James Hahn
CS1571 – Artificial Intelligence
Dr. Diane Litman
M/W 11am-12:15pm
Homework #1

Heuristic functions

Data Aggregation (refer to **note 3**) –

- *unicost* - choose the edge with the smallest time latency next
- *greedy* – continue on the path with the smallest most-recently-added edge
- *A** - continue on the path with least sum of [path cost by sum of edge latencies] and [most-recently-added edge weight]

Monitoring –

- *unicost* – go to the state with the "highest" cost. In other words, go to the state that currently monitors its targets for the longest amount of time
- *greedy* – go to the state with remaining sensors and targets located closest to their average location (refer to **note 2**)
- *A** - choose the state with the greatest sum of [cost, or longest time we can watch the targets] and [sum of distances of sensors and targets from their average location] (refer to **note 2**)

Pancakes –

- *unicost* – since we have no information about the state and we just move one step along the path at a time and nothing is weighted, uniform-cost search on this problem is exactly the same as BFS
- *greedy* – continue with the state that matches in similarity to the goal state as closely as possible (refer to **note 1**)
- *A** - go to the state with the lowest sum of [differences between it and the goal state] and [number of steps, "flips", taken so far] (refer to **note 1**)

Note 1 – In the Pancakes problem, we use the "similarity to the goal state" as the heuristic. This solution is unique to this problem and essentially what it does is scan through the current state of the pancakes, including its negatives, and does two things. The difference (from it and the goal state) starts as an integer at value 0. If the current pancake is negative, we add 1 to the difference. If the current pancake is not in the current index in the list, we add 1 to the difference. Because of this, any given pancake can add up to 2 points to the total difference variable. This means if all the pancakes were burnt-side up (negative) and none were in their correct order, the "difference" value would be 2N where N is the number of pancakes; this indicates that it is nowhere close to the goal state. Conversely, if the "difference" value was 0, we have reached the goal state.

Note 2 – For the monitoring problem, we have to choose creative approaches to the heuristic function. This is because we don't have any direct goal state that we're striving towards, much like the Pancakes problem. Rather, we just want any configuration that allows us to monitor targets for the longest amount of time. For this problem's heuristic function, I first gathered the list of sensors and the list of targets. Then, using the coordinates of the sensors, I calculated the average x and y coordinates of the sensors. The same was performed for the targets list. Now, when the heuristic function refers to the "sum of distances of sensors and targets from their average location", the problem scans over each state on the frontier. For each state, we look at all of the unassigned sensors and sum up their distance from the average sensor location. We do the same for the unassigned targets; sum up their distance from the average target location. Finally, we add the two sums together, refer to this as **sum(s, t)**

For the *greedy* approach, we want the lowest sum(s, t) indicating that the available sensors and targets are forming the closest cluster possible, hopefully ensuring that our next pick of sensor-target pairs is minimal. If all of the sensors/targets were far apart from eachother, this would surely limit the amount of time we can watch all of the targets.

The *A** heuristic isn't much different. We simply add sum(s, t) with the total cost of the state so far (maximum time we can watch all the assigned targets). Then, we take this final sum and choose the state that maximizes it. This works because most of the time the "cost" at a given state should play a significant factor in maximizing potential for a state. While we want to minimize sum(s, t), adding it to the current path cost and taking the maximum of that sum will hopefully have the path cost outweigh sum(s, t).

Note 3 – The approaches to the data aggregation problem seem to be similar to well-known algorithms such as Dijkstra's. My chosen heuristic functions are pretty straightforward. For *greedy*, just continue on the path that chose the smallest weighted edge last. So essentially we're trying to minimize cost to the goal state by always choosing the shortest option available to get there, or furthest distance away. In a perfect world for *greedy*, we would know every state's distance from the goal state. However, for the data aggregation problem, we just need the solution path to be length N for any solution where N is the number of nodes because we need to visit every node once. The problem with this is that you can't measure how far away from the goal state we are unless you count N-i being the distance where i is the length of the path so far. However, if that were the case, this problem would essentially just degrade to BFS because every path would be of weight 1.

