ECM2423 – Artificial Intelligence and Applications Continous Assessment

James Calnan, jdc235@exeter.ac.uk

Deadline: Thursday 16th March

Lecturer: Dr Ayah Helal (a.helal@exeter.ac.uk)

Question 1.1

Describe how you would frame the maze solver as a search problem. In your own words, write a short description of how the maze solver can be seen as a search problem.

In order to frame a maze solver as a search problem you would first have to interpret the maze as a traversable graph, with the start and end points being known. Converting the maze to a traversable graph can be done by iterating over the nodes/vertices and then recording adjacent nodes in order to construct an adjacency list for each node. An adjacency matrix can also be constructed, however in large mazes this would result in a large 2-dimensional array which could put a strain on memory. Once you have constructed an adjacency list for each node, you can then use graph traversal algorithms in order to find a path from the start and end node.

Once the graph has been constructed, graph traversal algorithms like Depth-First Search and Breadth-First Search can be used to find a path between two nodes in the graph. It's important to note that given an imperfect maze (a maze that has multiple solutions), neither of the two algorithms are able to guarantee an optimal solution. To solve this problem more advanced algorithms like Dijkstra's or A* can be used. These two algorithms are able to provide an optimal solution as they operate with the concept of weights.

Overall, framing the maze solver as a search problem involves converting the maze into a graph, selecting a graph traversal algorithm and then finding a path between the two nodes.

```
def returnNeighbours(maze: List[List[str]], x: int, y: int) ->
↪  List[Tuple[int, int]]:
    """
    Returns the coordinates of all the neighboring nodes that are
↪   valid paths (represented by a `-` character).
```

```python
    Args:
        maze (list[list[str]]): A 2D list of strings representing
↪   the maze.
        x (int): The x-coordinate of the node in the maze.
        y (int): The y-coordinate of the node in the maze.

    Returns:
        list[tuple[int, int]]: A list of coordinates of all the
↪   neighboring nodes that are valid paths.
    """
    # Initialize an empty list to hold the neighbors of the given
    ↪  node
    neighbours = []

    # Check the node above the given node
    if y - 1 >= 0 and maze[y - 1][x] == "-":
        # If the above node is within the boundaries of the maze
        ↪  and is a valid path, add it to the list of neighbors
        neighbours.append((x, y - 1))

    # Check the node below the given node
    if not (y + 1) == len(maze) and maze[y + 1][x] == "-":
        # If the below node is within the boundaries of the maze
        ↪  and is a valid path, add it to the list of neighbors
        neighbours.append((x, y + 1))

    # Check the node to the right of the given node
    if maze[y][x + 2] == "-":
        # If the node to the right is a valid path, add it to the
        ↪  list of neighbors
        neighbours.append((x + 2, y))

    # Check the node to the left of the given node
    if maze[y][x - 2] == "-":
        # If the node to the left is a valid path, add it to the
        ↪  list of neighbors
        neighbours.append((x - 2, y))

    # Return the list of neighbors
    return neighbours


def buildAdjacencyList(maze: List[List[str]]) -> defaultdict:
    """
    Build an adjacency list representation of the given maze.
```

```
    Args:
        maze (list of lists): A 2D grid representation of the
↪   maze, where "-" represents a node and " " represents an edge

    Returns:
        adjacencyList (defaultdict): A defaultdict where each key
↪   is a tuple representing a node in the maze, and the value is
↪   a list of its neighbours
    """
    bounds = (0, len(maze), 0, len(maze[0])) # set the bounds of
    ↪   the maze as a tuple

    adjacencyList = defaultdict(list) # initialize an empty
    ↪   defaultdict to store the adjacency list representation of
    ↪   the maze

    for y in range(bounds[1]): # loop through all rows in the
    ↪   maze
        for x in range(0, bounds[3], 2): # loop through every
        ↪   other column in the maze
            if maze[y][x] == "-": # if the current position is a
            ↪   node, add its neighbours to the adjacency list
                adjacencyList[(x, y)] = returnNeighbours(maze, x,
                ↪   y)

    return adjacencyList
```

Question 1.2

Solve the maze using depth-first search. This question is further divided into parts. This is a guide to help your in the coding process. You do not need to submitted different versions of the code, but just the last updated version.

