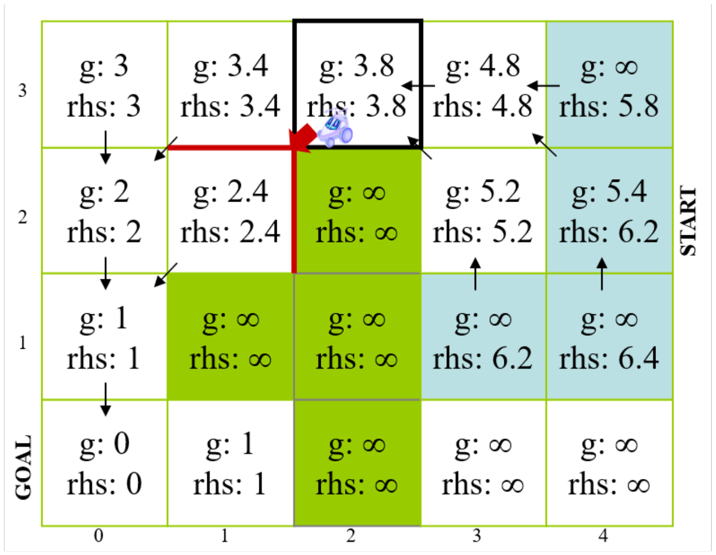
COMS W4733: Computational Aspects of Robotics

Homework 2

Problem 1: D* Lite (20 points)

Recall the D* Lite grid example that we covered in lecture. The robot is on its way toward the goal, but while at $(2, 3)$ it has again discovered an unexpected obstacle, this time at $(1, 2)$. While the robot must obviously reroute through $(1, 3)$ instead, let's work through the mechanics of D* Lite for replanning.



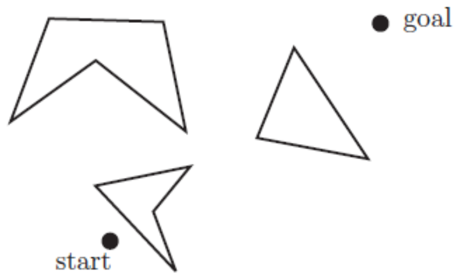
(a) The new obstacle cell $(1, 2)$ and its neighbors have their rhs values possibly updated via UpdateVertex(). Find the state of the open list of inconsistent nodes after this finishes and before ComputeShortestPath() is invoked. List all nodes in order of increasing priority, along with their g value and rhs value. Also indicate whether each is underconsistent or overconsistent.

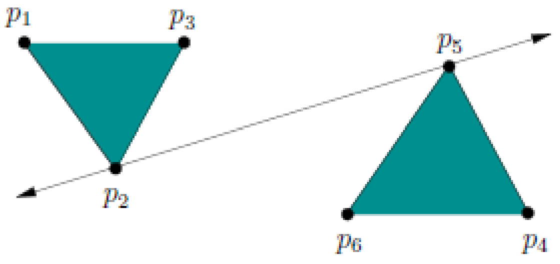
(b) ComputeShortestPath() is now invoked and the first node is popped from the open list. Give the node's updated g value. Then list all nodes whose rhs values are updated as a result of UpdateVertex(). Also indicate whether they are added (or re-added) to the open list.

(c) The next node is popped from the open list as ComputeShortestPath() continues. Repeat the steps in the previous part as a result of this event.

(d) How many times is the robot's current cell $(2, 3)$ added to and popped from the open list in total by the time ComputeShortestPath() finishes?

Problem 2: Visibility Graph (15 points)



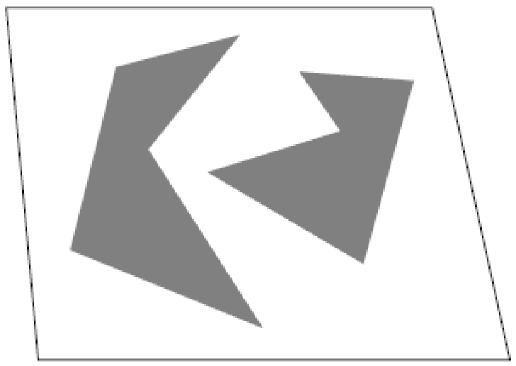


(a) Copy the left figure above and draw the reduced visibility graph on it. Use different colors to distinguish supporting edges and separating edges. Do not include unnecessary edges between obstacles. Remember that obstacle boundaries may also be considered as edges.

(b) Label the obstacle vertices in the visibility graph that you drew (e.g., using letters). Then draw the corresponding connectivity graph.

(c) It is relatively straightforward to identify supporting edges and separating edges between obstacles. Referring to the right figure, give a geometric description in terms of the six vertices that would identify either a separating edge or supporting edge (e.g., if one of the obstacles is reflected across the line) between p_2 and p_5. (Hint: Think about relative orientations.)

