CS440 Homework 1

Group Members: Nic Carchio, Zain Sayed, Ernest Lin

Questions

Part 0 - Setup your Environments

We generate our grid world environments by iterating through each cell in the 101 x 101 grid, assigning each one as blocked with 30% probability, and the rest as unblocked. Then we randomly choose two pairs of coordinates to be the start and goal. We use pygame to visualize our mazes.

Part 1 - Understanding the methods

a.

In Figure 8, the agent is current at E2 and will explore the following cells (along with their g, h, and f values), with the "shortest presumed-unblocked path" assumption:

E1: Unblocked, $g = 1, h = 4, f = 1 + 4 = 5$
D2: Unblocked, $g = 1, h = 4, f = 1 + 4 = 5$
E3: Unblocked, $g = 1, h = 2, f = 1 + 2 = 3$

The move corresponding to the lowest f value is E2 $\rightarrow$ E3, so the agent will travel to E3. Because the agent does not know initially which cells are blocked, it will travel to E3.

b.

The agent 1) does not re-expand cells that are in the closed list (i.e. removed from the open list) and 2) will expand all cells that are in the same unblocked cell cluster (group of unblocked cells that can be reached between one another, or in other words, a connected graph of unblocked cells).

There are a finite number of cells in the same unblocked cell cluster the agent can visit, so the agent will either 1) reach the target without expanding every cell or 2) expand every cell. In the first case, the agent just finds the path to the target. In the second case, if the target has not been found in the last possible cell the agent explored, then every cell is in the closed list and the agent discovers it's impossible to reach the target because no more moves can be done and no target has been found. If the target has been found, then the agent reaches the target. In all cases, because there is a finite number of unblocked cells the agent can expand and the agent doesn't re-expand cells in the closed list, it will either reach the target or discover that is impossible in finite time.

Below is the proof that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared:

Let m = number of moves of the agent (per computed normal A* path) until it reaches the target or discovers it's impossible, u = number of unblocked cells in the grid world.

In each iteration of Repeated Forward A*, the path to the goal from the current cell is computed with

1

normal A* and the "shortest presumed-unblocked path" assumption, and then the agent moves until it encounters a new blocked cell in its path, finds the target, or expanded all unblocked cells in its unblocked cell cluster. Between these 3 scenarios, the agent moves the most number of times if it encounters the last scenario, expanding all unblocked cells in its unblocked cell cluster, which can be at most u if the agent can reach all unblocked cells in the gridworld. In other words,

$$m \leq u$$

The agent wouldn't move more than the number of unblocked cells because the open list would be empty and the closed list would contain all unblocked cells it can reach. At this point, it would report that the target is found if the last cell is the target, otherwise it would report that the target cannot be found.

Again, in the worst case, the agent encounters a new blocked cell on its current path every time a new path is computed from normal A* in the Repeated Forward A* iteration. When this occurs, the agent must recompute a new A* path from its current cell, restarting its open list and closed list. However, the agent keeps new information on the blocked cells it discovers to ensure it won't get stuck on the same cell again, so the agent will recompute a new A* path AT MOST ONCE in each unblocked cell. If this is true, then the number of A* path computations, denoted by c, must be at most u, the number of unblocked cells in the gridworld, with the worst case including the scenario where the agent can reach all unblocked cells in the gridworld:

$$c \leq u$$

Since m is the number of agent moves for every normal A* path computation and c is the number of normal A* computations, mc is the total number of agent moves in the whole Repeated Forward A* algorithm. Using $m \leq u$ and $c \leq u$, it follows that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded above by the number of unblocked cells squared. In other words,

$$mc \leq u^2$$

Part 2 - The Effects of Ties

Implementation Comparison

INSERT HERE

Figure 9 - Larger g-values in favor

Step 0:
OpenList: A1
ClosedList:

Step 1:
Remove A1 from OpenList
A1's neighbors are A2, B1
OpenList: $A2(g = 1, h = 7, f = 8)$, $B1(g = 1, h = 7, f = 8)$
ClosedList: A1

Step 2:
Remove A2 from OpenList (arbitrary)
A2's neighbors are A1, B2, A3
OpenList: $B2(g = 2, h = 6, f = 8)$, $A3(g = 2, h = 6, f = 8)$, $B1(g = 1, h = 7, f = 8)$
ClosedList: A1, A2

Step 3:

Remove A3 from OpenList (largest g-value, arbitrary between A3 and B2)

A3's neighbors are A2, A4, B3

OpenList: A4($g = 3, h = 5, f = 8$), B3($g = 3, h = 5, f = 8$) B2($g = 2, h = 6, f = 8$), B1($g = 1, h = 7, f = 8$)

ClosedList: A1, A2, A3

Step 4:

Remove A4 from OpenList (largest g-value, arbitrary between A4 and B3)

A4's neighbors are A3, A5, B4

OpenList: A5($g = 4, h = 4, f = 8$), B4($g = 4, h = 4, f = 8$), B3($g = 3, h = 5, f = 8$) B2($g = 2, h = 6, f = 8$), B1($g = 1, h = 7, f = 8$)

ClosedList: A1, A2, A3, A4

Step 5:

