AI Assignment 1 Report

Thomas Heider tgh28, Youssouf Ouedraogo yo94, Paul Keough pmk95

October 14, 2019

1 Setup

Our 50 preimplemented gridworlds are included in the zip file as well as the program used to generate them. It should also be noted that we implemented our own minimum queue and used no imports for it.

2 Understanding the Methods

a. For this to be proper A* then the heuristic would need to be admissible. An admissible heuristic needs to always be optimistic and attempt the path that assumes there are no blocked cells in its path, and correcting it later on. The agent could not go north without prior information that A* does not support.

b. A* contains a closed list that will keep track of every cell that has already been searched. The closed list is added to as each cell is expanded and each of that cell's neighbors are added to the open list. If the current path is blocked, the algorithm will backtrack to the last viable node until all neighbors of the cells within the closed list are exhausted. So every node that has a neighbor that can connect to the start will be searched. If all nodes that connect to the start includes the goal, the goal will be found, otherwise a* will determine there is no path to the goal from the start. Because once a node has been expanded it won't be searched again, there can be no infinite loops. So as long as the gridworld is finite, the time to discover if there is a solution it will be finite.

   A* makes an initial search of what would be the shortest path assuming that there will be no obstacles in the way. It then proceeds to follow that path until it reaches the goal or discovers that it is no longer on the best path, whether that be in the form of an obstacle or discovering better action costs. At which point it performs the search again and follows the new path again until the goal is reached or is discovered to be unreachable. The worst case of the search would be one in which every unblocked cell would be in the presumed path. The worst case for the movement stage would be where it makes one move, and discovers it is no longer on the best path. Because A* uses a closed list, and cell will only be expanded only once, the maximum amount of times the pathfinding step will fail is the number of unblocked cells. So the overall worst case would be a path of the length of all of the unblocked cells, that has to perform a new search for each of the unblocked cells in the grid. Meaning the maximum amount of moves is the number of unblocked cells squared.

3 The Effects of Ties

File Name	Lower g Value	Higher g value
GW2.txt	0.487088 seconds	0.092810 seconds
GW3.txt	1.065974 seconds	0.862911 seconds
GW4.txt	0.032997seconds	0.026937 seconds
GW5.txt	0.552636 seconds	0.639292 seconds
GW7.txt	0.321184 seconds	0.310858 seconds
GW8.txt	0.136665 seconds	0.146676 seconds
GW9.txt	0.535566 seconds	0.503652 seconds
GW10.txt	0.630529 seconds	0.630344 seconds
GW11.txt	0.474172 seconds	0.416331 seconds
GW12.txt	0.243348 seconds	0.223407 seconds

Upon testing, we found that deferring ties to pick higher G values resulted in better run times on average. When the smaller values are favored, this means that our algorithm searches more nodes that are closer to the start rather than closer to the goal. Choosing the higher G values makes sure the algorithm stays on track with the continuously calculated, L-shaped, optimal paths towards the goal.

4 Forward vs. Backward

File Name	Forward	Backward
GW2.txt	0.092810 seconds	0.092102 seconds
GW3.txt	0.862911 seconds	1.044268 seconds
GW4.txt	0.026937 seconds	0.025020 seconds
GW5.txt	0.639292 seconds	0.642862 seconds
GW7.txt	0.310858 seconds	0.284222 seconds
GW8.txt	0.146676 seconds	0.158368 seconds
GW9.txt	0.503652 seconds	0.641641 seconds
GW10.txt	0.630344 seconds	0.671092 seconds
GW11.txt	0.416331 seconds	0.447232 seconds
GW12.txt	0.223407 seconds	0.250434 seconds

In most cases, we observed that forward A* ran slightly faster than backward A*. This can be attributed to the overall shape of the curves that each algorithm tries to make. Backward A* tends to resemble a concave path whereas forward A* resembles a convex path. This difference in shape is a visual indication of the slight differences between these two versions of the same algorithm.

5 Heuristic in the Adaptive A*

When using only the four cardinal directions, the Manhattan Distance is equivalent to the shortest path between two points. This is due to the fact that each of the four compass movements only allow you to move along the x-axis or y-axis, never both. Any path that would need to alternate moving vertically or horizontally due to obstructions still has the same vertical and horizontal distance between the two points. The Manhattan distance will be consistent when only using the four compass directions, because there is no possible path that could be shorter. If there exists a path through the obstructions that is equivalent to the shortest path possible, then the h value for any given node will be equal to the sum of moving to its neighbor and the h value of the path of the path from the neighbor to the goal. Otherwise, if the obstructions require the path to be higher than the path given no obstructions, the Manhattan distance will always be less than or equal to the path for one node to the next.

If we know that the initial h values are already consistent, increasing action costs will have no effect on the consistency of the h values. We know that the initial h value is consistent so for some initial cell s and its neighbor a:

$$h(s) \leq c(s, a) + h(a))$$

The only difference that Adaptive A* is that instead of the standard heuristic it uses:

$$h(s) = g(goal) - g(s)$$

By replacing the h values with their equivalents:

$$g(goal) - g(s) \leq c(s, a) + g(goal) - g(a)$$

Simplified you get:

$$g(a) - g(s) \leq c(s, a)$$

We already know that the initial heuristics were consistent. So the g value of s and a are their respective distances from the start. Meaning the difference is the shortest possible distance between the two cells.

6 Analysis of Heuristic in the Adaptive A*

File Name	Forward	Adaptive
GW2.txt	0.092810 seconds	0.100857 seconds
GW3.txt	0.862911 seconds	0.764882 seconds
GW4.txt	0.026937 seconds	0.025924 seconds
GW5.txt	0.639292 seconds	0.552343 seconds
GW7.txt	0.310858 seconds	0.307357 seconds
GW8.txt	0.146676 seconds	0.146566 seconds
GW9.txt	0.503652 seconds	0.483446 seconds
GW10.txt	0.630344 seconds	0.623988 seconds
GW11.txt	0.416331 seconds	0.404202 seconds
GW12.txt	0.223407 seconds	0.234832 seconds

In most cases, we observed that forward A* ran slightly faster than backward A*. This can be attributed to the overall shape of the curves that each algorithm tries to make. Backward A* tends to resemble a concave path whereas forward A* resembles a convex path. This difference in shape is a visual indication of the slight differences between these two versions of the same algorithm.

7 Memory Issues

In our implementation, we used several lists and dictionaries to keep track of association between integer values, tuples, and coordinates. We could have improved on run time by having a custom cell object that correspond to a specific cell, and contains all of of the values and fields that would be needed for that specific cell.

Based on our proposed improvements, implemented in Python, the cell object would have have six integers: one for the g value, one for the h value, one for the f value, one for the previous g value, one for the adaptive h value, and one to represent whether the cell is blocked, open, a start, or a goal. The cell would also have a tuple for the cell that leads to the shortest path. Overall this total storage for one cell would be $(28 * 6) + 40 = 208\, bits/cell$

Calculate the storage needed for a gridworld of 1001 x 1001:

$$1001 * 1001 = 1002001\, cells$$

$$1002001\, cells * 208\, bits = 26.05\, MegaBytes$$

Calculate the largest gridworld that could be implemented with 4MB:

$$\frac{4MB}{208\, bits/cell} = 153846\, cells$$

$$\sqrt{153846} \approx 392$$

The largest gridworld that could be made with 4MB would be 392 x 392.