Mazeworld Assignment Report

Paolo Takagi-Atilano

Introduction

The Mazeworld problem is explained as follows: There is a `n x m` maze that has open floor tiles and wall tiles. We represent these mazes using (the venerable tradition of) ASCII art. Below is an example (where `.` indicates a floor tile, and `#` indicates a wall tile):

```
.......
.##....
..##...
.......
..##...
#.###..
....##.
```

You have a robot, which can move in any of four directions: North (towards the top of the screen), East, South, and West. There are coordinates for the maze, where `(0,0)` is the bottom left corner tile, and `(n-1, m-1)` is the top right corner tile. The robot is both aware of where it is and what the diagram of the map is. The solution to any Mazeworld problem is a path that the robot can negotiate from the starting coordinate to the given goal coordinate. In order to represent the location of a robot, we a letter to denote it's location on the ASCII map. So for example, the robot on the below map is at coordinate `(0,0)`:

```
.......
.##....
..##...
.......
..##...
#.###..
A...##.
```

This report will show and explain different methods to solve different versions of the mazeworld problem. It will then discuss their corresponding strengths and weaknesses.

A-Star Search

The A* search algorithm is a method that can be used to find potential solutions to the Mazeworld problem. In order to understand how A* search works, one first must understand two other concepts: Uniform-Cost-Search and heuristics.

Uniform-Cost-Search:

Uniform-Cost-Search (UCS) is a version of (memoizing) DFS, where instead of using a stack data structure, a priority queue is used instead. In order for a priority queue to be helpful, there must be a way to rank different successor nodes. For UCS, we use a cost function in order to rank nodes, where the priority queue prioritizes the successor nodes with the lowest cost. For the Mazeworld problem example, we calculate the cost function as follows: Whenever a robot changes its coordinates by 1 tile, it will consume 1 unit of fuel (for a cost of 1). Note: in the multi-robot coordination problem outlined below, there will be some situations in which the robot should pass its turn, so that another robot can move instead. Hence when the robot passes its turn, it does not consume fuel.

Here is provided pseudocode for UCS:

```
function UCS(problem) return a solution, or failure
    node <- a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier <- a priority queue ordered by PATH-COST, with node as the only element
    visited_cost <- an empty dictionary

    loop do
        node <- POP(frontier) # chooses the lowest-cost node in frontier
        if problem.GOAL-TEST(node.state) then return SOLUTION(node)
        if node.state not in visited_cost.keys() or node.cost < visited_cost[node.sta
te]
            then visited_cost[node.state] = node.cost
        for child_state in problem.get_succesors(node.state) do
            child <- CHILD-NODE(state, node) # child node has its state and pointer t
o parent
            child.cost = problem.cost_func(child)
            if child.state not in visited_cost.keys() or child.cost < visited_cost[no
de.state]
                then visited_cost[child.state] = child.cost, frontier.ADD(child)
    return failure
```

The use of the `visited_cost` dictionary is a solution to some issues that are associated with trying to check if certain nodes already exist in the priority queue. The `visited_cost` dictionary solves this by updating the cost of nodes every single time one is reached that either has not been explored before, or has a lower cost than it previously had. Hence, this dictionary both keeps track of already visited notes as well as their costs. Apparently another solution is to make the `frontier` a fibbonaci heap, but from what I have

heard, that is much more complicated.

Heuristics:

A heuristic is an estimate of the cost between a given state and the goal state. It is usually reached by relaxing some of the constraints of the problem. It can be considered a function that takes state of input, and outputs a number that represents the cost estimate, denoted as `h(n)`. We will denote the actual cost to reach any node `n` from the goal node as `g(n)`.

Some heuristics are comsidered optimistic. An optimistic heuristic is one that underestimates cost of path to goal node from given node. It is important to know whether or not a heuristic is optimistic or not because using A* search (explained below) with an optimistic heuristic will produce shortest paths from start node to goal node (or an optimal path).

Also, below there is discussion as to whether or not some heuristics are monotonic. As such, the definition of a monotonic heuristic will be included here: A monotonic heuristic is a heuristic such that for every node `n` and every successor `n'` of `n` generated by the `search_problem`'s `get_successors` function, the estimated cost of reaching the goal from `n` (the value the heuristic produceses for node `n`) is not greater than the real cost of reaching `n'` from `n` plus the value the heuristic produces for the node `n'`. Mathematically, the condition for a heuristic to be monotonic looks like this: `h(n) <= [g(n') - g(n)] + h(n')`. Monotonicity is a stronger condition than optimisticity. All monotonic heuristics are also optimistic (but not necessarily the other way around).