Remove A5 from OpenList (largest g-value, arbitrary between A5 and B4)

A5's neighbors are A4, B5

OpenList: B5($g = 5, h = 3, f = 8$), B4($g = 4, h = 4, f = 8$), B3($g = 3, h = 5, f = 8$) B2($g = 2, h = 6, f = 8$), B1($g = 1, h = 7, f = 8$)

ClosedList: A1, A2, A3, A4, A5

Step 6:

Remove B5 from OpenList (largest g-value)

B5's neighbors are B4, C5 OpenList: C5($g = 6, h = 2, f = 8$), B4($g = 4, h = 4, f = 8$), B3($g = 3, h = 5, f = 8$) B2($g = 2, h = 6, f = 8$), B1($g = 1, h = 7, f = 8$)

ClosedList: A1, A2, A3, A4, A5, B5

Step 7:

Remove C5 from OpenList (largest g-value)

C5's neighbors are C4, D5 OpenList: C4($g = 7, h = 3, f = 10$), D5($g = 7, h = 1, f = 8$), B3($g = 3, h = 5, f = 8$) B2($g = 2, h = 6, f = 8$), B1($g = 1, h = 7, f = 8$)

ClosedList: A1, A2, A3, A4, A5, B5, C5

Step 8:

Remove D5 from OpenList (largest g-value)

D5's neighbors are D4, E5 OpenList: D4($g = 8, h = 2, f = 10$), E5($g = 8, h = 0, f = 8$) D5($g = 7, h = 1, f = 8$), B3($g = 3, h = 5, f = 8$) B2($g = 2, h = 6, f = 8$), B1($g = 1, h = 7, f = 8$)

ClosedList: A1, A2, A3, A4, A5, B5, C5, D5

Step 9:

Remove E5 from OpenList (largest g-value)

E5 is goal, so we stop.

Figure 9 - Smaller g-values in favor

If cells with smaller g-values were prioritized, then the agent would expand all of the cells in a breadth-first-search manner, resulting in more cells expanded.

Implementation Conclusion

To conclude, in general, Repeated Forward A* favoring larger g-values will be much faster than Repeated Forward A* favoring smaller g-values since the former will waste less time expanding cells closer to the start cell.

Part 3 - Forward vs. Backward

Proportion of Runs
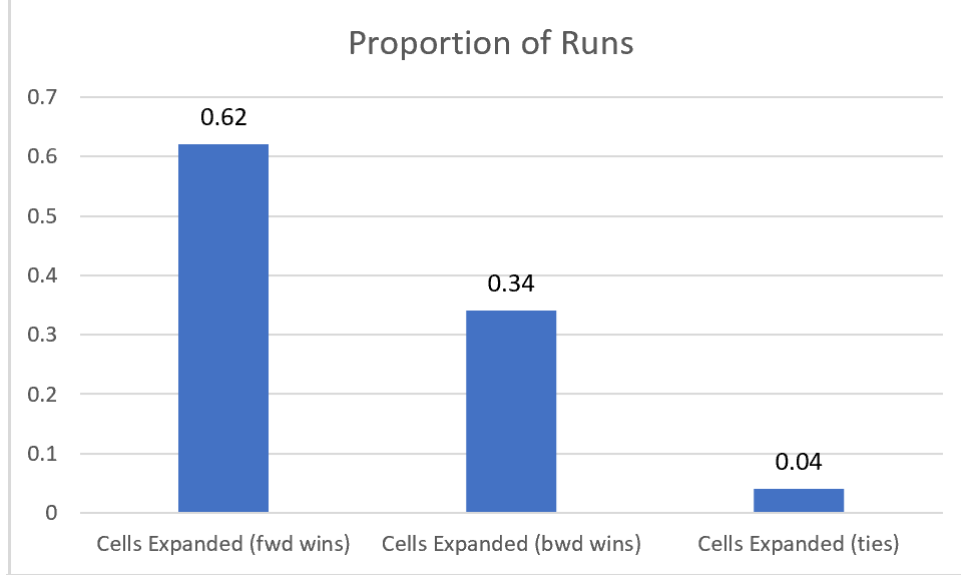
In the 50 test runs, Repeated Forward A* outperformed Repeated Backward A* 62% of the time (31 runs) in terms of the number of cells expanded. This is most likely because

Part 4 - Heuristics in the Adaptive A*

Proof 1

Prove that the Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions.

The consistent property says:

$$\forall(n, a, n') : h(n) \leq c(n, a, n') + h(n')$$

where $h(n)$ is the heuristic and $c(n, a, n')$ is the cost from node n' to a successor of node n by taking action a.

Assume the Manhattan distances are not consistent. That is,

$$\exists(n, a, n') : h(n) > c(n, a, n') + h(n')$$

Rearranging,

$$\exists(n, a, n') : h(n) - h(n') > c(n, a, n')$$

Case 1: If node n' has a greater Manhattan distance to the goal node G than node n, then $h(n')$ is greater than $h(n)$, which means $h(n) - h(n')$ is negative. Costs in the gridworld for moving from n to n' are always positive, so $h(n) - h(n') > c(n, a, n')$ cannot be true and is contradicted. Therefore, under case 1, the Manhattan distance is a consistent heuristic.

