COSC76, 2021 Fall Term, PA2, Kunal Jha

Implementation

astar_search.py

AstarNode class

This class would be used within the priority queue for A* search and encapsulated the state of the environment and other properties. One property inputted as a parameter was the heuristic cost of a node. The specifics of how the heuristic cost was calculated will be specified below. Other properties initialized within the node were the transition cost and the parent of the node. The transition cost was determined by doing the
parent node's transition cost + 1 = current node's transition cost
This was done to represent how each additional step the robot took incurred a cost, thus encouraging the robot to take less steps.

With all of these parameters, the next step was to define the priority of a node. Since A* combines greedy and uniform cost search, the priority of a node was an integer
priority = heuristic cost + transition cost
An additional method was used to compare the priority costs of two A* nodes within the priority queue, however, this method was not directly called.

Astar_search function

This function determines the best order to visit nodes based on their heuristic and actual path costs. It began by initializing a priority queue with the start state added in as a node. It also kept track of the visited cost of each state so that we could efficiently mark duplicated states with their shortest cost. In terms of the actual algorithm, the function followed the pseudocode provided in class.

As long as the priority queue was empty, it incremented the number of nodes visited by 1 and popped the lowest priority (lowest cost) node from the list using **heappop** from the imported *heapq* class. The *heapq* class allows us to push and pop items from the priority queue in **O(log n)** time, which makes it a relatively efficient way to maintain the priority order of the queue. However, since replacing items in a priority queue is quite difficult, the dictionary mentioned above kept track of all visited states and their lowest cost to be there. This allows us to add duplicated state nodes with different cost functions to the priority queue without sacrificing too much time, albeit at the cost of memory.

Once the current node was popped from the queue, its priority was compared to its lowest cost counterpart in the *visited* dictionary. If it had a higher visited cost than the lowest cost, path, the node was ignored and the loop moved on until the next iteration. Moreover, if the current node's state was the goal state, the pre-written backchaining algorithm, which found the path to the current node using its parent node, was executed and the solution to the goal was returned.

If none of these conditions held up, all of the successors to the current state were calculated using the *get_successors* method from the search_problem parameter. More on that implementation below. If there weren't any children of the current node, the node was added back into the priority queue and the number of repeats was incremented by 1. A repeat counter was kept to ensure that the number of times a state was visited never exceeded the number of robots in any given maze. If one robot had no children for a given state, then maybe a second robot did. However, if there are only two robots then you wouldn't want to be in the same state more than 2 times.

Assuming that children of the current state were found, the number of repeated states was reset to 0. Each of the children were iterated through, and if the state hadn't been visited or if the child had a lower visited cost than it's previously visited counterpart it was added/updated within the *visited* dictionary and it was pushed to the priority queue using **heappush**. As mentioned earlier, **heappush** automatically sorts items in the queue by priority so that accessing the minimum cost later would be efficient.

This entire process was repeated until the queue was empty or a solution was found, and a solution was returned.

MazeworldProblem.py

This class initialized the states, goal locations, and potential actions as tuples. The states locations parsed the robot locations in the *Maze* class and represented them as a tuple of tuples in the format
((x1, y1), (x2, y2), (x3, y3), … (xn, yn))
Where each *(x,y)* pair represented a single robot's location. The goal locations for each robot were represented in a similar format. A counter that represented the index of the current robot was also initialized within this class, so that only one robot was acting at any given time. The potential actions of any given state were "left", "right", "up", and "down", and their Cartesian Coordinate representation (i.e. "left": (-1,0)) were initialized in a set of tuples.

Get_successors function

The current robot's x and y coordinate was obtained by indexing into the inputted state using the robot counter mentioned in the above section. From there, a set of all possible children was initialized. The function then iterated through all possible actions and applied them to the x and y coordinates of the robot. If the robot's newX and newY were valid positions, a copy of the state was updated with the new robot's locations and this child state was added to the set of states. Once all of the actions had been iterated through, the set of all possible children was returned, and the current robot's index was incremented. To prevent the current robot's index was assigned as
robotIdx = (robotIdx + 1) / number of robots
The number of robots was calculated by determining the length of the state/length of the robot locations/2.

