ME570 Homework 4

Jay Lanzafane
Boston University Mechanical Engineering
Jlanzafa@bu.edu

1. Problem 1 Introduction

This problem includes the basic building blocks that define the graphVector structure as well as creating the graph_search function (which implements A*), and tests the functionality on the 2D manipulator.

1.1. Graph Search Function

In this section we develop the graph_search function to find the shortest path, if it exists, between 2 nodes idxStart and idxGoal in graphVector. The code is predominantly formatted following the pseudocode laid out in both the homework assignment and the text book. The search function has a few basic subsections which are detailed briefly here, but a more complete account can be found in the code comments.

Given a startIdx and goalIdx within the graph graphVector, we need to find the shortest path between the two. We start by expanding the node referenced by startIdx. This means we add the node to the priority queue pqOpen, which houses all the nodes which have a clear path back to the starting node. Ideally, the goal node will end up in this list and the function can then trace its way back to the start node. The priority queue is ordered (by cost) by the minimum possible distance to the goal from the start node through this node. By using the priority queue we explore the graph in an intelligent way to ensure the path we find is the shortest possible path. The function iterates through until it finds the goal, and on each iteration will find the neighbors of the "best" option in the priority queue and then add those possible routes to the priority queue. In addition the function keeps track of previously explored nodes as well as updating nodes that had less efficient backpointers. A* is a well known algorithm and has plenty of documentation in the public domain, but these are the major sections.

Note that a few helper functions were defined in the process, specifically, heuristic, getExpandList, expandVertex, and buildPath.

1.2. Graph Search Test

Here we develop the graph_search_test function which is a useful tool for verifying that graph_search is running properly. This function will either load an existing dataset or randomly generate a new dataset, depending on the call parameter bool. The graphVector object is then plotted on an x,y grid with dotted lines showing the connections between nodes. Xstart and Xgoal are also generated (or loaded) and graph_search is then called on this data. The results are plotted for visual inspection. See 1

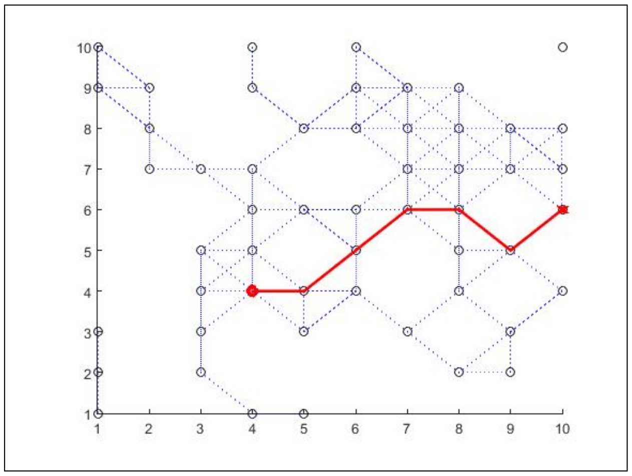

Figure 1. Randomly generated graphVector object with random start and end nodes. A* is then called to find the shortest path, which we can visually confirm to be correct.

1.3. Creating GraphVector

Here we define the function grid2graph(xGrid, yGrid, flag) which takes a NxM logical array, flag, and two generalized position vectors xGrid (Mx1) and yGrid (Nx1) and defines the graphVector structure. xGrid and yGrid contain the generalized x,y coordinates that correspond with the rows and columns of flag. We use an 8-neighbors connectivity to determine which nodes have connections. Details are found in the comments but the overarching structure follows; first, for each node that exists (flag = 1), create an entry in graphVector. Initialize these entries with their x,y coordinates in graphVector.x. Initialize empty vectors for neighbors, neighborsCost, g, and backpointer (we will fill in neighbors and neighborsCost momentarily). Once the graph has been initialized, the function searches for neighbors, which is simple except for the many if/else statements which are needed to prevent indexing into nonexistent entries. NeighborsCost is calculated simply as the euclidean distance between the neighbor and the node itself.

1.4. The Twolink Manipulator

In this problem we begin laying the groundwork for the twolink manipulator. This starts by defining the function twolink_freeSpace(points, NGrid). This function generates the two NGridx1 vectors theta1Grid and theta2Grid of evenly spaced points along [0 2*pi]. The entire space defined by the meshgrid of these two vectors is checked against the obstacles defined by the points variable. Returns flag, an NGridxNGrid array of logicals detailing whether the angle combination at that point is blocked or free, which we can utilize in the above function grid2graph. This is accomplished in the function twolink_freeSpace_makeData which utilizes twolink_freeSpace to make a 90x90 array covering the region, evaluating that graph's connectivity in the face of the obstacles, and turns that into a usable graphVector object which is saved for later use.

