Name: Jon Abrahamson
ID: 107084898

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 9 **Spring 2019, CU-Boulder**

Quick links 1a 1b 1c 1d 2 3

1. (30 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 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.

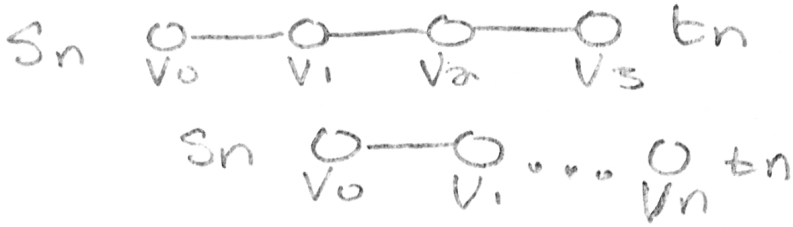
   As we'll see, although the worst-case running time of BFS and Bidirectional BFS are asymptotically the same, in practice Bidirectional BFS often performs significantly better.

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

   (a) (5 pts) Give 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 $n \to \infty$.

In the worst case of BFS and bidirectional BFS, the running time will be the same. In any straight line graph, this will be true. Straight line graphs are defined as all the $v \in V$ such that v_i has an edge to v_i and v_n only has an edge to v_{n-1}. s_n is always v_0 and t_n is always v_n in the worst case since the BFS will need to traverse all vertices.

1 (cont...)

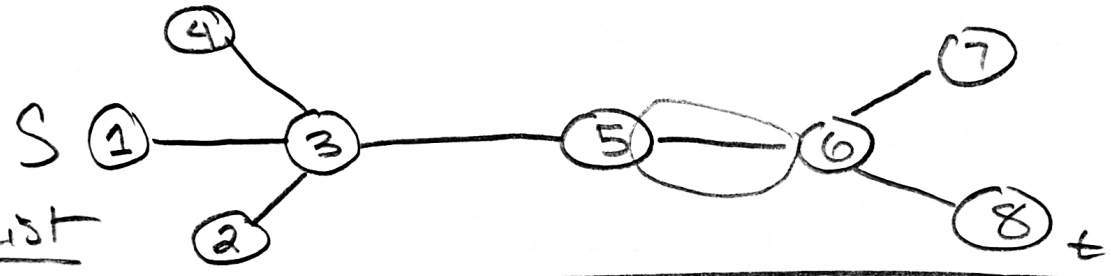


In a normal BFS from Sn to tn, the search will visit all vertices before reaching tn since the vertices are a straight line. Thus, the run time is O(n) as we traverse vertices. In the BFS we start at Sn and tn and stop when the paths cross. This is at the middle vertex of the straight line at V_2 in the first example. Sn will visit V_1, tn will visit V_3, and Sn will visit V_2 to meet tn. Although the paths are coming from either side, all vertices will still be visited in order to travel to the middle and will have a run time of O(n). Therefore, any straight line graph or linked list type graph will result in the same asymptotic run times in BFS and bidirectional BFS

Name: Jon Abrahamson
ID: 107084898

CSCI 3104 — **Profs. Grochow & Layer**
Problem Set 9 — **Spring 2019, CU-Boulder**

(b) (5 pts) Recall that in ordinary BFS we used a `visited` array (see Lecture Notes 8) to keep track of which nodes had been visited before. In bidirectional BFS we'll need *two* `visited` arrays, one for the BFS from s and one for the BFS from t. Let "naive bidirectional BFS" denote an attempted implementation of bidirectional BFS which uses only one `visited` array. Give an example to show what can go wrong if there's only one `visited` array. More specifically, give a graph G and two vertices s, t such that some run of a naive bidirectional BFS says there is no path from s to t when in fact there is one.



Visited List

1 8 3 6 4 2 7 5

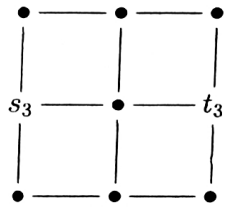
Here, the outcome of a single visited list. We start at S and t. The first round of BFS visits all the nodes 1 node away from S and t so 3 & 6. The next round visits nodes 2 away from their start so 2, 4, 5, & 7. From the path starting at t, 5 should also be visited in order to connect the path from S to t since it is 2 nodes away from t. However, since we already saw 5 in S path, t won't visit 5. All nodes have been seen & there will be no path. If each node had its own visited list, both paths would make it to node 5, path would connect, & the search would work.

