Multi-robot and Sensor-less Maze Solving Report

Ben Williams '25, September 22nd-27th, 2023

A-Star Search

The A-Star search begins by initializing an Astar node with the starting state. These Astar nodes wrap a state, its parent node on a path, its heuristic cost, and its transition cost. A visited-cost dictionary is also initialized, which will store the cost it took to visit a state - without counting the heuristic.

The first node is then pushed onto a priority queue. The priority of these Astar nodes is determined by adding a node's heuristic cost and its transition cost together.

We then loop through / pop from the priority queue either until it is empty (every possible state has been checked), or until the goal state has been found and we backchain to return the path. In the priority queue, we loop through all the successor states, first checking if there already exists a better cost to reach that state, and ignore that successor if so. Otherwise, we create a new Astar node for the successor, add its new visited cost to the dictionary, and add it to the priority queue.

We handle identical states appearing more than once in the priority queue by checking at the beginning if the Astar node's stored transition cost is greater than the cost stored in the visited-cost dictionary for that state. If it is, we ignore that node and continue on with the priority queue as it means that a better path to that state exists and has already been found.

Finally, there is an optional cost function for computing the cost of transitioning between two states. It takes the current and next states as parameters. This allows us to compute the fuel cost for the robots specifically, while keeping the Astar generalized. If a cost function is not provided, we assume the cost increases by one for each move.

Multi Robot Coordination

Discussion

Give an upper bound on the number of states in the system, in terms of n and k.

An upper bound for this would be $(n^{2)}k$, as there are approximately n^2 different possible locations for each of the k robots.

Give a rough estimate on how many of these states represent collisions if the number of wall squares is w, and n is much larger than k.

An upper bound to the number of possible states now excluding the wall collisions is ~(n^2 - w)^k, as there are now approximately n^2 - w possible locations for each robot. This does not, however, count collisions between robots.

Therefore, a good approximation for the number of collisions (or illegal states) is (n$^{2)}$k - (nˆ2 - w)ˆk.

If there are not many walls, n is large (say 100x100), and several robots (say 10), do you expect a straightforwards breadth-first search on the state space to be computationally feasible for all start and goal pairs? Why or why not?

It would not be feasible. Even if there are few walls, the BFS will likely still be searching an extremely large amount of states. Our upper bound from the previous question would say that there are approximately (10,000)ˆ10 different states which is infeasible to search. In fact, since there are fewer walls a manhattan-heuristic A-Star or Greedy search would perform even better than normal as the heuristic is more accurate.

Describe a useful, monotonic heuristic function for this search space. Show that your heuristic is monotonic. See the textbook for a formal definition of monotonic.

The Manhattan-distance is a pretty good heuristic for this space. Here is some psuedo code to calculate it:

```
manhattan_cost = 0
for each robot:
    x_difference = | robot_x - robot_goal_x |
    y_difference = | robot_y - robot_goal_y |
    manhattan_cost += (x_difference + y_difference)
```

The Manhattan Heuristic is monotonic, as if a robot gets one step closer to its goal location, then the Manhattan Heuristic will decrease by 1. However, because the cost of moving to that state is also 1, the Manhattan Heuristic is monotonic/consistent.

Describe why the 8-puzzle in the book is a special case of this problem. Is the heuristic function you chose a good one for the 8-puzzle?

The 8-puzzle can be thought of as 8 robots in a 3x3 maze, who each need to be lined up in the correct 1,2,3. . . 7,8 order.

It can be represented like this:

```
...
...
...
\robot 0 0
\robot 1 2
\robot 0 2
\robot 2 0
\robot 2 1
\robot 0 1
\robot 1 1
\robot 2 2
```

With the goal locations of (2, 0, 2, 1, 2, 2, 1, 0, 1, 1, 1, 2, 0, 0, 1, 0).

The Manhattan Heuristic is not bad at this problem. Experimentally, it had to search 7591 nodes, which is much better than the null-heuristic which had to search ~940K nodes.

The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this. (You do not have to implement this.)

