COMS W4733: Computational Aspects of Robotics

Homework 2

Ryan Napolitano (rn2473)
Professor Dear
February 24, 2021

Problem 1: D* Lite (20 points)

(a) At the cell (1,2) neighbors are (2,3),(1,3),(0,3),(0,2),(0,1). Here, all cells but for (2,3) have their predecessors from consistent nodes. For incoming edges, the transition costs are now ∞, *rhs* value of (1,2) is raise making it inconsistent. It is put on the queue. Next for all outgoing edges computed using (1,2), the *rhs* needs to be recomputed as transition cost is ∞ now. In this case, only (2,3) is computed using (1,2).

Node	g	rhs	Inconsistency	Priority
(1,2)	2.4	∞	Under consistent	1
(4,3)	∞	5.8	Over consistent	4
(4,2)	5.4	6.2	Under consistent	3
(4,1)	∞	6.4	Over consistent	6
(3,1)	∞	6.2	Over consistent	5
(2,3)	3.8	4.4	Under consistent	2

Table 1: Q1(a) Node state

(b) The node (1,2) is updated and the being under consistent, the new g value is $g = \infty$. Computing the rhs for its successors on expansion, routing via (1,3), we get new rhs for (2,3) as 4.4. Now, the node (2,3) has become over consistent. So, it will be inserted back into the open list.

(c) Now, we see that next minimum g is for node (2,3) and is 4.4. So, we pop that from list. Being under consistent, the g value for (2,3) is updated to ∞. Subsequently, (3,3) and (3,2) are updated for their rhs values. (3,2) updates rhs to 6.4(from (4,2)) and (3,3) updates rhs to 6.2 from (3,2). Both these becomes inconsistent and hence will be inserted to the open list again.

(d) By this process, (2,3) will be popped and added 3 times before the shortest path completes.

Problem 2: Visibility Graph (15 points)

a) The separating lines are denoted by blue and supporting lines are denoted as red color.

