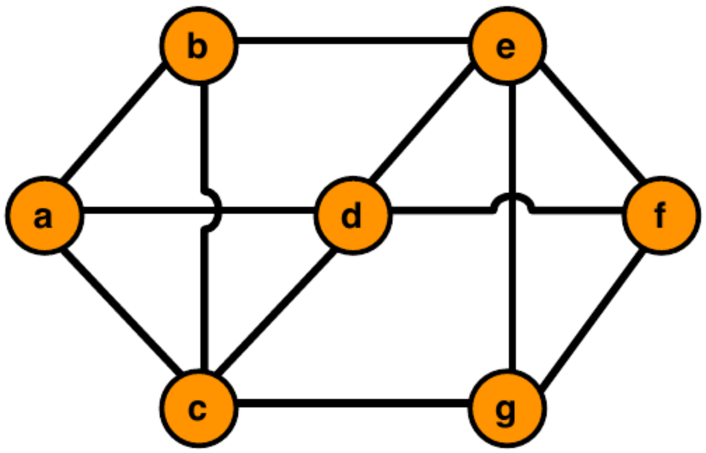
CSCI 3104: Algorithms **Instructor Buxton**
Problem Set 7 (110 total points) **Summer 2019, CU-Boulder**

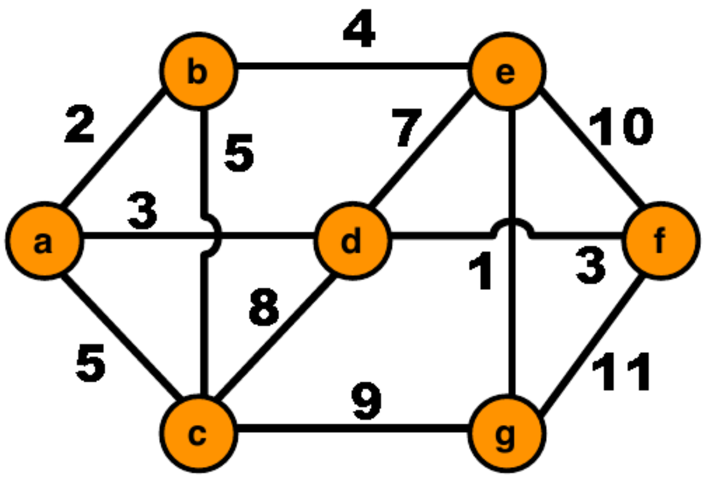
1. (30 points) Grog gives you the following unweighted graph and asks you to construct a weight function w on the edges, using positive integer weights only, such that the following conditions are true regarding minimum spanning trees (MST) and single-source shortest path trees (SSSP):

   - The MST is distinct from any of the seven SSSP trees.
   - The order in which Jarník/Prim's algorithm adds the safe edges is different from the order in which Kruskal's algorithm adds them.

   Justify your solution by (i) giving the edges weights, (ii) showing the corresponding MST and all the SSSP trees, and (iii) giving the order in which edges are added by each of the two algorithms.

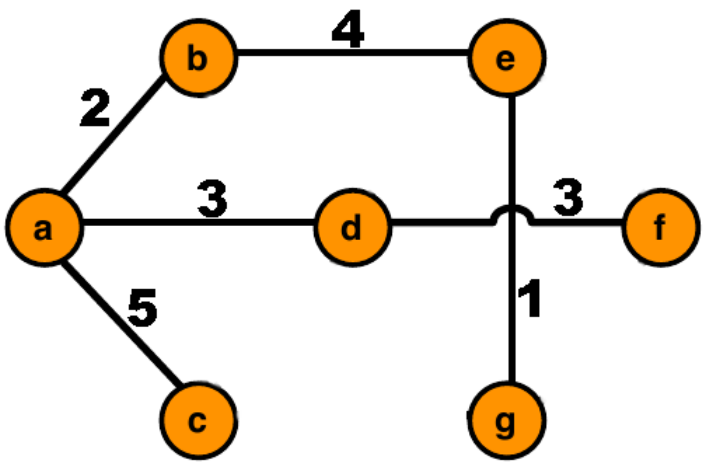


(i):



