Mazeworld

Ping-Jung Liu September 26th 2017

1. Introduction

This project aimed to solve the multirobot with sensors case and the sensorless robot case of the traditional mazeworld problem. The general objective is to move all robots to their own goals without hitting the edges, the walls, and other robots.

The main algorithm implemented in the project is A* search. A* could solve both cases of the mazeworld problem because they can both be represented with well-organized state machines. I will show the advantages of A* by comparing it with uniform-cost-search and BFS. I also implemented a few extra credit features that could provide huge boosts to the performance of the required A* search, which will be discussed near the end of this report.

2. A* Search

The A* search is similar the BFS, but it keeps tracks of the cost and heuristic of each state to make better judgements instead of searching state after state. I used a python priority queue to record the toVisit states and a dictionary to save the lowest priority of each visited state.

The required heuristic for this assignment is the manhattan_heuristic. The heuristic of a state is sum of all manhattan distances of each robot from its goal.

The codes of A* start as follow:

```
while len(pqueue) > 0:

    # pop the first element in the pqueue
    current = heappop(pqueue)

    if s_p.goal_test(current.state):
        #.........................
    elif current state is visited but priority is higher:
        continue

    solution.nodes_visited = solution.nodes_visited + 1

    successors = s_p.get_successors(current.state)

    for child in successors:

        # initialize a new child node
        new_node = AstarNode(child[0], heuristic_fn(child[0]))
        new_node.heuristic = heuristic_fn(child[0])
        new_node.parent = current

        # change the cost
        # ...............

        # determine whether to add the child or not
        if child is not visited or priority is lower:
            heappush(pqueue, new_node)
            visited_cost[child[0]] = new_node.priority()

solution.path = []

return solution
```

Note this is a pseudo code version. The actual codes are well-commented in astar_search.py.

Just to briefly demonstrate the usage of A* search, I will show the results of A* with manhattan_heuristic, uniform-cost-search, and BFS solving the following maze.

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

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

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

The results are as follow (without the path for simplicity):

```
Mazeworld problem:
attempted with search method Astar with heuristic
null_heuristic
number of nodes visited: 58190
solution length: 33
cost: 27

Mazeworld problem:
attempted with search method Astar with heuristic
manhattan_heuristic
number of nodes visited: 4172
solution length: 71
cost: 27

Mazeworld problem:
attempted with search method BFS
number of nodes visited: 24955
solution length: 31
cost: 0
```

It is obvious that with a decent heuristic, the runtime of Astar with manhattan_heuristic is far lower than that of uniform cost search(null heuristic) and BFS.

3-1. Multirobot Discussion

In the multirobot case, the state is a tuple that contains a robot ID which indicates whose turn it is, and the locations of all robots. If there k robots in a maze, we would then need 2 * k + 1 numbers to represent a state. Robot IDs start from 0. For example the state (1, 0, 3, 2, 4) shows that robot 1 is moving next, robot 0 is at (0, 3), robot 1 is at (2, 4).

In a M * N maze with k robots, the # of possible location combination is M * N permutes k. There are also k turns for the robots, so the upperbound of number of states is then k * (M * N)!/(M * N - k)!; let this number be S.

If there are W walls, the # of possible location combination is then M * N - W permutes k. Therefore, the number of states with collision will be S - k * (M * N - W)!/(M * N - W - k)!. If M * N is too large, then the number of states with collision should be about S * (W/M * N)

If there are not many walls and n is large, say 100 * 100, and there are around 10 robots, the number of possible states would be close to 10^41. If the starting locations of all robots are far from their goals, it is computationally infeasible for a traditional BFS to search through all these nodes.

A decent and monotonic heuristic function is the manhattan heuristic function, which is the sum of the manhattan distance of each robot from its goal. Manhattan distance indicates the length of the shortest paths from a location to the goal, which is undoubtedly an underestimate of the actual steps required. Manhattan heuristic is monotonic because manhattan distance is always less than or equal to the actual steps required.

The 8 puzzle in the book is indeed a special case of the multi robot case because the objective is still to move each tile to their goal position, which is why the previous heuristic will still work; it will still be a underestimate of the cost.