Problem 3: Cell Decomposition (15 points)



(a) Copy the figure above and draw a trapezoidal decomposition of the C-space. Then draw the corresponding roadmap by connecting each cell's approximate centroid with its neighbors via the midpoints of the adjacent edges. The leftmost and rightmost cells may have segments that connect to the midpoints of the environment boundaries.

(b) Repeat the above using a boustrophedon decomposition with vertical slices. Instead of overlaying a roadmap on the decomposition, draw an adjacency graph of the resultant cells.

(c) Suppose we now want to perform coverage on the boustrophedon decomposition that you found. Would a coverage pattern using *horizontal* long-segment laps be successful in this environment? Why or why not?

Problem 4: Brushfire for Navigation

Recall that the most basic functionality of the brushfire algorithm is to compute distances from obstacles on a grid. The points where wavefronts collide can be used to construct a generalized Voronoi diagram of the space. The wavefront collisions also admit a decomposition of the space, one that can be used for coverage. You will be exploring the first two applications in this assignment.

You will be writing a program to implement the tasks in each of the following sub-sections. As always, you are free to use whatever platform or language you like. The main input will simply be a grid or array of numbers. We provide several example inputs on which you can test your implementations, as well as an accompanying Python script to quickly generate these as well as custom environments and to plot your results. A 0 indicates a free cell, while $-i$ indicates that the cell is occupied by obstacle i. You may assume that obstacles do not intersect each other or the C-space boundary.

If you are using Python, you can load an environment from a file into a a NumPy array using the `numpy.loadtxt` function. If you are generating and saving new custom environments to a file, you can use either the `numpy.save` or `numpy.savetxt` functions. For these functions you should also input the `fmt="%g"` parameter.

4.1: Brushfire on a Grid (15 points)

The first task is to implement the brushfire algorithm to "fill in" the free grid cells with distances to the nearest obstacle, computed using 8-point connectivity. Cells immediately adjacent to any obstacle are labeled with a 1, the next ones adjacent have a 2, etc. This can be implemented as breadth-first style expansion, where all cells in a given layer of the search tree make up a single wavefront. Each of the four sides of the environment boundary should be treated as a distinct obstacle. Corner cells can be arbitrarily assigned.

Once the brushfire search is finished, your procedure should return the filled-in array. At that point there should be no more 0s in the grid. In Python, you can visualize a NumPy array using `matplotlib.pyplot.imshow`. In your writeup attach figures showing the results of brushfire on all provided input environments.

4.2: GVD Construction (15 points)

The vanilla brushfire search algorithm can be modified to compute the GVD. You can keep track of a grid of pointers indicating the original obstacle for each grid cell that has been expanded. If the same cell is expanded by multiple predecessors with different back pointers, it is part of the GVD. Remember that *all* adjacent free cells are considered neighbors, not just those that have not been expanded before.

Any cells that fall into the above scenario should be added to a growing set of GVD cells, which can then be returned at the end of brushfire (in addition to the filled-in grid). Note that GVD cells should no longer be considered for expansion in the next iteration once they are identified. Overlay the GVD on top of the filled-in grid for each of the sample environments. You may use the `plot_GVD` function provided in our Python script, which sets up `imshow` so that everything shows up distinctly.

4.3: Navigation (20 points)

The GVD can now be used as a roadmap for navigation. Write a second procedure that takes in the grid and GVD from your first procedure, as well as start and goal locations. A path should then be returned (e.g., as a list of adjacent cells) between the start and goal via the GVD roadmap.

You may find it helpful to break this task into two subroutines. The first would find a shortest path between a given cell location and the closest GVD cell via gradient ascent (this satisfies accessibility and departability). The second subroutine would search for a shortest path between any two cells on the GVD, for example using standard breadth-first search. Since the GVD is assumed to be connected, there should always be a solution. Then concatenate the three paths together: start to "GVD start", "GVD start" to "GVD goal", and "GVD goal" to goal.

Demonstrate your navigation procedure on each of the provided environments. You can come up with your own pair of start and goal locations as long as they are not already on the GVD, but please try to show longer or more complex paths, e.g. by placing the start and goal near opposite corners. You may again use our `plot_GVD` function to show the results, this time by also passing in the `path` list of cells that you found.

Submission

You should have one document containing your solutions and responses to all written questions. At the end of the document, create an appendix with printouts of all code that you wrote for problem 4. If you have multiple files for each part, ordering is not important, but it would be helpful to place each file or function on a new page. Archive all of your code files, and submit both your document and code on Gradescope. There should be separate bins for each—make sure you drop both in the right places by the deadline.