1.5. Twolink Search

We are almost ready to use our graph search algorithm as a path planning tool for the twolink manipulator, but need to adjust a few minor things first. We define twolink_search, the graph_search algorithm adapted for the twolink problem, which paths from position [theta1Start theta2Start] to [theta1Goal theta2Goal] within the constraints of graphVector (loaded from twolink_freeSpace_data). The start and goal location need not lie within the graphVector structure precisely, the nearest node to the desired start and end locations will be used for those positions instead. As such, the function needs to condition the inputs before passing them into graphSearch, and similarly conditions the outputs slightly; instead of a path defined by node indeces, we return the coordinate paths theta1Path and theta2Path.

1.6. Animating the Path

In order to visualize the motion of the twolink manipulator, we define twolink_animatePath. This function Draws the configurations of the twolink manipulator as it moves along the path defined by theta1Path and theta2Path. The draw speed will be done at fps frames per second. We run the path planner for two pairs of start and goal locations, given by [3.62, 0.85, 3.62, 5.54] seen in 2 and [2.77, 2.45, 1.06, 1.814] seen in 3. For convenience, the animation.m script can be used to quickly view the progress of the twolink manipulator for each of these configurations.

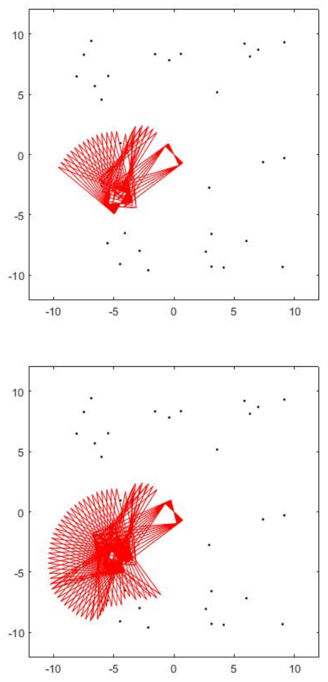

Figure 2. Generated path for the first start/stop coordinates, with partial path and full path snapshots.

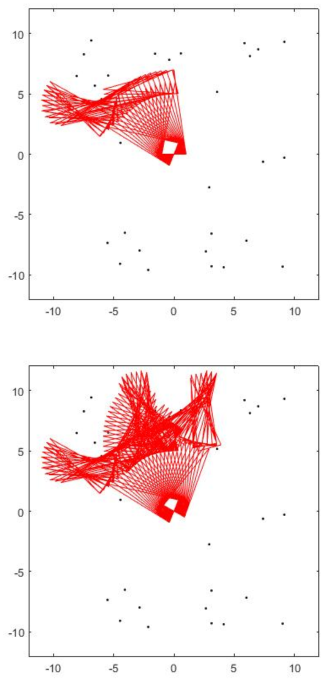

Figure 3. Generated path for the second start/stop coordinates, with partial path and full path snapshots.

1.7. Final Comments

It can be noted that we did not use the 'torus' definition when creating our graphVector. As such, the twolink manipulator treats the space exactly as we have defined it, ranging from [0 2*pi]. It has no conception of the fact that 2*pi will wrap to 0, i.e. it doesn't understand the periodic nature of working in angles. As such, it will never cross the boundary of 2*pi, or 0. This causes an unwinding phenomena in the first example, where the manipulator could have simply moved the second arm to get to the correct position, but would have needed to wrap at 2*pi, and thus does not consider it as a valid path.

Another observation is how close the manipulator comes to obstacles. There are a few cases where the manipulator may have collided with an obstacle in practice, as we haven't sampled finely enough to detect it (it may reside between two unobstructed states, which are connected). To remedy this, we could sample more finely, or artificially increase the size of our obstacles such that they are more disruptive. I would begin by increasing obstacle size, especially in this problem, as increasing the sampling resolution, while working with radial systems, and with point obstacles, means that the resolution would have to be drastically increased. This would slow down both the path planning and the graph generation dramatically. Increasing obstacle size, however, will only slightly affect runtime performance and would be more robust in this instance.