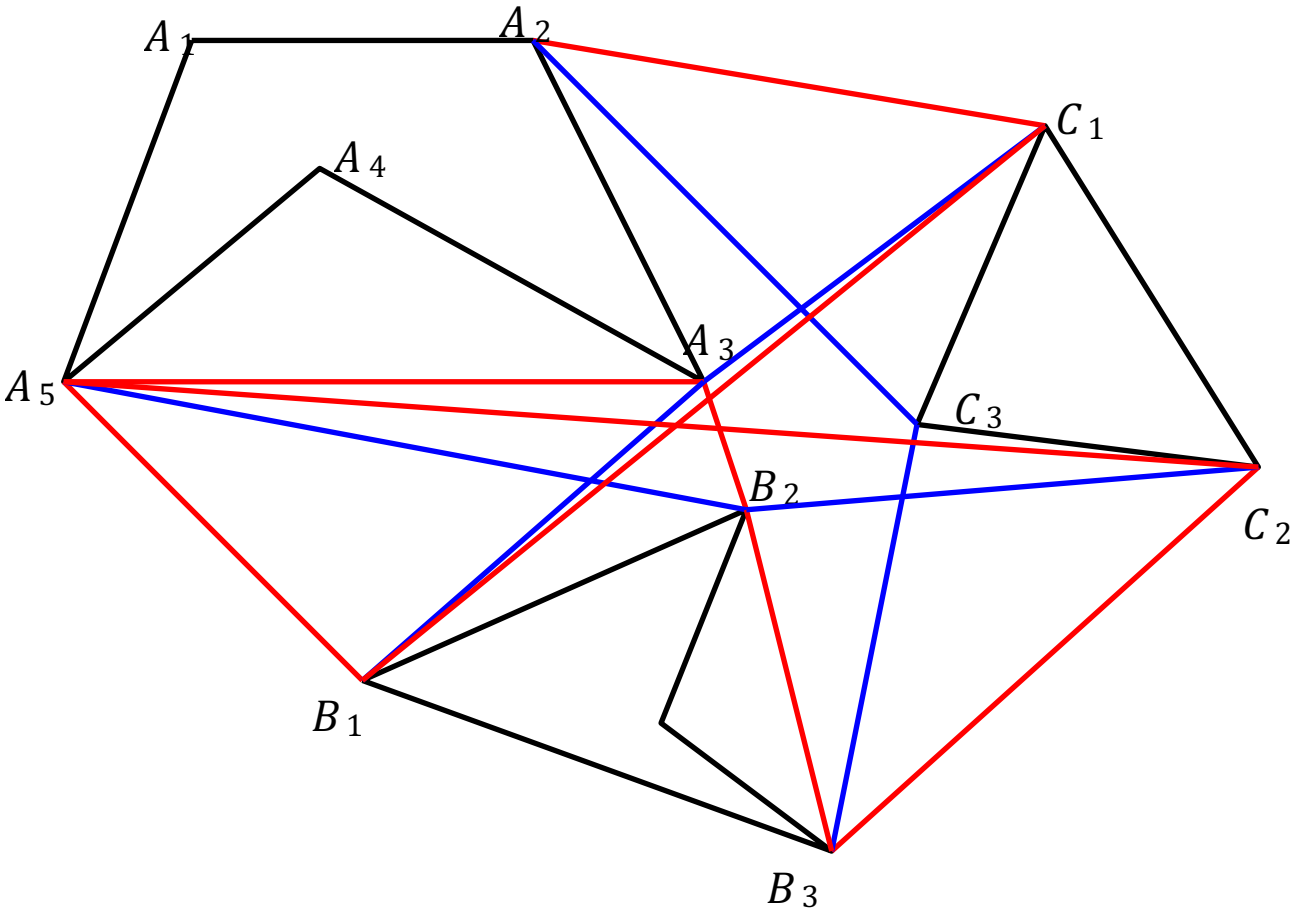

Figure 1: Q2(a) Reduced Visibility Graph

b) The connectivity graph is same as the Reduced Visibility graph above when we view each of the vertices as nodes of the graph. The black obstacle edges, red supporting and blue separating edges all together form the bidirectional connections in the connectivity graph.

c) An easy way to identify supporting of separating edge is as follows geometrically if it is in 2D space. First, take the pair of vertices, one each from each of the two obstacle vertex sets. Let them be P_i,P_j. Now, we compute the line through these two points. Then, we check if the equation of the line has the same sign for each of the other vertices in the obstacle sets. If all points have the same sign, it is a supporting line and if the sign is different for each vertex set, it is separating line. If the sign change is not same as vertex set partitioning, then it is neither supporting nor separating. Similar idea can apply by using planes in 3D space and 3D obstacles.

Problem 3: Cell Decomposition (15 points)

a) We have the trapezoidal decomposition and roadmap as follows. The dotted line is the roadmap.