IsValid function

This function took a state and a newX and newY as parameters. If the newX and newY were on a floor, as tested using the *isFloor* method from the *Maze* class, and if the newX and newY were not on the locations of other robots, the new state was valid and the function returned **True**. Otherwise, it returned **False**.

Manhattan Heuristic

This function iterated through the goal locations and the current state, inputted as a parameter. It can be described as
manhattan_distance = sum(abs(goalXi - robotXi) + abs(goalYi - robotYi)) for 0 <= i < number of robots< em>
By summing the manhattan_distance of each robot, you can determine a heuristic for the entire multi-robot system. This heuristic is useful when the robot cannot move diagonally, however, when the robot can move diagonally it may not be admissible because it overestimates how many steps a robot must take to change position. Since in this lab the robots can only move in 4 directions, the Manhattan heuristic is valid.

Goal Test function

This simply returned whether the current state (inputted as a parameter) was equal to the goal locations. If it was, the function returned *True*, and returned *False* otherwise.

SensorlessProblem.py

The goal locations and the actions were initialized similar to the standard *MazeworldProblem* initialization. However, the state was represented as a collection of all possible coordinates the robot could have been at. This was determined by iterating through every coordinate on the maze, and if it was a floor, adding it to the set of potential coordinates representing the state. There was also a dictionary which expressed the cartesian coordinate version of the actions as words, making the final path easier to understand.

Get_successors function

Similar to the *MazeworldProblem*'s implementation of this function, the *SensorlessProblem* created a set of all possible states and iterated through each action in the potential action space. For all potential coordinates in the state, the action was applied

and a *(newX, newY)* pair was created. If this pair was on a floor, it was added to a set representing the next possible state. Otherwise, the original coordinate was added to the potential state set (because it hit a wall). The reason for using a set was because it removes duplicate coordinates, so if enough intelligent actions are taken, the robot should localize to one possible location. Each set of possible coordinates was then converted into a tuple and added to the set of children. The tuple version of this child set was then returned.

Goal Test function

This method returned **True** if the length of the set of possible states was equal to 1.

States to Path function

This method was used to convert the complex representation of states into interpretable actions. It took in a state and a next state as parameters. It then iterated through each action, and if the result of applying that action to the original state was the next state, then the action's English translation was returned.

Back-chain to Path function

The method used the result of the back-chain method from the *astar_search* file as a parameter. With the path as an input, it ran the *statesToPath* method on each state, using the state at the next index as the next index in *statesToPath*. By adding the English version of each action to a path, the result was a much more readable version of how the robot localized in as few movements as possible.

Heuristic

This function took in a state as a parameter. It then returned *the length of the state - 1* as the heuristic cost. This was because in the best case, it takes that many steps to localize. For a more detailed discussion, see below.

Evaluation

Multi-Agent Search Problem

In all test cases of the multi-agent search problem, the AStar algorithm performed as it was intended. The first test case had two robots start within a confined space and have to pass by each other to get to their goals on the opposite side of the maze. This solution was found after 291 nodes were explored, and the total solution length was 42 steps. An interesting behavior that was noticed immediately from this first test was that the robot's did not know they could wait for another robot to pass them. This is because in the action space, they could only ever move left, right, up, or down, but had no action which said they could stay still. As a result, sometimes the robot would move strangely near the target because it was forced to move in one of the four directions rather than stay still. An interesting extension would be to see how the path cost changes when a robot stays still.

```
Mazeworld problem:
Start                Middle               End
#.##.####.##         #.##.####.##         #A##.####B##
#..#.##....#         #..#.##....#         #..#.##....#
#..##.#.####         #..##.#.####         #..##.#.####
#B#.....#..#         #.#.AB..#..#         #.#.....#..#
#A..#.##.#.#         #...#.##.#.#         #...#.##.#.#
```

The second test case was interesting because it asked 3 robots to reverse their order within a narrow column which was at most, 3 blocks wide.The robots managed to switch their order and move towards the goal in only 17 moves after exploring 161 possible nodes. Again, while this path was optimal given the constraints of the problem, the lack of a stationary action likely increased the number of moves each robot had to take. In attempting this maze by hand with a stationary move, the robots could have coordinated the best way to complete the task in only 10 moves compared to the 17 without it.

```
Mazeworld problem:
Start         Middle          End
##.##         ##.##           ##A##
#...#         #...#           #.B.#
#.C.#         #C.B#           #.C.#
##B##         ##A##           ##.##
#.A.#         #...#           #...#
```