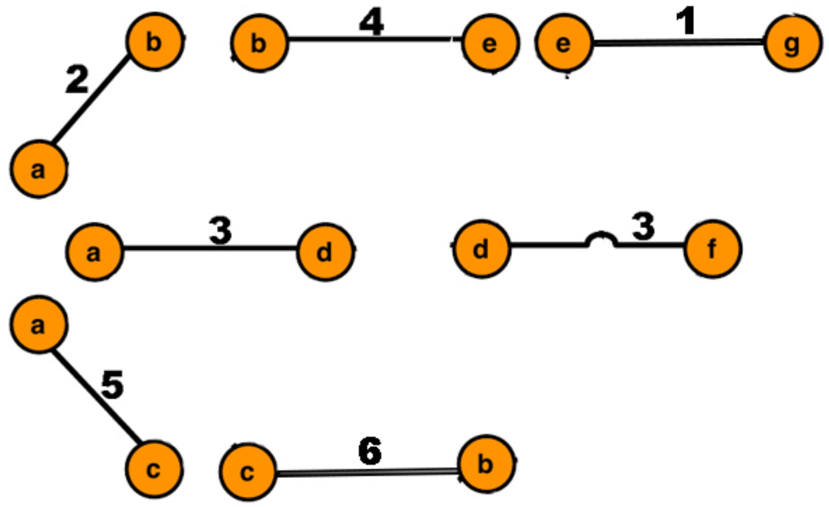
(ii):

MST:



$2 + 4 + 3 + 5 + 3 + 1 = 18$ (MST Total Weight)

SSSP:



(iii):

Prim's: (a, b), (a, d), (d, f), (b, e), (e, g), (a, c)

Kriskals: (e, g), (a, b), (a, d), (d, f), (b, e), (a, c)

2. (25 points) Harry and Shadow think they have come up with a way to get rich by exploiting the ore market. Their idea is to exploit exchange rates in order to transform one unit of gold into more than one unit of gold, through a sequence of exchanges. For instance, suppose 1 unit of gold buys 0.82 silver units, 1 silver unit buys 129.7 ethereum units, 1 ethereum unit buys 12 doge units, and finally 1 doge unit buys 0.0008 gold units. By converting their loot, Harry and Shadow think they could start with 1 gold unit and buy $0.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02$ gold units, thereby making a 2% profit! The problem is that those dwarves at Rocky Mountain Bank charge a transaction cost for each exchange.

   Suppose that Harry and Shadow start with knowledge of n ores $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of ore c_i buys $R[i, j]$ units of ore c_j. A traditional *arbitrage opportunity* is thus a cycle in the induced graph such that the product of the edge weights is greater than unity. That is, a sequence of ores $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that $R[i_1, i_2] \times R[i_2, i_3] \times \cdots \times R[i_{k-1}, i_k] \times R[i_k, i_1] > 1$. Each transaction, however, must pay Rocky Mountain Bank a fraction α of the total transaction value, e.g., $\alpha = 0.01$ for a 1% rate.

   (a) When given R and α, give an efficient algorithm that can determine if an arbitrage opportunity exists. Analyze the running time of your algorithm.

   Thormund's hint: It is possible to solve this problem in $O(n^3)$.

   (b) For an arbitrary R, explain how varying α changes the set of arbitrage opportunities that exist and that your algorithm might identify.

(a):

The sum of logs is equal to the log of our product. If we want positive weight cycles, and the Bellman-Ford algorithm identifies negative weight cycles. We want to include the transaction commission in each exchange rate. Then the algorithm finally is to simply compute the negative logarithm of (each exchange rate multiplied by (1-a)) and run Bellman-Ford to identify any resulting negative weights. $(-log_a + -log_b + -log_c = -(log_a + log_b + log_c) = -(log_{abc}))$. Because $-log_{abc}$ is negative, log_{abc} is positive, and abc is greater than one minus what we are looking for. This preprocessing set is $\theta(n^2)$ time, and is dominated by Bellman-Ford, so overall its in the same time as Bellman-Ford: $O(n^3)$.

(b):

A lower a would result in more or the same number of arbitrage opportunities. Higher values of a lead to fewer or equal arbitrage opportunities, because the commissions can end up costing more than whats gained.