The 8-puzzle has a set of states that are solvable, and a set of states that are un-solvable. One way to figure this out would be to perform a complete BFS going backwards from the goal state. Every state that is seen is part of the "solvable" set, while every state that is not seen would be in the "unsolvable" set. This BFS will find less than 9! total locations, which would mean there are some unreachable states from the goal.

Testing

I tested the Mazeworld algorithm on a few different mazes. The first was on maze3, which was given to us. Though not a super complicated maze, it is a good one to actually see the full path for:

```
Maze 3 with Manhattan Heuristic:
----
Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 244
solution length: 19
cost: 10
(Path excluded)

Start state:  (1, 0, 1, 1, 2, 1, 0)
Goal state:  (1, 4, 1, 3, 1, 2)
Mazeworld problem:
##.##
#...#
#.#.#
#...#
#BC.#
#A###

Mazeworld problem:
##.##
#...#
#.#.#
#...#
#BC.#
#A###
```

```
Mazeworld problem:
##.##
#...#
#.#.#
#B..#
#.C.#
#A###

Mazeworld problem:
##.##
#...#
#.#.#
#B..#
#.C.#
#A###

Mazeworld problem:
##.##
#...#
#.#.#
#B..#
#AC.#
#.###

Mazeworld problem:
##.##
#...#
#.#.#
#.B.#
#AC.#
#.###

Mazeworld problem:
##.##
#...#
#.#.#
#.B.#
#AC.#
#.###

Mazeworld problem:
##.##
#...#
#.#.#
#AB.#
```

```
#.C.#
#.###

Mazeworld problem:
##.##
#...#
#.#.#
#AB.#
#.C.#
#.###

Mazeworld problem:
##.##
#...#
#.#.#
#AB.#
#.C.#
#.###

Mazeworld problem:
##.##
#...#
#A#.#
#.B.#
#.C.#
#.###

Mazeworld problem:
##.##
#...#
#A#.#
#.B.#
#.C.#
#.###

Mazeworld problem:
##.##
#...#
#A#.#
#.B.#
#.C.#
#.###

Mazeworld problem:
##.##
#A..#
```

```
#.#.#
#.B.#
#.C.#
#.###

Mazeworld problem:
##.##
#A..#
#.#.#
#B..#
#.C.#
#.###

Mazeworld problem:
##.##
#A..#
#.#.#
#B..#
#C..#
#.###

Mazeworld problem:
##.##
#A..#
#.#.#
#B..#
#C..#
#.###

Mazeworld problem:
##.##
#A..#
#B#.#
#...#
#C..#
#.###

Mazeworld problem:
##.##
#A..#
#B#.#
#C..#
#...#
#.###
```

The other tests were more complex. robot_coordination (in /mazes) was a difficult one for the Manhattan and Null Heuristics. The maze looks simple, and a human can easily see the solution:

```
####.####
.........
\robot 0 0
\robot 1 0
\robot 2 0
```

With the goals at (8, 0, 7, 0, 6, 0), we have the results:

```
Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 1418
solution length: 49
cost: 34
(path excluded)

Mazeworld problem:
attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 2158
solution length: 49
cost: 34
(path excluded)
```

Finally, the 8 puzzle (which we already mentioned):

```
8-Puzzle with manhattan heuristic
----
Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 7591
solution length: 105
cost: 23
(path excluded)

8-Puzzle with null heuristic
----
Mazeworld problem:
attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 941183
solution length: 105
cost: 23
(path excluded)
```

Sensorless Robot Maze Solving

In the SensorlessProblem, the start state is simply every open floor space in the maze. The most interesting part of the implementation is the get_successors function. It takes a belief state as parameter, and loops through all possible actions in {N, E, S, W}. It then creates a set called "new_state", and loops through all locations in the belief state - attempting the action. If that action leads to an open space from the possible location, we add that space to the set, and if not, we add the old possible state to the set as we would have hit a wall. By using a set, we are able to ensure that duplicate states are just merged into one - reducing the overall size of the belief state. We finally convert it back into a tuple, and return it.

