Peter Schaldenbrand

Artificial Intelligence, Dr. Hwa

Project 1

September 26, 2016

**Jugs**

I created two heuristics for the jugs problem. The first one (“1”) returned how many jugs in the current state were not equal to the jugs in the goal state. This heuristic is not admissible, meaning it does not always give an optimistic (equal or lower than the actual) cost. For example, if the current state is (0,2), the capacities are (3,4) and the goal is (2,0), then the heuristic will return 2. But in actuality the state to goal could be completed by pouring one jug into the other which is a cost of 1. This heuristic is consistent, meaning that every node below a node has an equal or greater cost.

The second heuristic (“2”) I wrote returns the difference in jug sizes between the jugs in the current state versus the final state. This is not admissible for the same reasons and example as heuristic 1. It is also inconsistent. An example of its inconsistency is a jug problem where the capacity is (4,5), the current state is (0,0), and the goal is (3,4). The current cost is 7 and if you fill the second jug, the cost is 5.

The results of running search algorithms on a jug problem (test\_jugs.config) with capacities of (4,11), initial state of (0,0), and goal state of (1,0):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nodes Visited | Max Nodes in Frontier | Max Nodes in Explored |
| bfs | 1979 | 4619 | 1977 |
| dfs | 2641 | 201 |  |
| iddfs | 2832 | 9 |  |
| unicost | 1979 | 4619 | 1977 |
| greedy 1 | Out of memory before completion | | |
| greedy 2 | Out of memory before completion | | |
| astar 1 | 937 | 2026 | 935 |
| astar 2 | Doesn't Finish in 45min | | |
| idastar 1 | 4268 | 4619 |  |
| idastar 2 | 4268 | 4619 |  |

The results of the greedy algorithms make sense since the heuristics were not both admissible and consistent. In this case, the heuristic threw the algorithm off so it would follow the wrong path and take up huge amounts of memory.

The only reason that DFS got an answer in this problem is because I set a limit at a depth of 200. If it weren’t for this limit, then the execution would have never ended.

The results of running the search algorithms on a jug problem (jugs.config) with capacities of (3,4,5), initial state of (0,0,0), and a goal state of (0,2,2):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nodes Visited | Max Nodes in Frontier | Max Nodes in Explored |
| bfs | Doesn't Finish in 45 min | | |
| dfs | 21826 | 201 |  |
| iddfs | 9681 | 7 |  |
| unicost | Doesn't Finish in 45 min | | |
| greedy 1 | Doesn't Finish in 45 min | | |
| greedy 2 | 1291 | 11031 | 1289 |
| astar 1 | 227 | 1419 | 225 |
| astar 2 | 524 | 3816 | 522 |
| idastar 1 | 10883 | 48734 |  |
| idastar 2 | 10883 | 48734 |  |

Again, with this one the DFS would not have found an answer if it weren’t for that 200 height cap. It’s difficult to explain, with the greedy algorithm, why heuristic 2 worked for this problem but not for the previous one. The heuristic must have happened to persuade the algorithm to take the correct path at some point just by chance.

In these jug problems, as long as A\* is able to find an answer, then it is the best algorithm. But because the heuristics aren’t optimistic, then A\* is not guaranteed to be optimal and complete. Therefore, I do not think that this is the best algorithm for the jug problem.

I would argue that ID-DFS is the best because of memory management, with decent time complexity. With this jug problem, the optimal path is not going to be extremely long, but each node has so many children, so this can take up a lot of memory. ID-DFS only had a few nodes on the frontier at a time and no explored list. While it isn’t the fastest algorithm, it is complete, optimal, and space efficient which is why I would say that this is the best algorithm for the jug problem.

**Cities**

I created only one heuristic for the cities path finding problem. This heuristic returned the (Euclidian) distance between the two cities. Meaning that for city A(2,4) and city B(0,5) the distance would be calculated by sqrt( (2-0)^2 + (4-5)^2 ). This heuristic is admissible because this will always return the shortest distance between the two points which is optimistic because this is equal or less than the actual path cost. This heuristic is also consistent since every city’s children will have an equal or greater path cost.

