Assignment 2: Mazeworld

Avery Frankenberg

January 15, 2019

1 A Star Search

For this assignment our goal was to solve a maze. First we began by implementing an A Star search algorithm which combines the use of a heuristic and a transition cost which represents the accurate cost to get to that state. My implementation uses a few methods. First there is the class AStarNode. Each node contains the state, the parent, the heuristic value calculated from the heuristic function passed in, and the transition cost to get to that node. In addition there is a priority method which returns the priority and a method that allows the heap to work with this node type.

The search itself occurs in the astar search method. This method takes a specific search problem and a heuristic function and performs the search. It uses the data structure of a heap which is an array ordered by the priority of each node. As the other search algorithms we implemented were coded, we start by popping the first element onto the heap with its associated priority. The node is only poppped off though if the transition cost is not -1. A visited dictionary is created which stores each state and the associated node. In addition a solution is initialized which includes the heuristic, the search problem, and the search type. After these initializations, the while loop is entered which runs as long as there are nodes still in the heap and the goal has not been found. The next node in the priority is popped off, the visited cost dictionary is updated, and the current state is obtained for this node. For this current state the successors are obtained, which are all checked to be safe states in the get successors method in the maze problem method, and added to the heap with their associated priority. We also check to see if the visited cost dictionary already has the state in it. If it does and the cost of the current node is smaller than the cost of the node stored in the dictionary, the dictionary pointer is changed to point to the less expensive node and the transition cost of the node is changed to be -1. If the node that is popped off of the heap is the goal node then the backchain method is called. The backchain method takes the goal node and the solution and obtains the path cost and the path to get to the solution. This is then output to the console. The solution nodes visited value is incremented at the end of the astar method.

With the A star algorithm the heuristic can be changed to improve its efficacy for different problems or algorithms. However, for this first portion of the

assignment I implemented a simple manhattan heuristic which calculates the total distance without including diagonals to the goal from the current location of the robot. It disregards the walls. For the next portion of the assignment I changed my heuristic.

2 Multi-robot Coordination

2.1 Implementation

For this coordination implementation I modified the code that I had already created for the single robot problem. The state for each node is a tuple where the first number represents which robot number is currently moving and the rest of the values represent the three robot locations. The get successor method changed by having the robots move in succession, one at a time. Also instead of just calling get successors, there is now a get successors all method which either activates get successors or get successors multi which is the method for multiple robots in a maze. The goal test method has to change because instead of one goal now robot A has to get to goal location A, robot B has to get to goal location B, and robot C has to get to location C. In addition, the safe state has to ensure that these robots aren't overlapping even though there are now three of them in the maze at once.

For the first heuristic here I just implemented a manhattan heuristic which decreased the number of nodes visited from to 413 as compared to 3104 which were explored using uniform cost search.

2.2 Discussion Questions

1. If there are k robots in the system, I would represent the state with a tuple of size 2*k+1. The 2*k part represents the (x,y) coordinates for each robot's location and the +1 includes which robot's turn it is to move. In this way we can both recreate where all of the robots are and we know who moves next.
2. The upper bound for the number of states for this system in terms of n and k is n^{2k}. For a single robot there are n^2 possible locations and therefore the upper bound is that many possibilities for each of k robots.
3. A rough estimate of the number of these states that represent collisions is the upper bound of the total number of states minus the number of states with wall collisions $n^{2k} - (n^2 - w)^k$. This expression simplifies to w^k of these states represent wall collisions.
4. If there are not many walls in a 100x100 maze with around 10 robots, I do not expect a straightforward breadth first search to be computationally feasible for finding solutions for all start and goal pairs. If the robots are acting sequentially, one moves at each node but it can still move 5 different locations. From those 5 locations the next robot has the chance to move 5 new locations and this pattern continues to repeat. Because the state space is so big and there are no walls, almost all of these node states are safe and

therefore the tree expands rapidly. For this reason, conducting a standard breadth first search is not computationally feasible simply because the tree of possible states and paths is huge.
5. A monotonic function is a heuristic function where every node n and every child node n' generated have an estimated cost from the node to the goal that is no greater than the cost from n to n' plus the cost of getting to the goal from n'. The heuristic for my A* search algorithm is a wavefront heuristic. Before the search is even conducted, the cost from each location in the maze to the goal location for each robot is computed and stored in a table. If there are three robots there are three different heuristic look up tables created. Then when the heuristic is called on a specific state, the values in each robot's table for that same state location are added together and returned. This heuristic is optimistic because it is never overestimating the distance to the goal as all of the distances to the goal are already pre-computed and all that has to happen is these values are added together for each state. Because the estimated cost from the node to the goal is never greater than the cost from n to n' plus the cost of getting to the goal from n' (because all of the values are computed from the goal to the state to begin with anyway), this heuristic function is monotonic.
6. The mazes that I created are titled maze4.maz - maze8.maz. These mazes range in size and number of robots.
Maze01: The goal locations for maze01 are (1, 2, 1, 1, 1, 0). This maze is interesting because the robots start in a corridor in the order opposite of what they need to be in the future so the AI has to move these robots out of the way before putting them back in the appropriate order.
Maze02: The goal locations for maze02 are (4, 0, 1, 0). This maze is interesting because there are only two robots, which demonstrates that the algorithm doesn't only work with 1 or 3 in the maze. The robots have to switch places but both start in their own corridors but not in the correct order that they end up at the top. So first they have to get out of their corridor and then correctly arrange at the top of the maze.
Maze03: The goal locations for maze03 are (3, 5, 3, 5, 3, 5). This maze is interesting because the three robots have the same goal (which under our problem description is impossible because they cannot overlap).
Maze04: The goal locations for maze04 are (17, 16, 17, 18) and the robots start in opposite corners. This maze is interesting because it is very large and also very easily solvable by the human eye. There are two partial corridors toward the solution, one for each robot, and when a human solves it, it is very easy to follow these paths and get to the solution almost instantaneously. However, because the paths are not always in the direction that the manhattan distance heuristic would suggest, it takes longer for the AI to find the solution path. Hence, the number of nodes explored and the path itself is long.
Maze05: The goal location for maze05 is (31,27). This maze is 40x40 with a single robot. It is interesting because ideal path would be diagonally straight to the goal node. However, diagonal movement is not possible and therefore it has to step along toward the goal which means that more nodes have to be

traversed as compared to a diagonal path.
7. The 8 puzzle in the book is a special case of this problem because it is just an nxn maze with 8 agents and 8 goal locations in it. It is also a special case because instead of having many possible moves available at any given state, there are only ever 2-4 agents that can move from a given state. The heuristic function I chose is only okay for this problem. Because the maze space isn't very big (3x3), the numbers representing the distance from the goal for each agent never get very large. While the nodes closer to the goal appropriately have lower heuristic values, these values are all very close to one another. In addition, because there is only one empty space, many agents have to move away from their goal before eventually getting there. For these reasons, the wavefront heuristic is not ideal for this problem statement.
8. The state space of the 8 puzzle is made of two disjoint sets, the set of puzzles that lead to a solution and the set of puzzles that do not lead to a solution. We could modify our program by running all possible start configurations and keeping two sets. One set would contain the start states that did get to a solution and the other would contain the start states that never found a solution and failed. In this way we would be able to compare and demonstrate that these sets are completely disjoint, and therefore have no overlap.

3 Blind Robot Problem

For the blind robot problem we have to create a different problem class. This class contains a get successors all method, safe state method, goal test method, animate path method and a get start state method which creates the start set of all possible locations the robot could be in the maze. We use the same method names because that way the Astar search can still work with this problem. The get successors all method is different from the corresponding method in the MazeworldProblem class. In this version, all possible states are stored in a set. From that set, each possible motion is taken (north, south, east, west). For each state that generated from the motion north, it is checked if it is safe–i.e. if it is on the board for this blind robot problem. If so, then it is added to the set for that motion and then the set is added to the successors list. At the end this successors list is returned from the method.