1. Briefly outline the depth-first algorithm

The depth-first algorithm is a graph traversal algorithm that explores a graph or tree by visiting as far as possible along each branch before backtracking. It can be implemented iteratively using a stack data structure or recursively using a function call stack. The algorithm works by starting at a designated node, visiting its neighbors, and then visiting each neighbor's unvisited neighbors. If a dead end is reached, it backtracks to a previous node with unvisited neighbours (using the stack data structure or function call stack) and continues the search from there until all nodes have been visited or the loop can be terminated once the goal node is reached[2].

While this traversal is happening a solution map can be constructed which will keep track of how each node was reached. This can then be used to backtrack from the goal node to the start node giving you a path.

2. Implement depth-first to solve the maze

The solution of the maze should print the locations in the map that the algorithm will take to reach from the start to the goal. This question can be solved using the file 'maze-Easy.txt".

See file /solutions/maze-Easy-DFS-Solution.txt

Or:

```
# X # # # # # # # # # # # # # # # # # #
# X X X X X X X X X X X X X X X - - # #
# # # # # - # # - # # - # # # X # - - #
# - # # # - # - - # # - # # - X # - # #
# - # - - - # # # # # # # # # X # - - #
# - - - # - - - - # - # # - # X # # # #
# - # # # # - # - - - - - - - X X X - #
# - # # - - - # # # # # # # # # # X # #
# - - - - # - - # # - - - - - - - X X #
# # # # # # # # # # # # # # # # # # X #
```

```python
from collections import deque, defaultdict
from typing import Dict, List, Tuple


def depthFirstSearch(adjacencyList: Dict[Tuple, List[Tuple]],
↪   root: Tuple, goal: Tuple) -> Tuple[Dict[Tuple, Tuple], int]:
    """
    Traverses a graph represented by an adjacency list, starting
↪   from a specified root node, and searches for a goal node.

    Args:
            adjacencyList (dictionary of list): a dictionary that
↪   maps each node in the graph to a list of its adjacent nodes.
            root (tuple): the node to start the search from.
            goal (tuple): the node to search for.

    Returns:
            If the goal node is found, returns a tuple containing
↪   a dictionary that maps each visited node to its parent in the
↪   search tree, and the number of nodes explored during the
↪   traversal.
            If the goal node is not found, returns a tuple
↪   containing None for the path dictionary, and the number of
↪   nodes explored during the traversal.
    """

    # Initialize an empty set to keep track of discovered nodes.
    discovered = set()

    # Initialize a double-ended queue to store nodes to visit.
    S = deque()

    # Add the starting node to the queue.
    S.append(root)

    # Initialize an empty dictionary to keep track of the path
    ↪   from each visited node to its parent.
    cameFrom = {}

    # Initialize a counter to keep track of the number of nodes
    ↪   explored during the traversal.
    nodesExplored = 0

    # While there are nodes in the queue:
    while S:
```

```
        # Pop the last node from the queue.
        v = S.pop()

        # Increment the node counter.
        nodesExplored += 1

        # If the current node is the goal, return the dictionary
        ↪  map and the number of nodes explored.
        if v == goal:
            return cameFrom, nodesExplored

        # If the current node has not been discovered yet:
        if v not in discovered:

            # Mark it as discovered.
            discovered.add(v)

            # For each adjacent node w:
            for w in adjacencyList[v]:

                # If w has already been discovered, skip it.
                if w in discovered:
                    continue

                # Otherwise, add w to the queue and record the
                ↪  path from v to w.
                S.append(w)

                # Set the parent of the neighbour to the current
                ↪  node
                cameFrom[w] = v

    # If the goal was not reached, return None for the path and
    ↪  the number of nodes explored.
    return None, nodesExplored
```

Depth-First Search Implementation:[2]

3. Update your algorithm to calculate some statistics about its performance

The number of nodes explored to find the path, the time of the execution, and the number of steps in the resulting path.

Results were obtained running the algorithm on a Ryzen 5 3600

Table 1: Statistics for Depth-first Search on maze-Easy.txt (time taken averaged over 5000 runs)

Statistic	**Values**
Nodes visited	79
Percentage of maze explored	95%
Solution Length	27
Time taken to solve the maze	0.0000358 seconds
Solution percentage	33%

4. Update your algorithm to be generalised to read any maze in the same format. Then calculate the paths and statistics for all ‘maze-Medium.txt”/ ‘maze-Large.txt” / ‘maze-VLarge.txt”.