We can use BFS to search one arbitrary start state, then pick a state that has not been visited to be the next start state. The resulting sets of the two searches should add up to 9!, which is the total number of state in the 8 puzzle problem.

8-puzzle

3.2 Multirobot Model

The model of multirobot problem is implemented in MAzeworldProblem.py. As discussed earlier, the state can be represented by 2 * k + 1 integers given k robots. The get_successors function can obtain the possible children of a given state and indicate whether the cost need to be increased, because a robot might not move. The goal test function compare state[1 : len(state)] with goal because the first element is a robot ID.

goal_test as follow:

```
def goal_test(self, state):
    return state[1 : len(state)] == self.goal_locations
```

get_successors as follow:

```
def get_successors(self, state):

    # actions available for robots
    action = [[0, 0], [1, 0], [0, 1], [-1, 0], [0, -1]]
    maze = self.maze
    maze.robotloc = list(state[1 : len(state)])

    # whos turn
    turn = state[0]

    # number of robot in the maze
    robotNum = len(maze.robotloc) / 2

    # change turn
    state = list(state)
    state[0] = int((state[0] + 1) % robotNum)
    state = tuple(state)

    successors = []

    # loop through all available actions
    for i in range(0, len(action)):

        # if new positions are legal, append the new state
        check if new_state is valid
            new_state = list(state)
            new_state[turn * 2 + 1] = new_state[turn * 2 + 1] + action[i][0]
            new_state[turn * 2 + 2] = new_state[turn * 2 + 2] + action[i][1]
            new_state = tuple(new_state)

            # the second element indicates whether the cost should be increased or not
            # 1 for yes, 0 for no
            successors.append((new_state, 1))

        elif i == 0:
            # append no action state, cost remains the same
            successors.append((state, 0))

    return successors
```

Below is one of the test case I used to show the functionality :

```
..#....
.##.#..
#.##...
.......
#.##...
#.###..
....##.
________________________________________________________
print(test_mp.get_successors((1, 0, 0, 0, 1, 3, 1)))
________________________________________________________
[((2, 0, 0, 0, 1, 3, 1), 0), ((2, 0, 0, 1, 1, 3, 1), 1)]
```

I created numerous cases to test the multirobot model and the algorithm. In the animation part, I will skip many steps orelse the report would be too long. Feel free to skip the animation if they are tedious. The program will show the actual path of the solution, but some of them are long and I will not post them in the report. The provided part should be enough to show the program works.

maze5.maz, a simple maze with one robot, goal = (4, 0):

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

attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 5
solution length: 6
cost: 5

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

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

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

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

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

maze6.maz, a slightly harder maze with 3 robots, goal = (1, 0, 3, 1, 2, 5):

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

attempted with search method Astar with heuristic manhattan_heuristic
```

```
number of nodes visited: 34
solution length: 22
cost: 8

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

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

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

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

maze2.maz, a more complicated maze with several unreachable positions, goal = (4, 2, 5, 1, 6, 0):

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

attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 4172
solution length: 71
cost: 27

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

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

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

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

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

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

maze1.maz, a huge 40*20 maze with one robot, goal = (32, 17):

```
........................................
........................................
....#########............#.......#......
....#....................#.......#......
....#....................#.......#......
....#########............#########......
........................................
........................................
........................................
........................................
........................................
........................................
........................................
....#########............##########.....
....#.......#............#..............
....#.......#............#..............
....#.......#............#..............
....#.......#............##########.....
........................................
........................................
\robot 8 4

Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 292
solution length: 44
cost: 43

No animation for this one because it's too large.
```

maze3.maz, goal = (0, 4, 1, 4, 2, 4), a tricky corridor that requires the robots to switch order:

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

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

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

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

Mazeworld problem:
ABC....
.#####.
.#...#.
.#.#...
.#.#.#.