Case 2: If node n' has the same Manhattan distance to the goal node G as node n, then $h(n')$ is the

same as $h(n)$, which means $h(n) - h(n')$ is 0. Costs in the gridworld for moving from n to n' are always positive, so $h(n) - h(n') > c(n, a, n')$ cannot be true and is contradicted. Therefore, under case 2, the Manhattan distance is a consistent heuristic.

Case 3: If node n' has a smaller Manhattan distance to the goal node G than node n, then $h(n)$ is greater than $h(n')$, which means $h(n) - h(n')$ is positive. $h(n)$ and $h(n')$ are both Manhattan distances to the same goal node G, so $h(n) - h(n')$ is essentially the Manhattan distance between node n and node n', which we will denote with $h(n, n')$. Continuing from our earlier equation:

$$\exists(n, a, n') : h(n, n') > c(n, a, n')$$

From the definition of Manhattan distance:

$$\exists(n, a, n') : |x(n) - x(n')| + |y(n) - y(n')| > c(n, a, n')$$

We can think of $|x(n) - x(n')|$ and $|y(n) - y(n')|$ as the two legs of a triangle and $c(n, a, n')$ as the hypotenuse of the triangle. For such a $c(n, a, n')$ to exist so that the above inequality holds true, the agent must be allowed to move diagonally, which is not possible in our grid world. Because this isn't possible, the above inequality does not hold for any (n, a, n'). Therefore, by contradiction under case 3, the Manhattan distance is a consistent heuristic.

Proof 2

Prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase.

The Adaptive A* heuristic $hnew(s)$ is given by:

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

where $g(goal)$ is the smallest cost to the goal node and $g(s)$ is the smallest cost to node s, both under current knowledge of the gridworld computed by the previous A* run, which expanded node s.

Plugging into the consistent property:

$$\forall(n, a, n') : hnew(n) \leq c(n, a, n') + hnew(n')$$

$$\forall(n, a, n') : g(goal) - g(n) \leq c(n, a, n') + g(goal) - g(n')$$

$$\forall(n, a, n') : g(n') - g(n) \leq c(n, a, n')$$

Is there a case when $g(n') - g(n) > c(n, a, n')$? $c(n, a, n')$ is smallest when node n is on a shortest path from node s to node n'. In this scenario, $g(n') - g(n)$ is essentially the shortest distance between node n and node n', which is equivalent to or greater than $c(n, a, n')$ (equivalent if all cells between the two nodes are unblocked). Therefore, $g(n') - g(n) > c(n, a, n')$ cannot hold true for any (n, a, n') which confirms that $\forall(n, a, n') : g(n') - g(n) \leq c(n, a, n')$ is true, so the Adaptive A* heuristic $hnew(s)$ is consistent even if action costs can increase.

Part 5 - Heuristics in the Adaptive A*

REPEATED FWD VS ADPT, LARGER G CELLS

Part 6 - Memory Issues

We only use two 2-D lists for the grids, which hold integers representing the color to display (indicating unblocked, blocked, start, goal, etc.), indexed by the x and y coordinates of the cell. One 2-D list stores the real grid, and the other 2-D list stores the grid with current information. In other words, each list entry stores only the integer color. We have 6 different values for color (only need 3 bits to represent), so we can reduce the memory by storing color in a 1-byte integer instead.

We have a separate class called "Treenode" that stores the f (integer), g (integer), h (integer), Treenode parent (pointer), and coordinates 2-tuple (integer, integer). This adds up to $4 + 4 + 4 + 8 + 4 + 4 = 28$ Bytes per Treenode. Depending on the grid size, the coordinates, f, g, and h values can be compressed into smaller integers. For example, for the 101 x 101 grid, only 7 bits are required to represent an integer from 0 to 101. The x and y coordinates can be stored together in a 2-byte integer, with the x coordinate stored in the first 8 bits and the y coordinate stored in the last 8 bits (use bit shifts to access). As for the h value in a 101 x 101 grid, the maximum value would be the largest Manhattan distance, $101 + 101 = 202$, which can be represented by 1 Byte. The f and g values are upper bounded by 101 x 101 = 10201, so each can be represented by 14 bits, for a total of 28 bits, which can be accommodated by a 4-Byte integer. So the memory for one Treenode can be reduced from 28 Bytes to 4 (f and g) + 1 (h) + 2 (x and y coordinates) + 8 (parent pointer) = 15 Bytes, assuming the compressions are possible.

For a gridworld of size 1001 x 1001, the memory to store our 2 grids would be:

$$4 \text{ Bytes (color integer)} * 101 * 101 * 2 \text{ (each grid)} = 81608 \text{ Bytes}$$

With a memory limit of 4 MBytes, the largest gridworld to operate on would be:

$$4 * S * S * 2 = 4 * 1048576$$

$$S = 724, \text{ or } 724 \text{ x } 724 \text{ gridworld.}$$

The Treenodes are dynamically created and added to the open list as the A* algorithm runs, so we will exclude the memory used by Treenodes since that varies from grid to grid.