Name: Jon Abrahamson
ID: 107084898

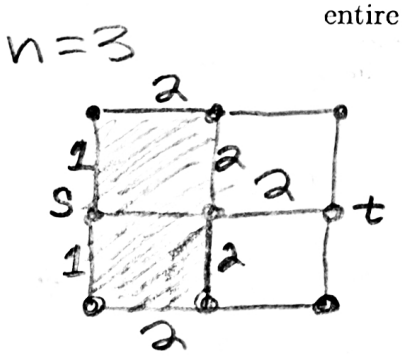
CSCI 3104
Problem Set 9

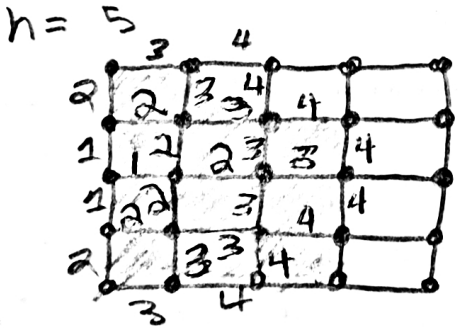
Profs. Grochow & Layer
Spring 2019, CU-Boulder

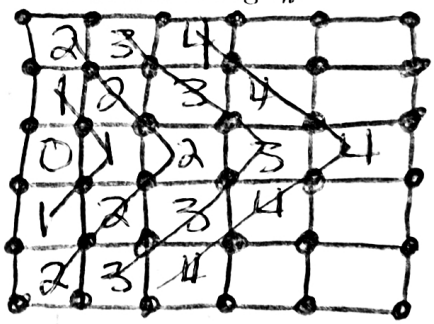
(c) Consider BFS vs. bidirectional BFS on grids. Namely, let G_n be an $n \times n$ grid, where each vertex is connected to its neighbors in the four cardinal directions (N,S,E,W). Vertices on the boundary of the grid will only have 3 neighbors, and corners will only have 2 neighbors. Let s_n be the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



i. (5 pts) Give an argument as to why BFS starting from s_n searches nearly the entire graph (in fact, a constant fraction of it) before encountering t_n.







Starting from s, BFS will cover 3/4 of the graph before encountering t due to the example above. Starting with n=3 and moving to n=5 nodes, we can see a pattern forming as we keep track of each layer of BFS. First we mark all nodes that are visited in the first layer (1 node away) with

(cont...)

a 1. Continuing this pattern until we hit ± we have a graph full of numbers corresponding to the layer of BFS that node was visited during. Visually, we can see that this fraction is close to 3/4. If you count all the nodes visited and divide this by the total number of nodes, it also is close to 3/4. If you continue to increase n, the same pattern holds true and gets closer & closer to 3/4.

In the second example, we can find a mathematical summation to find out how many nodes will be visited no matter what n is. We can see that until we hit the edges of the box, we will have three 1's, five 2's, seven 3's, etc. Once the edges are hit, however many levels we have left will even out at n nodes visited for each of those layers. We have the following summation for the first pattern that increases by 2 each layer that we came up with based on the pattern.

1 cont...)

Using Wolfram Alpha to reduce the summation

$$\sum_{k=1}^{\frac{n-1}{2}-1} 2k+1 = \frac{1}{4}(n^2-2n-3)$$

Next we add the rest of the nodes which have a constant amount of nodes visited in each round BFS. We take the rest of the layers and multiply it by n since we know that is how many nodes will be visited for that round

$$\frac{1}{4}(n^2-2n-3) + \frac{n+1}{2} \cdot n = \frac{2n^2+2n}{4} + \frac{1}{4}(n^2-2n-3)$$