3. (55 points) Bidirectional breadth-first search is a variant of standard BFS for finding a shortest path between two vertices $s, t \in V(G)$. The idea is to run *two* breadth-first searches simultaneously, one starting from s and one starting from t, and stop when they "meet in the middle" (that is, whenever a vertex is encountered by both searches). "Simultaneously" here doesn't assume you have multiple processors at your disposal; it's enough to alternate iterations of the searches: one iteration of the loop for the BFS that started at s and one iteration of the loop for the BFS that started at t.

   As we'll see, although the worst-case running time of BFS and Bidirectional BFS are asymptotically the same, in practice Bidirectional BFS often performs significantly better.

   Throughout this problem, all graphs are unweighted, undirected, simple graphs.

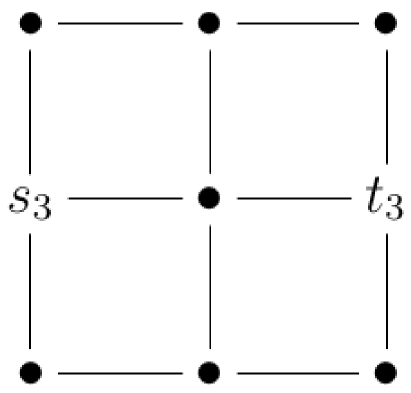
   (a) Implement from scratch a function `BFS(G,s,t)` that performs an ordinary BFS in the (unweighted, directed) graph G to find a shortest path from s to t. Assume the graph is given as an adjacency list; for the list of neighbors of each vertex, you may use any data structure you like (including those provided in standard language libraries). Have your function return a pair (d, k), where d is the distance from s to t (-1 if there is no s to t path), and k is the number of nodes popped off the queue during the entire run of the algorithm.

   (b) Implement from scratch a function `BidirectionalBFS(G,s,t)` that takes in an unweighted, directed graph G, and two of its vertices s, t, and performs a bidirectional BFS. As with the previous function, this function should return a pair (d, k) where d is the distance from s to t (-1 if there is no path from s to t) and k is the number of vertices popped off of both queues during the entire run of the algorithm.

   (c) For each of the following families of graphs G_n, write code to execute `BFS` and `BidirectionalBFS` on these graphs, and produce the following output:

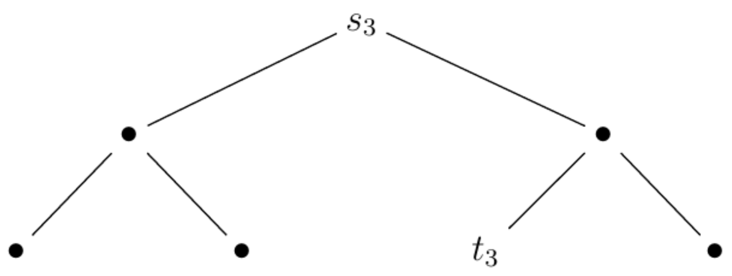
   - In text, the pairs (n, d_1, k_1, d_2, k_2) where n is the index of the graph, (d_1, k_1) is the output of `BFS` and (d_2, k_2) is the output of `BidirectionalBFS`.
   - a plot with n on the x-axis, k on the y-axis, and with two line charts, one for the values of k_1 and one for the values of k_2:

   i. Grids. G_n is an $n \times n$ grid, where each vertex is connected to its neighbors in the four cardinal directions (N,S,E,W). Vertices on the boundary of the grid will only have 3 neighbors, and corners will only have 2 neighbors. Let s_n be the midpoint of one edge of the grid, and t_n the midpoint of the opposite edge. For example, for $n = 3$ we have:



(When n is even s_n and t_n can be either "midpoint," since there are two.) Produce output for $n = 3, 4, 5, \ldots, 20$.

ii. Trees. G_n is a complete binary tree of depth n. s_n is the root and t_n is any leaf. Produce output for $n = 3, 4, 5, \ldots, 15$. For example, for $n = 3$ we have:

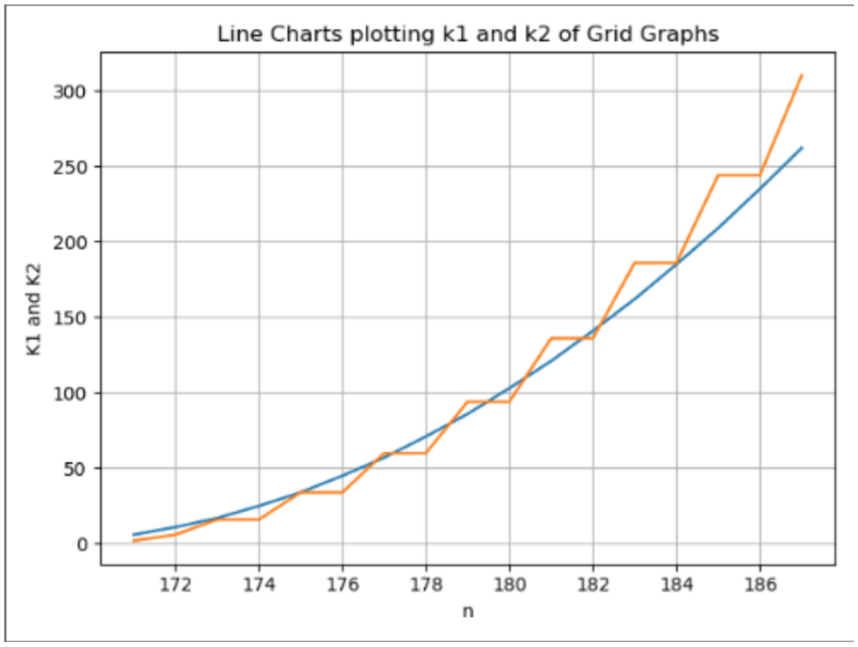


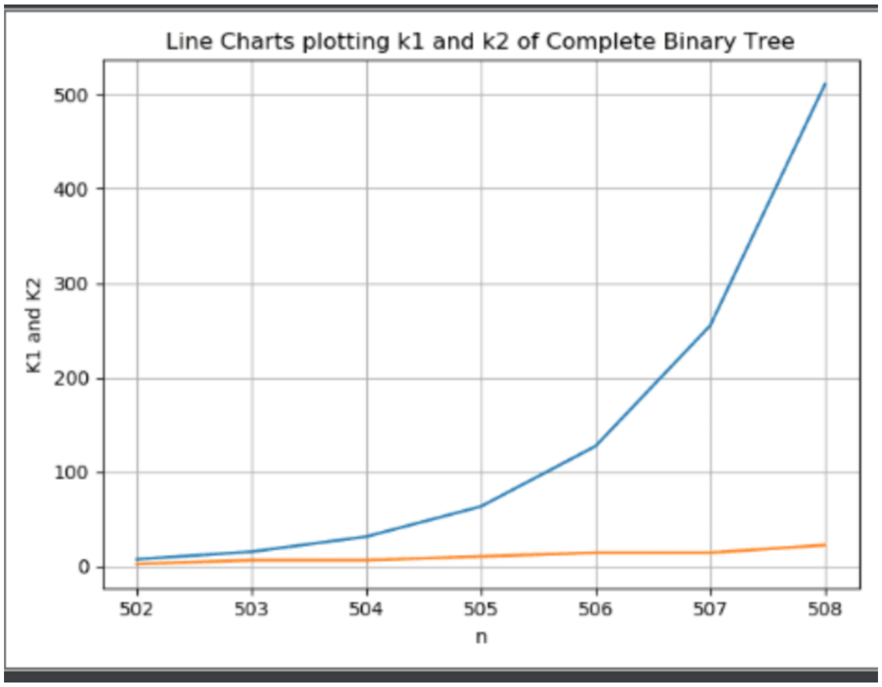
iii. Random graphs. G_n is a graph on n vertices constructed as follows. For each pair of of vertices (i, j), get a random boolean value; if it is `true`, include the edge (i, j), otherwise do not. Let s_n be vertex 1 and t_n be vertex 2 (food for thought: why does it not matter, on average, which vertices we take s, t to be?) For each n, produce 50 such random graphs and report just the average values of (d_1, k_1, d_2, k_2) over those 50 trials. Produce this output for $n = 3, 4, 5, \ldots, 20$.

