CSCI 3104 Spring 2018 Problem Set 9
Mohamed Al-Rasbi (07/25)

Problem 1

(10 pts) Let $G = (V, E)$ be a graph with an edge-weight function w, and let the tree $T \subseteq E$ be a minimum spanning tree on G. Now, suppose that we modify G slightly by decreasing the weight of exactly one of the edges in $(x, y) \in T$ in order to produce a new graph G'. Here, you will prove that the original tree T is still a minimum spanning tree for the modified graph G'.

To get started, let k be a positive number and define the weight function w' as

$$w'(u,v) = \begin{cases} w(u,v) & \text{if } (u,v) \neq (x,y) \\ w(u,v) - k & \text{if } (u,v) = (x,y) \end{cases}$$

Now, prove that the tree T is a minimum spanning tree for G', whose edge weights are given by w'.

When we want to find the MST, we have to find the safe edges. Those safe edges construct the MST path. Therefore, when we change one edge at the graph, there are two cases.

Case 1: The reduced edge is a safe edge. In this case we will get the same MST because a safe edge has the minimum value to a certain vertex. Reducing the minimum value will no change anything about the relativity of the edges to each other, that minimum value will still be the minimum.

Case 2: The reduced edge is a useless edge. In this case, we can either get the same MST or a different one. We will get the same MST in case the reduced edge remains useless. And we will get different MST in case the useless edge becomes safe.

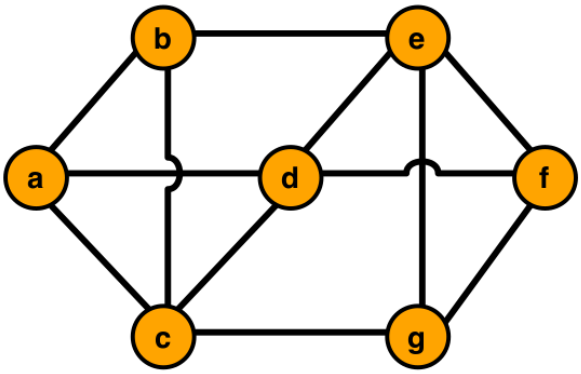
We see that changing the value of one edge can result a different MST and that is because in finding MSTs we care about safe edges.