$$= \frac{3}{4}(n^2-1)$$

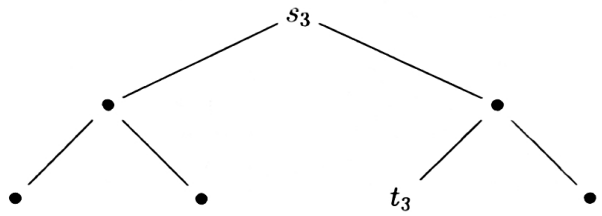
If we divide this total number of visited nodes by the total number of nodes in the graph (n^2), we will end up with the constant 3/4 which is the fraction we assumed to be correct

ii. (5 pts) Bidirectional BFS also searches a constant fraction of the entire graph before finding a path from s_n to t_n, but a smaller constant fraction than ordinary BFS. Estimate this constant, and give an argument to justify your estimate. Hint: as $n \to \infty$, if you "zoom out" the graph starts to look more like the unit square $[0,1] \times [0,1]$ in the real plane $\mathbb{R}^2$. Consider the "spreading" picture of BFS / bidirectional BFS and use basic geometric facts.

I don't know

Name: Jon Abrahamson
ID: 107084898
Profs. Grochow & Layer
Spring 2019, CU-Boulder

(d) Consider BFS vs. bidirectional BFS on trees. Let T_n is a complete binary tree of depth n. s_n is the root and t_n is any leaf. For example, for $n = 3$ we have:

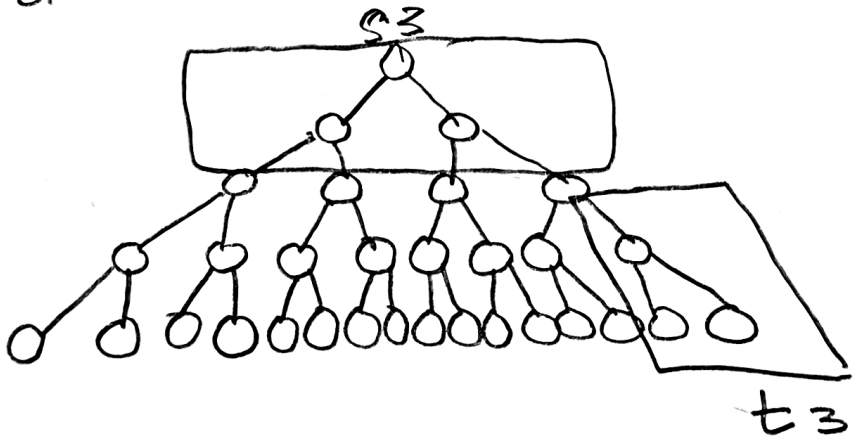


i. (5 pts) Prove the asymptotic running time of BFS on T_n starting at s_n, where the BFS can stop as soon as it finds a path from s_n to t_n.

In a binary search tree with height n, we know that there are 2^n-1 nodes. Since the node tn is one of the root nodes, BFS must search through all layers of the tree. It will first search all nodes distance 1 away from s, then 2 away, etc. until it reaches the root nodes. In worst case, tn will be the very last root node to be checked in that layer, so the BFS will search all nodes in the tree. Therefore, the run time would be $O(2^n-1)$.

ii. (5 pts) Prove the asymptotic running time of bidirectional BFS on T_n starting at (s_n, t_n).

The running time of bidirectional BFS on a binary tree is $O(2^{n/2})$. The height of the tree is still n, and the two starting points will meet eachother in the middle of graph or $n/2$. The leaf node t_3 will traverse a sub graph of the original tree and only traverses to the height of $n/2$. From s_3 to the middle, the path it traverses will also cross all layers of half the height as well. Since both paths traverse a height of $n/2$, and there are 2^{n-1} nodes total, the run time becomes $O(2^{n/2})$



Name: Jon Abrahamson
ID: 107084898

CSCI 3104
Problem Set 9

Profs. Grochow & Layer
Spring 2019, CU-Boulder

2. (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

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

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

If T is the path from u, v, than if any vertex on the path has a weight thats bigger than the new edge, then path T would no longer be the minimum spanning tree. If we remove the max weight edge in T, we can obtain a new minimum spanning tree for G'

