Fast Trajectory Replanning

Hammad Iqbal (HI60) and Ateeb Jamal (SAJ148)

February 27th, 2019

Setting Up The Environment

To generate our simple mazes, we looped through every node in our array to initialize it. Upon initialization, the constructor gives the node a 30If a node is chosen to be a wall, the node's blocked boolean is set to true and when printing out the grid, an 'X' will appear.

Understanding the Methods

A. In this example, the robot would initially try to go to the East before realizing that there is a blockage. This is because while at E2 the robot does not know about the blockage at E3, so as far as it can see going East would be the shortest path. To explain this further, within the terms of A*, we can define the g, h, and f values of the three nodes in the open list. While at E2 the open list would contain E1 (going East), D2 (going North), and E3 (going East). All three of these values would have a g (cost to go value) of 1. E1 would have an h value of 4 as it is for spaces away from the target T, and therefore an f value of 5. D2 would have an h value of 4 as it is four spaces away (Manhattan distance) from the target, and therefore an f value of 5 as well. Finally, E3 would have an h value of 2 as it is only 2 spaces away from the target T, and therefore an f value of 3. As we can see, E3 has the smallest overall f value and will be the first node popped from the open list.

B. In a finite gridworld the worst case scenario is that it has visited all the unblocked cells. The agent either gets to it's target or does not. If the agent has found the target the path has been found and the open list is empty; if the target has not been found but all cells, that are possible, have been visited the path is not found and open list is empty. The open list has a finite number of nodes and because of that will be empty in finite time.

The number of moves is the max amount of cells expanded multiplied the max amount of cells in a single expansion. Let's say that the letter "n" denotes unblocked nodes. The algorithm will expand each n cells each time. Therefore, the numbers will be both n so expanding n cells n time, which is n squared. There will be n squared maximum moves.

Effect of Ties

To implement this portion of our program, we used the object-oriented features in Java to our advantage. In Java, we used the built-in PriorityQueue to represent our heap. This allows us to choose a Comparator. So, we coded two different Comparators- one to break ties in favor of higher g values and one to break ties in favor of lower g values. We passed the desired comparator as a parameter when creating our PriorityQueue / Heap.

We observed that breaking ties in favor of lower g values was inherently inefficient. Logically, we can explain this as a higher g value means that we are further away from the starting position. Due to the way A* chooses its next node, we know that it may choose the higher g values for a reason. Consider a given node with a high g value was chosen. To be chosen, it must have had a lower overall f value. This means that despite the high g value, the h value was minimal (f = g + h). There are two ways to interpret this. 1) We know this is close to the target as the g value is great and we have traveled a greater distance from the start position. Or 2) We can infer that the h value is low which also indicates that we are close to the target position.

Foward vs. Backwards

Unlike the different tie-breaking conditions, we found that using a backward A* over a forward A* did not supply any meaningful difference. This makes a lot of logical sense as our start and end positions are randomly generated anyway. There is nothing dictating/constraining what can and can't be a start or end position. Therefore, comparing these two is no different from comparing forward A* on a given grid and forward A* on the same grid flipped about the diagonal (with the same start and end positions). Overall, the differences for these were minimal. The only time when it created a significant difference was if the start position was enclosed in a small box, say in the corner of the grid. Running forward A* allowed us to discover this quickly, while backward A* had to spend more time exploring every option to see there was no way into the box.

Heuristics in Adaptive A*

Manhattan distances are consistent in gridworlds because they are creating the shortest possible horizontal and vertical paths. The Manhattan distance between two points will always be the same in any combination of vertical and horizontal steps. Because the agent is only allowed to move in compass directions (North, East, South, West) and not diagonally, the Manhattan heuristic will never overestimate the cost of reaching the target, and if the target were to be allowed to move diagonally then the heuristic will overestimate the target.

The triangle inequality where h(n) ¡ or = to h(n') + c(n, n'); n is the cell, n' is the successor and c(n, n') determines the cost to move from the cell to next cell. For a triangle the three sides are represented by a ¿ b + c. Since the inequality will always be consistent so will the h values, there won't be any case where the inequality will be false. If we want to prove that h values hnew(n) consistent as well, let's substitute hnew(n) with g(goal) - g(n) and have hnew(n') be g(goal)-g(n'). The inequality becomes, g(goal) - g(n) ¡ or = to g(goal) - g(n') + c(n.n'). Simplify to g(n) ¿ or = to g(n') - c(n, n'). If the agent can only move in the four compass directions, this is always true. Since c(n,n') is always one, if g(n') is smaller than g(n), subtracting one from it will make it smaller. If g(n') is bigger than g(n), the only case is when their difference is one, which is true in this case. Therefore, the equality is satisfied and hnew(s) is always consistent. Furthermore if the action's cost is greater than one, add c' as the cost after increase. Then hnew(n) ¡ or = to hnew(n') + c(n,n') ¡ or = to hnew(n') + c'(n,n') which is consistent as well.

Heuristics in Adaptive A* Cont.

To implement Adaptive A*, only one small change was needed. Before running the next iteration of A*, we had to traverse all nodes on the established path and change their h values. Their h values were changed to the length of the path - the g cost of that node. This change helps to avoid the re-expansion of nodes while still finding the shortest path. Additionally, we know that these new values are admissible as explained above. Based on this it is clear that Adaptive A* should have a strong advantage.

That being said, when we ran our code, although Adaptive A* tended to have better results, they were not significantly better. For example, on one iteration we went from 237 in repeated A* to 223 in Adaptive A*. Comparing with other groups, similar results were found. We hypothesize that this is likely due to our grid size and would be something significantly more noticeable when running on extremely large grids (and game maps). We think this to be true as when we ran on extremely small grids such as 7x7 or 9x9, we received the same number of node expansions just about every time. Overall, while Adaptive A* algorithmically makes a lot of sense, a large-scale implementation is needed to maximize its benefit.

Memory Allocation

In our implementation, each node stored the following variables:
int g; (4 byte)
int h; (4 byte)
int f; (4 byte)
int x; (4 byte)
int y; (4 byte)
String psn; //These value was superflous and can be optimized out
boolean blocked; (1 byte)

The total byte size of these Java variables is 21 bytes. This means that in a 1001x1001 gridworld, our implementation, optimized would require 21042021 bytes or about 21.03 megabytes.

4MBytes is 4,194,304 bytes. 4,194,304 bytes / 21 bytes gives us a possibility of 199,728.67 bytes. The square root of that is 446.91. Rounding down, that means our implementation, when further optimized, can run an approximately 446x446 gridworld on just 4Mbytes of space.