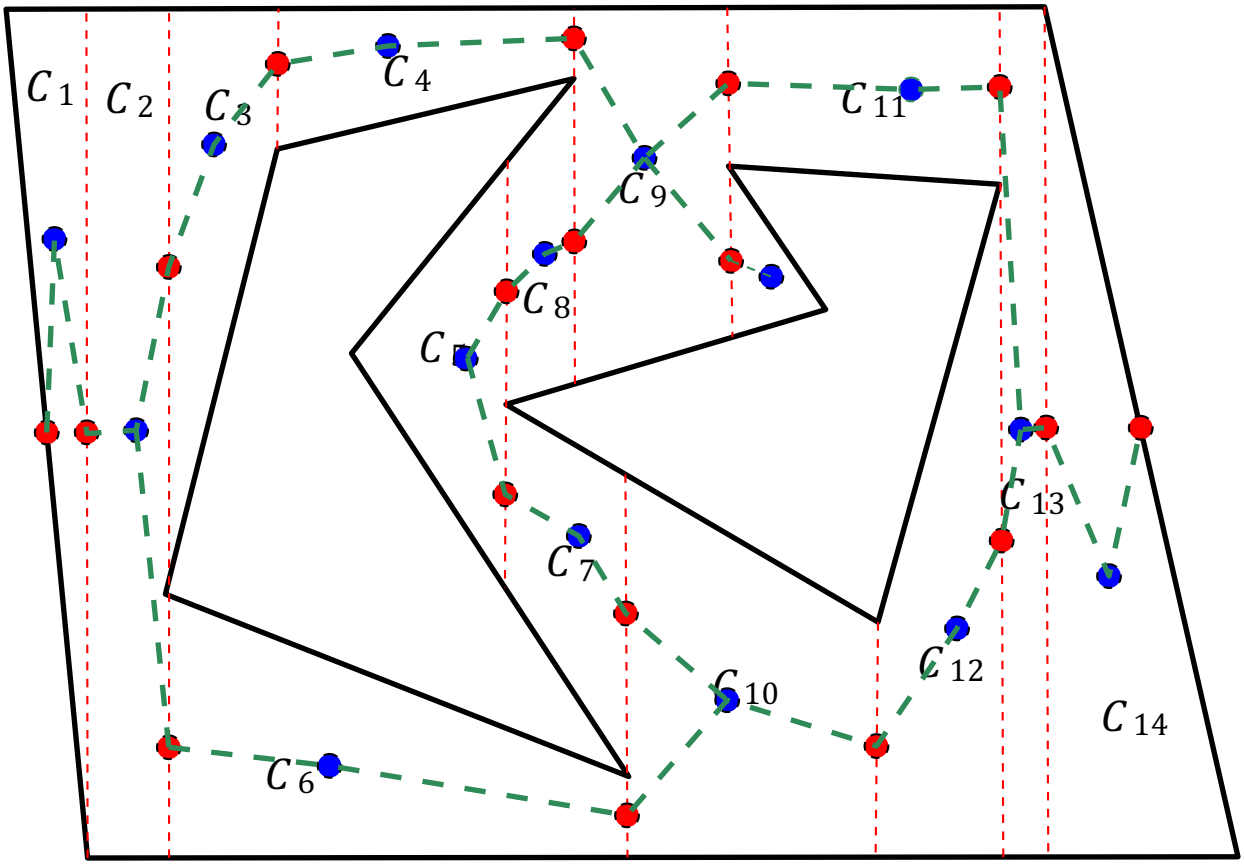

Figure 2: Q3(a) Trapezoidal decomposition and Roadmap

b) The Boustrophedon decomposition is as follows.