Heuristics

Describe what heuristic you used for the A search. Is the heuristic optimistic? Are there other heuristics you might use?*

The heuristic used for this search is the size of the belief state. A larger belief state would have a higher heuristic value, and therefore be valued less. This favored paths that decreased the size of the belief state the most.

It is not optimistic, though. A single move can remove several states from the belief state, rather than just one at a time. For example, in this simple maze with no walls:

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

At first, every open space is in the belief state, and the starting value of the heuristic would be 12 for all 12 floors. However, in only one move (North or South), we can reduce the belief state to only 6 floor spaces, far less than the cost of 12 that the heuristic had. Therefore, the value of 12 for the heuristic was not valid. The same could be said for moving East to start, reducing the belief state to size 10, and then moving North, reducing it to 5. The heuristic value of 10 was not an optimistic value for moving East to start.

As far as other heuristics go, one other possibility could have been to take the total distance between locations in the belief state. This would punish belief states who were made up of very far-apart possibilities that would take a long time to consolidate. However, calculating this would've taken much longer than the calculation for the size of the state space.

Testing

We test this approach on anything from maze1, maze2, and maze3 to more complex mazes in the *mazes* folder.

custom_maze1 has a very clear and intuitive solution. We can see the difference in the Astar versus the uniform-cost (null-heuristic) search here, as the null-heuristic has to search many more nodes in order to find the solution, while the state size heuristic finds an optimal solution quickly.

```
Testing empty maze w/ state_size_heuristic:
----
Blind robot problem:
attempted with search method Astar with heuristic state_size_heuristic
number of nodes visited: 21
solution length: 15
cost: 14
path = [((0, 0), (0, 1), (0, 2), (0, 3) ... , ((0,0),)

Testing empty maze w/ null_heuristic:
----
Blind robot problem:
attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 1096
solution length: 15
cost: 14
path = [((0, 0), (0, 1), (0, 2), (0, 3) ... , ((7,7),)
```

Then, we tested it on a larger but still not too complex maze on custom_maze2:

```
Blind robot problem:
attempted with search method Astar with heuristic state_size_heuristic
number of nodes visited: 16155
solution length: 20
cost: 19
path: [((0, 0), (0, 2), (0, 4), (0, 5) ... ((9,0),)
```

Note how much the number of nodes visited increased, despite there only being 40 floor spaces on it.

Finally, we tested on a maze meant to make this search algorithm struggle. Custom_maze3 is in the shape of an X:

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

This 10x9 maze did not finish in a reasonable amount of time (but it did finish quickly in the polynomial version!)

Polynomial Sensorless Maze Solving

First, we note that if the graph/maze is finite and all the nodes/floor spaces are in the same connected component, then there exists a solution for the sensorless problem.

Second, note that the size of the belief state can only decrease. This is illustrated in page 140 of the 3rd edition of the textbook (early on in section 4.4.1). This fact helped me think about and come up with the following algorithm, in psuedo-python code:

```
start_state = all floor spaces on the maze

current_state = start_state
belief_full_path = []

while current_state != goal_state # Just one belief
    to get the next state:
        location_1 = any belief/location in the current belief state
        location_2 = any belief/location != location_1 in the current belief state

        while location_1 != location_2:
            location_path = The path between location_1 and location_2

            # Find the sequence moves from {N, E, S, W} that the path takes
            moves = find_moves(location_path)

            # Perform the moves on the belief state, and track
            #   the path that the belief state takes
            # Adjust location_1 and location_2 in this function
            #   to wherever the moves would make a robot in location_1
            #   and location_2 end up
            new_state, belief_state_path = perform_moves(state, moves)

            # Permanantly log the path we have taken
            belief_full_path.extend(belief_state_path)

            current_state = new_state


        # Once location_1 and location_2 are equal,
        #   we must have eliminated at least one possibility from the
        #   belief state, so we take two new locations and repeat
        #   until there is only one state left

return belief_full_path # The search solution
```

Implementation details

We use *n* to refer to the total amount of floor-spaces in the maze.

The Search and Moving - Matching Location_1 and Location_2 We use a Bi-Directional BFS to find the location_path. Though the BFS will usually not have to (especially because it is bi-directional), it can search up to *n* nodes.