The results of running the search algorithms on test\_cities.config are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nodes Visited | Max Nodes in Frontier | Max Nodes in Explored |
| bfs | 16 | 193 | 14 |
| dfs | 213 | 201 |  |
| iddfs | 16 | 3 |  |
| unicost | 3710 | 66781 | 3708 |
| greedy | 3 | 24 | 3 |
| astar | 497 | 11180 | 495 |
| idastar | 19657 | 56826 |  |

As with the jug problem, DFS would not have completed if it weren’t for the 200 depth limit. Obviously with the nature of path planning problems, BFS, DFS, and ID\_DFS are useless unless the path cost is 1 for all transitions from nodes. In this case, the cost is not always 1, so these results aren’t helpful in finding the optimal path.

If you don’t care about optimality then greedy is obviously the best algorithm here since the heuristic function is really great at finding the path with the fewest cities.

The overall best algorithm would be A\*. unicost takes a very long time and uses a ton of memory as does IDA\*. A\* is optimal and complete because the heuristic is admissible and consistent. So not only does A\* find the optimal answer, but it does it pretty quickly. The only issue with A\* is that the frontier list can get pretty large which is not good for space efficiency.

**Pancake**

I created two heuristics for the pancake problem. The first (“1”) one compared the current state versus the goal state and returned how many pancakes were out of place or flipped the wrong way. This heuristic is neither optimistic nor consistent. As an example say we have a current state of (-5,-4,-3,-2,-1), then the heuristic would return 5 but the state can be turned into the goal state in just one flip. The second heuristic (“2”) returns the sum of the displacement of each pancake, not counting which way it’s flipped. For example, (1,-2,4,5,-3) would return 4. This heuristic is also not optimistic and consistent. So for the pancake problem A\* is neither complete nor optimal.

The results of running the search algorithms on test\_pancakes1.config where the start state was (-1, -11, -3, -6, -9, -4, -7, -10, -5, -8, -2) were completely inconclusive. No algorithm could find a path to the goal state in under 45 minutes.

I made another test file called test\_pancakes3.config where the start state was (3, -1, 2, 5, 4), and the results were as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nodes Visited | Max Nodes in Frontier | Max Nodes in Explored |
| bfs | 6363 | 25445 | 6361 |
| dfs | 15540 | 201 |  |
| iddfs | 7954 | 8 |  |
| unicost | 6363 | 25445 | 6361 |
| greedy 1 | 33 | 125 | 31 |
| greedy 2 | Out of memory before completion | | |
| astar 1 | 77 | 301 | 75 |
| astar 2 | 1028 | 4105 | 1026 |
| idastar 1 | 11243 | 25445 |  |
| idastar 2 | 11243 | 25445 |  |

DFS only found a solution because of a 200 maximum depth. If there had not been this max depth, then the algorithm would not have found a solution.

The only algorithm to not find a solution was the second heuristic on greedy which makes sense because the second heuristic was really terrible. The most surprising results were ID-DFS and BFS. ID-DFS and BFS found the solution in very good time, but this is because the solution was only 8 state changes. If the optimal solution were any longer, then these would not have been very good algorithms since the size increases quickly the longer the solution.

Because the heuristics were not admissible and consistent, A\* is not complete or optimal. So despite A\*’s very good performance on this problem, I would not recommend it because it won’t necessarily find the optimal solution and might take an indefinite amount of time to find any solution. So the best algorithm to choose for me is ID-DFS. ID-DFS was nearly as time efficient as BFS, but used nearly no memory because of the tiny size of the frontier list and no explored list. ID-DFS is also complete and optimal since all the path costs are 1, making this a very good algorithm for this problem, if A\* is not optimal.