One important additional aspect of the Blind robot problem is that the sets have to be added to the successor list as frozen sets. This means that they can be treated as one object, which is crucial for the A star algorithm. If not, a set is an unhashable type and therefore the visited cost functionality fails to work.

For the heuristic for this A* search on the blind robot, I decided to use the size of the set as a heuristic measure. I did this because we ideally want to get closer and closer to knowing our location, which only happens when the size of a set is 1. Therefore, the smaller a set is the higher priority it should have to be explored because we're closer to 1. knowing where the robot is and 2. getting to the goal location. This heuristic is not optimistic because there could be a

set of size 1 which is not the goal node. However, that node still is valuable because it tells us where the robot is at that point in time.

4 Bonus: Polynomial-time blind robot planning

To prove that a sensorless plan similar to the solution for blind robot problem is always possible as long as the size of the maze is finite and the start is in the same connected component as the goal, it is first important to understand what the same connected component means. I believe that two nodes are in the same connected component if there is a legal path to get from one to the other and therefore traversing a tree of possible nodes can get you from the start to the goal. The way that the sensorless search works is that it contains a set of all possible locations. From each of these possible locations it generates the next set of possible locations by applying each possible movement to each possible state. If the state space is finite, then eventually either the goal location should be reached or we should know where the robot is because all other possible locations have been eliminated. For example, locations are eliminated if the possible state set is (3, 1) (3, 2) and we move east but there is a wall at (3, 3) because that is at the end of the maze. Therefore the child state set would become (3, 1). In this way, as long as there is a path from the goal to the state, we will be able to find it using the sensorless plan outlined above.

The next task is to describe how to create a motion planner that runs in time that is linear or polynomial to the number of cells in the maze. Professor Balkcom suggests that for every step we should try and decrease the number of possible states by 1 at least. This analogy is not exactly perfect, but I was thinking about the game 2048 and I believe there are some similarities with the blind robot problem. In 2048, the goal is somewhat randomly achieved because the order that the tiles are generated and added to the board is seemingly random. However, there is an overall strategy that almost always results in a solution being obtained. This strategy is to alternate the movements enough that the bulk of the tiles stay concentrated in one corner. I think that considering the almost universal strategy for 2048 and applying some of the concepts to the blind robot problem could be helpful in reducing the number of possible states for every new layer of generated nodes. Instead of generating all possible successor states for every node by applying every possible motion, I think it would be interesting to try alternating motions, doing an east/west move and then doing a north/south move. In this way at every step the number of possible nodes would decrease more rapidly. For example if the set of possible states was (3,0), (2,0), (2, 1), (1, 0), (1, 1) and the motion was to move west, then then next set of possible states would be (3, 0), (2, 0), (1,). This immediately eliminated two possible states from the state set. If the strategy was altered to more intentionally vary the moves I believe it would reduce the time spent on the algorithm.

5 Previous Work: Not Graduate Student

To learn more about the multi-robot problem I read the article "Finding Optimal Solutions to Cooperative Pathfinding Problems" written by Trevor Standley from UCLA. This article focuses on creating a complete and optimal algorithm for solving this problem. They use two main techniques to achieve this goal: operator decomposition which reduces the branching factor of the problem and the use of independent sub-problems. They compare how a standard A* search algorithm with a perfect heursitic generates $(9^n)t$ nodes whereas their A* search with operator decomposition generates only $9nt$ nodes. Next, they figured out a method for creating more independent sub-problems. When nodes with the same heuristic value are available, the node with the the optimal path that has the fewest possible conflicts is returned. By using conflict avoidance tables they are able to improve the search time. Through these two primary methods, Standley was able to optimally solve 90.45 percent of the problems in under a second (177). From their experiments they were able to demonstrate how operator decomposition and independence detection drastically improved thier ability to solve the multi-robot problem.