Question 1:

a)

def cost\_to\_go(state):

    #if it is not in the memo table, we will get the

    #cost to go for the previous state (which should already

    # be in the table) and add it with the new edge

    if state not in memo:

        memo[state] = cost\_to\_go(state[:len(state)-1]) \

            + cost\_to\_go(state[len(state)-2:])

    return memo[state]

#represent the graph as a 2D array edges, where

#edges[i] takes the form ['A', 'B', 2], where A and

#B are nodes and the undirected edge between them

#has a weight of 2

alpha = 0

edges = [['A', 'B', 2],

         ['A', 'C', 1],

         ['A', 'D', 1],

         ['B', 'C', alpha],

         ['B', 'D', 2],

         ['C', 'D', 2]]

#set up a memoization dictionary where each entry will

#represent the cost to go of a certain state

#e.g. entry 'AB' will be 2 and 'ABD' will be 4

memo = {}

#add entries for all of the edges

#add an entry for both ways to traverse the edge

#since the graph is undirected (e.g. 'AD' == 'DA')

for edge in edges:

    state = edge[0] + edge[1]

    altState = edge[1] + edge[0]

    memo[state] = edge[2]

    memo[altState] = edge[2]

#compute the cost to go for each state in figure 2

cost\_to\_go\_final\_states = {}

states = ['ABCDA', 'ABDCA', 'ACBDA', 'ACDBA', 'ADBCA', 'ADCBA']

for state in states:

    cost\_to\_go\_final\_states[state] = cost\_to\_go(state)

print(cost\_to\_go\_final\_states)

Text

Description automatically generated

b)

memo = {}

def cost\_to\_go(state):

    #if it is not in the memo table, we will get the

    #cost to go for the previous state (which should already

    # be in the table) and add it with the new edge

    if state not in memo:

        memo[state] = cost\_to\_go(state[:len(state)-1]) \

            + cost\_to\_go(state[len(state)-2:])

    return memo[state]

#modify the edge cost for alpha and return true if

#it makes the given route a lowest cost path

def optimal\_route(final\_state):

    for state in states:

        cost\_to\_go\_final\_states[state] = cost\_to\_go(state)

    cost\_final\_state = cost\_to\_go\_final\_states[final\_state]

    for cost in cost\_to\_go\_final\_states.values():

        if cost < cost\_final\_state:

            return False

    return True

#represent the graph as a 2D array edges, where

#edges[i] takes the form ['A', 'B', 2], where A and

#B are nodes and the undirected edge between them

#has a weight of 2

#reset the memo table with edge weights based on alpha

def set\_memo\_table(alpha):

    edges = [['A', 'B', 2],

             ['A', 'C', 1],

             ['A', 'D', 1],

             ['B', 'C', alpha],

             ['B', 'D', 2],

             ['C', 'D', 2]]

    #set up a memoization dictionary where each entry will

    #represent the cost to go of a certain state

    #e.g. entry 'AB' will be 2 and 'ABD' will be 4

    memo = {}

    #add entries for all of the edges

    #add an entry for both ways to traverse the edge

    #since the graph is undirected (e.g. 'AD' == 'DA')

    for edge in edges:

        state = edge[0] + edge[1]

        altState = edge[1] + edge[0]

        memo[state] = edge[2]

        memo[altState] = edge[2]

    return memo

#compute the cost to go for each state in figure 2

cost\_to\_go\_final\_states = {}

states = ['ABCDA', 'ABDCA', 'ACBDA', 'ACDBA', 'ADBCA', 'ADCBA']

#for each final state, get the optimal interval of alpha

#as a two element array [start, finish]

#if thesre is no interval, enter the empty array []

#also cap the interval at 10 as the alpha value will not

#improve the interval after that (so the interval

# [0, 10] is equivalent to [0, infinity))

optimal\_alpha\_intervals = {}

for state in states:

    start = -1

    finish = -1

    alpha = 0

    while True:

        memo = set\_memo\_table(alpha)

        if start == -1 and optimal\_route(state):

            start = alpha

        elif start != -1 and not optimal\_route(state):

            finish = alpha - 1

            break

        alpha += 1

        #just cut off alpha at a point because it will not make

        #the path more optimal

        if alpha >= 10:

            finish = alpha

            break

    interval = [] if start == -1 or finish == -1 else [start, finish]

    optimal\_alpha\_intervals[state] = interval

print(optimal\_alpha\_intervals)



I set it so that the interval gets cut off at 10 because alpha will not improve after this.

