COSC 76 Project 2: Mazeworld

Seungjae Jason Lee

Introduction

Objective

The objective of the project is to implement A* search to solve two problems: multi-robot problem and blind robot problem. There are three parts in this project: implement A* search, build game models(classes), and implement heuristics.

Game Rule

For multi-robot game, the objective is to find a path that leads all of the robots to their destinations without any collisions. As for the blind robot, the robot does not know where it is located. Thus, the objective of the game is to build a plan that will help robot locate itself.

A* Search

First, I made AstarNode class, which stores a state of a game, its parent node, transition cost, and heuristic value. Within the class, there are functions that computes the cost of a state and make it comparable for priority queue.

Then, I used priority queue to implement A* Search. As the algorithm traverses through a tree, it chooses the least-cost path by using min priority queue. Whenever a new state is chosen along the way, I used dictionary to store a state as a key and cost as a value. Whenever an existing state is revisited, it checks if the cost is larger than the one in the dictionary. If so, it ignores the state. If not, it updates the lower cost and pushes the state with lower cost value. Although same states may be stored inside priority queue with different costs and causes inefficiency, the algorithm still works as it ignores the state with higher cost value.

Multirobot Problem

These are methods of Multirobot problem that makes states to build a graph. There are more methods for heuristic and graphic, which will be explained later.

- init method takes maze object and tuple of goal_locations. Then, it extracts static information from the maze object and stores it inside the class.
- get_successors(self, state): returns a list of legal successors(tuple of locations) of the given state
- goal_test(self, state): checks if every robot is at the right goal. Returns True or False.
- get_cost(self, state1, state2) returns the cost of moving from state 1 to state2. state1 and state2 must be local states. My initial model returns 0 if there are no robot moves from state 1 to state2. However, I slightly modified it, which will be explained under extension.

1.State: In each state, we need to store two things, location of k robots and the turn of the robot. So, for the first slot of the tuple, I stored a number from 0 to k-1, which represents index of the moving robot. After the first slot, I store x and y coordinates of k robots. Thus, each state(tuple) would have 2k + 1 components.

2. Upper bound: Since there are k robots, the first component has k possible options. All of the other components have n options as the maze is n by n. Thus, the upper bound is k*n^(2k).

3.Number of collision states: We know that there are total n*n-w possible locations for robots. Using combination, we know that there are C(n^2-w, k) possible states without collsion. Thus, there are n^2 -w - C(n^2-w, k) possible states with collision.

4. Using bfs on big maze? It would be a bad approach to use bfs on a big maze with many robots. Since there are at most k*n^(2k) possible states, it would be inefficient to search through every possible state for big k and n.

5. Heuristic functions: In this game, I have two different heuristic functions. Firstly, I used sum of Manhattan distance as a heuristic function. It is monotonic as the actual cost will increase whenever a robot faces walls or faces colliding situation. Also, since the robot cannot move diagonally in a single turn, Manhattan distance represents minimal(optimal) path when none of the robots were colliding and faced walls on its way.

Secondly, I used wavefront bfs to give the actual cost from a location to the goal. So, I started from each of the goal locations and computed cost of reachable locations using bfs. Then, I made dictionaries that store locations and costs to goal for each of the robots. Then, for the heuristic function, I added all of the costs to reach goal from given state. This heuristic function is definitely monotonic as it just computes the actual lowest cost from a state to the goal.

6. Examples: I used provided examples and alos made two examples of my own. I made a nice graphic display of the maze as an extension (explained later) and my five maze demonstrations can be viewed by running test_mazeworld.py or by watching attached video. (demonstrated in following order given below)

1) maze2.maz: small map with 1 robot. Checks if the basic case works

2) maze1.maz: 5 robots with mixed order

3) 8puzzle.maz: demonstrates that 8 puzzle is just a special case of mazeworldproblem without any walls

4)maze3.maz: provided sample. 3 robots with mixed order

5) maze4.maz, 10 by 10 maze with three robots. bigger state space. Used to compute efficiency of different heuristics.

7. 8-Puzzle: 8 Puzzle is a special case of the maze problem. In 3*3 maze, there are 8 robots and 8 goals, which is equivalent to 8 tiles and 8 locations for the tiles. I used Manhattan Distance as my heuristic and it still worked as the rules are still same as mazeworld: robots cannot collide and robots cannot move diagonally. I demonstrated it on test_mazeworld.py