A-Star Search:

A-Star search (A* search) works exactly the same was a UCS does, except rather than have the priority queue rank nodes by known cost from the start node, it has the priority queue rank nodes by their known class from start node plus estimated cost to goal node (calculated form heuristic). Hence, if the null heuristic is used (a heuristic that always returns 0), then A* search is equivalent to UCS. This is how UCS is simulated in my implementation.

Here is pseudocode for how A* search works (my implementation is based off of this):

```
function A*(problem) return a solution, or failure
    node <- a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier <- a priority queue ordered by PATH-COST, with node as the only element
    visited_cost <- an empty dictionary

    loop do
        node <- POP(frontier) # chooses the lowest g(node)+h(node) node in frontier
        if problem.GOAL-TEST(node.state) then return SOLUTION(node)
        if node.state not in visited_cost.keys() or node.cost < visited_cost[node.sta
te]
            then visited_cost[node.state] = node.cost
        for child_state in problem.get_succesors(node.state) do
            child <- CHILD-NODE(state, node, h())
            child.transition_cost = problem.cost_func(child)
            child.heuristic_cost = h(child.state)
            if child.state not in visited_cost.keys() or child.cost < visited_cost[no
de.state]
                then visited_cost[child.state] = child.cost, frontier.ADD(child)
    return failure
```

As explained above, it is extremely similar to UCS, except it factors in `h(n)` (heuristic) cost in addition to `g(n)` (transition) cost to priority queue ordering.

In order to compare the effeciveness of A* search in this report (with different heurstics), it will be compared to the solutions found using BFS.

Building the Model

Maze.py

Generates maze objects from .maz files. This was provided so I will not go into depth about how it works.

.maz files

Files containing information necessary to create maze object. Contains a depiction of the maze (following the same format as the example maze at the beginning of this report), and below some number of lines that say `\robot x y`, where x and y is the provided starting position for this particular robot. The number of robots in the problem is the same as the number of these `\robot` lines. Here is an example `.maz` file:

```
.......
.##....
..##...
.......
..##...
#.###..
....##.
\robot 0 0
```

SearchSolution.py

Object that contains information that contains infoormation necessary to presenting the solution. It is the same as the Missionares and Cannibals assignment, excent a new value, `self.cost` is added. This was provided so I will not go into depth about how it works.

MazeworldProblem.py

Contains the `MazeworldProblem` object that conforms to the set of rules that describes the Mazeworld problem. It is used for both single robot problems and multi-robot coordination. This object has the following information (not including completely provided functions and constructor):

Data

`maze` - the given maze object for this `MazeworldProblem` object.

`goal_locations` - the given tuple of where which robots must be in order to have achieved the goal for this specific `MazeworldProblem` object.

`total_robots` - total number of robots in this specific `MazeworldProblem` object.

`start_state` - start state for this specific `MazeworldProblem` object.

Functions

`get_successors(self, state)` - returns a set of successor state for given state for this specific `MazeworldProblem` object. Each successor state is a tuple.

`is_legal(self, state)` - determines if a given state is legal in this specific `MazeworldProblem` object. Used as a helper function for `get_successors(self, state)`.

`goal_test(self, state)` - determines if a given state qualifies as a goal state for this specific `MazeworldProblem` object.

`cost_func(self, node)` - cost function for this specific `MazeworldProblem` object. Returns the cost

of given node.

`is_equiv(self, state1, state2)` - determines if two states are the same besides their turn element (aka "equivalent"). Used as a helper function for `cost_func(self, node)`.

`manhattan_heuristic(self, state)` - returns the sum of manhattan distances from given state to goal state.

`straight_line_heuristic(self, state)` - return the sum of straight line distances from given state to goal state.

SensorlessProblem.py

Contains the `SensorlessProblem` object that conforms to the set of rules that describes the Blind Robot problem. This object has the following information (not including completely provided functions and constructor):

Data

`maze` - the given maze object for this `SensorlessProblem` object. `start_state` - the determined start state for this `SensorlessProblem` object.

Functions

`set_start_state(self)` - sets the start state of this `SensorlessProblem` object to every single floor tile in the `maze` object. This means that rhe initial start start is a belief state where the robot could be on any floor tile.

`goal_test(self, state)` - determines if provided state is a goal state for this `SensorlessProblem` object. Returns true if length of state is 1, false otherwise.

