Veronica Child & Liv Phillips
Artificial Intelligence
Prof. Howald
2/2/18

Assignment #2 Write-Up

Introduction

We used Python to build a simulation of a twelve room office floor, as well as an optimized HeatMiser to monitor the temperature and humidity of the environment. Each time a new office floor is created, it randomly assigns its twelve rooms a temperature and humidity (both within the given acceptable ranges), and randomly initializes the HeatMiser in one of the rooms. From there, it is the job of the HeatMiser to adjust the temperature/humidity of rooms on the floor until it reaches a floor average of 72 degrees F and 47% humidity. As long as the floor is not at that point, the HeatMiser determines the room that is the furthest from the standard deviation for temperature and the room that is the furthest from the standard deviation for humidity. Whichever room is further from the standard deviation becomes the HeatMiser's destination. From there, there are two search methods implemented to decide its path:

1) Basic Breadth-First Search
    a) Keep a queue of possible paths, starting with a path that is just the starting room.
    b) Keep a list of already visited rooms (so that the agent doesn't infinite loop).
    c) While the queue is not empty, dequeue a path from it, and look at the latest room in that path.
    d) -> If that room is the target room, return that path.
    e) Otherwise, for each neighbor room of the latest room, duplicate the path and add the neighbor room at the end. Then, append the path to the queue.
    f) Put the latest room in the list of explored rooms.
    g) Go back to step (c).
2) A* Search
    a) Keep a priority queue of possible rooms, beginning with the starting room. The number associated with each room is the lowest accumulated cost from the starting room to that room (seen thus far).
    b) Keep a list of already visited rooms.
    c) Keep a list that, for each room, *r*, holds the room, *p*, that comes before *r* on the lowest cost path from the start room to room *p* (that has been seen so far).
    d) Keep a list of the lowest accumulated edge cost (that has been seen so far) from the start room to each other room.
    e) While the priority queue is not empty, dequeue the lowest accumulated room.
    f) -> If that room is the target room, recreate the path (through use of the list in (c)), and return it.

g) Otherwise, add the room to the list in (b), and then for each neighbor of that room, check if the neighbor is already in the list in (b). If so, it's already been explored fully so move on.

h) If not, get a tentative total cost by adding the accumulation of edge costs from start to the current room from (d), plus the edge cost from the current room to the neighbor. If the neighbor has a lower value currently associated with it in (d), move on.

i) Otherwise, if the neighbor doesn't have a value in (d) yet, or if this value is less than the neighbor's current value in (d), put this value in (d) for the neighbor, and put the current room in (c). Then, add the neighbor's accumulated edge value from (d) to its heuristic value, and put that total cost as its associated value in the priority queue.

j) Go back to step (e).

The following table shows how we judge our temperature and humidity control agent type by the PEAS matrix (all bullets apply to both search (1) and search (2) unless noted otherwise):

Performance Measure	**Environment**	**Actions / Actuators**	**Sensors**
- Minimal office visits - Correct humidity percentage average - Correct temperature average - Correct room standard deviation - Least energy usage (2)	- 1 floor - 12 rooms - Settings: temperature, humidity - Energy costs between offices (2)	- Move to the next room (path determined by search); wheels + engine - Increase / decrease the temperature / humidity	- Thermometer - Humidity sensor - Knowledge of room number - Meter of current energy usage and potential energy usage

Analysis

Completeness

Since we were given the graph in advance, we know that it is connected and that there will always be a path between any two rooms the HeatMiser is trying to get to. Thus, to be complete, the algorithms would need to always return a path, and since neither of the algorithms we implemented will ever fail to find a path or endless loop, they are both complete.

Time Complexity

The time complexity of our baseline BFS search is $O(b^d)$, where b is the maximum branching factor of the search space and d is the depth of the solution. The number of nodes expanded is exponential in the depth of the solution, which would be the shortest path. Similarly, the worst case time complex of A* is also $O(b^d)$. The time complexity of A* depends greatly on how admissible the heuristic is;

a good heuristic lets A* ignore any node that wouldn't meet its criteria, which an uninformed search like BFS would still expand. In practice, a good heuristic would greatly diminish the time A* takes.

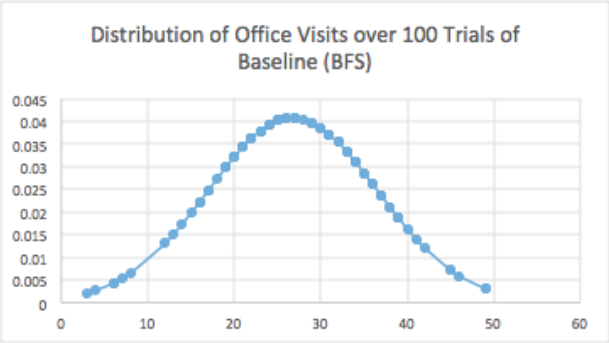
Space Complexity

The space complexity of our baseline BFS search is $O(b^d)$, where b is the maximum branching factor of the search space and d is the depth of the solution. Similarly, the space complexity of A* is $O(b^d)$.