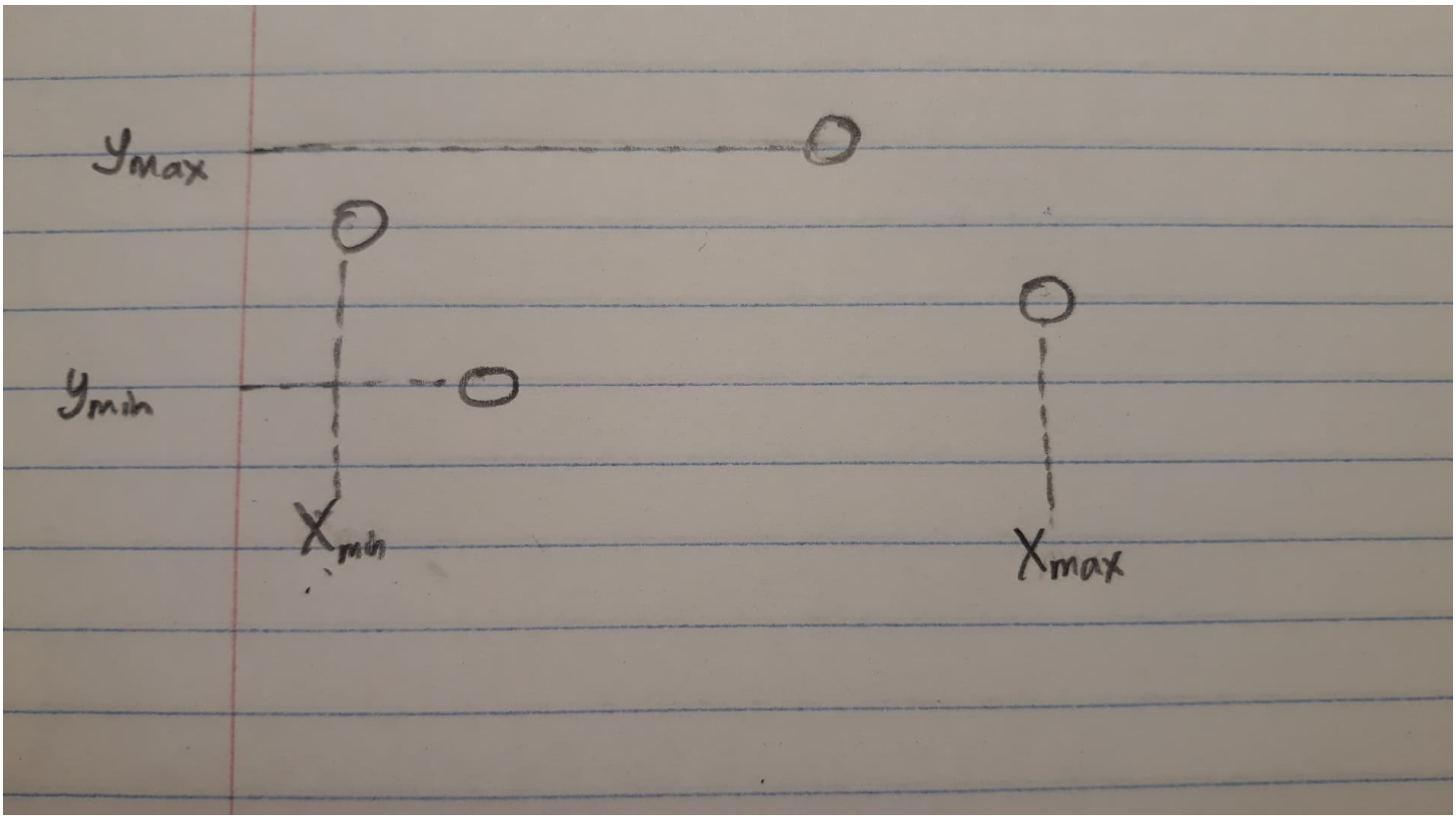
8. 8-puzzle state space: To make more efficient implementation of 8 puzzle, I would have two disjoint sets where one of them is movable puzzle pieces and the other one is immovable pieces. With this modification, it would avoid building unnecessary states.

Blind Robot

For Blind Robot, each state is a tuple with two components. First component is a tuple with belief states. Second component is a string that describes how the state is reached from the previous state. There are five possible strings: west, east, north, south and start. The class is implemented in SensorlessProblem.py.

- init method builds the blind robot object.
- get*successors returns a list of children(successors) of given belief state. I made build*successor as a helper function that builds a tuple of robot locations(new belief state) for a given belief state and direction.
- goal_test returns True if there is only one belief state. False otherwise.
- get_cost(state1, state2) takes two local states and returns 1.

Then, I made my own version of heuristic called sensorless_heuristic. It finds the max and min of x and y coordinate of belief state and computes the manhattan distance of it. (xmax-xmin+ymax-ymin)



Looking at the following image, we see that it takes ymax - ymin steps to have all of the locations on one y axis. Then, it takes xmax - xmin steps to gather all of the locations on one location. Since the following assumption holds when there is no wall on the way of robot locations, and walls would never reduce number of steps, we can say that the following heuristic is monotonic and is safe to use.

I used deque to implement queue and set to keep track of visited nodes in O(1) complexity. As for the backchaining, I used recursion to find the parents of nodes starting from the goal state to the root state.