The third test featured an extremely large maze that was randomly generated online at the link *https://www.dcode.fr/maze-generator* The width and height specified within the program was 40x40, but the resulting maze was approximately 81x122. A single agent had to work through this entire maze itself, however, since the maze was perfect (meaning it had a solution), the agent found its way to the goal in 294 moves, after exploring only 2595. This brings up a useful point about the runtime of AStar. Because it is contingent on a heuristic in part, AStar's runtime grows exponentially with relation to the length of the shortest path (d) and the branching factor (b). It can be written **O(b^d)**. In this maze, we used the manhattan heuristic, which judged the sum of the horizontal and vertical distances the robot was from the target. The agent might have visited several paths which resulted in a low manhattan distance but a dead-end nonetheless. Since AStar keeps all of the nodes in memory, this is taxing on the space constraints of a system as well. A better heuristic could have been that the robot should follow a wall until it finds a target, and if this heuristic was implemented, the robot may have found a solution to the large maze problem after exploring fewer nodes.

```
Drawing of path excluded for brevity, can run this example using instructions in "README.md" to see the anima

Mazeworld problem:
attempted with search method Astar with heuristic manhattan_heuristic
number of nodes visited: 2595
solution length: 294
cost: 294
path: [((0, 80),), ((1, 80),), ((1, 79),), ((1, 78),), ((2, 78),), ((2, 77),), ((3, 77),), ((4, 77),), ((5, 7
```

In the fourth test case, two agents were on opposite ends of the spiral, and had only one block of room to make way for each other. A solution was found in 67 steps after 479 were visited. This problem was interesting because it combined the constrained coordination aspects of problem two with the opposite goal locations of problem 1, and the robots behaved relatively efficiently in these conditions. The fifth test case was a larger variation of this problem, with two robots having criss-crossing goals and only one path to each goal, which meant they had to navigate a larger map with even more constraints than problem two. Again, the robots solved this task in a small amount of moves (39) after only visited 273.

```
Test Case 4:
Mazeworld problem:
Start           Middle          End
A######         .######         B######
.#....#         .#..A.#         .#....#
.#..#.#         .#.B#.#         .#..#.#
.#.##.#         .#.##.#         .#.##.#
.#.B#.#         .#..#.#         .#.A#.#
.####.#         .####.#         .####.#
......#         ......#         ......#


Test Case 5:
Mazeworld problem:
Start                 Middle              End
#A######B#            #.######.#         #.######.#
#.######.#            #.######.#         #.######.#
#.######.#            #.######.#         #.######.#
#........#            #.AB.....#         #........#
###.##.###            ###.##.###         ###.##.###
###.##.###            ###.##.###         ###.##.###
#...##...#            #...##...#         #...##...#
#.######.#            #.######.#         #.######.#
#.######.#            #.######.#         #.######.#
#..####..#            #..####..#         #B.####.A#
```

Sensor-less Problem

Once again, the AStar algorithm worked properly and helped the agent localize in as few moves as possible. Starting with the first test case, no solution existed which could help the agent localize, since certain floor tiles were surrounded by walls. The agent learned there was no solution after visiting 3179 nodes, which seems large but was necessary to determine the lack of a solution.

```
Blind robot problem:
attempted with search method Astar with heuristic heuristic
no solution found after visiting 3179 nodes
```

In the second test case, the agent had to localize in a mixed 7x7 maze where there was a solution. The agent only had to explore 232 nodes before determining 9 actions which would help it localize quickly and at the target. For all of these test cases, the robot was trying to localize at a target, which represents its actual location in the maze. This is the same as specifying the target of an actual location you would like the robot to get to, because the robot will take actions to become 100% certain it has arrived at a location if it knows the map.

```
Blind robot problem:
Start           Middle          End
#####M#         #####F#         #####.#
###GJL#         ###.EA#         ###A..#
###F###         ###B###         ###.###
###EIK#         ###.CG#         ###...#
##BD###         ##.H###         ##..###
##ACH##         ##.DI##         ##...##
#######         #######         #######

path: ['east', 'south', 'east', 'west', 'north', 'north', 'west', 'north', 'north']
```