They managed to switch the order.
```

4. Blind Robot

In this version of mazeworld. There is only one robot, but the robot is blind. It does not know where it is and has no idea when it hits a wall or edge. I decided to represent the state of this problem with two elements. The first is a list of possible locations of the robot, the second is the action the robot used to get here from parent. The possible locations of the start state is simple all locations in the maze that are not walls.

The main objective in this game is to shrink the possible locations to a set containing only the goal, which means the robot is sure it is at the goal. For example, given a starting location set ((0, 0), (1, 0), (2, 0)), if the robot move right, the set will become ((1, 0), (2, 0)) because it's impossible for the robot to be at the left most column if it did not hit a wall.

get_successors as follow:

```
def get_successors(self, state):

    # no point for the robot to stay in one robot case
    action = [[1, 0], [0, 1], [-1, 0], [0, -1]]
    maze = self.maze

    successors = []

    for i in range(0, len(action)):
        new_state = []

        # modify every possible positions
        for j in range(0, len(state[0])):
            newX = state[0][j][0] + action[i][0]
            newY = state[0][j][1] + action[i][1]

            # append the new position only if it's legal
            if maze.is_floor(newX, newY):
                new_state.append((newX, newY))
            # else append original position
            else:
                new_state.append((state[0][j]))

        # set() can eliminate duplicates in a list or tuple
        # make sure the new_state is different from original state
        if len(new_state) > 0 and not set(new_state) == set(state[0]):
            new_state = list(set(new_state))

            # note that the state has two elements: all locations, and action
            # the 1 indicates cost should be increased
            successors.append(((tuple(new_state), tuple(action[i])), 1))

    successors = list(set(successors))
    return successors
```

I tested this function with a simple test case as follow:

```
...
.#.

((((0, 1), (2, 0), (0, 0), (1, 1)), (-1, 0)), 1)
((((0, 1), (1, 1), (2, 1)), (0, 1)), 1)
((((2, 0), (0, 0), (1, 1), (2, 1)), (1, 0)), 1)
((((2, 0), (0, 0), (1, 1)), (0, -1)), 1)
```

The 1s at the end of each result indicates the cost should be increased. The second element of the states are actions.

I used the number of possible locations as the required heuristic for this part; though it is not really a good estimate, it is extremely easy to obtain and still provide an ok estimate. Additional heuristic will be discussed in the extra credit sections.

I also tested the blind robot with a few test cases.

maze4.maz, an extremely simple maze to show the program works, goal (2, 0):

```
...
.#.

attempted with search method Astar with heuristic num_heuristic
number of nodes visited: 8
solution length: 4
cost: 4
path: [(0, 1), (1, 0), (1, 0), (0, -1)]
# ALL PASS shows that the solution can work for every possible start_state
ALL PASS

# the possible positions of robot
((0, 0), (2, 0), (0, 1), (1, 1), (2, 1))
Blind robot problem:
...
A#.

((0, 1), (1, 1), (2, 1))
Blind robot problem:
A..
.#.

((1, 1), (2, 1))
Blind robot problem:
.A.
.#.
```

```
((2, 1),)
Blind robot problem:
..A
.#.

((2, 0),)
Blind robot problem:
...
.#A
```

maze5.maz, a normal maze with some complicated structure for a blind robot, goal (4, 3):

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

attempted with search method Astar with heuristic num_heuristic
number of nodes visited: 1746
solution length: 11
cost: 11
path: [(0, 1), (1, 0), (0, 1), (-1, 0), (0, 1), (0, 1), (1, 0), (0, 1), (0, -1), (1, 0), (1, 0)]

ALL PASS

# only part of the animation
# no possible positions because the page would be too messy
Blind robot problem:
##.#.
#....
#.#..
...#.
A#...

Blind robot problem:
##.#.
#....
#.#..
A..#.
.#...

Blind robot problem:
##.#.
#....
#A#..
...#.
.#...

Blind robot problem:
##.#.
#A...
#.#..
...#.
.#...
```

maze3.maz, again the corridor problem, goal (0, 4):

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

attempted with search method Astar with heuristic num_heuristic
```

```
number of nodes visited: 46298
solution length: 19
cost: 19
path: [(0, -1), (0, -1), (1, 0), (0, 1), (0, 1), (1, 0), (1, 0), (0, -1), (1, 0), (1, 0), (0, 1), (0, 1)
       (0, 1), (-1, 0), (-1, 0), (-1, 0), (-1, 0), (-1, 0), (-1, 0)]

ALL PASS

Blind robot problem:
.......
.#####.
.#...#.
.#.#...
.#A#.#.

Blind robot problem:
.......
.#####.
.#A..#.
.#.#...
.#.#.#.

