CS512– Artificial Intelligence

Lab Assignment - 1

By - Komal

2016csb1124

Q1) Finding a Fixed Food Dot using Depth First Search

Implementation details: The data structure used is Stack. For tracking path from start state to goal, I have used a python dictionary. In this dictionary, key is the state and value is a tuple of parent state and action required to reach this state from parent state. Also there is a list which stores all the expanded states so far. A state is not expanded if it is already in this list. Final path is tracked from the goal state to the start state using dictionary and then this path is reversed and returned.

1. python pacman.py -l tinyMaze -p SearchAgent

   Path length - 10 and search nodes expanded - 15

2. python pacman.py -l mediumMaze -p SearchAgent

   Path length - 130 and search nodes expanded - 146

3. python pacman.py -l bigMaze -z .5 -p SearchAgent

   Path length - 210 and search nodes expanded - 390

❏ Is the exploration order what you would have expected?

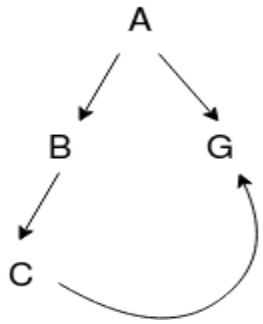
Yes, the exploration order is as expected in DFS.

❏ Does Pacman actually go to all the explored squares on his way to the goal?

No, the Pacman doesn't go to all the explored squares while going from initial state to goal.

❏ Does DFS find a least cost solution?

No, the solution returned by DFS is not least cost. This happens because DFS searches the deeper nodes first. For example: In the figure given below, least cost solution is A → G but DFS returns a path A → B → C → G. (A is the start state and G is the goal state)



==========

Q2) Breadth First Search

Implementation details: The data structure used is Queue. Just like DFS, there is a dictionary to keep track of the path. There is a list which stores visited nodes so that a state is not added to the queue if it is already there in queue. Final path tracking is same as used in DFS.

1. python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs

   Path length - 8 and search nodes expanded - 15

2. python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

Path length - 68 and search nodes expanded - 269

3. python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs

   Path length - 210 and search nodes expanded - 620

❏ Does BFS find a least cost solution?

Yes, BFS finds a least cost solution because it searches level by level. The search contours are spherical. For the above example, BFS would give the solution as A ➔ G.

==========

Q3) Uniform-Cost Graph Search

Implementation details: The data structure used is Priority Queue. Again, a dictionary is used for tracking path. There is a list which stores expanded nodes so far. A state is not expanded if it is already in this list. Here, a state is a tuple of coordinates and cost of reaching that node from start state. If there are two paths having different costs to reach the same state, then it is actually considered as two different states because state tuple differs for both.

1. python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

   Path cost - 68 and search nodes expanded - 269

2. python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

   Path cost - 1 and search nodes expanded - 186

3. python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

   Path cost - 68719479864 and search nodes expanded - 108

==========

Q4) A* Search

Implementation details: The only difference between UCS and A* is the cost on which the priority queue sorts the elements. Here, heuristic value ie the estimated cost from a state to goal is also used along with cost from start to that state.

1. python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar, heuristic=manhattanHeuristic

   Path cost - 210 and search nodes expanded - 549

2. python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=ucs, heuristic=manhattanHeuristic

   Path cost - 210 and search nodes expanded - 620

Different search strategies tested on openMaze:

1. python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=bfs

   Path cost - 54 and search nodes expanded - 682

2. python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=dfs

   Path cost - 298 and search nodes expanded - 576

3. python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=ucs

   Path cost - 54 and search nodes expanded - 682

4. python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

   Path cost - 54 and search nodes expanded - 535

DFS gives high cost solution whereas others give least cost solution because in this case cost of each action is equal. BFS and UCS give same answer because of equal cost of every step and A* helps in finding the solution faster.

==========

Q5) Finding All the Corners

Implementation details: The state representation is a tuple of coordinates and four booleans. This boolean is 0 if a corner still has food, otherwise it is 1. The order of corners is fixed ie (1,1), (1,top), (right, 1), (right, top). A state is goal state when all the four booleans become 1. For successors, the four booleans are same as that of parent's but if the successor state is one of the corners, then the corresponding boolean becomes 1.

1. python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

   Path cost - 28 and search nodes expanded - 252

2. python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

   Path cost - 106 and search nodes expanded - 1966

==========

Q6) Corners Problem: Heuristic

Heuristic Function: The distance measure used is manhattan distance. Heuristic value for a state is calculated as follows:

A. Out of the remaining corners (ie corners still having food), each corner is picked one by one and then steps B and C are followed. Now the current position is the picked corner.

B. From the current position, we calculate distance to the nearest corner (manhattan distance) and add it to the distance calculated so far.
C. Step B is repeated till there is a corner left with food.
D. Out of various distance values calculated for each remaining corner, we pick the smallest one and use it as heuristic value.

Steps B and C actually follow a greedy approach. These steps are repeated for each remaining corner because going to the nearest corner initially is not a good heuristic. This heuristic is admissible because it doesn't calculate actual cost (inculcating cost of walls), rather uses manhattan distance.

1. python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic

   Path cost - 106 and search nodes expanded - 741

==========

Q7) Eating All The Dots

Heuristic Function: The heuristic function used here is based on similar lines as for the above problem. Here instead of four corners, there can be varying number of dots. Step A written above is repeated for all the dots one by one. Steps B and C are repeated till all the dots are covered.

1. python pacman.py -l trickySearch -p AStarFoodSearchAgent

   Path cost - 60 and search nodes expanded - 6761

2. python pacman.py -l testSearch -p AStarFoodSearchAgent

   Path cost - 7 and search nodes expanded - 12

==========

Q8) Suboptimal Search

Implementation details: A state is a goal state of AnyFoodSearchProblem if that state contains food. For finding path to closest dot, AnyFoodSearchProblem can be solved using BFS as the search strategy as BFS searches in all directions and finds the closest goal.

Going to the closest dot doesn't give shortest possible path. For example: consider the below maze where A, B and D locations contain food. If pacman P goes to the closest dot always, then the path would be P → A → B → D which has a cost of 84 (BD is diagonal). Whereas shortest path is P → B → A → D which has a cost of 76.