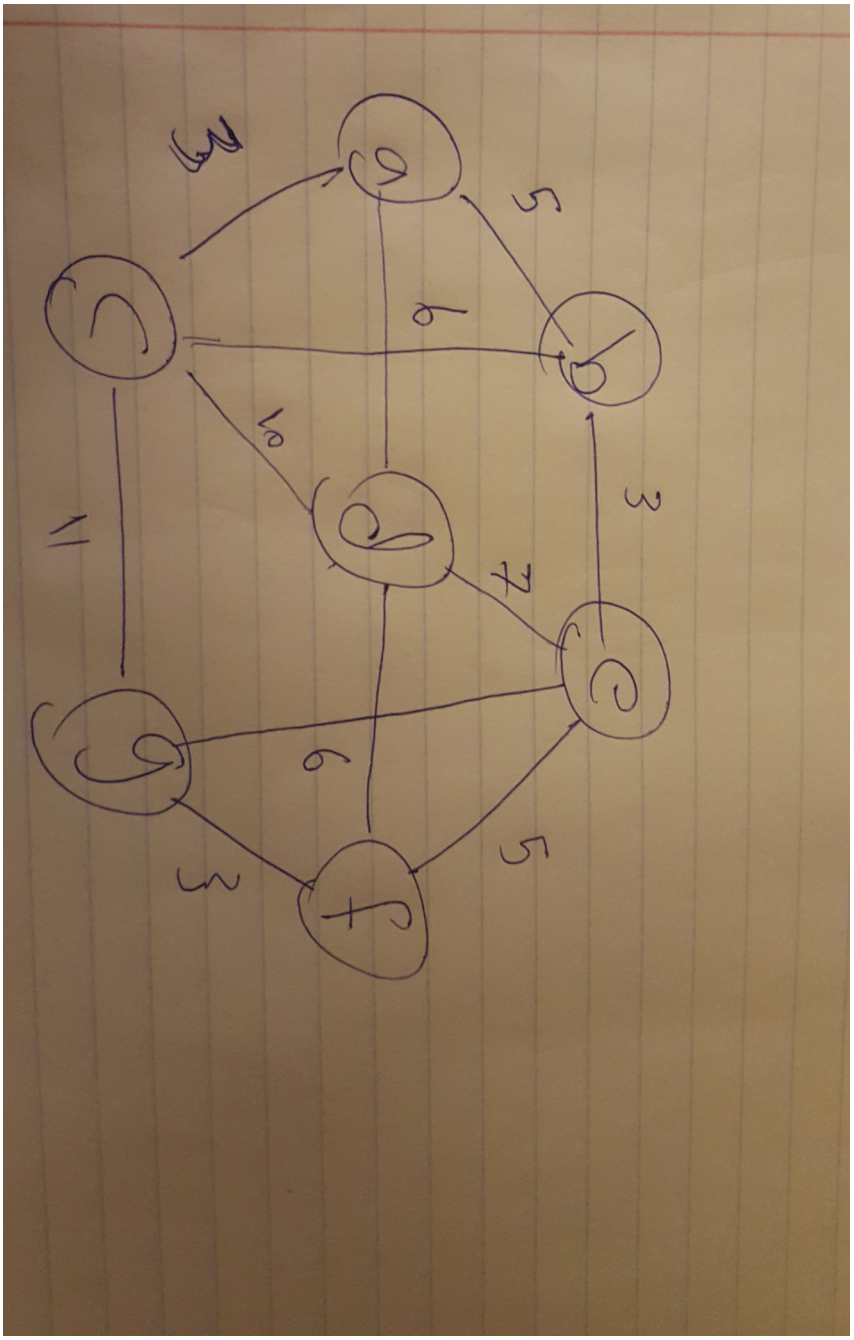
Problem 2

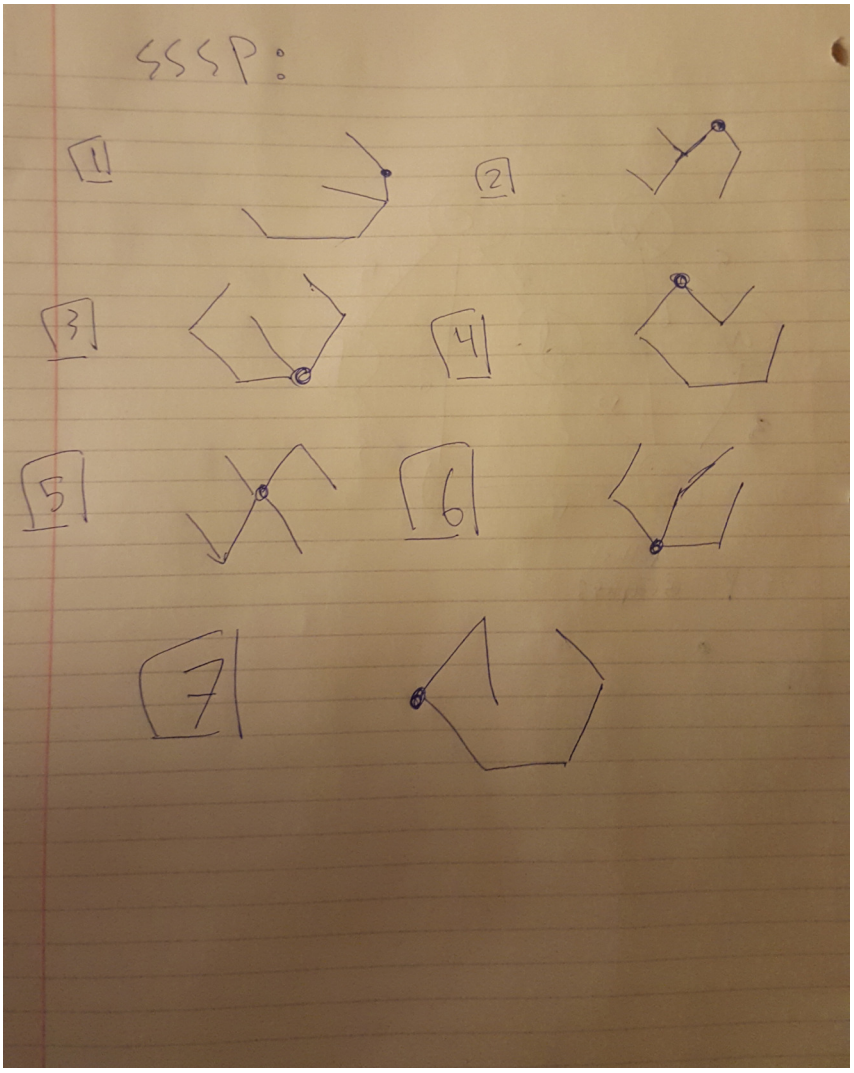
(20 pts) Professor Snape gives you the following unweighted graph and asks you to construct a weight function w on the edges, using positive integer weights only, such that the following conditions are true regarding minimum spanning trees and single-source shortest path trees:

- The MST is distinct from any of the seven SSSP trees.
- The order in which Jarnk/Prims algorithm adds the safe edges is different from the order in which Kruskals algorithm adds them.
- Boruvkas algorithm takes at least two rounds to construct the MST.