In the third test case, the agent explored a completely empty 10x10 map, and, similar to the solution we discussed in class, it worked its way up to a corner in only 18 moves after exploring 27 nodes. This was a relatively fast convergence for a large maze, but a big credit to this quick convergence was the lack of obstacles. As discussed in class, if there were obstacles in this large maze, the robot may confuse the corner for a wall and be stuck there for a certain amount of actions. Regardless, it is nice knowing that when unobstructed, the robot is still taking the fastest path to the target, which was the bottom left corner in this case.

```
Blind robot problem:
Middle            End
F.........        ..........
J.........        ..........
B.........        ..........
G.........        ..........
I.........        ..........
C.........        ..........
E.........        ..........
H.........        ..........
A.........        ..........
D.........        A.........

Start was omitted due to character formatting issues, can follow instructions in README.md to run this exampl

path: ['west', 'west', 'west', 'west', 'west', 'west', 'west', 'west', 'west', 'south', 'south', 'south', 'so
```

The fourth test case saw the robot, have to localize amidst an "H" shape in a 7x9 grid. The robot had to explore 817 nodes, but eventually found a solution within 17 actions. This example made me think about how few-shot learning may be implemented to solve such examples. As humans, we can observe the solution to a small portion of problems, but build quickly internalize heuristics that help solve new, similarly structured problems. Solving this personally, I believe I could arrive at the same conclusion as the robot, but I don't believe it would have taken me as many "nodes" explored. While I cannot formalize a more

efficient heuristic for this problem currently, I'd be curious to see how one specifically adapted to the sensor-less problem would perform.

```
Blind robot problem:
Start             Middle          End
G#######U         .#######.       .#######.
F#######T         .#######.       .#######.
E#######S         .#######.       .#######.
DHIJKLMNR         BIDFAGHCE       A........
C#######Q         .#######.       .#######.
B#######P         .#######.       .#######.
A#######O         .#######.       .#######.

path: ['north', 'north', 'north', 'north', 'north', 'north', 'south', 'south', 'south', 'west', 'west', 'west
```

The fifth test had a robot localize with zig-zagging rows in a 6x6 grid. While this problem was not extremely complex, I was surprised to see that the robot adopted a different strategy than what I had done. Working by hand, I localized each row to a single possible value, then built up from the zig-zags. The robot, on the other hand, engaged in seemingly spurious action but still found a solution in a few amount of moves. The robot determined where it was in 15 moves, after visiting a total of 1006 nodes. It was interesting seeing how the computationally efficient solution to this problem differed from my preconceived solution for the problem.

```
Blind robot problem:
Start         Middle          End
CFKORU        ...HDA          ..A...
##J###        ##.###          ##.###
BEINQT        ...E.F          ......
###M##        ###.##          ###.##
ADHLPS        ...BGC          ......
##G###        ##.###          ##.###

path: ['south', 'east', 'north', 'north', 'east', 'east', 'east', 'east', 'west', 'west', 'north', 'north', '
```

Evaluation takeaway

The biggest takeaway from the evaluation was that the robot performs well if the heuristic is optimized for the problem, but takes a long time to either find a solution or localize if the heuristic is not the best. This goes in line with our understanding of the time complexity of AStar search, and will be an important lesson about deciding on search algorithms moving forward. I'd be curious to see how a reinforcement learning agent performs on these test cases after sufficient training, and believe that it may internalize better heuristics to solve each problem. Still, the combination of a heuristic and the true path cost makes AStar a relatively successful search algorithm to use.

Discussion Questions

How to represent the state of the system with k robots?

Each robot has an *(x,y)* pair representing its coordinates on the maze. Moreover, if only one robot can be referenced at a time, you would want to keep track of which robot is currently eligible to move. Thus, you would want the state of the system to be

((x1,y1), (x2,y2), … , (xn, yn))

That is, a tuple containing the coordinates of all robots (represented as a tuple). I also initialized a counter that indicated the current robot's index. This counter was updated every time the *get_successors* method was called.

Give an upper bound on the state space for the system in terms of k and n.