Question 2:

a)

import heapdict

import math

from PIL import Image

import numpy as np

from matplotlib import pyplot as plt

'''a function that takes as input the start state s, goal state g, and

populated predecessor map pred, and returns the sequence of vertices on the optimal

path from s to g.'''

def RecoverPath(s, g, pred):

    curr = g

    result = []

    while(curr != s):

        result.insert(0, curr)

        curr = pred[curr]

    result.insert(0,s)

    return result

#do an A\* search algorithm and return an array representing the list of

#vertices along the optimal path to the goal g from the start s

#return the empty list if there was no optimal path

def A\_STAR\_SEARCH(V, s, g, N, w, h):

    CostTo = {} #cost (value) to get from the start node s to v (key)

    pred = {} #the previous node (value) from the target node v (key) on the shortest path

    '''is a map that assigns to each vertex v (key) the sum CostTo(v) + h(v,g) (value),

    the sum of the cost of the best known path to v and the predicted cost of the best

    path from v to the goal g; this is the estimated cost of the optimal path from the start

    s to the goal g that passes through vertex v.'''

    EstTotalCost = {}

    Q = heapdict.heapdict() #a priority queue in which elements with lower values are removed first.

    for v in V:

        CostTo[v] = math.inf

        EstTotalCost[v] = math.inf

    CostTo[s] = 0

    EstTotalCost[s] = h(s,g)

    Q[s] = h(s,g)

    while len(Q) != 0:

        v = Q.popitem()

        v = v[0] #get the vertex without the priority

        if v == g:

            return RecoverPath(s, g, pred)

        for i in N(v):

            pvi = CostTo[v] + w(v,i)

            if pvi < CostTo[i]:

                pred[i] = v

                CostTo[i] = pvi

                EstTotalCost[i] = pvi + h(i,g)

                Q[i] = EstTotalCost[i]

    return []

b)

Code:

import heapdict

import math

from PIL import Image

import numpy as np

from matplotlib import pyplot as plt

'''a function that takes as input the start state s, goal state g, and

populated predecessor map pred, and returns the sequence of vertices on the optimal

path from s to g.'''

def RecoverPath(s, g, pred):

    curr = g

    result = []

    while(curr != s):

        result.insert(0, curr)

        curr = pred[curr]

    result.insert(0,s)

    return result

#do an A\* search algorithm and return an array representing the list of

#vertices along the optimal path to the goal g from the start s

#return the empty list if there was no optimal path

def A\_STAR\_SEARCH(V, s, g, N, w, h):

    CostTo = {} #cost (value) to get from the start node s to v (key)

    pred = {} #the previous node (value) from the target node v (key) on the shortest path

    '''is a map that assigns to each vertex v (key) the sum CostTo(v) + h(v,g) (value),

    the sum of the cost of the best known path to v and the predicted cost of the best

    path from v to the goal g; this is the estimated cost of the optimal path from the start

    s to the goal g that passes through vertex v.'''

