Ish Davis

AI Project 1

**Report**

*Pancake Problem*

The pancake problem is the hardest computational problem that we had and caused quite a few memory errors. I first attempted to solve both problems using BFS, but on the first test file I received and out of memory error and the same happened with the second problem as well. The same problem occurred with Unicost because without any path costs it reverts to BFS, which just didn’t have enough memory to come to any solutions. I also tried running IDDFS and DFS, they didn’t run out of memory because of the nature of the algorithms but they ended up in an infinite loop. That’s especially not surprising for the second algorithm because it would have to be a very lucky path to have all of the pancakes sorted and not burned without any heuristics. The IDA\* algorithm took too long with this problem because of the constant recreation of nodes at the top of the tree, so it took more than 30 minutes to complete. There were two algorithms that did work on the two test problems that we received, greedy and A\* search. The first heuristic that I implemented rated children based on how sorted they were at the end of the list, so for example [1, -2, 3, 4] would be picked before [1, 3, 2, 4] because the first set had the last two numbers in order while this one only had 4 where it should have been. This worked quite well on the first pancake problem, however it didn’t work on the second problem and took over 30 minutes to complete. After running this heuristic several times I knew that I needed an addition to what I already had in order to make it more performant. So I essentially made a heuristic that started at the end of the list and moved toward the beginning, and the more items it had sorted at the end, the better the heuristic score. The other part that I added starts at the beginning of the list and looks for the largest unsorted number to this point and prioritizes the child that has that number at index 0 of the list. I did this because if that number is at the beginning of the list, the pancakes can simply be flipped from the 0th index to the highest unsorted index, and it would essentially sort the list from the end to the beginning. It turned out that a number being positive or negative didn’t necessarily matter for my heuristic, because if it was negative at index 0 I would rank it low so that it could be flipped to positive on the next iteration and then moved to the sorted section. As for the searches, A\* and greedy search worked identically because there were no path costs associated with each flip, so they each created 265 nodes on test\_pancakes1, which I have illustrated below in the chart. On test\_pancakes2 they took 8,300 node creations and the frontier size was about 8,000. The algorithms solve the puzzle in around 1 second on test\_pancakes1 and about 5 seconds on test\_pancakes2.

*City Path Problem*

The path-planning problem was interesting in that we had actual path costs to utilize with our search algorithms. I first attempted to use BFS to find a solution and the output was an enormous 14,697 created nodes, because the algorithm only checked the explored list and not the frontier so there were a lot of duplicates. The DFS and IDDFS algorithms both ran into infinite loops because of cycles, and the parent check that the professor recommended wasn’t enough to stop those cycles from occurring. It would have taken a node to check all of its parents to ensure that no cycles occurred, which would have been a costly operation. The searches that performed the best were Unicost, A\*, and Greedy search. Greedy search worked best for this algorithm because the heuristic was very good at choosing the next state to visit. The heuristic that I implemented was a distance function that found the straight line path from the current city to the goal city. That is as optimistic as possible because the straight line is the shortest distance to any place, so the heuristic would choose the city that had the least straight-line distance to the goal. With that heuristic the greedy algorithm generated less than ¼ of the nodes than Unicost and A\* search. It also had much fewer nodes on explored, but a few more nodes on the frontier list. Unicost and Astar performed identically in all 3 statistics, however in the output the path that Unicost takes is the shortest at just 17 units. The best search in this case is dependent on what the user wants, if you want the shortest possible path then Unicost is the answer but if you want the program to run as quickly as possible then the Greedy algorithm is the search to use.

*Jugs*

Utilizing the search algorithms were slightly harder to do on the jug problem because we had to account for two sub-problems, the two jug and three jug problems. After testing several searches, which I have illustrated below, the greedy search required roughly half of the node creations than that of the other searches. It took only about 26 nodes to find the answer, while BFS, Unicost, and A\* all took at least 51 node creations. It was also twice as efficient as those other algorithms when taking into account the nodes on the explored list, which would be a huge savings in memory for more difficult problems. The DFS algorithm didn’t work on this particular problem because of infinite loops, even after implementing the parent check that the professor requested it would still get caught in a cycle. I also ran the IDDFS and IDA\* algorithms but they each took over 100,000 node creations, mostly due to the multiple creations of nodes at the top of the graph. For my heuristic function I returned the sum of the distance for each jug from its goal value. For example if I had 3 gallons in the 11 gallon jug and 2 in the 4 gallon jug, I would return |3-0| + |2-1| = 4. This is admissible because it was optimistic based on the trial runs that I executed, and also satisfied the consistency principle. Overall I believe the greedy search was the best search for this problem because it provided the quickest performance and utilized ½ of the memory that the other searches did.