MAE 242: Robot Motion Planning
Project #1

Assigned April 5, due April 15

Homework Instructions:

This first assignment has two parts (an analytical and programming part) with different due dates, check it out on canvas. Please submit your answers on the corresponding link on gradescope. The Collaboration Policy for this course is detailed on the syllabus and you have to provide the names of at most 2 other students you have collaborated with to do this homework. Please check the late homework policy in the syllabus.

Programming exercises:

1. (Programming problem) In this exercise, you will implement and demonstrate the performance of various deterministic search algorithms in different maze environments. To do this, please download the folder "search" from canvas. In it you will find several files. The file `mazemods.py` contains main functions to plot a maze, check for collisions, calculate costs, and draw paths in the maze environment. The files `smallMaze.py`, `mediumMaze.py`, and `bigMaze.py` are mazes of several sizes. A maze is a 2D grid world of $n \times m$ states given by $x = (a, b)$, with $0 \leq a \leq n-1$, $0 \leq b \leq m-1$, and a list of forbidden cells in it described by O. Transitions at each state are given by legal controls $u \in U = \{(1,0), (-1,0), (0,1), (0,-1)\}$, which do not incur into collision, and which can be obtained by simple addition $x + u = x'$. The illegal or forbidden transitions are those resulting into a wall collision, and this is evaluated by the function `collisionCheck` inside `mazemods`.

   The file `search.py` is the file that you will have to edit and submit as your solution to this problem.

   (a) (10 points) Define a function `depthFirstSearch`(x_I, x_G, n, m, O) implementing a DFS algorithm on a maze. The inputs of the function are an initial state, x_I, a goal state, x_G, and problem environment parameters, n, m, O. The outputs of the function should be a list of legal control actions following DFS, the cost of this sequence of actions, the number of nodes explored in the search, and the plot of a final path from start to goal. To compute the cost of the action sequence use the function `getCostOfActions`(x_I, x_G, n, m, O) in `mazemods.py`. Note that any illegal action will result in a very large cost. Is the exploration order what you would have expected? Is this a least cost solution? If not, think about what depth-first search is doing wrong.

   (b) (10 points) Define a function `breadthFirstSearch`(x_I, x_G, n, m, O) implementing a BFS algorithm on a maze. The inputs of the function are an initial state, x_I, a goal state, x_G, and problem environment parameters, n, m, O (maze size and obstacles). The outputs of the function should be a list of legal control actions following BFS, the cost of this sequence of actions, the number of nodes explored in the search, and a plot of the final path from start to goal. To compute the cost of the action sequence use the function `getCostOfActions`(x_I, x_G, n, m, O) in `mazemods.py`.

   (c) (10 points) While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Implement a `DijkstraSearch`$(x_I, x_G, n, m, O,$ `cost=westCost`$)$

implementing Dijkstra's algorithm over a maze. The inputs of this function are x_I, x_G, n, m, O as before as well as a cost function. The outputs should be a list of legal control actions following Dijkstra's, the cost of the sequence of actions, the number of nodes explored in the search, and a plot of the final path from start to goal. Provide example plots and results based on the cost function that favors that the agent stays West; $\ell((a,b),u) = \ell(a',b') = a'^2$, and the cost function that favors the agent staying East $\ell((a,b),u) = \ell(a',b') = (x_{\max} - a')^2$, where $x_{\max}$ is the furthest east coordinate of the environment. These functions can be found in `mazemods.py`.

(d) (20 points) A^* implementation. A^* expands Dijkstra's algorithm by making use of a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the goal state in the problem. Implement a function `aStarSearch`$(x_I, x_G, n, m, O,$ `heuristic=givenHeuristic`$)$ that implements the A^* search. To do this, use the regular cummulative cost function `getCostOfActions`. You also have to define two heuristic functions to replace `givenHeuristic` as follows:

- Using $|a' - a| + |b' - b|$ as a heuristic (`manhattanHeuristic`)
- Using $\sqrt{(a'-a)^2 + (b'-b)^2}$ as a heuristic (`euclideanHeuristic`)

As an example, the trivial `nullHeuristic` is given for you in `search.py`. Evaluate A^* on various grid-based problems of your choice. Compare the performance for the two different cases: Which heuristic is superior? Explain your answer.

Programming Instructions:

- You will fill in portions of search.py in search.zip during the assignment. You should submit search.py with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than these files.
- The appropriate function calls used for evaluating your functions are:
  (a) `out1, out2, out3 = depthFirstSearch(vstart,vgoal,n,m,O)`,
  (b) `out1, out2, out3 = breadthFirstSearch(vstart,vgoal,n,m,O)`,
  (c) `out1, out2, out3 = DijkstraSearch(vstart,vgoal,n,m,O,'westcost')`,
  (d) `out1, out2, out3 = DijkstraSearch(vstart,vgoal,n,m,O,'eastcost')`,
  (e) `out1, out2, out3 = aStarSearch(vstart,vgoal,n,m,O,'manhattan')`,
  (f) `out1, out2, out3 = aStarSearch(vstart,vgoal,n,m,O,'euclidean')`,

  where the outputs are in the order of input sequences, total cost and number of visited nodes. The inputs include start and goal positions, grid parameters and heuristic strings for Dijkstra's and A-star algorithms.
- Your code will be verified for basic technical correctness. Please do not change the names of any provided functions, classes or file names, as this will confuse the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score. The autograder acts primarily as a filter. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.
- Getting Help:
  (a) A byte code file 'autoverify.cpython-39' has been provided for your help. Please keep it in the same folder as the other files and run it as any other python file. This file tests 'search.py' functions for any data structure issues and the validity of the output u-sequence.
  (b) Some sample code has been provided towards the end of the search file to understand the use of functions. Please start early on the coding assignments. If you are stuck on something, please use office hours and Piazza.
- The autograder currently considers numpy and matplotlib libraries. If you are using other libraries, please email the TA to add those Python libraries in the autograder environment.