Then, once we find the path between location_1 and location_2, we perform the moves necessary to execute this path on **the entire belief state**. We also keep track of how location_1 and location_2 shift with the moves - as if there were a robot on each location and we wanted them to eventually end on the same spot.

After performing the moves on the belief state, it is possible (and may happen a few times) that location_1 != location_2 after the moves are performed. Nevertheless, they are closer together, since each move will either make location_1 closer to location_2, or they will stay the same distance apart. Since location_1 and location_2 can be at most *n* nodes apart, the moves will get them closer together at the end.

The following figure shows how we have to find a few paths between loc_1 and loc_2, and how the belief state changes. While the belief state shrinks significantly in this example, **it is only guaranteed to shrink by one**:

This can mean several BFS's per location-matching, but note that the BFS will look at far fewer nodes after the first time. In fact, the worst scenario would have the Bi-directional BFS look at *n* nodes the first time, then still have to look at ~ *n/2* nodes the second time, and decrease from there. We have A represent loc_1, and B represent loc_2:

(A screenshot is in ./figures in case it doesn't load right)

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

This type of maze tries to sync loc_2's movements up with loc_1's as much as possible. This means that there will be several searches, but note that these searches will be seeing much fewer nodes as they continue. With this, we argue

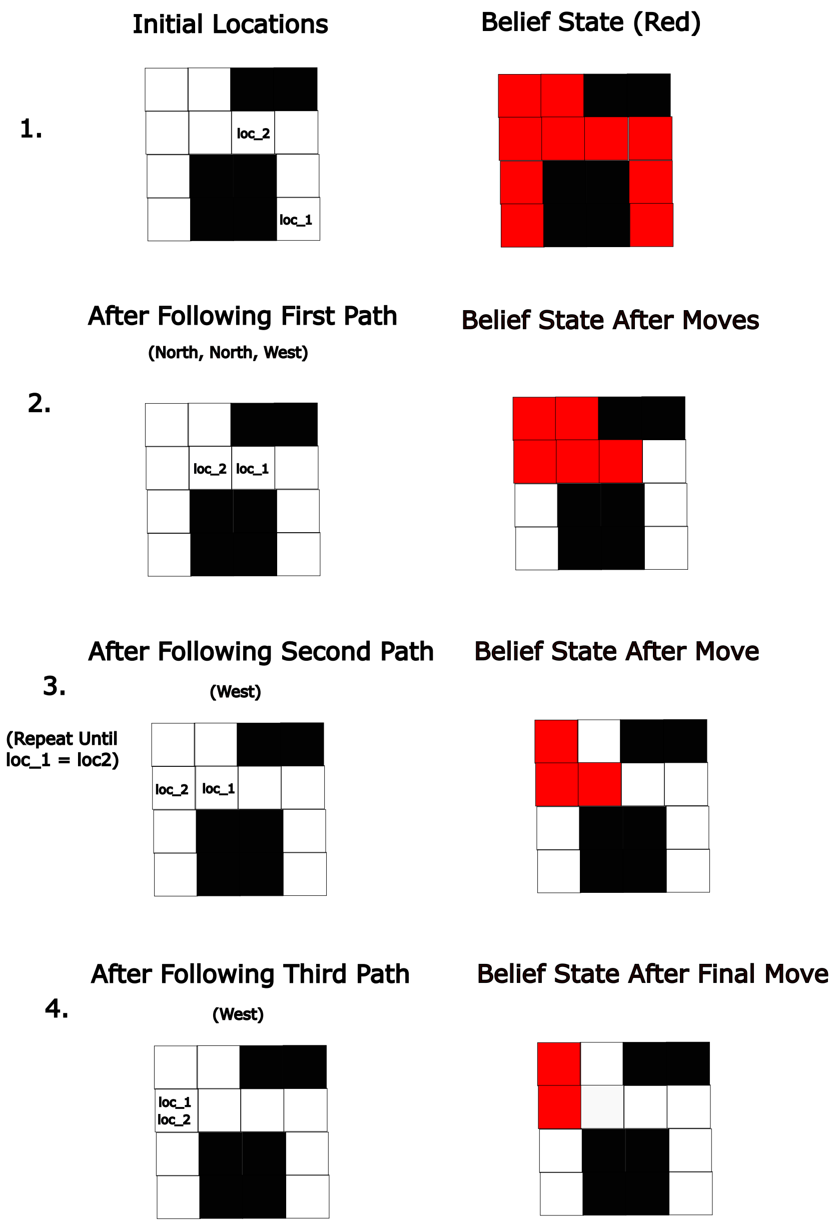

Figure 1: location matching

that the overall search cost for each transition is O(c * n) = O(n), even though we search a few times.

Then, by matching up loc__1 and loc__2, we ensure that **at least one** of the belief states is removed. In practice, we often remove much more especially at the beginning. Since we remove at least one belief state each time, and since we would need to do that at most *n* times for the starting number of belief states, we have an overall time complexity of O(n^2).

Again, in practice, this is often much better. Here is one example of a randomized 40x40 maze with ~40 walls. So, we have *n* = 360, and the following result:

```
Polynomial Time Blind Robot Problem:
attempted with search method Single-Possibility Removal through repeated BiDirectional BFS
number of nodes visited: 3654
solution length: 343
cost: 0
path: [((0, 0), (0, 1), (0, 2), (0, 3), (0, 4), ...
        ((0,0),)]
```

Where 3654 nodes is much less than n^2 = 129,600 nodes. In this case, it is much closer in scale to *n log n* (n log__2(n) ~ 3000), which is incredible. This may vary though as location__1 and location__2 are chosen at random from the belief state (see end of report)

