CSE 537 Artificial Intelligence

Assignment 1 Report

Submitted By

Prakhar Avasthi
111432497

Rajat Jain
111452437

Q1. Depth First Search

Data Structure Used

- **Stack**: - To keep track of the nodes we have explored at every level down the tree. We are popping the element from the stack and putting its unvisited successors into the stack. We are doing this procedure until the stack is empty or we hit the goal which is our destination.
- **Array** (Visited): - To keep track of the nodes we have already visited so as not to visit them again in the procedure or otherwise it will go into an infinite loop.
- **Array** (Path): - To keep track of the node's parents. When we hit our goal, we extract the information from this array (to know from where we reached the goal) and return it.

Q2. Breadth First Search

Data Structure Used

- **Queue**: To keep track of the nodes we have explored at every level down the tree. We are popping the element from the queue and putting its unvisited successors into the queue. We are doing this procedure until the queue is empty or we hit the goal which is our destination.
- **Array** (Visited): To keep track of the nodes we have already visited so as not to visit them again in the procedure or otherwise it will go into an infinite loop.
- **Array** (Path): To keep track of the node's parents. When we hit our goal, we extract the information from this array (to know from where we reached the goal) and return it.

Q3. Uniform Cost Search

Data Structure Used

- **Priority Queue**: To keep track of the nodes we have explored at every level down the tree. After popping a node from a priority queue, we are putting its unvisited successors along with their cost as priority into the queue. We are selecting the next nodes based on the minimum cost (cost of path from the source node) to reach that node. Since we always need the minimum path at every stage, we are taking priority queue as priority queue will always pop node with least priority. We are doing this procedure until the queue is empty or we hit the goal which is the destination.

- **Array** (Visited): To keep track of the nodes we have already visited so as not to visit them again in the procedure or otherwise it will go into an infinite loop.
- **Array** (Path): To keep track of the node's parents. When we hit our goal, we extract the information from this array (to know from where we reached the goal) and return it.

Q4. A* Search

Data Structure Used

- **Priority Queue**: To keep track of the nodes we have explored at every level down the tree. After popping a node from a priority queue, we are putting its unvisited successors into the queue. We are selecting the next nodes based on the minimum cost (cost of node from the source node + node's heuristic) to reach that node. Since we always need the minimum path at every stage, we are taking priority queue. We are doing this procedure until the queue is empty or we hit the goal which is the destination.

- **Array** (Visited): To keep track of the nodes we have already visited so as not to visit them again in the procedure or otherwise it will go into an infinite loop.

- **Array** (Path): To keep track of the node's parents. When we hit our goal, we extract the information from this array (to know from where we reached the goal) and return it.

Q5. Corners Problem

As in this problem, Goal state is reached when Pac-Man touches all the corners and it should take a shortest path.

There can be many paths on which Pac-Man touches all the corners and therefore in each state we must maintain the corners that are reached in that path.

Therefore, we need to maintain a dictionary of corners which is reached in a path. Below modifications were required to solve this problem.

- State
  As mentioned above, at any state, we need to maintain position, direction, cost and dictionary of corners reached before that state.
  A typical state will be **[(position, direction, cost), {}]**

- Starting State
  Since we don't have any direction or cost at start state, so Starting state will be just like state but without direction and cost.
  getStartState will return start state as (position, {})

- Goal State
  A node with all corners in the dictionary of the state is a goal state, so in **isGoalState** method, we will check whether the length of dictionary is the total number of corners.

- Get Successors
  In **getSuccessors** method, we will find the nearby successors of the provided state and if provided state is a corner then we will append current state in the dictionary of the successors.

Q6. Heuristic for the corner problem:

Heuristics is the lower bound on the cost of reaching the goal. In Corner Heuristics, the Goal is to reach all the corners with shortest path. The heuristic that we have implemented, firstly, finds the Manhattan distance from state (S) to nearest corner (C_i). Then it finds the Manhattan distance from C_i to nearest corner (C_j) and continues to do until the goal is reached.

This Heuristic is both admissible and consistent, because, firstly, the greedy approach for finding Manhattan distance from state S to Goal G will ensure that cost of shortest path will not be below Heuristic. Secondly, cost between two states can be equal or greater that Manhattan distance, which will ensure that cost + heuristic will be monotonically increasing and hence it will be consistent also.

Heuristics Calculation

- We are calculating the node's distance from the nearest unvisited corner. (Let's say d)
- Then from that corner, we are taking the minimum distance from that corner to all the other unvisited corners (Let's say c1 + c2 + c3)
- We then add the distance calculated in step 2 and add it to the distance calculated in step 1 to obtain the heuristic for that node as shown in below diagram.
  Heuristic = d + (c1 + c2 + c3)

Q7. Food Heuristic

In Food Heuristics, the Goal is to collect all the food. The heuristic that we have implemented, firstly, finds the Manhattan distance from state (S) to farthest corner (C_i) in food grid.

This Heuristic is both admissible and consistent, because, firstly, to reach goal, all foods needs to be collected and Manhattan distance from any state to farthest food is the lowest possible cost to Goal. In case, where all food are in a straight line, Heuristic value will be equal to cost of reaching Goal. Secondly, cost between two states can be equal or greater that Manhattan distance, which will ensure that cost + heuristic will be monotonically increasing and hence it will be consistent also

Heuristics Calculation

- We are calculating the Manhattan distance to the farthest food in food grid of the state.
- At any state, cost of reaching a goal state will always be higher or equal but not lesser than the Manhattan distance to farthest food.
- For example: all foods are next to each other, Goal state will be when all food is finished.
- Cost of reaching the goal state in above case will be the Manhattan distance of farthest food, as it will be collected last.
- Therefore, we can be sure that our Heuristics is both admissible and consistent.