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 three algorithms. (For Boruvkas algorithm, be sure to denote which edges are added simultaneously in a single round.)






SSSP:
1
2
3
4
5
6
7

MST

Problem 3

(10 pts extra credit) Crabbe and Goyle think they have come up with a way to get rich by playing the foreign exchange markets in the wizarding world. Their idea is to exploit these exchange rates in order to transform one unit of British wizarding money into more than one unit of British wizarding money. For instance, suppose 1 wizarding penny bought 0.82 French wizarding pennies, 1 French wizarding penny bought 129.7 Russian wizarding pennies, 1 Russian wizarding penny, and one Russian wizarding penny bought 0.0008 British wizarding pennies. By converting these coins, Crabbe and Goyle think they could start with 1 British wizarding penny and buy $0.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02$ *British wizarding pennies, thereby making a 2% profit! The problem is that those gnomes at Gringots charge a transaction cost for each exchange...*

Suppose that Crabbe and Goyle start with knowledge of n wizard monies $c_1, c_2, \cdots, c_n$ *and an* $n \times n$ *table R of exchange rates, such that one unit of wizard money* c_i *buys* $R[i, j]$ *units of wizard money* 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 currencies* $\langle c_{i_1}, c_{i_2}, \cdots, c_{i_k} \rangle$ *such that* $R[i_1, i_2] \times R[i_2, i_3] \times \cdots \times R[i_{k1}, i_k] \times R[i_k, i_i] > 1$*. Each transaction, however, must pay Gringots 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.*

Hermiones hint: It is possible to solve this problem in O(n 3). Recall that Bellman-Ford can be used to detect negative-weight cycles in a graph.

TODO

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

TODO

Problem 4

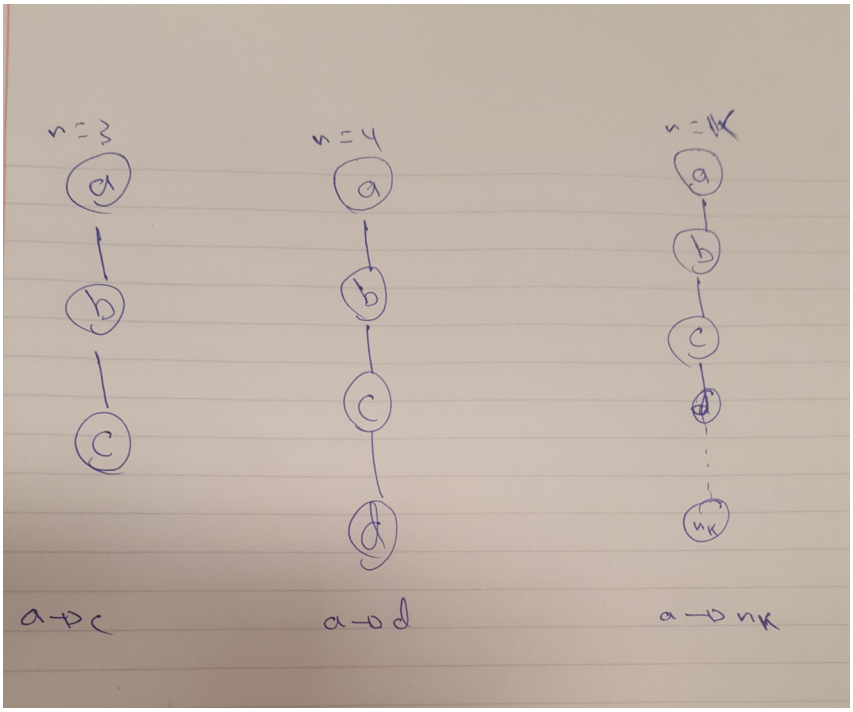
(40 pts) Bidirectional breadth-first search is a variant of standard BFS for finding a shortest path between two vertices $s, t \in V(G)$. The idea is to run two breadth-first searches simultanesouly, one starting from s and one starting from t, and stop when they meet in the middle (that is, whenever a vertex is encountered by both searches). "Simultaneously" here doesnt assume you have multiple processors at your disposal; its 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 well 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.