    EstTotalCost = {}

    Q = heapdict.heapdict() #a priority queue in which elements with lower values are removed first.

    for v in V:

        CostTo[v] = math.inf

        EstTotalCost[v] = math.inf

    CostTo[s] = 0

    EstTotalCost[s] = h(s,g)

    Q[s] = h(s,g)

    while len(Q) != 0:

        v = Q.popitem()

        v = v[0] #get the vertex without the priority

        if v == g:

            return RecoverPath(s, g, pred)

        for i in N(v):

            pvi = CostTo[v] + w(v,i)

            if pvi < CostTo[i]:

                pred[i] = v

                CostTo[i] = pvi

                EstTotalCost[i] = pvi + h(i,g)

                Q[i] = EstTotalCost[i]

    return []

# Read image from disk using PIL

occupancy\_map\_img = Image.open('occupancy\_map.png')

# Interpret this image as a numpy array, and threshold its values to → {0,1}

M = (np.asarray(occupancy\_map\_img) > 0).astype(int)

def N(v):

    row = v[0]

    col = v[1]

    neighbors = [(row-1,col-1),

                 (row-1, col),

                 (row-1,col+1),

                 (row,col-1),

                 (row,col+1),

                 (row+1,col-1),

                 (row+1,col),

                 (row+1,col+1)]

    unoccupied\_neighbors = []

    for n in neighbors:

        #make sure our original vertex was not on an edge or corner

        if n[0] < 0 or n[0] >= len(M) or n[1] < 0 or n[1] >= len(M[0]):

            continue

        #only add unoccupied neighbors

        if M[n[0]][n[1]]:

            unoccupied\_neighbors.append(n)

    return unoccupied\_neighbors

#euclidean distance between two vectors v1 and v2

def d(v1, v2):

    x1, y1, x2, y2 = v1[0], v1[1], v2[0], v2[1]

    return math.sqrt((x2-x1)\*\*2 + (y2-y1)\*\*2)

def totalPathLength(pathArr):

    totalLen = 0

    for i in range(len(pathArr)-1):

        totalLen += d(pathArr[i], pathArr[i+1])

    return totalLen

#get the vertex set V from the occupancy grid

V = set()

for i in range(len(M)):

    for j in range(len(M[0])):

        V.add((i,j))

opt\_path = A\_STAR\_SEARCH(V, (635,140), (350,400), N, d, d)

print(totalPathLength(opt\_path))

xs, ys = [], []

for i in opt\_path:

    xs.append(i[1])

    ys.append(i[0])

plt.plot(xs, ys)

plt.imshow(occupancy\_map\_img)

plt.show()

Plot with line added:

A picture containing chart

Description automatically generated

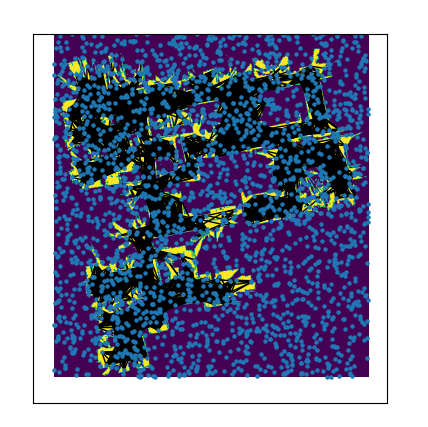
Total Length:

Text

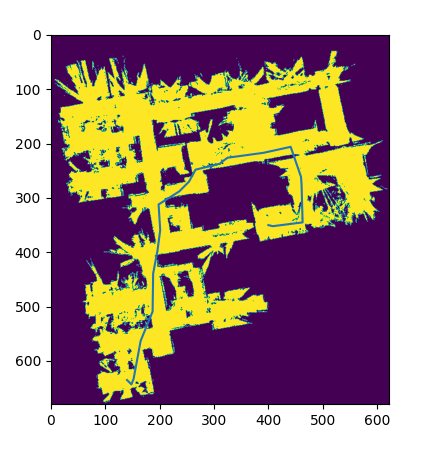
Description automatically generated

c)

PRM Plot for iv:



Optimal Path Plot for v:



Code:

import random, math

from PIL import Image

import numpy as np

import networkx as nx

from matplotlib import pyplot as plt

#sample a vertex uniformly and randomly from

# the vertex set of the of the occupancy map M

def sampleVertex():

    #get the vertex set V from the occupancy grid

    V = []

    for i in range(len(M)):

        for j in range(len(M[0])):