`get_successors(self, state)` - returns a set of successor state for given state for this specific `SensorlessProblem` object. Each successor state is a tuple that is the belief state. The belief state is a tuple of all possible states. So in all, this returns a set of tuples of tuples.

`is_legal(self, node)` - determines if a certain state is legal (if the robot is on a floor tile in the maze object for this `SensorlessProblem` object). To determine if a belief state is legal, every single certain state in the belief state is teseted by this function.

`cost_func(self, node)` - Cost function for this `SensorlessProblem`. Returns cost of given node. There an assumption here that robots never pass their turn (because it does not make sense for a robot to do so in the Blind Robot problem).

uninformed_search.py

Contains the BFS search algorithm that was written for the Missionaries and Cannibals assignment. It is the exact same as that BFS algorithm expect is typecasts the `start_node.state` to a tuple when adding to the visited set, and adds given cost to the solution (thus makes the extra assumptoin that given `search_problem` has a `cost_func(state)`).

astar_search.py

Contains the `AstarNode` object which is the search node implementation for A* search. The only part of this that I wrote (besides the constructor) was the `priortiy(self)` function, which returned the sum of the corresponding heuristic value and transition*cost value. If we wanted to use UCS, we would instead merely return the tansition*cost value, and similarly if we wanted to use greedy search, we would return the heuristic value.

This file also contains my implementation of the A* search algorithm. How it works is described above in its own section.

test_mazeworld.py

Creates, runs, then prints all of the tests used for the single robot problems, and then runs and prints all of the tests for the multi robot problems.

test_sensorless.py

Creates, runs, then prints all of the tests used for the blind robot problems.

Single Robot

This is the most simple version of the Mazeworld problem, which follows all the rules set forth in the introduction.

Tests:

The following heuristics were used:

`null_heuristic(state)` - returns 0.

`manhattan_heuristic(state)` - returns manhattan distance between given state and goal state (manhattan distance is distance between two tiles if wall tiles were floor tiles).

`straight_line_heuristic(state)` - returns the straight line distance between given state and goal state (using Pythagorean Theorem).

Below are maps that I used and corresponding discussion of results. To see precise outputs, please refer to the Raw Output section of this report. NOTE: `S` denotes the robot's start position, and `G` denotes the robot's goal position:

Maze 1:

```
......G
.......
.......
.......
.......
.......
S......
```

Discussion:

This was a very basic map, with zero walls, and the goal is to get the robot from the bottom left corner to top the top right corner. I thought that this was an appropriate baseline first test, which would be useful to compare the other tests too. We saw that BFS and UCS both found the optimal path after visitng 215 nodes. Both the `manhattan_heuristic` and the `straight_line_heuristic` were improvements from those searches, at 185 and 197 nodes visited, respectively. The optimized versions of `manhattan_heuristic` and `straight_line_heuristic` were significantly faster, at 106 and 107 nodes visited, respectively (by optimized, I mean multiplied by some constant where the A* search visits the fewest nodes).

Maze 2:

```
............
............
............
............
............
............
S..........G
............
............
............
............
............
............
```

Discussion:

I decided to give this map a try beause I was interested in what would happen when there were many nodes

that would not be part of the final solution path (all the tiles on top or below the row with both the start and goal tiles). I doubled both the height and width of the maze from the Maze 1 because I wanted the optimal solution length to be the same. This way we could see how the different search algorithms fared in an environment with a vey similar solution but more potential nodes. Unsurprisingly, they all did worse, as they all had more nodes to explore. That being said, how they did compared to each other was remarkably similar. The optimized `straight_line_heuristic` visited about half as many nodes as the BFS algorithm, similar to the Maze 1 outputs. This time, the UCS visited slightly fewer nodes than the BFS. Unsurprisingly, the not optimized `manhattan_distance` did better than BFS and the UCS, but worse than optimized algorithms.

Maze 3:

```
......G
.##....
..##...
.......
..##...
#.###..
S...##.
```

Discussion:

This map was meant to be the same as the first map, but with walls inhibiting the number of potential paths to the goal. Both BFS and UCS visited fewer nodes than Maze 1, which makes sense, as there were fewer potential floor tiles that they could have visited. Again, these two algorithms were comparable in terms of nodes visited to each other. Both of the A* searches did better than their Maze 1 counterparts, but not on the sacale the the BFS and UCS searches did. This makes sense, because the potential benefit from a heurstic is minimized when there are fewer nodes that will not lead to a goal node. In other words, the BFS and UCS are more likely to choose a succesful path (despite being uninformed), because the walls get rid of many possible unsuccesful paths.