(a) and (b): python

```
import queue
import matplotlib.pyplot as plt
import networkx as nx
import scipy as sp


def generate_grid_graph(n):
    graph = nx.grid_2d_graph(n, n)

    s = int(n/2)*n
    t = int(n/2+1)*n-1
    result = list()
    result.append(s)
    result.append(t)
    result.append(graph)
    return result


def generate_complete_binary_tree(h):
    graph = nx.balanced_tree(2, h-1)
    #nx.draw(graph)
    #plt.show()
    s = 0
    t = 2**h - 2
    result = list()
    result.append(s)
    result.append(t)
    result.append(graph)
    return result


def adj_matrix_to_list(matrix, n):
    vertices = n
    adj_list = dict()
    for i in range(0, vertices):
        for j in range(0, vertices):
            listt = list()
            if matrix[i][j] == 1:
                if i in adj_list:
                    listt = adj_list.get(i)
                    listt.append(j)
                    adj_list.update({i:listt})
                else:
                    listt.append(j)
                    adj_list.update({i:listt})
    return adj_list


def reverse_the_list(listt):
    new_listt = list()
    length = len(listt)
    count = length-1
    while count >= 0:
        new_listt.append(listt[count])
        count-=1

    return new_listt


def BFS(G, s, t):
    sparse_matrix = nx.adjacency_matrix(G)
    matrix = sparse_matrix.toarray()
    size = len(matrix)
    adj_list = adj_matrix_to_list(matrix,size)
    mainQueue = queue.Queue(1000)
    visited = list()
    visited.append(s)
    value = adj_list.get(s)
    parent_map = dict()
    for i in range(0, len(value)):
        mainQueue.put(value[i])
        parent_map.update({value[i]:s})
    while not mainQueue.empty():
        parent = mainQueue.get()
        visited.append(parent)
        if parent is not t:
            values = adj_list.get(parent)
            for i in range(0, len(values)):
                if values[i] not in visited:
                    if values[i] not in mainQueue.queue:
                        parent_map.update({values[i]:parent})
                        mainQueue.put(values[i])
        else:
            break

    curr = t
    distance = list()
    distance.append(t)


    while curr != s and curr != None:
        parent = parent_map.get(curr)
        distance.append(parent)
        curr = parent
    return [len(distance), len(visited)]


def bidirectional_bfs(G, s, t):
    sparse_matrix = nx.adjacency_matrix(G)
    matrix = sparse_matrix.toarray()
    size = len(matrix)
    adj_list = adj_matrix_to_list(matrix, size)

    StartingQueue = queue.Queue(1000)
    GoalQueue = queue.Queue(1000)

    visited1 = list()
    visited2 = list()
    map1 = dict()
    map2 = dict()
    temp = 0

    visited1.append(s)
    value = adj_list.get(s)
    for i in range(0, len(value)):
        StartingQueue.put(value[i])
        map1.update({value[i]: s})


    visited2.append(t)
    value = adj_list.get(t)
    for i in range(0, len(value)):
        GoalQueue.put(value[i])
        map2.update({value[i]: t})
        if value[i] in StartingQueue.queue:
            temp = value[i]

    if temp == 0:

        check = 0
        while not StartingQueue.empty() and not GoalQueue.empty():
            p1 = StartingQueue.get()
            visited1.append(p1)

            if p1 is not t and p1 not in StartingQueue.queue and p1 not in GoalQueue.queue:
                values = adj_list.get(p1)
                for i in range(0, len(values)):
                    if values[i] not in visited1 and values[i] not in visited2:
                        map1.update({values[i]: p1})
                        StartingQueue.put(values[i])

                    elif values[i] in visited2:
                        check = 1
                        temp = values[i]
                    elif p1 in visited2:
                        temp = p1
                        check = 1

            elif p1 in GoalQueue.queue:
                check = 1
                temp = p1

            p2 = GoalQueue.get()
            visited2.append(p2)
            if p2 is not t and p2 not in StartingQueue.queue and p2 not in GoalQueue.queue:
                values = adj_list.get(p2)
                for i in range(0, len(values)):
                    if values[i] not in visited1 and values[i] not in visited2 :