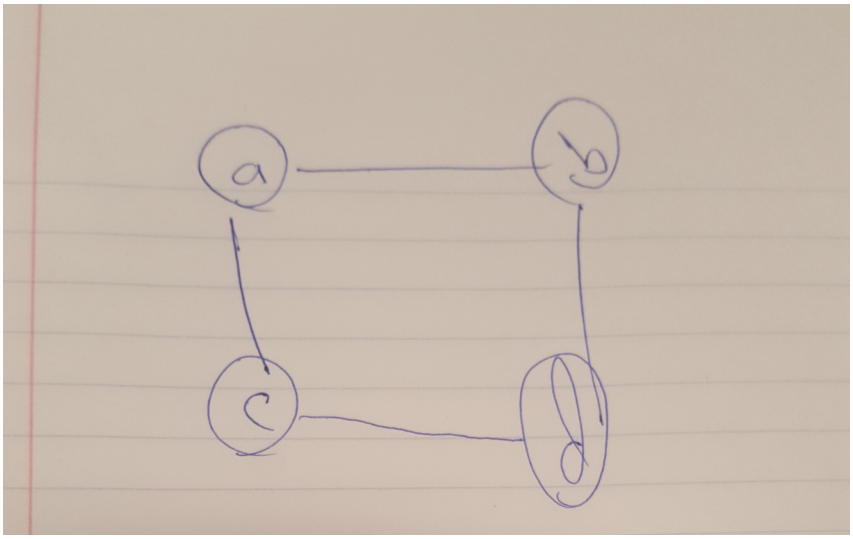
(a) *Given examples to show that, in the worst case, the asymptotic running time of bidirectional BFS is the same as that of ordinary BFS. Note that because we are asking for asymptotic running time, you actually need to provide an infinite family of examples (G_n, s_n, t_n) such that $s_n, t_n \in V(G_n)$, the asymptotic running time of BFS and bidirectional BFS are the same on inputs (G_n, s_n, t_n), and $|V(G_n)| \to \infty$ as $n \to \infty$*

Both algorithms, BFS and BidirectionalBFS, have the same asymptotic running time because both of them depend on how many algorithms added and popped out of the queue(s).
`(*see the algorithms below)` In BFS, in each iteration we pop out only one vertex, but in BidirectionalBFS we pop out two vertices. However, in BFS, there is only one for loop inside the while loop, but in BidirectionalBFS, there are two for loops inside the while loop.
In the family of examples below, we find the path from starting vertex (for example, a) and the ending vertex (for example, d). BFS and BidirectionalBFS give the same asymptotic running time because the same number of vertices are added and popped out of the queue(s), so that gives the same number of iterations. Hence, the asymptotic running time for both algorithms in the worst case is the same.



(b) *Recall that in ordinary BFS we used a* `state` *array (see Lecture Notes 8) to keep track of which nodes had been visited before. In bidirectional BFS we will need two* `state` *arrays, one for the BFS from s and one for the BFS from t. Why? Give an example to show what can go wrong if theres only one state array. In particular, give a graph G and two vertices s, t such that some run of a bidirectional BFS says there is no path from s to t when in fact there is one.*

`(*see the algorithms below)` In BST and BidirectionalBST algorithms, I used two state arrays, and I check in each iteration for each array, if that certain vertex was visited by the other array. That is how I knew I found a path between two vertices.
However, if we use only one state array, the only way that we can know we reached the destination vertex by comparing each not visited vertex with the destination vertex (if they are equal), and by doing that we will not find a path between two vertices in most cases.



For example, if we want to find the path between 'a' and 'd' in the graph above, first we will add 'a' and 'd' to the queue. 'c' and 'b' will be added and marked visited. Then, we will go through the adjacency list of 'd' and find that 'c' and 'd' are visited, so nothing will happen. And the same thing will happen when we go through the adjacency lists of 'b' and 'c'. Hence, no path will be found, and that is wrong.