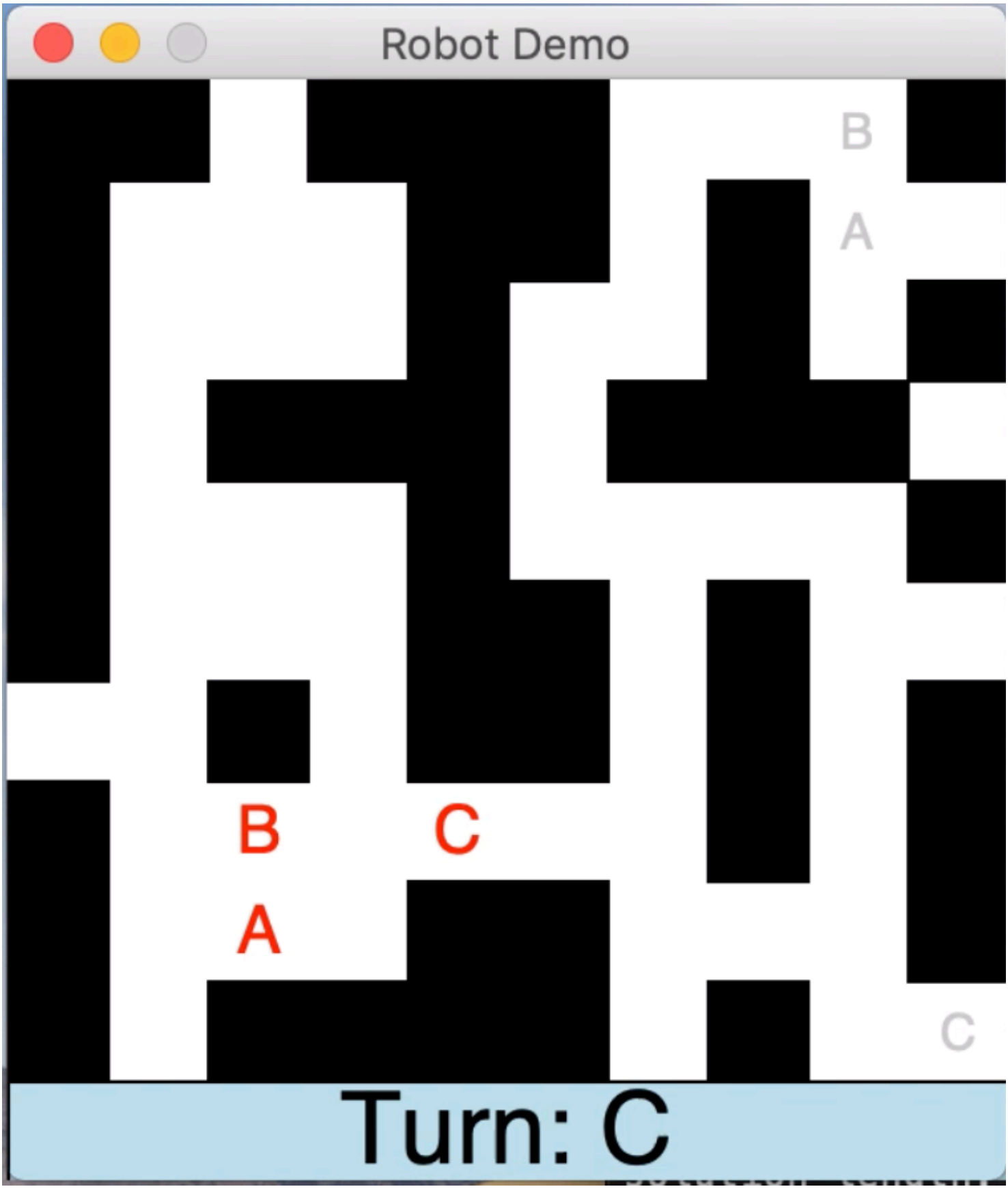
I tested blind*robot with null*heuristic and my heuristic. My own version of heuristic visited less number of nodes compared to null_heuristic and still gave lowest-cost path. The result of two different heuristic is shown below.