Comparison to the standard Astar Search / SensorlessProblem

Just to see how much better this solution was, I wanted to compare the number of nodes visited on the normal SensorlessProblem compared to the SensorlessPolynomial problem. We refer to the names of the mazes in the mazes folder. The polynomial algorithm clearly **vastly** outperforms the normal search method.

small__maze: Nodes visited normally: 50

Nodes visited with polynomial algo: 28

custom__maze1 (empty maze): Nodes visited normally: 21

Nodes visited with polynomial algo: 46

I think it intuitively makes sense why the state__size__heuristic would outperform the algorithm in the empty maze case, as it bee-lines to the solution by design

custom__maze2 (10x6 maze): Nodes visited normally: 16155

Nodes visited with polynomial algo: 76

custom__maze3 (x maze): Nodes visited normally: Takes too long

Nodes visited with polynomial algo: 174

custom_maze4 (several x maze) Nodes visited normally: Takes too long

Nodes visited with polynomial algo: 828

random_maze1 (40x40 maze): Nodes visited normally: Takes too long

Nodes visited with polynomial algo: 3654

Other Details and Discussion

We do not use a wrapper node for the Bi-Directional BFS.

The get_successors() method of the SensorlessPolynomial is **not** meant for anything except finding the successors of a certain location (used for getting the neighbor floor spaces in the BFS). This was done so that the Bi-Directional BFS could be made problem-ambivalent, but the SensorlessPolynomial cannot be searched over normally.

When you animate the maze, you can pass in the move list to more easily see which move is made between states. Just uncomment the animate_path call in the bonus_test_polynomial_sensorless file.

Why Bi-Directional BFS?

Firstly, just to try and implement it as something new. Secondly, it was the simplest way to input two locations into a search algorithm. It is possible to use the other algorithms due to the modified get_successors and is_goal_state that are not meant to be used outside the get_next_state's search method.

We could use a slightly modified version of the Astar search that took a start state and end_state as parameter. It could help a bit, but the algorithm is already fast enough, and it likely wouldn't make a significant difference.

Choice of location_1 and location_2

At first, I just picked the first and last states in the belief state - especially since they were likely to be far away. However, picking the belief states for location_1 and location_2 was actually a lot faster. In fact, for the random maze, it searched only ~3500 nodes instead of ~7000. It will of course vary due to the randomness, so my numbers here are just from one trial.