As for *A**, sum up the total path cost and weight of the most recently added path edge. Then, find the state that minimizes this and continue down that path. This is textbook *A**, adding the path cost so far with what our future cost to the goal state looks like. Unfortunately, it's not the optimal choice as we just mentioned in the above paragraph, but it's close enough.

Did outcomes make sense?

Yes. As far as each algorithm goes, *bfs* creates the most nodes by far, as expected, because it is an exhaustive search essentially. Depending on the problem, *iddfs* creates a much smaller frontier than *bfs* and the others because we're doing a DFS down the tree; if the branching factor at each level is small, even if the solution path is long, the frontier size will be very minimal. Another point is *unicost* tends to produce optimal solutions more often than both *bfs* and *iddfs*, which is great news. Also, depending on the problem we're searching, *greedy* may return the optimal solution or may not; it definitely performs poorly with test_pancakes3.config, returning a result just as bad as *iddfs*. The *greedy* algorithm performs quicker than all the other algorithms when utilizing test_pancakes3.config; this makes sense because we're just trying to make it through the search space to the goal state as quickly as possible, attempting to travel along the "quickest" route there. Unfortunately, it performs poorly in terms of achieving the optimal solution.

In the Pancakes problem, *unicost* returns the same results as *bfs*. This makes complete sense because a "flip" has no weight to it; every flip performed essentially has weight 1. If you refer to the textbook and slides, *unicost* performs the same as *bfs* if this very fact is true. So because of this, this result makes perfect sense.

Also for the Pancakes problem, when ran on test_pancakes3.config with *A**, it does not produce the optimal solution. This is fine because our textbook states when forming a heuristic that should be optimal: "The next step is to prove that whenever A* selects a node n for expansion, the optimal path to that node has been found." This piece of information directly indicates that *A** cannot be optimal for the Pancakes problem because given any configuration of pancakes in a stack, although we can measure similarity to the goal state, we will never truly know exactly how many flips we are from the goal state. Think back to our classroom examples with the cities and distances to the goal state; we had that information, which helped ensure that our informed search, *A**, would always move closer to the goal state. That was an easy example because we knew our precise distance from the goal. However, in the Pancakes problem, we can measure similarity to the goal state, but we cannot tell the exact number of flips, or distance in this case, from the goal state. Because of this, it makes sense that *A** does not produce optimal results for the Pancakes problem for some configuration files.

One thing to note is that the *A** algorithm on the Pancakes problem takes quite a bit longer than I would've liked. I think this can be explained by the fact that even if we believe we're moving closer to the goal state and reducing the differences between our current state pancake stack and the goal state pancake stack, that flip may in turn only allow us to make worse flips from there, which actually move us further away from the goal state. Take for example the goal state (1, 2, 3, 4, 5) and the current states **t** (1, 2, 3, 4, -5) and **u** (-2, -1, 3, 4, 5). With the similarity metric I provided earlier, **t** has a similarity of 1, indicating it's close the goal state. Meanwhile, **u** has a similarity of 4. However, **u** requires one flip (between the -1 and 3) to reach the goal state and **t** requires many more than that because we need to flip the -5, or bottom pancake. As we can see, in this scenario, *A** would choose to go to **t**, but we'd actually be moving further away from the goal state.

All of the algorithms are complete, as expected. The only algorithm I wasn't expecting to necessarily be complete was *greedy*, simply because of the Pancakes problem. You can technically just go in a circular pattern with what seems to be infinite flips. With *greedy* just clinging onto what seems to be the path closest to the goal state, we can actually end up going further away from the goal state in the long run as mentioned above. It's complete on the test_pancakes3.config, but it takes too long to measure completeness even on input with fifteen pancakes.