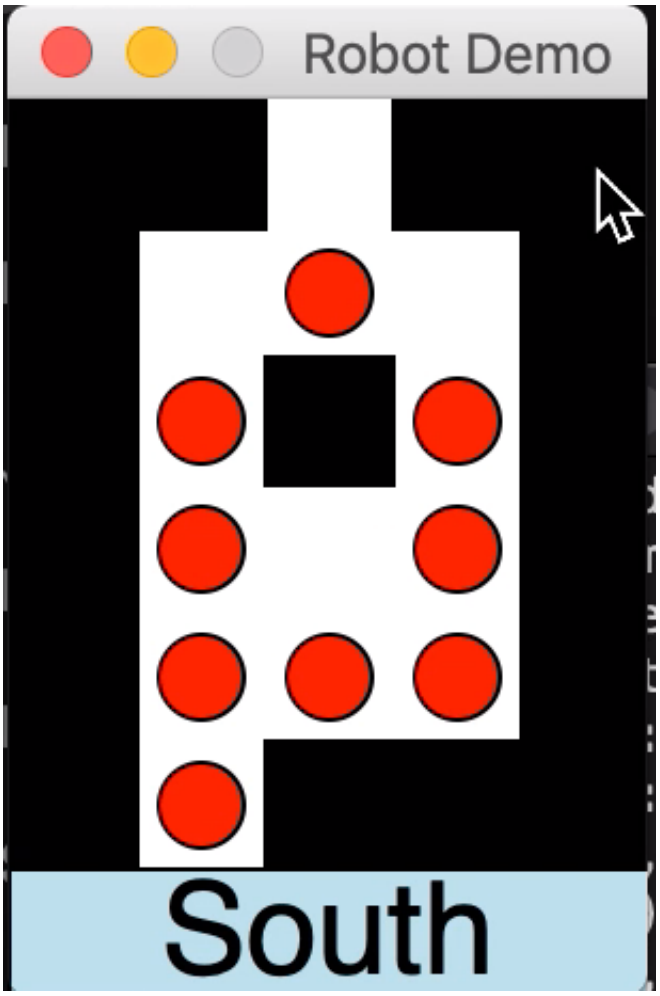
```
----
Blind robot problem:
attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 1366
solution length: 9
cost: 8
path: [((1, 2), (3, 2), (1, 3), (3, 3), (3, 1), (1, 4), (2, 1), (2, 4), (2, 5), (2, 2), (1, 0), (3, 4), (1, 1)), 'start'), (((1, 2), (3,
2), (1, 3), (3, 3), (1, 4), (2, 2), (2, 5), (3, 4), (1, 1)), 'North'), (((3, 2), (1, 3), (3, 3), (2, 1), (2, 2), (2, 5), (3, 4), (2, 4)),
'East'), (((1, 2), (3, 2), (3, 3), (3, 1), (2, 1), (2, 4)), 'South'), (((3, 2), (3, 3), (3, 1), (2, 2), (3, 4)), 'East'), (((3, 2), (3, 1)
, (3, 3), (2, 1)), 'South'), (((3, 2), (3, 1), (3, 3)), 'East'), (((3, 2), (3, 1)), 'South'), (((3, 1),), 'South')]

----
Blind robot problem:
attempted with search method Astar with heuristic sensorless_heuristic
number of nodes visited: 148
solution length: 9
cost: 8
path: [((1, 2), (3, 2), (1, 3), (3, 3), (3, 1), (1, 4), (2, 1), (2, 4), (2, 5), (2, 2), (1, 0), (3, 4), (1, 1)), 'start'), (((1, 2), (3,
2), (1, 3), (3, 3), (3, 1), (2, 1), (2, 4), (1, 0), (1, 1)), 'South'), (((1, 2), (1, 3), (3, 3), (2, 1), (1, 4), (2, 2), (1, 0), (1, 1)),
'West'), (((1, 2), (3, 2), (1, 3), (2, 1), (1, 0), (1, 1)), 'South'), (((1, 2), (1, 3), (2, 2), (1, 0), (1, 1)), 'West'), (((1, 2), (1, 0)
, (1, 1), (2, 1)), 'South'), (((1, 0), (1, 1), (2, 1)), 'South'), (((1, 0), (1, 1)), 'West'), (((1, 0),), 'South')]
```

Extension

Extension 1: Instead of printing text of every path, I made a nice graphic display of the maze path. For both sensorlessproblem and mazeworldproblem, I made a method called animate*path*extension. This method takes the path from SearchSolution object as an input and visualizes the path. To make this possible, I made a class called RobotDisplay that takes path and maze and converts them to a nice visual representation. Video demonstrations are can be found in the attached file.


Robot Demo
B
A
B
C
A
C
Turn: C



Extension 2: One of the issues with using number of steps to measure the cost is that the output may take lowest step but not lowest time. Thus, often robots just decides to stay at their position during their turns when they can actually move closer to their goals. To solve the problem, instead of returning 0 for the cost when the robot is not moving, I return 0.001 as the cost for non-moving robots. Then, I got shortest path and lowest cost as the output of the search.

```
def get_cost(self, state1, state2):
    step = 0
    for i in range(int(len(self.maze.robotloc))):
        step += abs(state1[i+1]-state2[i+1])
    if step*self.fuel == 0:
        return 0.001
    return step*self.fuel
```