                        map2.update({values[i]: p2})
                        GoalQueue.put(values[i])

                    elif values[i] in visited1:
                        check = 1
                        temp = values[i]
                    elif p2 in visited1:
                        check = 1
                        temp = p2


            elif p2 in StartingQueue.queue:
                check = 1
                temp = p2

            if check == 1:
                break
    curr = temp
    distance1 = list()
    distance1.append(curr)
    while curr != s:
        if curr in map1:
            parent = map1.get(curr)
            distance1.append(parent)
            curr = parent
        else:
            break

    curr = temp
    distance2 = list()
    while curr != t:
        if curr in map2:
            parent = map2.get(curr)
            distance2.append(parent)
            curr = parent
        else:
            break
    part1 = reverse_the_list(distance1)
    final_distance = part1+distance2
    return [len(final_distance), len(visited1+visited2)]

from pylab import *


def line_charts(range_start, range_end , k1s, k2s , graph_name):

    if graph_name == "Complete Binary Tree":

        t = arange(range_start+502, range_end-1, 1)
    if graph_name == "Grid Graphs":
        t = arange(range_start , range_end - 1, 1)

    if graph_name == "Random Graph":
        t = arange(range_start, range_end - 1, 1)

    plot(t,k1s)
    plot(t,k2s)
    xlabel('n')
    ylabel('K1 and K2')
    title('Line Charts plotting k1 and k2 of '+ graph_name)
    grid(True)
    show()


def print_grid_graph_result(start, end):
    print("For grid graphs : ")
    k1s = list()
    k2s = list()
    for i in range(start, end):
        res = generate_grid_graph(i)
        start = res[0]
        end = res[1]
        graph = res[2]
        d1, k1 = BFS(graph, start, end)
        d2, k2 = bidirectional_bfs(graph, start, end)
        k1s.append(k1)
        k2s.append(k2)
        print("For n: ", i)
        print("d1:" +str(d1)+ ", k1:"+str(k1)+", d2:"+str(d2)+", k2:"+str(k2))
    line_charts(start, end, k1s, k2s, "Grid Graphs")


def print_complete_binary_tree_graph_result(start, end):
    k1s = list()
    k2s = list()
    print("For grid graphs : ")
    for i in range(start, end):
        res = generate_complete_binary_tree(i)
        start = res[0]
        end = res[1]
        graph = res[2]
        d1, k1 = BFS(graph, start, end)
        d2, k2 = bidirectional_bfs(graph, start, end)
        k1s.append(k1)
        k2s.append(k2)

        print("For n: ", i)
        print("d1:" + str(d1) + ", k1:" + str(k1) + ", d2:" + str(d2) + ", k2:" + str(k2))
    line_charts(start, end, k1s, k2s, "Complete Binary Tree")


import random


def generate_random_graph(n):

    G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
    for i in range(0, n):
        for j in range(0, n):
            if random.randint(0, 1) == 0:
                G.add_edge(i, j)
    result = list()
    result.append(0)
    result.append(n-1)
    result.append(G)
    return result



def print_random_graphs(start, end):
    k1s = list()
    k2s = list()
    print("For random graphs : ")
    for i in range(start, end):
        sumd1 = 0
        sumd2 = 0
        sumk1 = 0
        sumk2 = 0
        for j in range(0, 50):
            res = generate_random_graph(i)
            start = res[0]
            end = res[1]
            graph = res[2]
            d1, k1 = BFS(graph, start, end)
            d2, k2 = bidirectional_bfs(graph, start, end)
            sumk1 = sumk1 + k1
            sumk2 = sumk2 + k2
            sumd1 = sumd1+ d1
            sumd2 = sumd2 + d2
        avr_k1 = sumk1/50
        avr_k2 = sumk2/50
        avr_d1 = sumd1/50
        avr_d2 = sumd2/50
    k1s.append(avr_k1)
    k2s.append(avr_k2)
    print("For n: ", i)
    print("d1:" + str(avr_d1) + ", k1:" + str(avr_k1) + ", d2:" + str(avr_d2) + ", k2:" + str(avr_k2))
    line_charts(start, end, k1s , k2s, "Random Graph")


#print_grid_graph_result(3,20)
#print_complete_binary_tree_graph_result(3,10)
#print_random_graphs(3, 20)
```

(c):




Line Charts plotting k1 and k2 of Complete Binary Tree
K1 and K2
0
100
200
300
400
500
502
503
504
505
506
507
508
n