As far as space goes, everything ran as expected. To go more in-depth, *bfs* literally explored a large chunk of the search space, going across instead of down (like *dfs*/*iddfs*). This caused *bfs* to create a much larger frontier and sheer number of nodes than the other algorithms. Next, *iddfs* had the smallest frontier. As we move down the search space, we're essentially disregarding all other states from a given action until we know for sure our current path doesn't contain a solution. This works great for the Pancakes problem because we just want to keep on flipping pancakes rather than exploring every single option. Finally, *A**, *greedy*, and *unicost* all perform differently depending on whether the cost for each step in the problem is weighted or not, whether we have detailed information regarding distance to the goal state, and even if we have detailed information about the current distance/cost that we've travelled so far to a given state as well. Usually, they weren't the worst in terms of space (except for *unicost* in Pancakes). However, there were points when they were the best, such as *greedy* in Pancakes. It was all relative to the information we knew prior in the problem.

Best Search for each problem?

Data Aggregation – For this problem, we should use *A** because it's a relatively good example problem for this type of algorithm. This means that we have a concrete path cost (sum of edge latencies) and we have some goal state, which is how many nodes we haven't connected. Although we don't use the number of non-connected nodes as a metric, we use the weight of the most recently added edge. Then, we sum the two values together and choose the minimum of all the states. This will quickly search down the search space and hopefully discover the optimal solution most often because we are travelling along the path with the lowest cost and most immediate lowest weight edge to the goal.

Monitoring – For this problem, there is no good metric. I strongly believe *unicost* is most optimal in this scenario because it continues with the state on the frontier that monitors its targets for the most amount of time. By following this rule, we're hopefully only increasing the time we monitor our targets. Compared to other algorithms, such as *greedy* and *A**, that use locations of the nodes to find heuristics, we don't really need that information; it clogs up the search space and doesn't add much to the value of the problem because in the end we just want the sensor-target pairs to be as close to eachother as possible and not necessarily closest to their average, central location. Although utilizing average sensor/target location may be helpful, it doesn't really add much and only complicates the algorithm and state.

Pancakes – This one is tough. If we want optimality, our only guarantee is to choose *bfs* or *unicost* because they act identical in this problem as mentioned earlier. We have no idea how far we truly are from the goal state and every flip is of the same weight so we can't really gain an advantage by using flips as a cost metric. If we want speed, which is completely viable due to the pure branching factor of this problem, just use *greedy* because we want to traverse down the search space as fast as possible and just make as many flips as we need to in order to reach the goal state, rather than across with *bfs* (due to the branching factor). In this problem, *greedy* will just choose the state that "seems" to be most similar to its goal state. Due to the lack of information given in the problem, traversing down the search space in a somewhat heuristic-ish fashion will be our best bet.

Anything surprising?

Nothing was extraordinarily surprising. One thing that was a bummer in this project is the branching factor of any given problem for the search space. If the Pancakes problem has input of twenty pancakes, there are twenty different positions that you can flip the pancakes for any given configuration. This means the branching factor is already size twenty and this doesn't become smaller as we scan the search space. So, in reality, the worst-case runtime for the Pancakes problem with twenty pancakes is 20^n where n is the level of any given path.

One true difficulty was choosing heuristic functions. This was definitely surprising to me because we always cover the easy examples in class. The cost of a given path is direct, we know the location of the goal state, and we know how far we are from the goal state. As mentioned several times in this report already, we could never truly and accurately calculate our distance from the goal state. And if we could calculate the distance, it was with limited information and somewhat of a guess. I wish we covered much more difficult examples in class that actually applied to the projects so we had a vague idea of what we should be looking for in these toy problems.

Overall, the runtime was the worst thing to deal with in this project and I hope we're not judged too harshly on how our project performs in terms of runtime because right now mine can only realistically run on input of smaller sizes because the branching factor of the search space becomes ridiculous and searching through every node is unfeasible otherwise.

Final note: I set the timeout length to 10 minutes. If the problem hasn't been solved within 10 minutes, we report that no solution is found.