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 extremely big 2-dimensional arrays 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 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))

        # 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))

        # Return the list of neighbors
        return neighbours


def buildAdjacencyList(maze):
        adjacencyList = {}

        for y in range(len(maze)):
            for x in range(len(maze[y])):
                # Iterate over each point in the maze
                if maze[y][x] == "-":
                    # Check if the current point is a
                    ↪  node and if so add the
                    ↪  current nodes neighbours to
                    ↪  the dictionary
                    adjacencyList[(x, y)] =
                    ↪  returnNeighbours(maze, x, y)

        # Return the adjacency list
        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[4].

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

```
def depthFirstSearch(adjacencyList, root, goal):
        # Initialize an empty list to hold the vertices that have
        ↪  been discovered
        discovered = []
        # Initialize an empty stack to hold the vertices to be
        ↪  explored
        S = []
        # Initialize an empty dictionary to hold the vertices
        ↪  that lead to each discovered vertex
        cameFrom = {}

        # Add the root vertex to the stack
        S.append(root)

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

        # While there are still vertices to be explored
        while len(S) > 0:
                # Pop a vertex from the stack
```

```
        v = S.pop()
        # Increment the counter
        nodesExplored += 1

        # If the goal has been reached, return the
        ↪ cameFrom dictionary and the number of nodes
        ↪ explored
        if v == goal:
                return cameFrom, nodesExplored

        # If the vertex has not been discovered yet
        if v not in discovered:
                # Add it to the list of discovered
                ↪ vertices
                discovered.append(v)

                # For each neighbor of the current vertex
                for w in adjacencyList[v]:
                        # If the neighbor has already
                        ↪ been discovered, skip it
                        if w in discovered:
                                continue

                        # Add the neighbor to the stack
                        ↪ to be explored
                        S.append(w)
                        # Record the current vertex as
                        ↪ the one that leads to the
                        ↪ neighbor
                        cameFrom[w] = v

# If the goal was not found, return 0 for both the
↪ cameFrom dictionary and the number of nodes explored
return 0, 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	53
Percentage of maze explored	63%
Solution Length	28
Time taken to solve the maze	0.00401 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	6695
Percentage of maze explored	73%
Solution Length	400
Time taken to solve the maze	1.23553 seconds
Solution percentage	4%

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

Statistic	Values
Vertices visited	10939
Percentage of maze explored	13%
Solution Length	1121
Time taken to solve the maze	3.42631 seconds
Solution percentage	1%

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

Statistic	**Values**
Vertices visited	87822
Percentage of maze explored	9%
Solution Length	3692
Time taken to solve the maze	722.31954 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. The heuristic function 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 quicker through optimisations such as a binary heap[3] priority queue to dequeue the next node with the lowest heuristic score.

Overall, by using A* search, we can expect to achieve a faster and 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

Binary Heap:

```
class BinaryHeapPriorityQueue:
        # Init method which is called when an object of the class
        ↪  is instantiated
    def __init__(self):
            # Initialize an empty list to store the heap and a
            ↪  variable to store the size of the heap
        self.heap = []
        self.size = 0

    # Method called cameFrom that takes an index as input and
    ↪  returns the index of its parent node
    def cameFrom(self, i):
        return (i - 1) // 2

    # Method called left_child that takes an index as input and
    ↪  returns the index of its left child
    def left_child(self, i):
        return 2 * i + 1

    # Method called right_child that takes an index as input and
    ↪  returns the index of its right child
    def right_child(self, i):
        return 2 * i + 2

    # Removes and returns the element with the lowest priority
    ↪  from the heap
    def extractMin(self):
            # If the size of the heap is 0, return None
        if self.size <= 0:
            return None
        # If the size of the heap is 1, remove the element and
        ↪  decrease the size by 1
        if self.size == 1:
            self.size -= 1
            return self.heap.pop()
            # Otherwise, remove the root element from the heap,
            ↪  replace it with the last element in the heap, and
            ↪  decrease the size by 1
        root = self.heap[0]
```

```python
        last_element = self.heap.pop()
        self.size -= 1
        # If the heap is not empty, call _minHeapify to maintain
        ↪  the min-heap property
        if self.size > 0:
            self.heap[0] = last_element
            self._minHeapify(0)
        # Return the root element that was removed
        return root

    # Adds an element to the heap with a given priority and value
    def enqueue(self, priority, value):
            # Append the element to the end of the heap and
            ↪  increase the size by 1
        self.heap.append((priority, value))
        self.size += 1
        # Bubble up the element to its correct position in the
        ↪  heap
        i = self.size - 1
        while i != 0 and self.heap[self.cameFrom(i)][0] >
        ↪  self.heap[i][0]:
            self.heap[i], self.heap[self.cameFrom(i)] =
            ↪  self.heap[self.cameFrom(i)], self.heap[i]
            i = self.cameFrom(i)

    # Maintains the min-heap property for a given node in the
    ↪  heap
    def _minHeapify(self, i):
        l = self.left_child(i)
        r = self.right_child(i)
        smallest = i
        # If the left child has a lower priority than the current
        ↪  node, set it as the smallest
        if l < self.size and self.heap[l][0] < self.heap[i][0]:
            smallest = l
        # If the right child has a lower priority than the
        ↪  current node, set it as the smallest
        if r < self.size and self.heap[r][0] <
        ↪  self.heap[smallest][0]:
            smallest = r
        # If the smallest is not the current node, swap the
        ↪  elements and recursively call _minHeapify on the
        ↪  smallest node
        if smallest != i:
            self.heap[i], self.heap[smallest] =
            ↪  self.heap[smallest], self.heap[i]
```