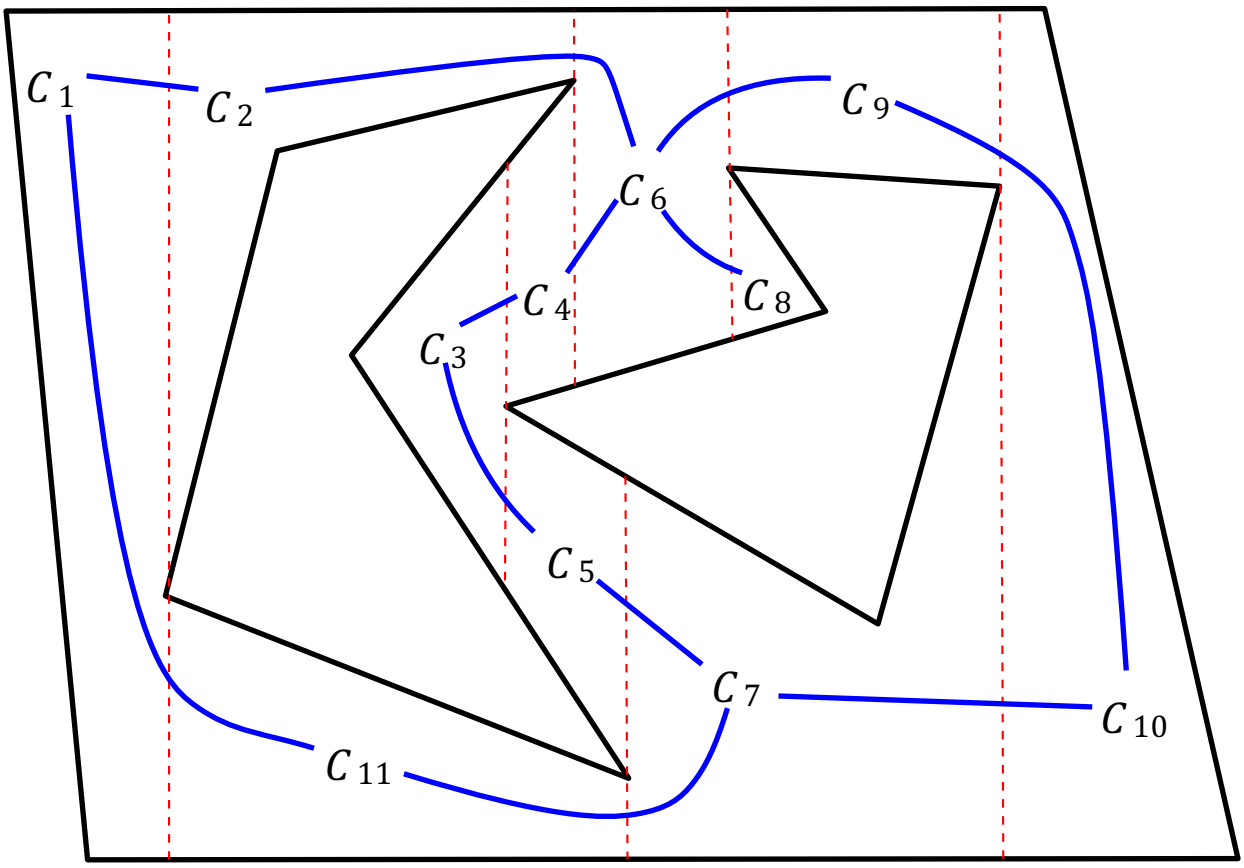

Figure 3: Q3(b) Boustrophedon decomposition and adjacency graph

c) For coverage, we have generated the adjacency graph using vertical slices. So, adjacency is of vertical slices. If we do long segment laps, it does not meet obstacle/environment boundary within a cell. And also, the number of long segments becomes too large making the search costly as turns and non-straight paths are costly and to be avoided.

Problem 4: Brushfire for Navigation

4.1 Brushfire on a Grid

The code is done in Q4.ipynb and the results for the 4 worlds are attached.