Before Modification

```
----
Mazeworld problem:
attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 419647
solution length: 63
cost: 46
path: [(0, 1, 0, 1, 1, 2, 1), (1, 1, 0, 1, 1, 2, 1), (2, 1, 0, 1, 1, 2, 1), (0, 1, 0, 1, 1, 3, 1), (1, 1, 0, 1, 1, 3, 1), (2, 1, 0, 2, 1, 3, 1), (0, 1, 0, 2, 1, 3, 1), (1, 1, 1, 2, 1, 3, 1), (2, 1, 1, 2, 1, 3, 1), (0, 1, 1, 2, 1, 3, 1), (1, 1, 2, 2, 1, 3, 1), (2, 1, 2, 2, 1, 3, 1), (0, 1, 2, 2, 1, 3, 1), (1, 2, 2, 2, 1, 3, 1), (2, 2, 2, 2, 1, 3, 1), (0, 2, 2, 2, 1, 3, 1), (1, 3, 2, 2, 1, 3, 1), (2, 3, 2, 2, 2, 3, 1), (0, 3, 2, 2, 2, 3, 1), (1, 4, 2, 2, 2, 3, 1), (2, 4, 2, 3, 2, 3, 1), (0, 4, 2, 3, 2, 3, 1), (1, 5, 2, 3, 2, 3, 1), (2, 5, 2, 4, 2, 3, 1), (0, 5, 2, 4, 2, 3, 1), (1, 6, 2, 4, 2, 3, 1), (2, 6, 2, 4, 2, 3, 1), (0, 6, 2, 4, 2, 3, 1), (1, 6, 3, 4, 2, 3, 1), (2, 6, 3, 5, 2, 3, 1), (0, 6, 3, 5, 2, 3, 1), (1, 6, 4, 5, 2, 3, 1), (2, 6, 4, 6, 2, 3, 1), (0, 6, 4, 6, 2, 3, 1), (1, 6, 5, 6, 2, 3, 1), (2, 6, 5, 6, 3, 3, 1), (0, 6, 5, 6, 3, 3, 1), (1, 6, 5, 6, 3, 3, 1), (2, 6, 5, 6, 4, 3, 1), (0, 6, 5, 6, 4, 3, 2), (1, 6, 5, 6, 4, 3, 2), (2, 6, 5, 6, 4, 3, 2), (0, 6, 5, 6, 4, 4, 2), (1, 5, 5, 6, 4, 4, 2), (2, 5, 5, 6, 4, 4, 2), (0, 5, 5, 6, 4, 4, 2), (1, 5, 6, 6, 4, 4, 2), (2, 5, 6, 6, 5, 4, 2), (0, 5, 6, 6, 5, 4, 2), (1, 5, 7, 6, 5, 4, 2), (2, 5, 7, 5, 5, 4, 2), (0, 5, 7, 5, 5, 4, 2), (1, 6, 7, 5, 5, 4, 2), (2, 6, 7, 5, 6, 4, 2), (0, 6, 7, 5, 6, 4, 2), (1, 6, 8, 5, 6, 4, 2), (2, 6, 8, 5, 7, 4, 2), (0, 6, 8, 5, 7, 4, 2), (1, 6, 9, 5, 7, 4, 2), (2, 6, 9, 5, 7, 4, 2), (0, 6, 9, 5, 7, 5, 2), (1, 7, 9, 5, 7, 5, 2), (2, 7, 9, 5, 7, 5, 2), (0, 7, 9, 5, 7, 6, 2), (1, 7, 9, 5, 7, 6, 2), (2, 7, 9, 6, 7, 6, 2), (0, 7, 9, 6, 7, 6, 2), (1, 7, 9, 6, 7, 6, 2), (2, 7, 9, 6, 8, 6, 2), (0, 7, 9, 6, 8, 6, 2), (1, 8, 9, 6, 8, 6, 2), (2, 8, 9, 6, 8, 6, 2), (0, 8, 9, 6, 8, 6, 2), (1, 8, 8, 6, 8, 6, 2), (2, 8, 8, 6, 9, 6, 2), (0, 8, 8, 6, 9, 6, 2), (1, 8, 8, 6, 9, 6, 2), (2, 8, 8, 7, 9, 6, 2), (0, 8, 8, 7, 9, 6, 2), (1, 8, 8, 7, 9, 6, 2), (2, 8, 8, 8, 9, 6, 2), (0, 8, 8, 8, 9, 6, 1), (1, 8, 8, 8, 9, 6, 1), (2, 8, 8, 8, 9, 6, 1), (0, 8, 8, 8, 9, 7, 1), (1, 8, 8, 8, 9, 7, 1), (2, 8, 8, 8, 9, 7, 1), (0, 8, 8, 8, 9, 8, 1), (1, 8, 8, 8, 9, 8, 1), (2, 8, 8, 8, 9, 8, 1), (0, 8, 8, 8, 9, 8, 0), (1, 8, 8, 8, 9, 8, 0), (2, 8, 8, 8, 9, 8, 0), (0, 8, 8, 8, 9, 9, 0)]