At this point, I realized that the `straight_line_heuristic` and the `manhattan_heuristic` use the same mathematical concept, and thus result in the same number of nodes visited in an A* search when optimized. In other words, they are essentially the same heuristic when giving optimal answers. While it was worth trying to implement an additional heuristic, it appears that this one does not lead anywhere new.

Multi-Robot Coordination

State:

If there are `k` robots in a `n x n` maze, I would represent the state as such: `(t, x1, y1, x2, y2, ... , xk, yk)`, where `x1` and `y1` are integers corresponding to the x and

y values of the first robot, `x2` and `y2` are integers corresponding to the x and y values of the second robot, and so on. `t` is also an integer that represents which robot's turn it is. With this state, if we were to somehow forget where all of the robots were, we could reconstruct all of their locations and know which robot's turn it was. Note: I wrote the program so that a `t` value of 0 indicates the first robot's turn to move (in order to make math more simple by staying consistent with zero-indexing used in lists).

Since there are a total of `n*n` locations in the make and we have `k` robots, an upper bound estimate for total possible states (without considering their legality) is `n*n choose k`.

If there are `w` wall squares and `n` is much larger than `k`, then there are `n*n - w` free spaces. Hence, there are `[n*n - w] choose k` non-collisions, so there are an estimated `[n*n choose k] - [(n*n - w) choose k]` total collisions.

A monotonic heuristic for this state is the sum of each robot's manhattan distance. Since the manhattan distance is a positive monotnonic heuristic, the sum of manhattan distances is monotonic. The following is an explanation of why the manhattan distance is monotonic: There is a robot at location `x`, the goal is at location `y`, and a sucessor state of `x` is `x'`, which is just every tile next (not diagonal) to the corresponding tile with `x`. There are two options. First, `x'` is farther away from the goal than `x` (from a bird's eye view), in which case `h(n) < h(n')`, so `h(n) < h(n') + 1`. Second, `x'` is closer to the goal by 1 unit, in which case `h(n) = h(n)+1`. Either way, the manhattan distance heuristic is monotonic.

I find it easiest to interpret the 8-puzzle as follows: each number tile is a robot, that needs to move its corresponding goal state. This puzle is a special case becuase there are not walls that limit movement, but rather the scarcity of tile spaces wihout robots on them. But at its heart, the puzzle is still the same, and follows the same rules. The sum of manhattan distances is still a good heuristic to use, because it is still monotonic, the best case scenario would be that each numer tile only move its manhattan distance to its corresponding goal, so we see that our heuristic is still monotonic.

The state space of the 8-puzzle is made of two disjoint sets. One set is the set of all state spaces that yield a solveable 8-puzzle, and the other set is the st of all state spaces that do not yield a solveable 8-puzzle. One method would be to implement a form of Waveform BFS similar to what I did for extra credit in my report for the missionaries and canibals problem. I would provide the goal state as input, and from that determine every single possible starting state that leads to the goal state, and return then all in a set. Then I would write a method that would return every single possible stating space as a set. I would then compare the two sets to see if they had the same elements. I would expect that they would not, which would prove that there were the two aforementioned state spaces. After further exploration, I found this: (https://math.stackexchange.com/questions/293527/how-to-check-if-a-8-puzzle-is-solvable) which provides a method to determine whether or not a given 8-puzzle is solveable. If it has an even number of inversions it is solveable, otherwise it is not. This provides another method to prove that there are the two disjoint state spaces. I could feed my current BFS algorithm one 8-puzzle that I know to be solveable, and one that I know

that is not solveable. I would then expect the solveable puzzle to output a solution, and the not solveable puzzle to output that no solution was found. That would prove that the aforementioned disjoint state spaces existed.

Testing

The `MazeworldProblem` object was designed to handle multiple robots, so the exact same code for the single robot problems could be used for the multi robot problems. The only difference were the .maz files that were used. Below is their outline and discussion. See Raw Output for their precise outputs. For maze diagrams, letters denote robot starting positions, and numbers denote robot goal positions (A = 1, B = 2, C = 3)

Maze 1:

```
..#..
A.#.1
B...2
C.#.3
..#..
```

Discussion:

This was the first Multi Robot maze that I tested. I tried to keep it realatively simple, with the only obstace being a bottleneck that all of the nodes had to pass through. There were some interesting results. First, the BFS and UCS visited approximately the same number of nodes. BFS produced a solution path that was optimal with respect to length, and UCS produced a solution path that was optimal with respect to cost. Both the manhattan distance and straight line heuristics visited significantly fewer nodes than the uninformed search methods (as expected). The manhattan heuristic visited slightly fewer nodes than the straight line, but produced a suboptimal solution (cost). It is worth noting that adding robots to a Mazeworld problems appears to make the problem significantly more computationally difficult. This makes sense, because there are many more potential states that are produced.

