CSE – 537 (ARTIFICIAL INTELLIGENCE)
Project 1 Report

Submitted by – Gaurav Ahuja (111500318), Rakshit Gautam (111168264)

Question 1- DFS

Implementation -
1) Used *iterative* version of DFS.
2) Used **Stack** data structure as fringe list
3) Explored the nodes exactly in the order as it happens in recursive version of DFS
4) Initially pushed the start state in the fringe list
5) Until the goal state is not found or fringe list is not empty, we do the following -
    - Get the top element state from the fringe list, and test if it is a goal state
    - If yes, then break from the while loop
    - If not, add **a** non-visited successor of current state into the fringe list, and mark it visited so that it is not inserted again
    - If currentState has no unvisited successor, pop it from the fringeList

The states present in fringeList when goal state is found give the path from start state to goal state.

Question 2 - BFS

Implementation -
1) Used Q**ueue** data structure as fringe list (stores the states)
2) Initially pushed the start state in the fringe list
3) Until the goal state is not found or fringe list is not empty, we do the following -
    - Pop a state from the fringe list, and test if it is a goal state
    - If yes, then break from the while loop
    - If not, add the non-visited successors of current state into the fringe list, and mark them visited so that these are not inserted again
    - Also, store the predecessor of non-visited states. This helps in backtracking the path when we reach the goal state.

Predecessors are stored in a dictionary. Predecessor of a state is a (state, move) tuple. If we make the move m from state s_i to reach state s_j then we use s_j as key and map (s_i, m) as value to this key.

Question 3 – Uniform Cost Search

Implementation -
1) Used **Priority Queue** data structure as fringe list (stores the (state, minimum cost to reach this state from the start state) tuples, with **cost as priority and state as value**)
2) Initially pushed the (start state, 0) in the fringe list
3) Until the goal state is not found or fringe list is not empty, we do the following -
    - Pop a (current state, current cost) tuple from the fringe list, and test if the current state

is a goal state

- If yes, then break from the while loop
- If not, update or add the (successor state, cost) tuples into the fringe list, where successor state represents the successor of current state and cost represents the minimum cost of reaching successor state from the start state. If the successor has already been added to the priority queue before, we update the cost of (successor state, previous cost) tuple to the minimum of (previous cost, current cost + edge cost)
- Also, store the predecessor of successor states. This helps in finding the minimum cost path when we reach the goal state. The predecessor of a successor state is the state, using which we can reach the successor state with minimum cost.

Predecessors are stored in a dictionary. Predecessor of a state is a (state, move) tuple. If we make the move m from state s_i to reach state s_j then we use s_j as key and map (s_i, m) as value to this key.

Question 4 - A-star

Implementation -
This is implemented in a similar way as Uniform Cost Search. Instead of storing the (state, actual cost to reach the state from the start node) tuples into the priority queue, we store the (state, actual cost to reach the state from the start node + heuristic cost to reach the goal state from this intermediate state) tuples into the priority queue. Updates are done using this new cost rule.

Question 5 – State for Corners Problem

The problem is thought of as -
We have 4 dots, one in each corner placed for the pac-man. The pac-man is supposed to eat the 4 dots while traversing the shortest path possible.

A state is identified by the parameters – current position of pac-man on the maze, positions of the 4 corners on the maze and 4 integers representing if a corner is visited by the pac-man.

State = ((i,j), (corner_location_1 , visited_status_1), (corner_location_2, visited_status_2), (corner_location_3, visited_status_3), (corner_location_4, visited_status_4))

where (i,j) is the current position of the pac-man on the maze, corner_location_i is the location of i_th corner of the maze, and visited_status_i is 1 if i_th corner is visited else it is marked as 0

Only this information is sufficient to represent the CornerSearchProblem. We do not need to store anything else in order to test whether a state is a goal state and return the successors of a state.

Goal Test – To test whether a state is a goal state, check if all of visited_status_i are 1 for i in [1,2,3,4]. If yes, then the state is a goal state else not.

Successors of a state – Pac-man has choice to move UP, DOWN, LEFT or RIGHT. If the resultant position after making a move is legal, then we check if the resultant position is a corner position and accordingly create the successor state.

Question 6 – Heuristic for Corners Problem

Relax the game – Assume that there are no walls in the maze. So Manhattan Distance becomes a lower bound on the distance between two points. If walls are introduced in the game, actual distance between any two points will always be greater than or equal to manhattan distance between the two points.

Nodes expanded for mediumCorners = 741

Suppose pac-man is at position *p* and he is **yet to** visit corners *C1.....Cn*.
Pac-man can visit all the **n** corners in **n!** Permutations.