2. Problem 2 Introduction

In this problem we consider the related procedure of path planning using visibility graphs. Here we examine a world of multiple polygons. Each vertex of the polygons will be a node within a graphVector structure, as well as the start and goal locations. The process for generating graphVector is entirely different than for the graph in Problem 1. Here we need to check visibility between nodes instead of checking adjacency. Note, however, that once our graph is properly created, it is a simple matter of using the already developed graph_search function to plan paths through the environment.

2.1. Viewing the world

One of the first and most simple things we'd like to do is be able to visualize the world. This is a simple process, we start by defining the funciton polygon_draw, which takes a list of vertices. We simply plot each of those vertices, and then close off the polygon by plotting the line from the last vertex to the first vertex. With the ability to draw a single polygon, we create visibility_plotWorld. This function takes a world structure (defined in the homework and comments) and iterates through all of it's polygons, calling polygon_draw on each of them. This Creates a single plot of our entire world, which can be seen in 4.

2.2. Edge Collision

The first and most obvious type of obstruction to visibility between two points is edge collision. What is meant by this is that the line of sight (LOS) between two points intersects an edge of any of the polygons. The most basic building block of this check is determining if two lines intersect or not (the LOS and an edge of a polygon, for example). We define the function edge_checkCollision which takes as input the two sets of endpoints for each line and computes their intersection point, if it exists. Note that if the two lines intersect at an endpoint, then visibility is not blocked. Also, if the lines are co-linear, then the LOS is also considered not blocked. This function draws on the built-in function polyxpoly, which does almost exactly what we want, but then checks for co-linearity as well as endpoint overlap to be allowed as exceptions.

2.3. Self Occlusion

The other possible situation in which LOS is blocked between two points is called self occlusion. This is when the LOS between 2 vertices on a single polygon may have line of sight to one another by the edge collision check, but that the line traverses the inside section of the polygon. Clearly, the vertex can not see through the interior of the assumed-solid obstacle. To check for this we define the function visibility_checkSelfOcclusion. This function essentially determines if the LOS from an initial vertex to a destination vertex falls between the lines that travel from the initial vertex to its adjacent vertices. We use normalized dot products to check the angles between the 3 vectors, defined by the 4 points above (initial-previous, initial-next, initial-destination). We can then determine if the LOS is blocked by the obstacle itself via self occlusion.

2.4. Checking Visibility

With the building blocks of checking visibility in hand, we define the function visibility_check which will determine the visibility of any one point to all the vertices within our world structure. This is done by iterating over every vertex in world and setting it is the destination node. We then first check for self occlusion, followed by edge collision. Edge collision needs to be performed on every edge within the world, so we iterate over every polygon, calculate the lines between vertices (the same as polygon_draw) and then check collision. If the destination node is not self occluded or blocked by edge collision, then it is added to the list of visibile vertices and returned. Note that self occlusion changes on the boundary polygon very drastically. Here we want the reverse effect; instead of ensuring the LOS isn't inside the polygon, we want to make sure that is is inside the polygon. This is because the boundary is effectively filled externally, as opposed to the obstacles which are filled internally. To check for this we check whether the polygon is labeled clockwise or counterclockwise. CCW denotes an obstacle, while CW denotes a boundary. We check this before checking self occlusion to know whether we are on a boundary or obstacle.

2.5. Testing Visibility

With the visibility_check function in hand, we proceed to a test function to validate our efforts thus far. We create the visibility_check_test function. This function loads the data in polygonalWorld.mat, plots the world using our visibility_plotWorld function, and then calculates visibility from a predefined point. We then draw lines in red between the given point and each vertex in world that it has LOS to. An example of this can be seen in 4. Note this location happens to be one of our start locations used subsequently.

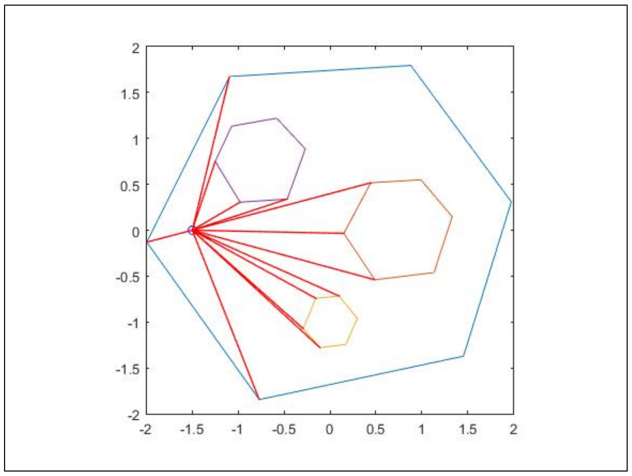

Figure 4. Results of visibility_check function called on the first start location.