Maze 2:

```
........3.
........#.
.......C#.
........#.
........#.
........#.
........#.
........#.
........#2
AB......#1
```

Discussion:

This problem I tried to make more difficult by making one's robots final location the location of the bottleneck that the other robots must pass through. It seems like this make the problem significantly more difficult, as millions of nodes had to be visited to find solutions. This problem produced results similar to Maze 1, except the informed search methods did not perform as well (still significantly outpreformed uninformed search). I suspect that this was because the bottleneck that they had to move through was directly on the way to their goal states, and instead had to move out of the way of the solution to pass through the bottleneck, which both heuristics would penalize.

Maze 3:

```
..........#........
.#.#.#.#...#........
A#.#.#.#####........
B#.#.#.#............
C#######........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.#.#........#.#.
.#.#.############.#.
.#.#..............#.
.#.################.
3#..................
2#..................
1#..................
```

Discussion:

The idea of this maze was to create one in which all 3 robots would back to move away from the goal in order to reach it. In addition, unreachable areas were added (partially in order to make the problem run faster). Similar to Maze 1, BFS and UCS visited about the same number of nodes, and the UCS solution was cost optimal (BFS was not). The manhattan heuristic significantly outpreformed the straight line heuristic which significantly ourpreformed the uninformed search methods. I found this result interesting. Perhaps it is because the way the straight line distance heuristic is calculated, it could encourage robots to fall down the "traps" to the right of their starting position, which manhattan distance could aviod that.

Blind Robot

This version of the Mazeworld problem is the same as the Single Robot version outlined above, except for that the robot is blind, and as such does not know where it starts (the robot still knows the map of the maze and which direction is which). Hence belief states are used to show all the possible locations a robot could be in at a certain point of time. Since the state is fundamentally different from the previous two problems, different heuristics must be used for the A* search. I decided to use the following heuristics:

null heuristic (`null_heuristic(state)`) - always returns 0

state length heuristic (`state_len_heuristic(state)`) - returns the length of given belief state

optimistic state length heuristic (`op_state_len_heuristic(state)`) - returns length of given belief state divided by `n` (width == depth), in order to force it to be optimistic.

unique x and y values heuristic (`uniq_x_y_heuristic(state)`) - returns the sum of the number of unique x values and number of unique y values in the given belief state

Testing

In order to test, the same maps were used as the ones for the Multi-Robot Coordination section. Below is discussion for each map. See Raw Output for their precise outputs.

Maze 1:

```
..#..
..#..
.....
..#..
..#..
```

Discussion:

This is the bottleneck map that I used for the multi robot problem. Both the BFS and UCS searached approximately 10,000 nodes (BFS slightly lower than UCS). They both found optimal paths with regard to length and cost. The state length heuristic vistied the fewest nodes (1.6k), but returned a solution path with a cost of 16. The optimized state length heuristic visited 8.7k nodes, but managed to return a solution path with optimal cost. Finally, the unique x y heuristic found a cost optimized solution path by only visiting 3.3k nodes.

Maze 2:

```
.......
.......
.......
.......
.......
.......
.......
```

Discussion:

This is the simple map I used in the single robot probem. This map seems boring, but the results are interesting. The BFS and UCS found a cost optimal solution after visiting about 19k nodes apiece (BFS marginally fewer nodes than UCS). The state length heuristic found a solution after visiting ony 205 nodes (!!!), but was not cost optimal. The optimized state length heuristic managed to find a cost optimal solution, but only after visting 13k nodes. Finally, the unique x y values heuristic found a solution that had a cost 1 higher than optial (better than state length solution), and only visited 513 nodes. It appears that in situations without many walls, the informed search really starts to shine in terms of nodes visited (although perhaps by sacrificing optimality).

Maze 3:

```
##.##
#...#
#.#.#
#...#
#...#
#.###
```

Discussion:

This was a map that was found on the website page for the assigment. The BFS and UCS both found cost optimal solutions after visting 4.3k and 4.5k nodes accordingly. The state length heuristic found a solution that had a cost 2 higher than optimal after only visiting 917 nodes, and the optimized state length heuristic found an optimal solution after visting 4.3k nodes (marginally fewer nodes than BFS search). The unique x y values

heurstic found a cost optimal solution, and only visited 345 nodes in doing so, making it far and away the king heuristic for this problem.