(1, 0, 1, 1, 2, 1)
----
Mazeworld problem:
attempted with search method Astar with heuristic wavefront_bfs_heuristic
number of nodes visited: 25493
solution length: 90
cost: 46
path: [(0, 1, 0, 1, 1, 2, 1), (1, 1, 0, 1, 1, 2, 1), (2, 1, 0, 1, 2, 2, 1), (0, 1, 0, 1, 2, 3, 1), (1, 1, 1, 1, 2, 3, 1), (2, 1, 1, 1, 2, 3, 1), (0, 1, 1, 1, 2, 3, 1), (1, 2, 1, 1, 2, 3, 1), (2, 2, 1, 1, 2, 3, 1), (0, 2, 1, 1, 2, 3, 1), (1, 2, 2, 1, 2, 3, 1), (2, 2, 2, 1, 2, 3, 1), (0, 2, 2, 1, 2, 3, 1), (1, 3, 2, 1, 2, 3, 1), (2, 3, 2, 1, 2, 3, 1), (0, 3, 2, 1, 2, 3, 1), (1, 4, 2, 1, 2, 3, 1), (2, 4, 2, 1, 2, 3, 1), (0, 4, 2, 1, 2, 3, 1), (1, 5, 2, 1, 2, 3, 2), (2, 5, 2, 1, 2, 3, 2), (1, 6, 2, 1, 2, 3, 2), (2, 6, 2, 1, 2, 3, 2), (0, 6, 2, 1, 2, 4, 2), (1, 6, 2, 1, 2, 4, 2), (2, 6, 2, 1, 2, 4, 2), (0, 6, 2, 1, 2, 5, 2), (1, 6, 3, 1, 2, 5, 2), (2, 6, 3, 1, 2, 5, 2), (0, 6, 3, 1, 2, 5, 2), (1, 6, 4, 1, 2, 5, 2), (2, 6, 4, 2, 2, 5, 2), (0, 6, 4, 2, 2, 5, 2), (1, 6, 5, 2, 2, 5, 2), (2, 6, 5, 3, 2, 5, 2), (0, 6, 5, 3, 2, 5, 2), (1, 5, 5, 3, 2, 5, 2), (2, 5, 5, 4, 2, 5, 2), (0, 5, 5, 4, 2, 6, 2), (1, 5, 5, 4, 2, 6, 2), (2, 5, 5, 5, 2, 6, 1), (0, 5, 5, 5, 2, 6, 1), (1, 5, 5, 5, 2, 6, 1), (2, 5, 5, 5, 2, 6, 1), (0, 5, 5, 5, 2, 7, 1), (1, 5, 6, 5, 2, 7, 1), (2, 5, 6, 5, 2, 8, 1), (0, 5, 6, 5, 2, 8, 1), (1, 5, 7, 5, 2, 8, 1), (2, 5, 7, 5, 2, 8, 1), (0, 5, 7, 5, 2, 8, 0), (1, 6, 7, 5, 2, 8, 0), (2, 6, 7, 5, 2, 8, 0), (0, 6, 7, 5, 2, 9, 0), (1, 6, 8, 5, 2, 9, 0), (2, 6, 8, 6, 2, 9, 0), (0, 6, 8, 6, 2, 9, 0), (1, 6, 9, 6, 2, 9, 0), (2, 6, 9, 6, 3, 9, 0), (0, 6, 9, 6, 3, 9, 0), (1, 7, 9, 6, 3, 9, 0), (2, 7, 9, 6, 4, 9, 0), (0, 7, 9, 6, 4, 9, 0), (1, 7, 9, 6, 4, 9, 0), (2, 7, 9, 6, 5, 9, 0), (0, 7, 9, 6, 5, 9, 0), (1, 8, 9, 6, 5, 9, 0), (2, 8, 9, 5, 5, 9, 0), (0, 8, 9, 5, 5, 9, 0), (1, 8, 9, 5, 5, 9, 0), (2, 8, 9, 5, 6, 9, 0), (0, 8, 9, 5, 6, 9, 0), (1, 8, 9, 5, 6, 9, 0), (2, 8, 9, 5, 7, 9, 0), (0, 8, 9, 5, 7, 9, 0), (1, 8, 9, 5, 7, 9, 0), (2, 8, 9, 6, 7, 9, 0), (0, 8, 9, 6, 7, 9, 0), (1, 8, 8, 6, 7, 9, 0), (2, 8, 8, 6, 8, 9, 0), (0, 8, 8, 6, 8, 9, 0), (1, 8, 8, 6, 8, 9, 0), (2, 8, 8, 6, 9, 9, 0), (0, 8, 8, 6, 9, 9, 0), (1, 8, 8, 6, 9, 9, 0), (2, 8, 8, 7, 9, 9, 0), (0, 8, 8, 7, 9, 9, 0), (1, 8, 8, 7, 9, 9, 0), (2, 8, 8, 8, 9, 9, 0)]
```