2.6. Creating the Visibilty Graph

We now have all the tools necessary to transform our polygonal world into a connected graphVector structure, which we do in the function visibility_graph. As previously stated, connections will exist between nodes if they have LOS to one another, as can now be checked using our function visibility_check. This function starts by iterating through each node and initializing a corresponding node in graphVector. We simultaneously calculate visibility from each node using visibility_check and add the results to the neighbors field. Finally, we calculate the euclidean distance between each node in order to define the neighborsCost parameter. We have now generated a usable graphVector over which we could use our graph_search algorithm.

2.7. Searching the graph

While we could call graph_search directly on our graphVector structure as is, there are a few things we'd like to modify and adapt for first. We do this in the visibility_search function. Firstly, we want to be able to search between any two points within the world, but not necessarily constrained to the polygonal obstacles within the world. As such, this function takes an xStart and xGoal location. We then need to add these two points into the pre-existing graphVector. This is done relatively simply by using visibility_check on each of them to get their neighbors, and then adding them into graphVector the same way we did in visibility_graph above. Note that we also need to add the goal node's index to neighbor lists of each of goal node's neighbors to ensure that a path can be found. This is required because the graph structure is effectively directional. Note we don't need to do this for the start node, though it would be perfectly fine to do so. With our start and goal nodes inserted into the graph we can then use graph_search to find the shortest path between these two points. We then modify the output into the x,y coordinate pairs that trace the path from start node to goal node. Note that direct paths between start node and goal node will not be found in this schema. To ensure that the true shortest path is found, we need to check if a direct path does exists between start and goal. This is done in the helper function visibility_quickCheck.

2.8. Testing

This concludes all the development required for the desired functionality, and we wrap up with a quick function to perform our testing. This is done in visibility_search_test. This function loads our data from polygonalWorld.mat, uses visibility_graph to create a graphVector from the data, and iterates through each of the five loaded xStart locations and finds their shortest paths to the goal location. We then use visibility_plotWorld to draw the world and then plot the paths on top of this. The results can be seen in 5. We can visually confirm that all the shortest paths are found, and our algorithm is working properly.

2.9. Final Thoughts

This problem is very similar to the sphereworld problem we addressed in homework 3, as the polygons are close representations of the spheres we saw in that problem. There are a few major advantages to this graphVector representation. The first is that we do not need to do any tuning in order to avoid local minima or ensure convergence, which is a big positive. We are also sure that we find the minimum path distance, which is something potential field methods can not guarantee in all cases, which we saw in the previous homework assignment. The potential field methods generally run much faster, however. Also note that as the complexity of the obstacles increases (more points to define the obstacle)

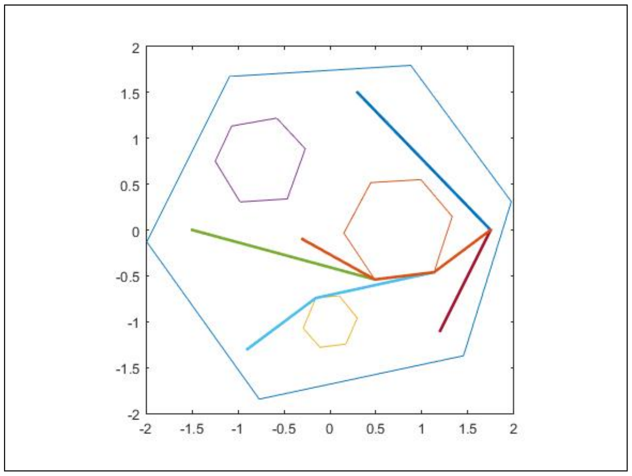

Figure 5. Results of visibility_search_test function called on the 5 start locations (same goal location).

the performance will slow down. Thus it is difficult to accurately represent a circular/spherical object using the methods employed here. Overall I like this method more, especially for this problem, but potential fields do have their advantages and use cases.