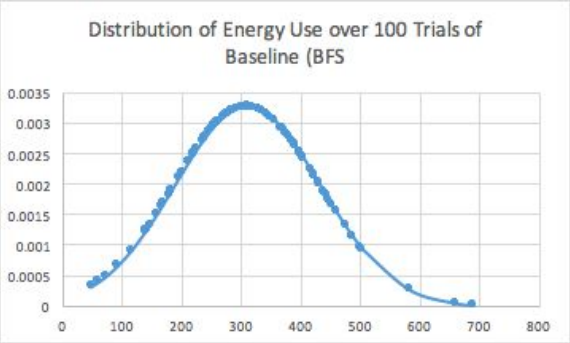
Optimality

Depending on the utility that we are trying to optimize, either one of these searches could be considered optimal. If we are trying to optimize the number of rooms that the HeatMiser visits, then BFS would be optimal, while A* would not. In one 100-trial run, BFS HeatMiser had an average room visit of approximately 27 visits per trial while A* HeatMiser had an average of 29 (see figures 1 & 3, respectively). On the other hand, if we were attempting to optimize energy use, then the A* HeatMiser would prevail over the BFS HeatMiser. For energy use, in one 100-trial run we found that the BFS HeatMiser had an average energy use of 309 per trial, while the A* HeatMiser had an average energy use of 278 per trial (see figures 2 & 4, respectively).

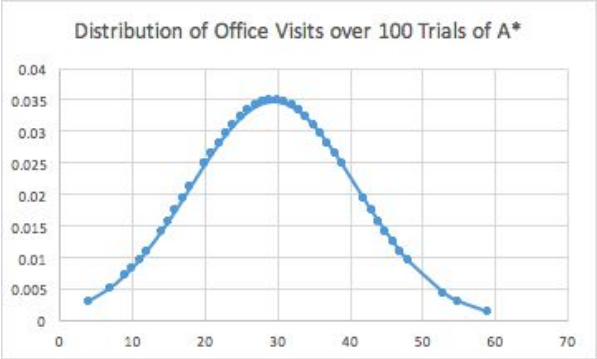
These differences demonstrate that the trade-off between their optimalities wasn't equal -- since more room visits means more edge weights to consider, the A* HeatMiser typically only visited a few more rooms than the BFS HeatMiser. Yet, due to the BFS HeatMiser's disregard for the edge weights (as well as the structure of the graph itself), it often chose paths entirely comprised of the most heavily weighted edges. Thus, when considering the optimization of both room visits *and* energy cost at the same time, the A* HeatMiser wins out. Thus, since the space/time complexity of the two algorithms are the same, A* is the overall better search algorithm.



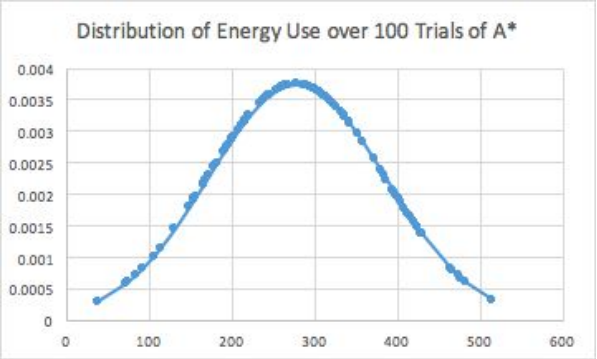
(Figure 1: Distribution of BFS Office Visits)



(Figure 2: Distribution of BFS Energy Usage)



(Figure 3: Distribution of A* Office Visits)



(Figure 4: Distribution of A* Energy Usage)

In terms of implementation, BFS was much easier as it is an uninformed search and therefore has fewer factors to keep track of and consider. Still, because much of the basic structure of the two algorithms is the same, once we had implemented BFS, we only needed to add in the specifics of A*. As for the heuristic used in our implementation, it's clear that straight line distance did minimize the overall energy use which points to it being relatively good. Our results make us inclined to keep straight line distance as the heuristic of choice for keeping energy consumption low.

Future Considerations

This version of HeatMiser is already far improved from our last iteration due to its ability to set room temperature and humidity to a specific temperature. To further improve its performance of minimal room visits, it would be ideal if our HeatMiser could go directly to the room with the highest standard deviation, bypassing the constraints of the graph. The number of room visits made by our agent would decrease and the speed of our agent would increase.

In terms of structure, there may be ways of preprocessing the room graph that would allow the HeatMiser to minimize room visits and energy use. For instance, perhaps the HeatMiser could start in the room that had the highest temperature/humidity variation (as opposed to being randomly placed in the room). Additionally, the HeatMiser often passed through rooms that it would later decide to go back to in order to change their temperature/humidity. This means that it backtracked, and could have expended less energy if it had been able to decide which rooms to change before it started, and then simply found the least cost path between those rooms. In addition to these features, we could also try implementing an optimized agent with Greedy Search instead of A* and see how the performance differs in terms of energy usage.