(c) *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.*

```
def BFS(G, s, t):
    '''
    G is a dicitionary, key is a vertex name, value is an adjacency list
    s and t are names of vertices
    '''
    #initilize visited dict, key is vertex name and value is initially False
    visited = {}
    for i in G:
        visited[i] = False

    dist = {}
    for i in G:
        dist[i] = 0

    queue = []
    queue.append((s, G[s]))
    visited[s] = True
    found = False
    nodesPopped = 0
    iterations = 0
```

```
    while(len(queue) and not found):
        iterations += 1
        nodesPopped += 1
        element = queue.pop(0)
        currVer = element[0]
        currAdj = element[1]
        for vertex in currAdj:
            iterations += 1
            if(visited[vertex] == False):
                visited[vertex] = True
                dist[vertex] = dist[currVer] + 1
                queue.append((vertex, G[vertex]))

                if(vertex == t):
                    found = True

    d = dist[t]
    if(dist[t] == 0):
    d = -1

    return (d, nodesPopped)
```

(d) *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.*

```
def BidirectionalBFS(G, s, t):
    '''
    G is a dicitionary, key is a vertex name, value is an adjacency list
    s and t are names of vertices
    '''
    #initilize visited dict, key is vertex name and value is initially False
    visited_s = {}
    for i in G:
        visited_s[i] = False

    visited_t = {}
    for i in G:
        visited_t[i] = False

    dist = {}
    for i in G:
        dist[i] = 0

    queue_s = []
    queue_s.append((s, G[s]))

    queue_t = []
    queue_t.append((t, G[t]))

    visited_s[s] = True
```

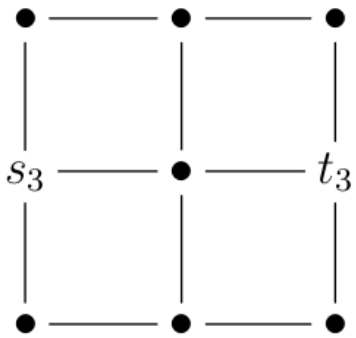
```
visited_t[t] = True

found = False
nodesPopped = 0
iterations = 0
foundVertexDistance = -1
while(len(queue_s) and len(queue_t) and not found):
    iterations += 1
    element_s = queue_s.pop(0)
    nodesPopped += 1

    currVer_s = element_s[0]
    currAdj_s = element_s[1]
    for vertex in currAdj_s:
        iterations += 1
        if(visited_s[vertex] == False):
            visited_s[vertex] = True
            dist[vertex] += dist[currVer_s] + 1
            queue_s.append((vertex, G[vertex]))

            if((vertex == t or visited_t[vertex] == True) and s != t):
                found = True
                foundVertexDistance = dist[vertex]


    if(not found):
        element_t = queue_t.pop(0)
        nodesPopped += 1

        currVer_t = element_t[0]
        currAdj_t = element_t[1]
        for vertex in currAdj_t:
            iterations += 1
            if(visited_t[vertex] == False):
                visited_t[vertex] = True
                dist[vertex] += dist[currVer_t] + 1
                queue_t.append((vertex, G[vertex]))

                if((vertex == s or visited_s[vertex] == True) and s != t):
                    found = True
                    foundVertexDistance = dist[vertex]



return (foundVertexDistance, nodesPopped)
```

(e) *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 tuples (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*

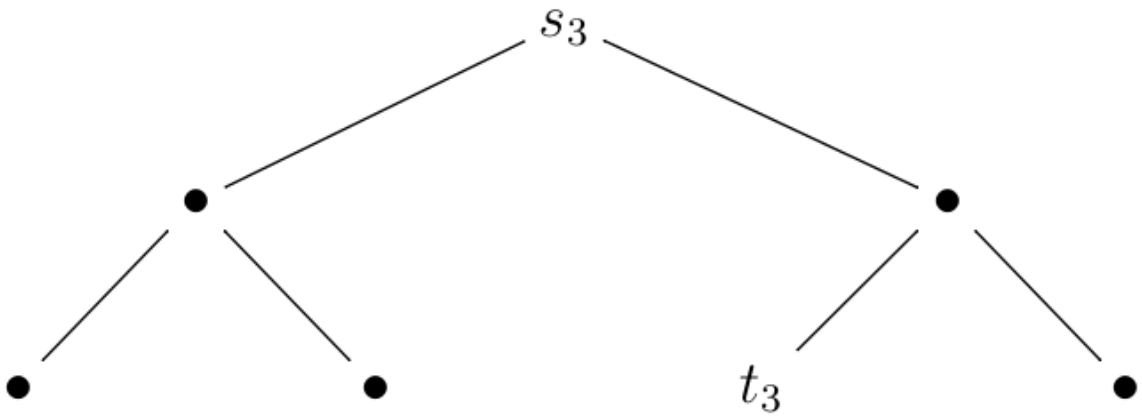
the values of k_1 and one for the values of k_2:

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:*



Produce output for $n = 3, 4, 5, \cdots, 20$.