The first robot can be at *n^2* possible locations. The second robot can be at *n^2 - 1* possible locations. This pattern continues for all *k* robots in the system. Thus, the upper bound on the state space becomes

The product of all n^2 - i for i in range(0,k robots)

We can view this upper bound as a factorial by doing

(n^2)! / ((n^2) - k)!

Give a rough estimate of how many of the states represent collisions if w represents a wall

If we take the total number of states to be

(n^2)! / ((n^2) - k)!
we can view the total number of valid states to be

((n^2) - w)! / ((n^2) - w - k)!

Thus, we can view the number of states that represent the number of collisions as
((n^2)! / ((n^2) - k)!) - (((n^2) - w)! / ((n^2) - w - k)!)

Would BFS be feasible given a large maze with few obstacles and many robots?

No, this would not be computationally efficient. Each robot would explore 4 nodes each time and add these points to their memory. This is because each time a robot is revisited, its neighbor robots may have left, so it has to expand 4 nodes each time. While a single robot may be able to complete this search on its own, multiple robots performing these calculations would be very intensive on memory. The Astar search gives the robots a guide to get to their goal efficiently by considering path costs and heuristic cost. However, BFS considers none of these, and the program could backtrack or explore more nodes than necessary to find a target that may be relatively close by.

Show a useful monotonic heuristic for this search space

A useful monotonic heuristic would be the manhattan heuristic for this search space. The manhattan distance calculates the distance between the robot's coordinates and the goal's coordinates, assuming the robot cannot move diagonally. This is monotonic because for all actions a robot can take, the path cost from the robot's location to its neighbors will be increased by 1. This leaves two options: either the neighbor could be closer to the goal, in which case its manhattan distance is reduced by one, or the neighbor is farther from the goal, in which case its manhattan distance is increased by one. Regardless of the neighbor's location, the path cost to the neighbor added to the neighbor's heuristic cost will always be greater than or equal to the robot's current heuristic cost, making the heuristic monotonic.

Describe why the 8-puzzle is an example of the multi-agent problems

If you consider the location of each number in the 8-puzzle problem as part of the state, then they each have a coordinate on the 3x3 grid. Since there is one empty space, you can treat the space as a robot which you move around. The heuristic comes from how far each number is from its corresponding goal location (i.e. number 1 should be in top left corner, so find distance from top left to number 1's location). This is a modification of the robot search problem, since all of the numbered tiles represent free spaces and the empty tile represents the robot. The heuristic still holds up because a path cost of 1 paired with the sum of the manhattan distance for each numbered tile on the board results in a monotonic heuristic.

Prove that the 8-Puzzle is made of two disjoint sets

For an 8-puzzle to be made of two disjoint sets the puzzle would have to be unsolvable. This means the goal solution is not reachable from the start state, and vice versa. I would prove this to be true by writing two functions. The first would be a helper function called *countInversions* and would take in an array representing the puzzle. For each item in the array, if the item's value

was greater than the value of the item at the next index, the number of inversions was incremented by one. From there, a driver function called *solvable* which takes a puzzle as a parameter, runs *countInversions* on every state individually, then continues until the pqueue is empty. After each call to *countInversions*, the total number of people has increased. If the final result is even, the puzzle is solvable. Otherwise, the sets are disjoint and the puzzle is not solvable.

What heuristic was used for A star search with a sensor-less robot?

I used *the length of the state - 1* as the heuristic for the sensor-less problem. This is because if a robot is unsure of its location (meaning the length of the state is greater than 1), in the worst case it would have to travel across the entire board to localize. However, at best it would only have to travel *the length of the state - 1* to become sure of its location. As the state size reduces, the agent is more confident about its location, so the heuristic cost of discovering its next position is reduced. This heuristic is admissible because it always underestimates the moves required to localize or provides the exact amount of moves needed to localize. This heuristic is optimistic because it predicts the cost of localization to be less than or equal to the actual cost of localization.

Another heuristic I considered was the null heuristic. This would ensure the robot localized if a solution existed, however, it comes at the cost of memory and time. The null heuristic turns Astar into uniform cost search. This means that it will explore nearly every node in trying to find a solution, which can be inefficient. While both search methods are optimal, the null heuristic doesn't look to minimize the time and space complexity of the search program to the same extent that Astar does, and as such, I would prefer the previously defined heuristic over the null heuristic for this search problem.