Blind robot problem:
.......
.#####.
.#...#.
.#.#A..
.#.#.#.

Blind robot problem:
.......
.#####.
.#...#A
.#.#...
.#.#.#.

Blind robot problem:
A......
.#####.
.#...#.
.#.#...
.#.#.#.
```

Some of the larger mazes took a lot longer so I decided not to include them here.

5.1 Extra Credit: BFS length as heuristic:

For the multi robot case, I wrote another heuristic that basically treat each robot as individual and perform BFS on each fo them. The heuristic is the sum of the length of these solutions. Though the runtime is undoubtedly higher, but the search space is only 2D, which is acceptable because the dimension of all locations is 6D with 3 robots.

bfs heuristic code as follow:

```
def bfs_heuristic(self, state):
    maze = self.maze
    start_state = maze.start_state
    robotloc = maze.robotloc
    heuristic = 0
    robotNum = int(len(state) / 2)

    individual_loc = []

    for i in range(0, robotNum):
        individual_loc.append((state[i * 2 + 1], state[i * 2 + 2]))

    # perform bfs on each location
    for i in range(0, len(individual_loc)):

        start = individual_loc[i]
        maze.robotloc = start
        maze.start_state = (0, start[0], start[1])

        new_mp = MazeworldProblem(maze, (self.goal_locations[i * 2], self.goal_locations[i * 2 + 1]))
        result = bfs_search(new_mp)
        heuristic = heuristic + len(result.path) - 1

    maze.robotloc = robotloc
    maze.start_state = start_state

    return heuristic
```

I will compare the results using null heuristic, manhattan, and this BFS heuristic on maze3.maz:

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

attempted with search method Astar with heuristic null_heuristic
number of nodes visited: 30522
solution length: 55
cost: 46

attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 17100
solution length: 88
cost: 46

attempted with search method Astar with heuristic bfs_heuristic
number of nodes visited: 7191
solution length: 55
cost: 46
```

It is obvious that bfs heuristic provides tremendous decrease in number of nodes visited.

5.2 Extra Credit: Spam Heuristic:

I implemented an additional heuristic for the blind robot case. Given the set of all possible locations, the heuristic function will find the range of the Xs and Ys then add them up to create the heuristic. For example ((0, 0), (1, 0), (2, 0)) will have a heuristic of 2 + 0 = 2.

Spam Heuristic code as follow:

```
def spam_heuristic(self, state):
    xs = []
    ys = []

    for i in range(0, len(state[0])):
        xs.append(state[0][i][0])
        ys.append(state[0][i][1])

    x_dist = max(xs) - min(xs)
    y_dist = max(ys) - min(ys)

    return x_dist + y_dist
```

I will test the spam heuristic on maze5.py, goal (4, 0):

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

Blind robot problem:
attempted with search method Astar with heuristic num_heuristic
number of nodes visited: 229
solution length: 10
cost: 10

Blind robot problem:
attempted with search method Astar with heuristic spam_heuristic
number of nodes visited: 17
solution length: 9
cost: 9
```

The new heuristic is not only as fast, but again largely decrease the number of visited nodes and even find better plannings.

5.3 Extra Credit: Same Priority Condition:

In this modification, I changed the **LT** function so that when two states have the same priority, compare their heuristic instead. Given the setup of the maze, it is very likely that many states would have same priority.

```
if self.priority() < other.priority():
        return True
elif self.priority() > other.priority():
    return False
else:
    return self.heuristic < other.heuristic
```

I will show the results of the original blind heuristic running on maze5.py again:

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

# original __LT__
Blind robot problem:
attempted with search method Astar with heuristic num_heuristic
number of nodes visited: 2302
solution length: 12
cost: 12

# modified __LT__
Blind robot problem:
attempted with search method Astar with heuristic num_heuristic
number of nodes visited: 1849
solution length: 12
cost: 12
```

The modification also save a decent amount of runtime.

5.4 Polynomial time blind robot

Suppose the maze is N * N and all locations are connected. The robot can then access all possible x and y values, excluding the starting position, and eliminate enough rows and columns that only one location is left in the set.

The robot can simply perform a left-hand-rule search to run through the whole maze, in which the robot stick to the left walls no matter what. The worst condition will be the robot making N^2 moves, which is still polynomial time.