Assignment 1

Christian Pervan

February 9, 2016

1 DFS

In A1, I used a pretty standard method of implementing this search algorithm. In my algorithm, I have a stack that takes in a 3 element tuple - element one being the state we are currently on, the 2nd element being the directions to get to that state from the start, and the 3rd element being the path of states to get to the current state from the beginning. While the stack is not empty, we pop the most recently pushed tuple off the stack, and we look for the successors of the popped states. For each popped state, we add the directions to that state to the directions list, the state itself to the path list, and we check if the successor is the goal state. If so, we return the list of directions, and if not, we push the successor and the new updated directions and path lists in a tuple onto the stack. We repeat this for each successor. After this, we pop the most recently input element off the stack and repeat the whole process.

This is clearly a correct DFS implementation since the stack represents the depth of the DFS and because it is a stack, it is a FIFO data type, so we pop the most recently visited state in depth terms, so it works as a DFS because we go depth first and traverse backwards using the stack. We also intercept other branches on the backwards traversal by going through the successors when possible.

2 BFS

Again, I used a pretty standard method of implementing this search algorithm. In my algorithm, I have a queue that takes in a 2 element tuple - element one being the state we are currently on and the 2nd element being the directions to get to that state from the start. While the queue is not empty, we dequeue the most recently pushed tuple off the queue, and we look for the successors of the dequeued states. For each dequeued state, we add the directions to that state to the directions list and we check if the successor is the goal state. If so, we return the list of directions, and if not, we enqueue the successor and the new updated directions and path lists in a tuple onto the queue. We repeat this for

each successor. After this, we dequeue the most recently input element off the stack and repeat the whole process.

This is clearly a correct BFS implementation since the queue represents the depth and because it is a queue, it is a FILO data type, so we pop the oldest visited state in depth terms, so it works as a BFS because we go across the forks and the breadth of the graph first and traverse in a normal, straightforward fashion through the graph.

3 UCS

Once more, we use a similar sort of structure when it comes to this, but some more changes here. As before, we have a 2 element tuple - element one being the state we are currently on and the 2nd element being the directions to get to that state from the start. However, this time, we use a priority queue where the priority is the cost of the directions. While the priority queue is not empty, we pop the least costly tuple off the priority queue. Next ,we check if the popped state is the goal state. If so, we return its directions, if not, we continue. Next, we check for popped state's successors and traverse through them. For each successor, we find the cost of its directions and push a tuple of the successor state and its directions to the priority queue and have the priority be its just calculated cost. Afterwards, we pop the least costly tuple off the priority queue and repeat until empty.

This is a correct UCS implementation since it searches through the tree and returns the path with the lowest cost. It does this by prioritizing the lowest costs in the priority queue. Otherwise, it asks just like a normal queue, which means it reverts to a BFS, which is fine since that is the fastest non-priority based algorithm.

4 A*

This algorithm is very similar in nature to UCS, and thus the structure is basically identical. However, there is one significant difference - measurement of priority. Now, instead of just having one cost determine it, it is the sum of 2 costs. The local cost, which is the same cost as previously in the UCS algorithm, and the heuristic cost, which is determined by running an appropriate heuristic model for the problem at hand (more on this later). The combination of these two is what determines priority. Otherwise, the structure of the implementation is virtually identical to UCS.

5 Corners Problem

Here, we shift gears and have to implement a class to handle this problem in multiple facets: finding the start state, goal state, and successors. The start state is trivial - just return the start state and an empty list representing how many of the 4 corners have been touched. Next, for the goal state, we add the corner currently being touched if Pacman is currently positioned on a corner to the corner list and check if the length of that list is 4 (i.e. all 4 corners are accounted for). If so, we have reached the goal. If not, return false. The successor is slightly more complicated. This involves measuring the Manhattan distance from the current state to its successor and copying our list of touched corners to the successor and adding to the list if the successor is a corner, and then updating a list of successors to the 3 element tuple of the successor state, it's direction (we check in all directions and make sure it is legal, in that we don't hit the wall), and it's cost, which is just a uniform incremental 1 in this instance. Afterwards, we return the list of successors. As a result, we have a new search problem that still has the same return value and functional properties as the others, even though it operates far differently. Now, we have successfully solved the corners problem.

6 Corners Heuristic

This one was a much tougher one. This involves using distances and constantly trying to find the smallest distance possible. Here, we use the Manhattan distance. But first, we create a list of untouched corners, which is obviously the complement to the touched corners list. We use this list to main tin a cycle while it still exists. Through each cycle, we traverse it and find the Manhattan distances to all the untouched corners from where Pacman is currently stationed. We then add the minimum distance to the total, set the current position to the corner at the end of that shorter distance, and remove that corner from untouched list, and cycle again until the untouched corner list empties. After, we return the distance as the cost, since each step is of incremental cost. This is clearly an admissible heuristic since it produces a reasonable output speed and is also consistent since it does not sway in output, although this could've been inferred very likely fro the sole fact that the heuristic is admissible.

7 Food Heuristic

Since this one is more open ended, this one tends to take a fair amount longer. However, this is ironically much simpler to implement. We have a grid, and the Grid class is kind enough to have height and width stored, so we traverse through the grid as if it was a 2D array (which it is), and for all instances of True (meaning there is food at that location), we measure Manhattan distance

from the current position of the algorithm to the food. However, this time, we take the maximum distance as the final answer instead of the minimum. This is because food is a lot more frequently instanced and bunched up, meaning there is going to be residue you will have to come all the way back around for when you are nearing completion.

8 Suboptimal Search

This was extremely easy to implement. We implement a function that allows us to find the out if we are on top of food currently. In the main function we must implement, we just run BFS on the problem, since it is the quickest non-priority search (meaning it is suboptimal). However, because we have a specific subclass of problem, when we go for the goal state in the process of the BFS, we are easily able to determine it. This is the least inefficient suboptimal search we can do under these circumstances, since we are able to run it at the recommended cost.