See files:

1. /solutions/maze-Medium-DFS-Solution.txt
2. /solutions/maze-Large-DFS-Solution.txt
3. /solutions/maze-VLarge-DFS-Solution.txt

Table 2: Statistics for Depth-first Search on maze-Medium.txt (time taken averaged over 5000 runs)

Statistic	Values
Nodes visited	2027
Percentage of maze explored	22%
Solution Length	575
Time taken to solve the maze	0.0008746 seconds
Solution percentage	6%

Table 3: Statistics for Depth-first Search on maze-Large.txt (time taken averaged over 5000 runs)

Statistic	Values
Nodes visited	70512
Percentage of maze explored	85%
Solution Length	1050
Time taken to solve the maze	0.0378546 seconds
Solution percentage	1%

Table 4: Statistics for Depth-first Search on maze-VLarge.txt (time taken averaged over 5000 runs)

Statistic	Values
Nodes visited	134794
Percentage of maze explored	14%
Solution Length	4049
Time taken to solve the maze	0.0883288 seconds
Solution percentage	0%

Question 1.3

Suggest an improved algorithm for this problem.

1. Briefly outline another algorithm to solve the maze. That decision should be justified by what you expect to improve in the performance over depth-first search.

One possible algorithm to solve the maze is the A* search algorithm[1]. A* search is an informed greedy search algorithm that uses a heuristic function to guide the search more efficiently than depth first search. The heuristic function, which in this case will just be the Manhattan distance between two points, estimates the distance between the current node and the goal state by summing the absolute difference between the x and y values of the coordinates. A* search uses this information to prioritise the exploration of seemingly more promising paths.

By using A* search instead of depth-first search, we can expect to improve the performance of the algorithm in several ways. Firstly, A* search can identify the shortest path to the goal node more efficiently than depth-first search. This is because A* search uses the heuristic function to guide the search towards the most promising paths, leading to quicker convergence to the goal node. Secondly, the heuristic can be adjusted to be admissible (heuristic value never overestimates) which will allow for the most optimal solution to be found. This can be done by applying a multiplier to the value returned by the heuristic function. Additionally, A* search can avoid getting stuck in local optima by exploring other promising paths, unlike depth-first search which can get stuck exploring a sub-optimal path.

Furthermore, the A* search algorithm has a time complexity of $O(b^d)$[1], where b is the branching factor and d is the length of the shortest path. This means that A* search can be more efficient in terms of the number of nodes explored than depth-first search. Finally, A* search can be potentially quicker using a priority queue[3] to dequeue the next node with the lowest heuristic score.

However, there can also be some drawbacks to using the A* Search Algorithm. Firstly, the performance is heavily based on the heuristic function and if the heuristic isn't optimal the solution may not be the best possible and the performance can suffer greatly. Secondly, A* search can potentially be slower than DFS as it is required to keep track of all the of the nodes that have been visited and nodes that are waiting to be visited.

Overall, by using A* search, we can expect to achieve a more optimal solution to the maze compared to depth-first search as well as potentially reducing the time and space required to find that solution.

2. Implement the suggested algorithm to solve the maze. It should also calculate the same statistics about its performance as the previous depth-first algorithm. The solution of the maze should print the locations in the map that the algorithm will take to reach from the start to the goal.

See files:

1. /solutions/maze-Easy-ASTAR-Solution.txt
2. /solutions/maze-Medium-ASTAR-Solution.txt
3. /solutions/maze-Large-ASTAR-Solution.txt
4. /solutions/maze-VLarge-ASTAR-Solution.txt

Results were obtained running the algorithm on a Ryzen 5 3600.

Table 5: Statistics for A* (.8 Multiplier) on maze-Easy.txt (time taken averaged over 5000 runs)

Statistic	Values
Nodes visited	35
Percentage of maze explored	42%
Solution Length	28
Time taken to solve the maze	0.0000524 seconds
Solution percentage	33%

Table 6: Statistics for A* (.8 Multiplier) on maze-Medium.txt (time taken averaged over 5000 runs)

Statistic	Values
Nodes visited	456
Percentage of maze explored	5%
Solution Length	322
Time taken to solve the maze	0.0007493 seconds
Solution percentage	3%

Table 7: Statistics for A* (.8 Multiplier) on maze-Large.txt (time taken averaged over 5000 runs)

Statistic	**Values**
Nodes visited	41752
Percentage of maze explored	50%
Solution Length	975
Time taken to solve the maze	0.0673739 seconds
Solution percentage	1%

Table 8: Statistics for A* Search (.8 Multiplier) on maze-VLarge.txt (time averaged over 5000 runs)

Statistic	**Values**
Nodes visited	74999
Percentage of maze explored	8%
Solution Length	3692
Time taken to solve the maze	0.1387081 seconds
Solution percentage	0%

A* Algorithm:

```
from heapq import heappush, heappop
from typing import Dict, List, Tuple
from collections import deque, defaultdict

def aStarSolver(adjacencyList: Dict[str, List[str]], root: str,
↪  goal: str) -> Tuple[Dict[str, str], int]:
        """
        A* algorithm implementation to find the shortest path
↪   between two nodes in a graph.

        Args:
                adjacencyList (dictionary of list): a dictionary
↪   that maps each node in the graph to a list of its adjacent
↪   nodes.
            root (tuple): the node to start the search from.
            goal (tuple): the node to search for.

        Returns:
            If the goal node is found, returns a tuple containing
↪   a dictionary that maps each visited node to its parent in the
↪   search tree, and the number of nodes explored during the
↪   traversal.
```

```
        If the goal node is not found, returns a tuple
↪   containing None for the path dictionary, and the number of
↪   nodes explored during the traversal.
    """

    # Set the heuristic multiplier.
    multiplier = .8

    # Create a dictionary to hold the distances from the root
    ↪   to each node.
    # Initialize the distance to the root to be the heuristic
    ↪   distance.
    distance = defaultdict(lambda: float('inf'))
    distance[root] = heuristic(root, goal, multiplier)

    # Create a dictionary to hold the parent of each node in
    ↪   the shortest path from the root to that node.
    cameFrom = {root: None}

    # Create a binary heap priority queue to store the nodes.
    prioQueue = []

    # Enqueue the root with a priority of 0.
    heappush(prioQueue, (0, root))

    # Keep track of the number of nodes explored.
    nodesExplored = 0

    # Create a cache for the heuristic values.
    heuristicCache = defaultdict()


    # While there are nodes in the heap.
    while prioQueue:
            # Extract the node with the lowest priority.
            _, current = heappop(prioQueue)
            nodesExplored += 1

            # If the current node is the goal, return the
            ↪   shortest path.
            if current == goal:
                    return cameFrom, nodesExplored

            # For each neighbor of the current node.
            for neighbor in adjacencyList[current]:
```

```
# Calculate the tentative distance from
↪  the root to the neighbor through the
↪  current node.
tentative_distance = distance[current] +
↪  1 #heuristic(neighbor, current)

# If the tentative distance is less than
↪  the current distance to the neighbor,
↪  update the distance.
if tentative_distance <
↪  distance[neighbor]:
        distance[neighbor] =
        ↪  tentative_distance

        # Check if the heuristic value
        ↪  for the neighbor is already
        ↪  cached.
        if neighbor in heuristicCache:
                # Use the cached value.
                heuristic_value =
                ↪  heuristicCache[neighbor]
        else:
                # Calculate the heuristic
                ↪  value and cache it.
                heuristic_value =
                ↪  heuristic(neighbor,
                ↪  goal, multiplier)
                heuristicCache[neighbor]
                ↪  = heuristic_value

        # Calculate the priority of the
        ↪  neighbor as the sum of the
        ↪  tentative distance and the
        ↪  heuristic distance to the
        ↪  goal.
        priority = tentative_distance +
        ↪  heuristic_value
        # Enqueue the neighbor with the
        ↪  calculated priority.
        heappush(prioQueue, (priority,
        ↪  neighbor))
        # Set the parent of the neighbor
        ↪  to the current node.
        cameFrom[neighbor] = current
```

```
        # If there is no path from the root to the goal, return
        ↪  None for the path and the number of nodes explored.
        return None, nodesExplored


def heuristic(current: Tuple[int, int], goal: Tuple[int, int], m:
↪  int = 1) -> int:
    """
    Calculate the Manhattan distance between two nodes.

    Args:
            current (Tuple[int, int]): The current node.
            goal (Tuple[int, int]): The goal node.
            m (int): The weight to multiply the Manhattan
↪   distance by.

    Returns:
            int: The calculated heuristic value.
    """
    return (abs(current[0] - goal[0]) + abs(current[1] -
    ↪  goal[1])) * m
```

A* Search Algorithm Implementation[1]

4. Discuss based on your results, analysis how the suggested algorithm performed better than the depth-first search algorithm.

Based on the results obtained it can be seen that the A* algorithm is required to visited much fewer nodes than Depth-First Search in order to reach the goal node. However, despite needing to visited fewer nodes due to the overhead and more complex operations in the A* algorithm (ordering priority queue, storing heuristic values) the execution time is always longer. This can also be due to the nature of how I coded the adjacency list. When I create the adjacency list the neighbouring nodes are placed into the dictionary in this order: above the current node, below the current node, to the right of the current node and to the left of the current node. This means that when the neighbouring nodes are added to the stack in the depth first search it is the node below takes precedence over the nodes to the right and the left. This works well for the inputted mazes as the search starts from the top of the maze and works its way to the bottom. When this ordering is changed to below, right, left, above the performance of the algorithm suffers:

Table 9: DFS on maze-VLarge.txt with different adjacency list ordering

Statistic	**Values**
Nodes visited	778817
Percentage of maze explored	84%
Solution Length	6131
Time taken to solve the maze	0.60223 seconds
Solution percentage	0%

In the above example the depth first search visited 84% of the maze instead of just 14% with the optimal neighbour ordering. The solution length is also much longer at 6131 instead of 4049 with the original neighbour ordering.

Table 10: Comparison of solution lengths

Maze File	**Algorithm**	**Solution length**
maze-Easy.txt	A*	27
	DFS	27
maze-Medium.txt	A*	321
	DFS	575
maze-Large.txt	A*	974
	DFS	1050
maze-VLarge.txt	A*	3691
	DFS	4049

As can be seen from the table above (Table 10), the A* algorithm was able to find a solution which has a shorter length on all but the easy maze (which both algorithms found a value of 27). Based on these results the A* algorithm can be seen to performed better than the depth first search algorithm. Additionally, the A* algorithm wont suffer from the same ordering of the adjacency list problem that Depth-First Search suffers from due to the use of a priority queue.

Table 11: Comparison of maze exploration

Maze File	Algorithm	Nodes Visited	% of maze visited
maze-Easy.txt	A*	35	42%
	DFS	79	95%
maze-Medium.txt	A*	456	5%
	DFS	2027	22%
maze-Large.txt	A*	41,752	50%
	DFS	70,512	85%
maze-VLarge.txt	A*	74,999	8%
	DFS	134,794	14%

From the statistics in the table above (Table 11) it can be observed that the A* algorithm was required to visit less nodes than Depth-First Search in order to find a solution.

Table 12: Comparison of time taken to solve the maze

Maze File	A*	DFS
maze-Easy.txt	0.0000524 seconds	0.0000358 seconds
maze-Medium.txt	0.0007493 seconds	0.0008746 seconds
maze-Large.txt	0.0673739 seconds	0.0378546 seconds
maze-VLarge.txt	0.1387081 seconds	0.0883288 seconds

The above table (table 12) compares the time taken to solve the four different mazes using two algorithms: A* and Depth-First Search (DFS). The results show that, in most cases, DFS is faster than A*. However, it's important to note that A* is still the superior solution in terms of scalability and finding shorter paths, as demonstrated by its ability to find the solution without visited as many nodes as DFS. This means that A* algorithm is more efficient in situations where the goal state is far away from the start state, as it guides the search in the right direction.

One interesting observation is that A* was faster than DFS on the maze-Medium.txt file. This could be the ordering of neighbors/adjacent nodes. As mentioned earlier, DFS is advantageous for certain mazes, which explains its faster performance on most files.

Despite its longer completion times, A* is a more reliable and efficient solution overall. It's worth noting that A* found the shorter paths on all files, except for the maze-Easy.txt file which both algorithms found the shortest path.

Therefore, in cases where finding the shortest path is a priority or when dealing with larger mazes, A* should be the preferred algorithm.

References

[1] Wikipedia. A* search algorithm, 2023.

[2] Wikipedia. Depth-first search, 2023.

[3] Wikipedia. Priority queue, 2023.