After Modification

```
----
Mazeworld problem:
attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 462146
solution length: 60
cost: 46
path: [(0, 1, 0, 1, 1, 2, 1), (1, 1, 0, 1, 1, 2, 1), (2, 1, 0, 1, 2, 2, 1), (0, 1, 0, 1, 2, 3, 1), (1, 1, 1, 1, 2, 3, 1), (2, 1, 1, 1, 2, 3, 1), (0, 1, 1, 1, 2, 3, 2), (1, 2, 1, 1, 2, 3, 2), (2, 2, 1, 1, 2, 2, 3, 2), (0, 2, 1, 2, 2, 4, 2), (1, 3, 1, 2, 2, 4, 2), (2, 3, 1, 2, 2, 4, 2), (0, 3, 1, 2, 2, 4, 2), (1, 3, 2, 2, 2, 4, 2), (2, 3, 2, 2, 2, 4, 2), (0, 3, 2, 2, 2, 5, 2), (1, 4, 2, 2, 2, 5, 2), (2, 4, 2, 3, 2, 5, 2), (0, 4, 2, 3, 2, 6, 2), (1, 5, 2, 3, 2, 6, 2), (2, 5, 2, 4, 2, 6, 2), (0, 5, 2, 4, 2, 6, 1), (1, 6, 2, 4, 2, 6, 1), (2, 6, 2, 5, 2, 6, 1), (0, 6, 2, 5, 2, 6, 1), (1, 6, 3, 5, 2, 6, 1), (2, 6, 3, 6, 2, 6, 1), (0, 6, 3, 6, 2, 6, 1), (1, 6, 4, 6, 2, 6, 1), (2, 6, 4, 6, 3, 6, 1), (0, 6, 4, 6, 3, 6, 1), (1, 6, 5, 6, 3, 6, 1), (2, 6, 5, 6, 4, 6, 1), (0, 6, 5, 6, 4, 7, 1), (1, 5, 5, 6, 4, 7, 1), (2, 5, 5, 6, 5, 7, 1), (0, 5, 5, 6, 5, 8, 1), (1, 5, 6, 6, 5, 8, 1), (2, 5, 6, 5, 5, 8, 1), (0, 5, 6, 5, 5, 8, 0), (1, 5, 7, 5, 5, 8, 0), (2, 5, 7, 5, 6, 8, 0), (0, 5, 7, 5, 6, 8, 0), (1, 6, 7, 5, 6, 8, 0), (2, 6, 7, 5, 7, 8, 0), (0, 6, 7, 5, 7, 8, 0), (1, 6, 8, 5, 7, 8, 0), (2, 6, 8, 6, 7, 8, 0), (0, 6, 8, 6, 7, 8, 0), (1, 6, 9, 6, 7, 8, 0), (2, 6, 9, 6, 8, 8, 0), (0, 6, 9, 6, 8, 8, 0), (1, 7, 9, 6, 8, 8, 0), (2, 7, 9, 6, 9, 8, 0), (0, 7, 9, 6, 9, 8, 0), (1, 8, 9, 6, 9, 8, 0), (2, 8, 9, 7, 9, 8, 0), (0, 8, 9, 7, 9, 9, 0), (1, 8, 8, 7, 9, 9, 0), (2, 8, 8, 8, 9, 9, 0)]

----
Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 244151
solution length: 60
cost: 46
path: [(0, 1, 0, 1, 1, 2, 1), (1, 1, 0, 1, 1, 2, 1), (2, 1, 0, 1, 2, 2, 1), (0, 1, 0, 1, 2, 3, 1), (1, 1, 1, 1, 2, 3, 1), (2, 1, 1, 2, 2, 3, 1), (0, 1, 1, 2, 2, 3, 2), (1, 2, 1, 2, 2, 3, 2), (2, 2, 1, 2, 2, 4, 2), (0, 3, 1, 2, 2, 4, 2), (1, 3, 2, 2, 2, 4, 2), (2, 3, 2, 2, 2, 5, 2), (0, 3, 2, 2, 2, 6, 2), (1, 4, 2, 2, 2, 6, 2), (2, 4, 2, 3, 2, 6, 2), (0, 4, 2, 3, 2, 6, 2), (1, 5, 2, 3, 2, 6, 2), (2, 5, 2, 4, 2, 6, 2), (0, 5, 2, 4, 2, 6, 1), (1, 6, 2, 4, 2, 6, 1), (2, 6, 2, 5, 2, 6, 1), (0, 6, 2, 5, 2, 7, 1), (1, 6, 3, 5, 2, 8, 1), (2, 6, 3, 6, 2, 8, 1), (0, 6, 3, 6, 2, 8, 1), (1, 6, 4, 6, 2, 8, 1), (2, 6, 4, 6, 3, 8, 1), (0, 6, 4, 6, 3, 8, 0), (1, 6, 5, 6, 3, 8, 0), (2, 6, 5, 6, 4, 8, 0), (0, 6, 5, 6, 4, 9, 0), (1, 5, 5, 6, 4, 9, 0), (2, 5, 5, 6, 5, 9, 0), (0, 5, 5, 6, 5, 9, 0), (1, 5, 6, 6, 5, 9, 0), (2, 5, 6, 5, 5, 9, 0), (0, 5, 6, 5, 5, 9, 0), (1, 5, 7, 5, 5, 9, 0), (2, 5, 7, 5, 6, 9, 0), (0, 5, 7, 5, 6, 9, 0), (1, 6, 7, 5, 6, 9, 0), (2, 6, 7, 5, 7, 9, 0), (0, 6, 7, 5, 7, 9, 0), (1, 6, 8, 5, 7, 9, 0), (2, 6, 8, 6, 7, 9, 0), (0, 6, 8, 6, 7, 9, 0), (1, 6, 9, 6, 7, 9, 0), (2, 6, 9, 6, 8, 9, 0), (0, 6, 9, 6, 8, 9, 0), (1, 7, 9, 6, 8, 9, 0), (2, 7, 9, 6, 9, 9, 0), (0, 7, 9, 6, 9, 9, 0), (1, 8, 9, 6, 9, 9, 0), (2, 8, 9, 7, 9, 9, 0), (0, 8, 9, 7, 9, 9, 0), (1, 8, 8, 7, 9, 9, 0), (2, 8, 8, 8, 9, 9, 0)]