Conclusions:

The optimized state length heuristic usually not much faster than uninformed search methods (BFS and UCS), but it would always return a cost optimized solution. The state length and unique x y values heuristics were both significantly faster than the other search methods, but both came at the price of being optimistic. While both were not optimistic as shown above, the unique x y values heuristic appeared to be more optimistic than the state length heuristic, in that it found an optimal solution path more often than state length, and when it did not, the solution path found was closer to optimal than was the state length heuristic produced. Most of the time, the state length heuristic visited fewer nodes than the unique x y values heuristic (but not always, see Maze 3). From my tests, it seems like the unique x y heuristic occupies the middle ground of both visiting not very many nodes, and finding close to optimal solutions.

Polynomial-Time Blind Robot Planning

Whenever the robot in the Blind Robot problem moves to a new `x` value (a `x` value that has not previously been reached, not an `x` value that is being reached an additional time), the belief state of the problem decreases by a column. Similarly, when the robot move sto a new `y` value, an the belief state is decreased by a row. Since the robot occupies a space with one `x` and one `y` value, there are `n -1` new x values at the beginning, and `n-1` new y values at the beginning. Hence, there must be `2n-1` moves in order to make the belief state a singleton set.

Hence, a potential motion planner would be to move `n-1` moves North, then `n-1` moves East, then `n-1` moves South, then `n-1` moves West. Then repeat this process until the belief state is a singleton set. This amounts to `4(n - 1)` times some constant `k`. This is hence an algorithms that occurs in linear time.

Previous Work (undergraduate student)

For the previous work, I looked at the "Finding Optimal Solutions to Cooperative Pathfinding Problems" article by Trevor Standley. This article discusses how A* search can be an effective and fast algorithm for problems in which there is only one robot, but becomes much less effective when multiple robots are present. The solution presented uses A* search for each individual robot. Then, if any of the solution paths conflict, the algorithm will perform a sort of tie breaking to find some other path that is optimal for both robots that does not involve any collisions. If this is not possible, only then will the algorithm try to solve the problem with both states in consideration. The paper found that as the number of agents increased, the runtime of the proposed algorithm began to beat normal A* search (my algorithm) by more and more. The approach taken by this novel algorithm is intersting because it acknowledges a problem showed by my findings, which is that A* search becomes slower and slower as the number of robots increases. This algorithm limits the amount that the factoring in

multiple robots is used.

Further Exploration

As a further exploration, I tried two things. First, I created a second cost function for the `MazeworldProblem` object that was based on time spent rather than fuel expended. This change makes sense because fuel expended has no penalty for robots deciding not to move. That means that there could be an "optimal" path that has every robot start off by passing their first 100 turns. Which does not necessarily make sense. There is potential for basing cost off of time to result in the algorithm to be less fuel efficent, but it is still an interesting problem. To test this, I chose to use Maze 1 for the Multi Robot problem because the BFS generated an optimal solution with respect to solution length, and the UCS generated an optimal solution with respect to cost (based on fuel) originally. After using the modified cost function, they both BFS and UCS had the same cost. I am convinced the cost was correct because it was equal to the solution length minus one, which makes sense considering that the solution path includes both the start and goal states, and cost measures the turns taken (in this version).

Second, I created a comparison operator with a tiebreaker for the A* search algorithm. I chose to have the tiebreaker be the node that has the lower hueristic value, as that node is closer to the goal (if the heuristic is correct). This should generally result in fewer nodes visisted, but could potentially backfire if the heuristic is not accurate. I tested this on Maze 1 of the Multi Robot problem. I found this to be successful. Without tiebreaking, the null heuristic visited 77k nodes, and the manhattan distance heuristic visited 21k nodes. With tiebreaking, the null heuristic visited 32k nodes, and the manhattan heuristic visited 19k nodes. Perhaps the null heuristic was optimized by more because it potentially had more tiebreaks. Regardless, the tiebreaking appears to be an upgrade in terms of minimizing number of nodes visited. Yet while it appears to be an upgrade in number of nodes visited, both solutions found were significantly more costly than their non-tiebreaking counterparts. This makes me suspect that I have not completely implemented tiebreaking properly.

Note: I found that all of the raw output from the search algorithms was rather substantial, so in order to keep this report concise I dedided not to include them. Rather, there is a pdf file in this zip called `raw_output.pdf` which will have all of this information.