```
            self._minHeapify(smallest)

    # Method to return the size of the queue
    def getSize(self):
        return self.size

    # Method to return true if an item is in the queue
    def contains(self, value):
            # Iterate over the items in the queue and check if
            ↪  its equal
        for _, v in self.heap:
            if v == value:
                return True
        return False

    # Method to return the contents of the queue
    def toString(self):
        return str(self.heap)
```

Binary Heap Priority Queue Implementation

A* Algorithm

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

        # 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 = {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.
        heap = BinaryHeapPriorityQueue()

        # Enqueue the root with a priority of 0.
        heap.enqueue(0, root)

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

        # While there are vertices in the heap.
        while heap.getSize() > 0:
                # Extract the vertex with the lowest priority.
                current = heap.extractMin()[1]
                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] +
                        ↪  heuristic(neighbor, current)
```

```
                        # If the tentative distance is less than
                        ↪  the current distance to the neighbor,
                        ↪  update the distance.
                        if neighbor not in distance or
                        ↪  tentative_distance <
                        ↪  distance[neighbor]:
                                distance[neighbor] =
                                ↪  tentative_distance
                                # Calculate the priority of the
                                ↪  neighbor as the sum of the
                                ↪  tentative distance and the
                                ↪  heuristic distance to the
                                ↪  goal.
                                priority = tentative_distance +
                                ↪  heuristic(neighbor, goal,
                                ↪  multiplier)
                                # Enqueue the neighbor with the
                                ↪  calculated priority.
                                heap.enqueue(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 0, 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* (10 Multiplier) on maze-Easy.txt

Statistic	**Values**
Vertices visited	35
Percentage of maze explored	42%
Solution Length	28
Time taken to solve the maze	0.0176 seconds
Solution percentage	33%

Table 6: Statistics for A* (10 Multiplier) on maze-Medium.txt

Statistic	**Values**
Vertices visited	469
Percentage of maze explored	5%
Solution Length	322
Time taken to solve the maze	0.02187 seconds
Solution percentage	3%

Table 7: Statistics for A* (10 Multiplier) on maze-Large.txt

Statistic	**Values**
Vertices visited	14405
Percentage of maze explored	17%
Solution Length	1097
Time taken to solve the maze	0.128 seconds
Solution percentage	1%

Table 8: Statistics for A* Search (10 Multiplier) on maze-VLarge.txt

Statistic	**Values**
Vertices visited	117216
Percentage of maze explored	12%
Solution Length	3738
Time taken to solve the maze	1.72872 seconds
Solution percentage	0%

Based on the results shown in the tables, the A* Search algorithm with a binary heap and heuristic multiplier of 10 performs in general much better than Depth-First Search.

However, it can be observed that on the easy maze (maze-Easy.txt) Depth-First Search completed the maze much quicker, 0.0176 seconds for A* and 0.00401 seconds for DFS. It can be seen also that A* required less of the maze to be explored in order to reach the end vertex, 35 vertices or 42% explored for A* and 53 vertices or 63% explored for DFS.

On the larger mazes, the A* Search algorithm performs much better than Depth-First Search finding a solution to the mazes in a much shorter length of time.

On the medium maze (maze-Medium.txt), the A* Search algorithm took 0.02187 seconds to find a solution whereas Depth-First Search took 1.23553 seconds. Additionally, the A* search algorithm found a solution which had a length of 322 vertices whereas the solution found by Depth-First Search had a length of 400 vertices.

On the large maze (maze-Large.txt), the A* Search algorithm took 0.128 seconds to find a solution whereas Depth-First Search took 3.42631 seconds. The A* search algorithm found a solution which had a length of 1097 vertices whereas the solution found by Depth-First Search had a length of 1121 vertices.

On the very large maze (maze-VLarge.txt), the A* Search algorithm took 1.72872 seconds to find a solution whereas Depth-First Search took much longer at 722.31954 seconds or 12 minutes. The A* search algorithm found a solution which had a length of 3738 vertices whereas the solution found by Depth-First Search had a length of 3692 vertices. In this case the A* Search Algorithm found a solution which had a higher length than DFS, when adjusting the heuristic multiplier to 0.6 the A* Search Algorithm has the following statistics:

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%

When adjusting the heuristic multiplier to 0.6, the algorithm will more closely resemble Dijkstra's algorithm, which will result in the best possible path. Which in this case is a solution length of 3692. In doing this you make the algorithm less greedy and more comprehensive and therefore there are more vertices visited. 117216 vertices with a heuristic multiplier of 10 and 616003 vertices visited with a heuristic multiplier of 0.6.

When comparing the A* algorithm with a less greedy heuristic (0.6 multiplier) to DFS, despite the A* algorithm having visited more than six times more vertices (67% of the maze explored vs 9%) it still has a much quicker completion time (5.52698 seconds vs 722.31954 seconds), this is due to the binary heap data structure that is used to efficiently sort the vertices so that when the vertex with the minimum heuristic value is dequeued it can be done so quickly.

Overall, the A* Search algorithm performs much better than Depth-First Search and through the use of a heuristic you can also adjust the greediness of the algorithm to make it closely resemble Dijkstra's Algorithm that will allow for the an optimal solution.

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. Binary heap, 2023.

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