Suppose one such permutation *perm* is (Corner1, Corner4, Corner2).
Let **C(*perm*)** = ManhattanDistance(CurrentPosition, Corner1) + ManhattanDistance(Corner1, Corner4) + ManhattanDistance(Corner4, Corner2)

Cost incurred to traverse this path (CurrentPosition → Corner1 → Corner4 → Corner2) is less than or equal to C(perm).

We take the minimum cost across all these permutations and use this as a heuristic.
H(state) = minimum(C(permutation) for permutation in [all possible permutations of yet-to-visit-corners])

Admissibility of Heuristic -
Suppose the pac-man is at position currentPosition and is yet to visit the corners C1, C2, C3, C4.
Suppose in the least cost path, pac-man visits the corners in order C2, C1, C4, C3.
Let,
cost_actual = ActualCost(currentPosition, C2) + ActualCost(C2, C1) + ActualCost(C1, C4) + ActualCost(C4, C3)
cost_manhattan = ManhattanDistance(currentPosition, C2) + ManhattanDistance(C2, C1) + ManhattanDistance(C1, C4) + ManhattanDistance(C4, C3)

Since,
ActualCost(currentPosition, C2) >= ManhattanDistance(currentPosition, C2)
ActualCost(C2, C1) >= ManhattanDistance(C2, C1)
ActualCost(C1, C4) >= ManhattanDistance(C1, C4)
ActualCost(C4, C3) >= ManhattanDistance(C4, C3)

Summing all the 4 inequalities listed above, we can say that **cost_actual>=cost_manhattan**

Since our heuristic always chooses the minimum of C(permutation), the Heuristic Value for currentPosition is <= cost_manhattan. Thus, **cost_actual >= cost_manhattan >= heuristic value**

This is valid for any currentPosition in general.
Thus, H(state)<=H*(state)

This makes our heuristic admissible.

Consistency of Heuristic -

Suppose pac-man is at position *p* and he is **yet to** visit corners *C1.....Cn*. Also assume that while computing the H(currentState), the order visit of corners which had minimum cost was given by the permutation *perm=(*C1, C2, C3*).*

If the pac-man makes *a* move (reaching a new state n'), then the Manhattan distance from current position to C1 changes by +1 or -1, and the Manhattan distance from C1 → C2 and C2 → C3 does not change.

H(n') = H(currentState) + 1 or H(currenState) - 1

For consistency, the following condition should be met :

H(currentState) <= ActualCost(currentState → n') + H(n')

where ActualCost(currentState → n') is 1

In both the cases mentioned above, the consistency condition is met.

Since, the above condition holds for a single move of pac-man, it also holds after pac-man has made n moves. Proving consistency of heuristic after one move is sufficient to prove consistency of heuristic after n moves.

Hence, our heuristic is consistent.

Question 7 – FoodHeuristic for FoodSearchProblem

Relax the game – Assume that there are no walls in the maze. So the distance between two points is Manhattan Distance. If walls are introduced in the game, distance between any two points will always be greater than or equal to manhattan distance between the two points.

Heuristics tried:

1) Number of food items remaining on the grid (more than 12000 nodes expanded)
2) Maximum Manhattan distance between pac-man's current position and a food point (more than 9500 nodes expanded)

Heuristic used – Horizontal distance between pac-man's current position and farthest point in horizontal direction + Vertical distance between pac-man's current position and farthest point in vertical direction

Nodes expanded for tinySearch = 9419

Admissibility of Heuristic -

To eat all the food dots in the maze from its current position, pac-man needs to go to the farthest point in horizontal direction and that in vertical direction (*They can be the same point, in which case our heuristic will be equal to manhattan distance between current position and the farthest food position*). Since our heuristic always underestimates the actual cost to eat all the food dots, it is admissible.

Consistency of Heuristic -

Suppose the pac-man is currently at position (px, py) and the maximum distances from any food

point in x and y direction are fx and fy respectively.

Now suppose the pac-man makes a move in horizontal direction, then fy will not change and fx can either change to (fx-1) or (fx+1) or remain fx.
H(new_state) = fx+fy+[-1,0,1] = H(old_state) + [-1,0,1]
H(old_state) = H(new_state) + [1,0,-1]
H(old_state)<= H(new_state) + 1
H(old_state)<= H(new_state) + ActualCost(old_state → new_state)

Similar argument holds when pac-man makes a move in vertical direction.

Since, the above condition holds for a single move of pac-man, it also holds after pac-man has made n moves. Proving consistency of heuristic after one move is sufficient to prove consistency of heuristic after n moves.

Hence our heuristic is consistent.