----
Mazeworld problem:
attempted with search method Astar with heuristic wavefront_bfs_heuristic
number of nodes visited: 6537
solution length: 60
cost: 46
path: [(0, 1, 0, 1, 1, 2, 1), (1, 1, 0, 1, 1, 2, 1), (2, 1, 0, 1, 2, 2, 1), (0, 1, 0, 1, 2, 3, 1), (1, 1, 1, 1, 2, 3, 1), (2, 1, 1, 2, 2, 3, 1), (0, 1, 1, 2, 2, 3, 2), (1, 2, 1, 2, 2, 3, 2), (2, 2, 1, 2, 2, 4, 2), (0, 3, 1, 2, 2, 4, 2), (1, 3, 2, 2, 2, 4, 2), (2, 3, 2, 2, 2, 5, 2), (0, 3, 2, 2, 2, 6, 2), (1, 4, 2, 2, 2, 6, 2), (2, 4, 2, 3, 2, 6, 2), (0, 4, 2, 3, 2, 6, 2), (1, 5, 2, 3, 2, 6, 2), (2, 5, 2, 4, 2, 6, 2), (0, 5, 2, 4, 2, 6, 1), (1, 6, 2, 4, 2, 6, 1), (2, 6, 2, 5, 2, 6, 1), (0, 6, 2, 5, 2, 7, 1), (1, 6, 3, 5, 2, 8, 1), (2, 6, 3, 6, 2, 8, 1), (0, 6, 3, 6, 2, 8, 1), (1, 6, 4, 6, 2, 8, 1), (2, 6, 4, 6, 3, 8, 1), (0, 6, 4, 6, 3, 8, 0), (1, 6, 5, 6, 3, 8, 0), (2, 6, 5, 6, 4, 8, 0), (0, 6, 5, 6, 4, 9, 0), (1, 5, 5, 6, 4, 9, 0), (2, 5, 5, 6, 5, 9, 0), (0, 5, 5, 6, 5, 9, 0), (1, 5, 6, 6, 5, 9, 0), (2, 5, 6, 5, 5, 9, 0), (0, 5, 6, 5, 5, 9, 0), (1, 5, 7, 5, 5, 9, 0), (2, 5, 7, 5, 6, 9, 0), (0, 5, 7, 5, 6, 9, 0), (1, 6, 7, 5, 6, 9, 0), (2, 6, 7, 5, 7, 9, 0), (0, 6, 7, 5, 7, 9, 0), (1, 6, 8, 5, 7, 9, 0), (2, 6, 8, 6, 7, 9, 0), (0, 6, 8, 6, 7, 9, 0), (1, 6, 9, 6, 7, 9, 0), (2, 6, 9, 6, 8, 9, 0), (0, 6, 9, 6, 8, 9, 0), (1, 7, 9, 6, 8, 9, 0), (2, 7, 9, 6, 9, 9, 0), (0, 7, 9, 6, 9, 9, 0), (1, 8, 9, 6, 9, 9, 0), (2, 8, 9, 7, 9, 9, 0), (0, 8, 9, 7, 9, 9, 0), (1, 8, 8, 7, 9, 9, 0), (2, 8, 8, 8, 9, 9, 0)]
```

Extension 3: I implemented wavefront*bfs in mazeworldproblem.py. It find the actual cost from the goal state to a given state, which can be used as a heuristic function. Although it took some computation to find all of the*

actual cost of each state, this heuristic turned out to be way fastest heuristic function because it reduced down number of nodes visited. An example is demonstrated in the picture above. Mannhattan Distance visited about 244000 nodes while wavefront bfs only visited 6537 nodes.

Extension 4: To make a better analysis, I also found computation time of three different heuristics. Using time class, I was able to find computation time for three different heuristic functions. I used maze4.maz since it has big state space and found out that wavefront is the fastest by alot.

```
time taken for null_heuristic(s): 34.0300989151001
----
Mazeworld problem:
attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 462146
solution length: 60
cost: 46
```

```
time taken for manhattan(s):  21.646970987319946
----
Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 244151
solution length: 60
cost: 46
```

```
time taken for wavefront bfs(s):  0.5391857624053955
----
Mazeworld problem:
attempted with search method Astar with heuristic wavefront_bfs_heuristic
number of nodes visited: 6537
solution length: 60
cost: 46
```