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 $n = 3, 4, 5, \cdots, 15$. 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 $n = 3, 4, 5, \cdots, 20$.*

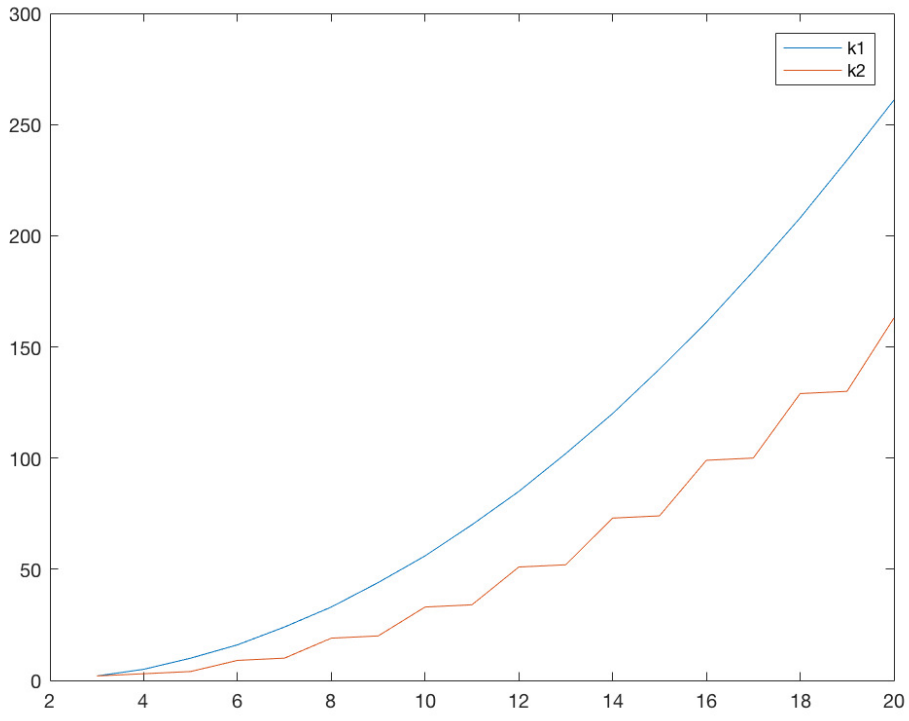
i:This is the code I wrote to create the graph and create x, y1, y2 lists in python and then used them in matlab to plot.

```
def createGraph(n):
    x = 1
    graph = dict()
    for i in range(n):
        for j in range(n):
            if(j == 0):
                if(i == 0):
                    graph[x] = [x+1, x+n]
                elif(i == n-1):
                    graph[x] = [x+1, x-n]
                else:
                    graph[x] = [x+1, x+n, x-n]
            elif(j == n-1):
                if(i == 0):
                    graph[x] = [x-1, x+n]
                elif(i == n-1):
                    graph[x] = [x-1, x-n]
                else:
                    graph[x] = [x-1, x+n, x-n]
            else:
                if(i == 0):
                    graph[x] = [x+1, x-1, x+n]
                elif(i == n-1):
                    graph[x] = [x+1, x-1, x-n]
                else:
                    graph[x] = [x+1, x-1, x+n, x-n]

            x += 1

    return graph
x = []
y1 = []
y2 = []
for n in range(3, 21):
    graph = createGraph(n)
    s = ((((n + 1)/2) - 1) * n) + 1
    t = s + n - 1
    d1, k1 = BFS(graph, s, t)
    d2, k2 = BidirectionalBFS(graph, s, t)
    print "(n:%d, d1:%d, k1:%d, d2:%d, k2:%d)" %(n, d1, k1, d2, k2)
    x.append(n)
    y1.append(k1)
    y2.append(k2)

print x
print y1
print y2
```

text:

(n:3, d1:2, k1:2, d2:2, k2:2)
(n:4, d1:3, k1:5, d2:3, k2:3)
(n:5, d1:4, k1:10, d2:4, k2:4)
(n:6, d1:5, k1:16, d2:5, k2:9)
(n:7, d1:6, k1:24, d2:6, k2:10)
(n:8, d1:7, k1:33, d2:7, k2:19)
(n:9, d1:8, k1:44, d2:8, k2:20)
(n:10, d1:9, k1:56, d2:9, k2:33)
(n:11, d1:10, k1:70, d2:10, k2:34)
(n:12, d1:11, k1:85, d2:11, k2:51)
(n:13, d1:12, k1:102, d2:12, k2:52)
(n:14, d1:13, k1:120, d2:13, k2:73)
(n:15, d1:14, k1:140, d2:14, k2:74)
(n:16, d1:15, k1:161, d2:15, k2:99)
(n:17, d1:16, k1:184, d2:16, k2:100)
(n:18, d1:17, k1:208, d2:17, k2:129)
(n:19, d1:18, k1:234, d2:18, k2:130)
(n:20, d1:19, k1:261, d2:19, k2:163)

plot:



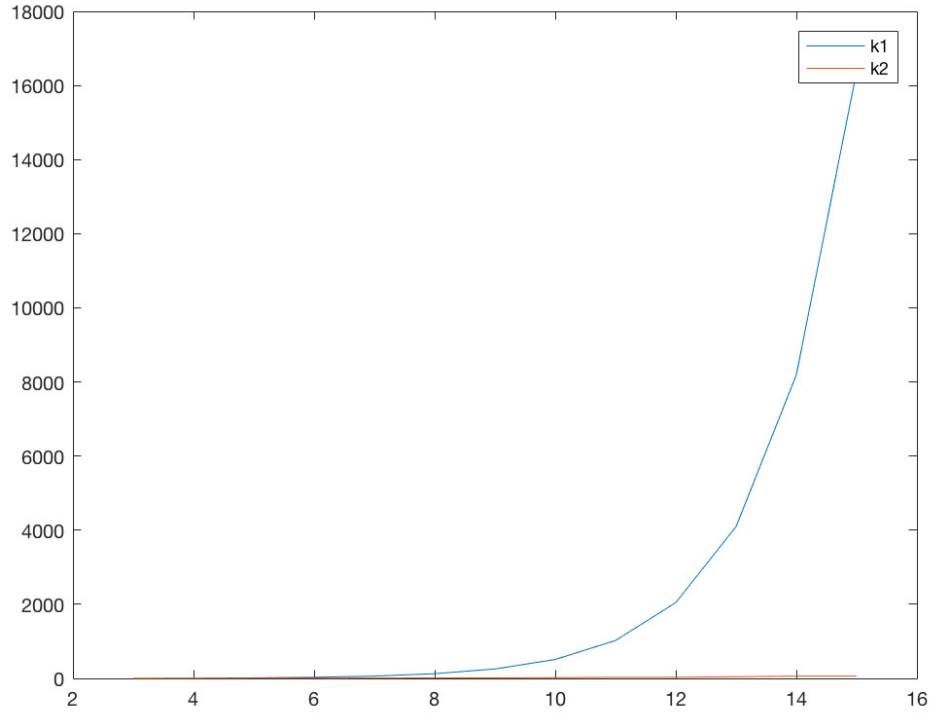
ii:This is the code I wrote to create the tree and create x, y1, y2 lists in python and then used them in matlab to plot.

```
def createTree(n):
    graph = dict()
    graph[1] = [2, 3]
    graph[2] = [1]
    graph[3] = [1]

    x = 4
    for k in range(3, n+1):
        for i in range(2**(k-1)):
            graph[x] = [x/2]
            graph[x/2].append(x)
            x += 1


    return graph

x = []
y1 = []
y2 = []
for n in range(3, 16):
    tree = createTree(n)
    s = 1
    t = 2**n - 1
    d1, k1 = BFS(tree, s, t)
    d2, k2 = BidirectionalBFS(tree, s, t)
    print "(n:%d, d1:%d, k1:%d, d2:%d, k2:%d)" %(n, d1, k1, d2, k2)
    x.append(n)
    y1.append(k1)
    y2.append(k2)

print x
print y1
print y2
```

text:

(n:3, d1:2, k1:3, d2:2, k2:2)
(n:4, d1:3, k1:7, d2:3, k2:4)
(n:5, d1:4, k1:15, d2:4, k2:5)
(n:6, d1:5, k1:31, d2:5, k2:6)
(n:7, d1:6, k1:63, d2:6, k2:10)
(n:8, d1:7, k1:127, d2:7, k2:13)
(n:9, d1:8, k1:255, d2:8, k2:14)
(n:10, d1:9, k1:511, d2:9, k2:22)
(n:11, d1:10, k1:1023, d2:10, k2:29)
(n:12, d1:11, k1:2047, d2:11, k2:30)
(n:13, d1:12, k1:4095, d2:12, k2:46)
(n:14, d1:13, k1:8191, d2:13, k2:61)
(n:15, d1:14, k1:16383, d2:14, k2:62)

plot:



iii:This is the code I wrote to create the random graph and create x, y1, y2 lists in python and then used them in matlab to plot.

