report.md

MazeWorld

Assignment 2
CS 276, Artificial Intelligence
Fall 2021, Dartmouth College
Spencer Bertsch

Description

1. How do your implemented algorithms work?

Multi-Robot Coordination Problem

The multi-robot coordination problem is a problem in which more than one (k) robots are initialized in a maze and each is trying to get to a given goal. Each of the robots can't travel over a wall or over another robot. In order to solve this using A* we needed to implement a heuristic function *h(n)* which found the best position for robot k to optimize not only that robot's position relative to its goal, but the overall location of all robots and all goals. For this reason, the heuristic used was the mean heuristic for each robot in the maze; my implementation simply used the mean Manhattan heuristic for each robot in the maze.

The other important aspect of this implementation was the *get_successors* function. This function allowed us to take a state representing each robot's position, and which robot was going to move next, and let us iteratively update the correct robot's position. The state in this problem is represented by an odd-length tuple starting with an integer representing which of the robots should be moved next, and then pairs of (X,Y) coordinates representing the positions of each respective robot.

The actual A* algorithm was also an important part of the implementation. As we learned in class and read in the textbook, A* functions by taking both the heuristic *h(n)* and the cost *c* into account. For the multi-robot search problem, the robots burned one unit of gas if they moved, and zero units of gas if they stayed still. This transition cost, along with the mean Manhattan heuristic, were the components that allowed our A* to solve this problem.

Sensorless Robot Problem (Pacman Physics)

The Sensorless Robot Search problem is an interesting problem with a different heuristic *h(n)* and a different *get_successors* function than the multi-robot search proboem seen above. For the sensorless problem, the state was simply represented as a tuple of tuples, the latter of which represent all of the (X,Y) coordinates that the robot could be in. As A* searched through the state space moving the robot North, East, South, or West, the length of the state shrank accordingly as spaces were removed from possibility. The appropriate heuristic, therefore, was to examine the number of (X,Y) positions in the state. The fewer the positions in the state, the closer we are to the goal.

The *get_successors* function was also an important aspect of the sensorless-robot problem. As mentioned, the state exists as a tuple of 2-length tuples, each of which represent the possible (X,Y) coordinates of the robot. The get_successors function is divided into four parts, one for North, one for East, one for South, and one for West. At any given time, the successor state would be all the possible places that the robot could travel if it moves in any one of those four directions. If the robot was in an illegal space after it moved, then the *get_successors* function would keep the initial state in the successors, but if the robot was in a legal space after it moved, then the get successors function would remove the initial (X,Y) coordinate from the set of possible successors. This process was repeated through A* until only one space was left. As A* searched, the heuristic pushed the search into the direction that minimized the number of (X,Y) coordinates still in the state until it hit a state with only one remaining (X,Y) coordinate, at which time the search terminated and the path was returned.

2. What design decisions did you make?

We were given skeleton code for the problem set, but there were still several design decisions that helped me complete the project. One thing was to create a similar copy of the A* search algorithm specifically for the sensorless robot problem. I initially started using a single A* function and passing in a `sensorless` bool which could be used as a switch to activate several `if sensorless: ...do something` logical components in the A* function. After starting down this road, I realized there were so many changes that needed to be made to the A* algorithm that it would be easier to create a copy that was tailored specifically for the sensorless-robot problem.

One interesting addition to the SensorlessAStarNode class was the addition of the path instance variable so that we could store the directions (North, East, South, West) that the robot travelled so that it could be sure of where it was in the maze. This was the path that was printed at the end after the SensorlessSearch finished.

3. How you laid out the problems?

I laid out the problems by structuring most of the Sensorless code in the SensorlessProblem.py file, and most of the multi-robot search code in the astar_search.py and MazeworldProblem.py files. As seen in the README.md file which is also in the MazeWorld directory, the two frameworks can be tested by running either the test_mazeworld.py file as a script, or running the test_sensorless.py file as a script. The heuristics are also defined in these two files.

Evaluation

Multi-Robot Coordination Problem

The multi-robot coordination proboem is working as expected. The robots seem to work together, moving out of tunnels and reorienting themselves in the correct way before re-entering tight spaces so that they can all reach their intended targets. See below for a somewhat long example of how three robots (A, B, and C) coordinate to orient themselves so that they can enter the tunnel in the correct allignment.

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

...

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

There were two real keys to this problem: understanding that the heuristic should apply to the robots as a whole, and the get_successors function should store the information regarding which robot's turn it is to move. See the **Appendix** at the end of this report to see a full multi-robot search spet-by-step.

Blind Robot Evaluation

After much experimentation with the heuristic function and implementation, the blind robot functionality essentially works as expected.

Let's imagine a trivial maze with no blocks and only floor space. The robot could reduce the state space by either a row or column by simply moving either north or south, then moving either east or west until there is only one remaining space left. That is the result we see when running test_sensorless.py on the trivial sensorless maze:

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

```
....
```

And we can see that we get the expected output below. Note that the solution length is one larger than it should be because the initial SensorlessNode was initialized with a direction of None.

```
Solution found! Path to solution: ['North', 'North', 'East', 'East', 'East']
SOLUTION PATH LENGTH: 5
```

The solutions for nontrivial sensorless problems always seem to be missing the last direction in the path. (See below). It looks like the sensorless A* search always converges on the correct solution, but it doesn't print the last move (direction) in the path. I'm still trying to figure out why!

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

This input, as seen from our quiz a few days ago, renders the following output:

```
Solution found! Path to solution: ['North', 'East']
SOLUTION PATH LENGTH: 2
```

We know that the real answer is [West, North, West, West], and a sub optimal solution could be [North, East, East, South, West], so the algorithm is going in the correct direction and it capable of reducing the state space to a single coordinate, but it doesn't track the path in the correct way. I'm still looking into this issue.