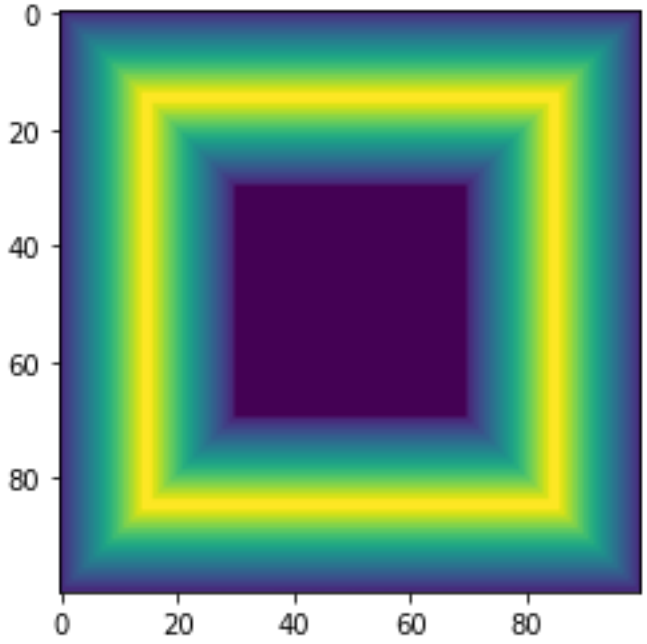

Figure 4: Q4(a) World 1

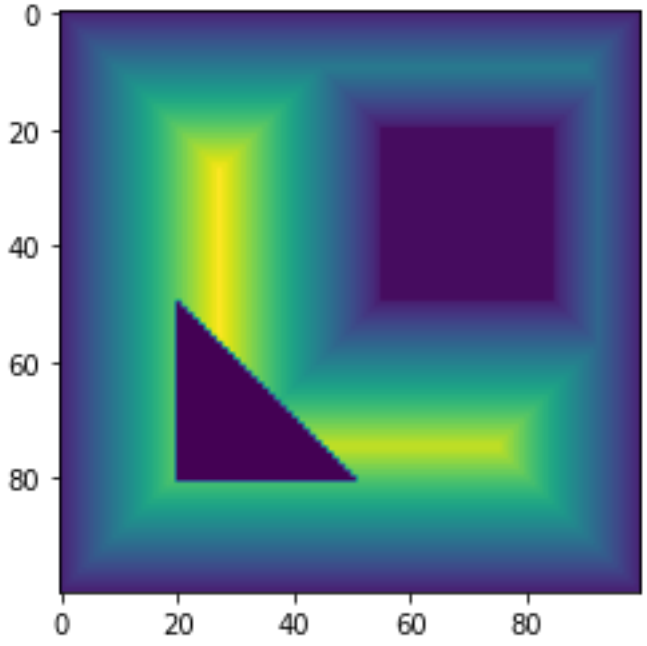

Figure 5: Q4(a) World 2

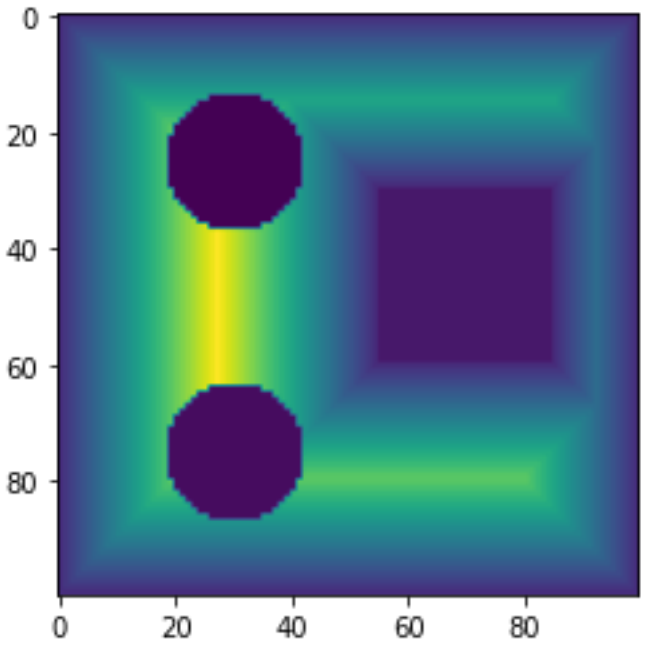

Figure 6: Q4(a) World 3

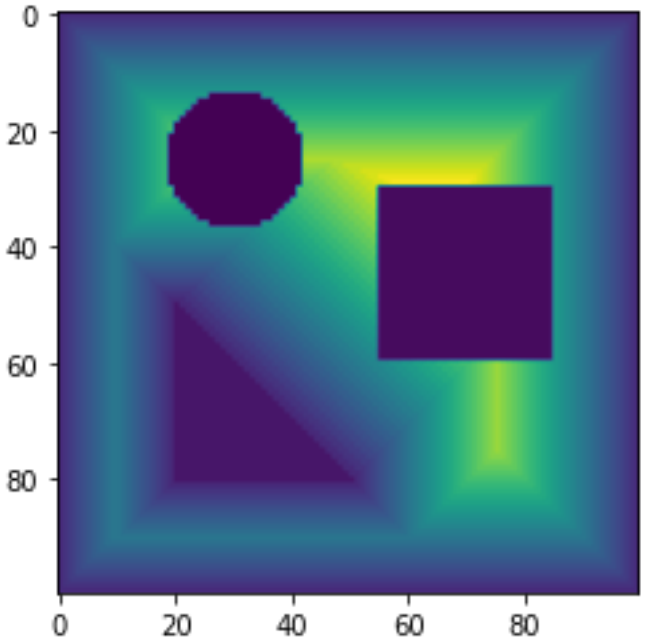

Figure 7: Q4(a) World 4

Appendix

Appendix 1:

Problem 4.1 Code

```
#%%

import numpy as np
import copy as cp
import matplotlib.pyplot as plt
wmapi = np.loadtxt('world2.txt')

#%%

wmapi
nRow = wmapi.shape[0]
nCol = wmapi.shape[1]
print(nRow,' rows and ', nCol, ' columns')

#%%

wmap = cp.deepcopy(wmapi)
iter = 0
print('Initialized world')
print(wmap)
while np.amax(wmap == 0.) == 1.:
    iter = iter + 1
    print('Iteration ',iter)
    wmap1 = cp.deepcopy(wmap)
    for i in np.arange(nRow):
        for j in np.arange(nCol):
            if wmap[i,j] == 0.:
                nbr_ridx = np.array([i-1,i-1,i-1,i,i,i+1,i+1,i+1])
                nbr_cidx = np.array([j-1,j,j+1,j-1,j+1,j-1,j,j+1])
                if
max(np.amax(nbr_ridx<0),np.amax(nbr_ridx>=nRow),np.amax(nbr_cidx<0),np.amax(nbr_cidx>=nCol)) ==
:
                    #print('Environment bounday detected')
                    wmap1[i,j] = 1.
                else:
                    if np.amin(wmap[np.ix_(nbr_ridx,nbr_cidx)]) == -1.:
                        #print('Obstacle detected')
                        wmap1[i,j] = 1.
                    elif np.amax(wmap[np.ix_(nbr_ridx,nbr_cidx)]) > 0.0:
                        #print('Distance increment detected')
                        wmap1[i,j] = np.amax(wmap[np.ix_(nbr_ridx,nbr_cidx)]) + 1.
                    #else:
                        #print(np.amin(wmap[np.ix_(nbr_ridx,nbr_cidx)])+1)
                        #print('Nothing detected')
    wmap = cp.deepcopy(wmap1)
    del wmap1
print('Brushfired world with distances')
print(wmap)

#%%

plt.imshow(wmap)

#%%
```