It does not matter, on average, which vertices we take s, t to be because on average each vertex has one edge, so the number of vertices popped off of the queue(s) is constant.

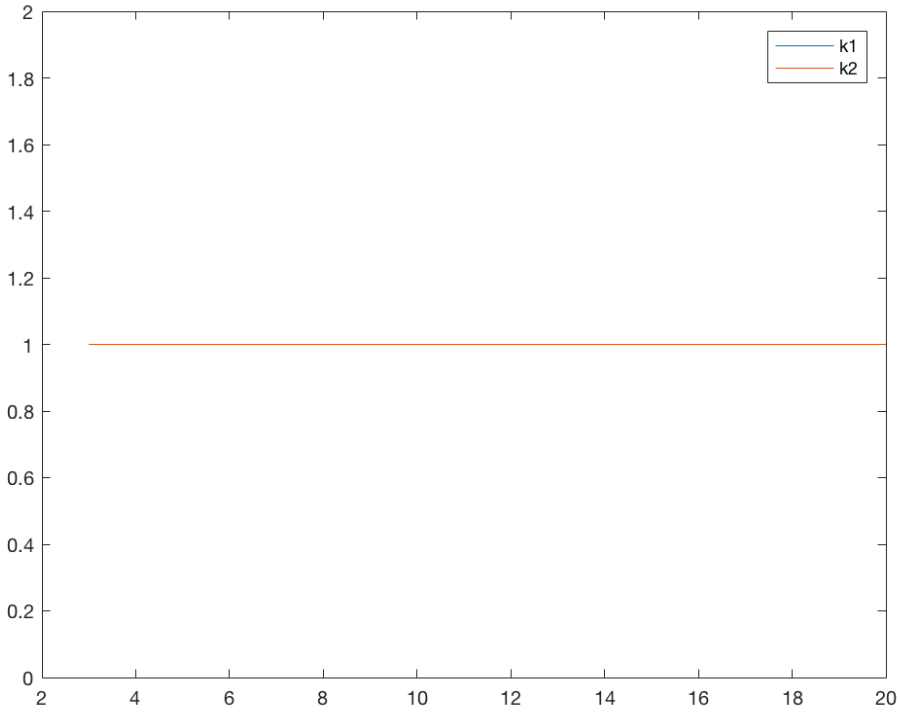
```
x = []
y1 = []
y2 = []
for n in range(3, 21):
    average_d1 = 0
    average_d2 = 0
    average_k1 = 0
    average_k2 = 0
    for i in range(50):
        graph = createRandom(n)
        s = 1
        t = 2
        d1, k1 = BFS(graph, s, t)
        d2, k2 = BidirectionalBFS(graph, s, t)
        average_d1 += d1
        average_d2 += d2
```

```
        average_k1 += k1
        average_k2 += k2

    average_d1 /= 50
    average_d2 /= 50
    average_k1 /= 50
    average_k2 /= 50

    print "(n:%d, d1:%d, k1:%d, d2:%d, k2:%d)" %(n, average_d1, average_k1, average_d2, average_k2
    x.append(n)
    y1.append(average_k1)
    y2.append(average_k2)


print x
print y1
```

text:

(n:3, d1:0, k1:1, d2:0, k2:1)
(n:4, d1:0, k1:1, d2:0, k2:1)
(n:5, d1:0, k1:1, d2:0, k2:1)
(n:6, d1:0, k1:1, d2:0, k2:1)
(n:7, d1:0, k1:1, d2:0, k2:1)
(n:8, d1:0, k1:1, d2:0, k2:1)
(n:9, d1:0, k1:1, d2:0, k2:1)
(n:10, d1:0, k1:1, d2:0, k2:1)
(n:11, d1:-1, k1:1, d2:-1, k2:1)
(n:12, d1:0, k1:1, d2:0, k2:1)
(n:13, d1:0, k1:1, d2:0, k2:1)
(n:14, d1:0, k1:1, d2:0, k2:1)
(n:15, d1:-1, k1:1, d2:-1, k2:1)
(n:16, d1:0, k1:1, d2:0, k2:1)
(n:17, d1:-1, k1:1, d2:-1, k2:1)
(n:18, d1:-1, k1:1, d2:-1, k2:1)
(n:19, d1:-1, k1:1, d2:-1, k2:1)
(n:20, d1:-1, k1:1, d2:-1, k2:1)

plot:


k1
k2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
2
4
6
8
10
12
14
16
18
20

Worked with Abdullah Bajkhaif