Sujay Jakka

Svj0007@auburn.edu

COMP 5600

February 21, 2025

Assignment 1 Report

In this report, I will detail my algorithmic and design choices for Assignment 1. This report will consist of 8 sections, each dedicated to explaining my thought process for each question of the assignment. Furthermore, most of everything that is said in this report is also commented in my code to prevent any confusion.

Question 1

When implementing the Depth First Search function I decided to go with the iterative approach rather than the recursive approach. I keep track of my explored states with a set called “explored” and I keep track of my frontier with a stack called “dfs_stack”. Furthermore, I also have an additional stack that stores the corresponding path for each state in dfs_stack. The path to each state is an array containing the actions to a specific state. I loop until either the goal state is found or if the frontier(stack) is empty. I pop from my “dfs_stack” to get the current state and I also pop from my “paths” stack to get the corresponding path of the current state. I check if my current state is the goal state, if it is I then return the path of the goal state and terminate my function. Next, I add my current state to explored. It is important to note that when implementing DFS iteratively it is important to add a state to explored once its popped from the stack not when the algorithm discovers the state as a successor state. In the recursive approach, you are able to implement it both ways. However, in the iterative approach if you add the successor states to explored when you are iterating over them, depending on the graph, it may cause unnecessary backtracking causing the algorithm to not behave in a traditional DFS manner. Back to the algorithm, now the algorithm loops through the current state's successor states and add them to "dfs_stack" if they have not been explored and also adds the successor state's corresponding path of action to "paths".

Question 2

For Breadth First Search, my algorithm was similar to my algorithm for Depth First Search with some key differences. Just like my algorithm for DFS I had a set to keep track of explored states called "explored". However, unlike DFS I have initialized queues for both the frontier and paths of actions of states called "queue" and "paths" respectively. Next, I add my start state to my explored set and push it onto the "queue"(frontier) as well. Furthermore, I also pass an empty list of actions to "paths" for the start state as it took no actions to reach the start state. Similar to my DFS algorithm, I loop until the goal state is reached or if the frontier is empty. The algorithm then pops from both queues to get the current state from the frontier and its corresponding path of actions from the frontier. The algorithm checks if the current state is the goal state, if it is, it will return the current state. Lastly, the algorithm will loop through the current state's successor states and check if they have been explored. If they have not been explored, we add the successor states to the explored set and push them onto the frontier. Additionally, we also push the path of actions that led to this state as well onto the "paths" queue. One important thing to note is that unlike DFS, I am adding states to my explored set when they are successor states which will avoid the problem of pushing the same successor states onto the frontier.

Question 3

Now question 3 asks for a Uniform Cost Search implementation. The algorithm begins with initializing a set for explored states called “explored” and initializing a priority queue called “priority_queue”. Next I initialize two dictionaries. One dictionary stores the path of actions to a particular state where the state is the key. The other dictionary stores the cost to reach a state where the state is the key. The dictionaries are named “path_to_state” and “cost_to_state” respectively. I push my start state onto the “priority_queue”(frontier) with a cost of 0 as it took no actions to reach the state. Just like the previous two algorithms, I loop until I reached the goal state or have no more states in my frontier. I then pop from my frontier, to get the current state. The algorithm then checks if the current state is the goal state, if it is the algorithm will return the LEAST cost path to the goal state. If the current state is not the goal state, I then add the current state to my explored set. Next, I loop through the successor states and check if the successor states have been explored. If they have not, I find the cost of the successor state from the current state by adding the cost to get to the current state plus the cost to find the successor state from the current state. Once I have this cost, I check if this is the minimum cost path to the successor state seen so far by the algorithm. If it is, I then update my “path_to_state” dictionary to reflect this new list of actions that lead to a lower cost of finding the successor state and also update the corresponding “cost_to_state” dictionary. Lastly, the algorithm updates the “priority_queue” with a new key for the successor state.

Question 4

The A* star implementation is almost entirely the same as the Uniform Cost Search implementation so I will only be discussing the major differences between the two implementations. Both algorithms use the exact SAME data structures. The main thing to note here is that states may be explored multiple times to find the optimal path to the goal state. The reason is because the heuristic for the algorithm may not be perfect and cause the algorithm to reach a state sub optimally. This is why the “explored” set takes a tuple that consists of a state and the cost to that state. Unlike UCS, we check if the successor state and this cost to the successor state from the start state has been explored before. If it has not, the we check if this is the minimum cost path the algorithm has seen so far to the successor state. If it is, similar to UCS we update the “path_to_state” and “cost_to_state” dictionaries to reflect this better path of actions. However, unlike UCS for our priority queue the key will be updated not to this new minimum cost to the successor state but to this cost plus the heuristic from the successor state to the goal state.

Question 5

I will now be discussing my implementation for the getStartState, isGoalState, and getSuccessors functions for the CornersProblem class. My state representation is a tuple consisting of the x location, y location, and a set of corners that have been explored so far by the path of actions that led to this state. For my getStartState function I can extract the x and y coordinates from the starting position attribute of the CornersProblem class. I then check if the starting state is one of the 4 corners, if it is, I add this location to the set of corners explored and return the tuple. If it not, then I return the tuple with the set of corners being empty. For my implementation, I use a frozen set because this type of set is immutable and many of the search algorithms I implemented use a set data structure requiring its elements to be immutable. For the isGoalState function, I check to see if there are 4 corners in the frozen set of explored corners. If there are, then that means the path that led up to this state reached all 4 corners and the goal has been reached returning True. If not, the function will return False. Lastly, for the getSuccessors function I extract the x location, y location, and the frozen set of explored corners from my current state. I then check if the successor state is a corner state. If it is, then I add the corner to my set of explored corners and append to the list of successors a tuple consisting of a successor’s x location, successor’s y location, and the modified frozen set. If the successor is not a corner, I just append to the list of successors a tuple consisting of a successor’s x location, successor’s y location, and the current state’s frozen set.

Question 6

For my Corners Heuristic, I implemented a simple but effective heuristic. The heuristic function extracts the x location, y location, and the frozen set of corners explored from the state tuple. If the frozen set has a length of 4, this means the path that led up to this state has already accomplished the goal of visiting the 4 different corners in the maze. The algorithm will return 0. The algorithm then creates a list of unexplored corners and returns the Manhattan distance from the current state to the farthest unexplored corner.

Question 7

The heuristic for question 7 will aid in finding the a path where Pacman eats the food in as few steps as possible. My algorithm does a BFS from the current state to every other remaining food location. It then returns the cost of the food location that took the longest to reach using BFS, as the heuristic. This approach balances optimism and pessimism well for a heuristic. For example, I first decided to return the largest Manhattan distance to a remaining food from the current state however this approach was too optimistic causing the algorithm to explore many states. However, this approach is optimistic enough to where its admissible for every state.

Question 8

For question 8, we need to implement the findPathToClosestDot function in which the function returns the cost of reaching the closest food. I first implemented the isGoalState function in the AnyFoodSearchProblem class which was trivial. I return true if the state that is passed contains food and false if it does not. For implementing the findPathToClosestDot function, this was very trivial as well. I just ran a BFS search on the AnyFoodSearchProblem instance. However, I would have ran UCS if the costs to successor states did not always have a cost of 1.