            V.append((i,j))

    return random.choice(V)

#euclidean distance between two vectors v1 and v2

def d(v1, v2):

    x1, y1, x2, y2 = v1[0], v1[1], v2[0], v2[1]

    return math.sqrt((x2-x1)\*\*2 + (y2-y1)\*\*2)

#return the neighbor of v1 with the lowest distance to v2

def lowestDistNeighbor(v1, v2):

    row = v1[0]

    col = v1[1]

    neighbors = [(row-1,col-1),

                 (row-1, col),

                 (row-1,col+1),

                 (row,col-1),

                 (row,col+1),

                 (row+1,col-1),

                 (row+1,col),

                 (row+1,col+1)]

    valid\_neighbors = []

    for n in neighbors:

        #make sure our original vertex was not on an edge or corner

        if not(n[0] < 0 or n[0] > len(M) or n[1] < 0 or n[1] > len(M[0])):

            valid\_neighbors.append(n)

    min\_distance = math.inf

    min\_dist\_v = ()

    for vn in valid\_neighbors:

        dist = d(vn, v2)

        if dist < min\_distance:

            min\_distance = dist

            min\_dist\_v = vn

    return min\_dist\_v

#returns a boolean representing if v2 is reachable from

# v1 by a straight line path through the occupancy grid

def reachabiliyCheck(v1, v2):

    curr = v1

    while curr != v2:

        next = lowestDistNeighbor(curr, v2)

        nextRow = next[0]

        nextCol = next[1]

        if not M[nextRow][nextCol]:

            return False

        curr = next

    return True

def AddVertex(G, vNew, dMax):

    G.add\_node(vNew)

    for v in G.nodes:

        dist = d(v, vNew)

        if v != vNew and dist <= dMax and reachabiliyCheck(v, vNew):

            G.add\_edge(v, vNew, weight=dist)

#return a probabalistic road map corresponding to the occupancy grid M,

#number of desired samples N, and a maximum search radius dMax

def ConstructPRM(N, dMax):

    G = nx.Graph()

    for k in range(N):

        print(k)

        v = sampleVertex()

        AddVertex(G, v, dMax)

    return G

def totalPathLength(pathArr):

    totalLen = 0

    for i in range(len(pathArr)-1):

        totalLen += d(pathArr[i], pathArr[i+1])

    return totalLen

# Read image from disk using PIL

occupancy\_map\_img = Image.open(r'C:\Users\Shaun\College\Fall 2022\Mobile\_Robotics\Robotics\hw3\occupancy\_map.png')

# Interpret this image as a numpy array, and threshold its values to → {0,1}

M = (np.asarray(occupancy\_map\_img) > 0).astype(int)

'''

#How the graph functions work

# Create empty graph

G = nx.Graph()

# Add vertex v1 at position (r1, c1)

G.add\_node((0,0))

# Add vertex v2 at position (r2, c2)

G.add\_node((2,2))

# Add an edge between v1 and v2 with weight w12

G.add\_edge((0,0), (2,2), weight=10)

print(G.nodes)

'''

G = ConstructPRM(2500, 75)

posDict = {}

for v in G.nodes:

    posDict[v] = (v[1], v[0]) #switch the row and column for representation on image

#plt.imshow(occupancy\_map\_img)

#nx.draw\_networkx(G, posDict, node\_size=5, with\_labels=False)

#plt.show()

G.add\_node((635,140))

G.add\_node((350,400))

opt\_path = []

while opt\_path == []:

    try:

        opt\_path = nx.astar\_path(G, (635,140), (350,400), d)

    except nx.NetworkXNoPath:

        AddVertex(G,sampleVertex(), 75)

print(opt\_path)

print(totalPathLength(opt\_path))

xs, ys = [], []

for i in opt\_path:

    xs.append(i[1])

    ys.append(i[0])

plt.plot(xs, ys)

plt.imshow(occupancy\_map\_img)

plt.show()

Code output:

