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.

```
def returnNeighbours(maze, x, y):
    # Initialize an empty list to hold the neighbors of the given
    ↪  cell
        neighbours = []

        # Check the cell above the given cell
        if y - 1 >= 0 and maze[y - 1][x] == "-":
                # If the above cell 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 cell below the given cell
        if not (y + 1) == len(maze) and maze[y + 1][x] == "-":
                # If the below cell 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 cell to the right of the given cell
        if maze[y][x + 2] == "-":
                # If the cell to the right is a valid path, add
                ↪  it to the list of neighbors
                neighbours.append((x + 2, y))

        # Check the cell to the left of the given cell
        if maze[y][x - 2] == "-":
                # If the cell 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):
    bounds = (0, len(maze), 0, len(maze[0]))
    adjacencyList = defaultdict(list)

    for y in range(bounds[1]):
        for x in range(0, bounds[3], 2):
            if maze[y][x] == "-":
                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 or recursively using a function call stack. The algorithm works by starting at a designated node, visiting its neighbors, and then recursively visiting each neighbor's unvisited neighbors. If a dead end is reached, it backtracks to the previous node and continues the search from there until all nodes have been visited[3].

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:

1. maze-Easy-DFS-Solution.txt

```python
def depthFirstSearch(adjacencyList, root, goal):
    # Initialize an empty set to keep track of discovered
    ↪  vertices.
    discovered = set()

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

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

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

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

    # While there are vertices in the queue:
    while S:
        # Pop the last vertex from the queue.
        v = S.pop()

        # Increment the node counter.
        nodesExplored += 1

        # If the current vertex is the goal, return the path to
        ↪  it and the number of nodes explored.
        if v == goal:
            return cameFrom, nodesExplored

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

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

            # For each adjacent vertex 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)
                cameFrom[w] = v

    # If the goal was not reached, return 0 for the path and the
    ↪  number of nodes explored.
    return None, nodesExplored
```

Depth-First Search Implementation[1]

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.

Table 1: Statistics for Depth-first Search on maze-Easy.txt

Statistic	**Values**
Vertices visited	79
Percentage of maze explored	95%
Solution Length	28
Time taken to solve the maze	0.00356 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. maze-Medium-DFS-Solution.txt
2. maze-Large-DFS-Solution.txt
3. maze-VLarge-DFS-Solution.txt

Table 2: Statistics for Depth-first Search on maze-Medium.txt

Statistic	**Values**
Vertices visited	2027
Percentage of maze explored	22%
Solution Length	576
Time taken to solve the maze	0.00303 seconds
Solution percentage	6%

Table 3: Statistics for Depth-first Search on maze-Large.txt

Statistic	**Values**
Vertices visited	70512
Percentage of maze explored	85%
Solution Length	1051
Time taken to solve the maze	0.08182 seconds
Solution percentage	1%

Table 4: Statistics for Depth-first Search on maze-VLarge.txt

Statistic	**Values**
Vertices visited	134794
Percentage of maze explored	14%
Solution Length	4050
Time taken to solve the maze	0.13179 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[2]. A* search is an informed greedy search algorithm that uses a heuristic function to guide the search towards the goal state 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 state and the goal state, and A* search uses this information to prioritize the exploration of 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 be more efficient in terms of the number of nodes explored as it doesn't need to explore a path to completion like depth-first search. Secondly, the heuristic can be adjusted to make the algorithm less greedy and more resembling Dijkstra's which will allow for the most optimal solution to be found. Finally, A* search can can be potentially be quicker using a priority queue[4] to dequeue the next node with the lowest heuristic score.

Overall, by using A* search, we can expect to achieve a more optimal solution to the maze compared to depth-first search.

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

A* Algorithm

```
def aStar(adjacencyList, root, goal):
    # Set the heuristic multiplier to 10.
    multiplier = .8

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

    # Create a binary heap priority queue to store the vertices.
    # Create a priority queue to store the vertices.
    # heap = heapdict()
    prioQueue = PriorityQueue()

    prioQueue.put((0, root))

    # Enqueue the root with a priority of 0.
    # heap[root] = 0

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

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


    # While there are vertices in the heap.
    while not prioQueue.empty():
        # Extract the vertex with the lowest priority.
        # current, priority = heap.popitem()
        _, current = prioQueue.get()
        nodesExplored += 1

        # If the current vertex is the goal, return the shortest
        ↪  path.
        if current == goal:
            return cameFrom, nodesExplored
        # For each neighbor of the current vertex.
        for neighbor in adjacencyList[current]:
            # Calculate the tentative distance from the root to
            ↪   the neighbor through the current vertex.
            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.
                # heap[neighbor] = priority
                prioQueue.put((priority, neighbor))
                # Set the parent of the neighbor to the current
                ↪   vertex.
                cameFrom[neighbor] = current

    # If there is no path from the root to the goal, return 0 for
    ↪   the path and the number of nodes explored.
    return None, nodesExplored

# Heuristic function that calculates the manhatten distance
↪   between two points
def heuristic(current, goal, m=1):
        return (abs(current[0] - goal[0]) + abs(current[1] -
        ↪   goal[1])) * m
```

A* Search Algorithm Implementation

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

Table 5: Statistics for A* (.8 Multiplier) on maze-Easy.txt

Statistic	**Values**
Vertices visited	35
Percentage of maze explored	42%
Solution Length	28
Time taken to solve the maze	0.003 seconds
Solution percentage	33%

Table 6: Statistics for A* (.8 Multiplier) on maze-Medium.txt

Statistic	**Values**
Vertices visited	456
Percentage of maze explored	5%
Solution Length	322
Time taken to solve the maze	0.00696 seconds
Solution percentage	3%

Table 7: Statistics for A* (.8 Multiplier) on maze-Large.txt

Statistic	**Values**
Vertices visited	41752
Percentage of maze explored	50%
Solution Length	975
Time taken to solve the maze	0.23718 seconds
Solution percentage	1%

Table 8: Statistics for A* Search (.8 Multiplier) on maze-VLarge.txt

Statistic	**Values**
Vertices visited	74999
Percentage of maze explored	8%
Solution Length	3692
Time taken to solve the maze	0.47306 seconds
Solution percentage	0%

Table 9: Statistics for A* Search (0.6 Multiplier) on maze-VLarge.txt

Statistic	**Values**
Vertices visited	616003
Percentage of maze explored	67%
Solution Length	3692
Time taken to solve the maze	5.52698 seconds
Solution percentage	0%

References

[1] James Calnan. Maze-generation-python. https://github.com/jamescalnan/Maze-Generation-Python, 2023. GitHub repository.

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

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

[4] Wikipedia. Priority queue, 2023.