Discussion Questions Responses

Multi-robot coordination problem

1. Representing the state of the system correctly is one of the most important aspects of getting this problem right. The state of the system requires (2 * k robots) + 1 integers, or (2 * k robots) + 1 pieces of information about the state. This is clearly an odd number, so the smallest state would consist of 3 integers: (robot_to_move, X_position, Y_position). In this case there is only a single robot, so the first integer would always be zero. As the number of robots in the game increases, then we would increase the number of ints in the state from 3 to 5, then to 7, then to 9 and so on. Each robot adds an additional X, Y coordinate to the state and an additional increment to the first int in the state telling the A* algorithm which robot's turn it is to move next. Structuring the state in this way is one of the keys of setting up the multi-robot coordination problem correctly.

2. An upper bound on the number of states in the system can be found by thinking about the way we represented the state. We represent the state with a tuple (k, m, m, ...) for a square maze of length m and k robots where the first robot is represented by 0. The upper bound on the size of the state would be m*m*(k+1) if k starts with 0 as represented in my implementation, or (m*m*k).

3. As the number of wall squares increases, collisions will increase as well. Boxing the robots in so that they are stuck in hallways or paths will force robots to collide and potentially back out of a hallway in order to reorient themselves to reach the desired goal. In terms of an expression, I would probably need to run a simulation in which I generate random, large mazes with differing wall densities and measure the number of robot-robot collisions that take place during the search for each maze. Without doing this, I could say that the number of collisions can be expressed by *collisions = (alpha(w)/n) + epsilon* for some small constant epsilon and a scaling factor alpha. Epsilon and alpha in this case would depend on *n*, and the larger *n* is the smaller epsilon would be. Similarly, we include an *n* term when we consider *w* so that we can get the relative density of the maze instead of simple the number of walls.

4. Good question, I think that BFS would probably not be a good solution here because it would likely cause the machine to run out of memory. We know that if n is 100, there are very few walls, and k=10, then each iteration will likely present a very large state space. We know that BFS holds all previous states in memory, giving it a space complexity of O(b^d). With 10 robots capable of moving North, South, East, or West, the branching factor could be as large as 40, and the depth of the shallowest solution (in the worst case) would be 99 moves north or south, and 99 moves east or west for a single robot, which makes depth = 198. This means that the space complexity for this problem would be on the order of 1.6*10^317 bytes assuming 1000 bytes per node in the BFS graph.

5. In addition to understanding how to represent the state space for the problem, understanding the right way to represent the heuristic is for multi-robot search is a key to success. There are many admissible or consistent heuristic that can be used for a simple two-dimensional puzzle such as this, but an example of a monotonic (consistent) heuristic that I used was the mean manhattan distance for all robots to all goals. This was crucial for success in the multi-robot search problem: the heuristic needs to reflect all robots and all goals, not an iteration of each robot and its respective goal! Remember that monotonicity or consistency just means that the heuristic at any node *n* is not greater than the heuristic at the next node *n'* plus the cost of reaching *n'*.

$$h(n) \leq h(n') + c$$

5. (Continued) The mean manhattan distance for all robots is monotonic because the fuel needed to move one square is 1. In the worst case, even if the robot is moving in the exact opposite direction from its goal (which does sometimes happen), consistency is held because the cost would be 1 and the new heuristic *h(n') would increase by 1, allowing the inequality to hold strongly. For the mean manhattan heuristic, and the manhattan, heuristic in general, moving laterally also yields the same result in which the cost increase is added to the additional heuristic, allowing the inequality to strongly hold. In the case where the robot is moving directly towards its goal, then the *h(n')* term would be one less than *h(n)*, but the cost *c* in this case would be exactly 1, allowing the inequality to hold.

6. The 8-puzzle in the book is a special case of this problem because each numbered square in the 8-puzze can be treated as a robot! Each numbered square has a starting state and a goal state and each needs to work together to get into the goal state. I think that A* and our implementation could be used to implement the 8-puzzle. As discussed in class, a good heuristic to use would be the overall manhattan distance for the all game pieces and all goal states, which is the exact heuristic that I implemented for the multi-robot search problem.

7. The 8-puzzle can be represented by two mutually disjoint sets which can be represented by robots that can move and robots that cannot move at any given time. The two sets can be thought of the robots that can move (robots that are not boxed in), and the robots that can't move (robots that are boxed in). The robots that "can't" move have the only option of staying in place and burning no gas in the process. The robots that can move will have more than the trivial child state. We could easily modify our program to print the disjoint sets on every iteration of the *get_successors* function in order to simply observe which robots are capable of only staying in the same place, and which robots can move towards the goal in order to improve their heuristic.

APPENDIX

Multi-Robot Search Complete Example

Please see below for a complete example of a multi-robot search problem in which all three robots needed to shift their orientation in order to achieve the goal state.

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

MazeWorld problem:

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

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

MazeWorld problem:

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

MazeWorld problem:

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

MazeWorld problem:

#######
#.#####
#.#####
#C#####
#AB...#
#.#####

MazeWorld problem:

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

MazeWorld problem:

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

MazeWorld problem:

#######
#.#####
#.#####
#C#####
#.AB..#
#.#####

MazeWorld problem:

#######
#.#####
#.#####
#C#####
#.AB..#
#.#####

MazeWorld problem:
```

```
#######
#.#####
#.#####
#.#####
#CAB..#
#.#####

MazeWorld problem:

#######
#.#####
#.#####
#.#####
#CAB..#
#.#####

MazeWorld problem:

#######
#.#####
#.#####
#.#####
#CAB..#
#.#####

MazeWorld problem:

#######
#.#####
#.#####
#.#####
#.AB..#
#C#####

MazeWorld problem:

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