Say $(u,v) = (x,y)$

$w'(T') = w(T) - k$

T' such that $w(T) \leq w(T')$

if $(x,y) \notin T'$ $\rightarrow$ $w'(T') = w(T')$

$\therefore w(T') \geq w(T) > w'(T)$

if $(x,y) \in T'$ $\rightarrow$ $w'(T') = w(T') - k \geq w(T) - k = w'(T)$

$\therefore w'(T') \geq w'(T)$

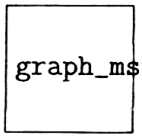
So, $w'(T) \leq w'(T')$ Always

$\therefore T$ is a minimum spanning tree for w'

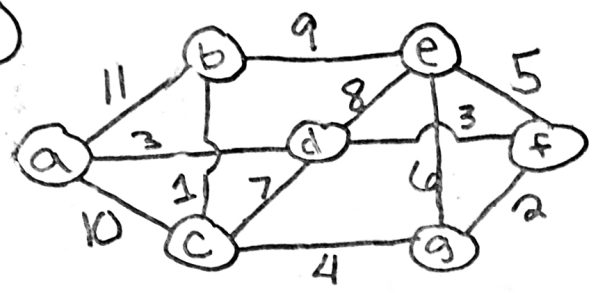
3. (20 pts) Professor Snape gives you the following unweighted graph and asks you to construct a weight function w on the edges, using positive integer weights only, such that the following conditions are true regarding minimum spanning trees and single-source shortest path trees:

   - The MST is distinct from any of the seven SSSP trees.
   - The order in which Jarník/Prim's algorithm adds the safe edges is different from the order in which Kruskal's algorithm adds them.
   - Borůvka's 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 Borůvka's algorithm, be sure to denote which edges are added simultaneously in a single round.)



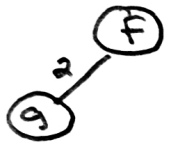
3.1 (i)



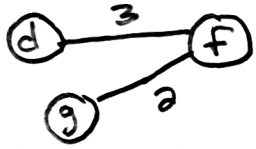
(ii) <u>using prim's algorithm</u>

Chose any vertex, let's say F

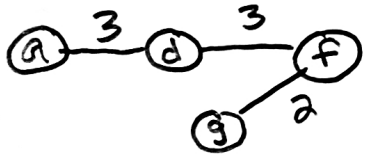
Choose the minimum edge weight



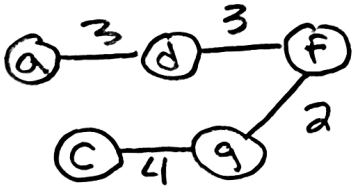
Choose second minimum weight



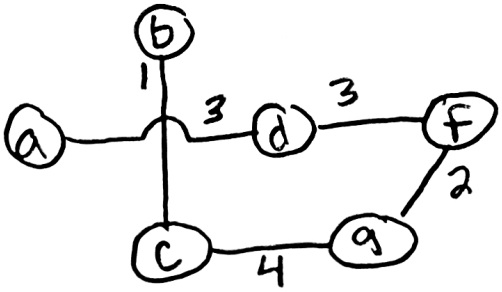
On that node chose minimum edge weight



go to node (g) and choose minimum edge weight

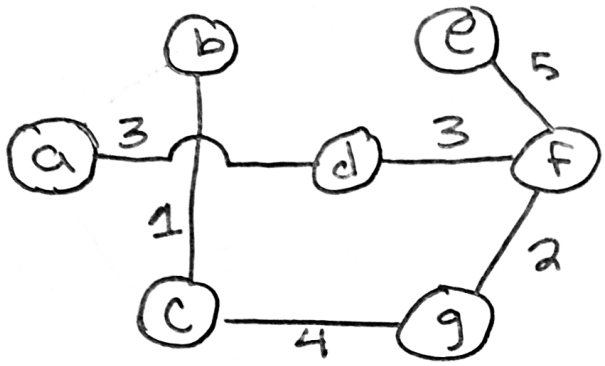


On that node choose minimum edge weight



3 Cont...)

go back to node (f) and chose the last minimum edge



All vertices accounted for

$\therefore (f,e) \rightarrow (f,d) \rightarrow (d,a) \rightarrow (g,c) \rightarrow (c,b) \rightarrow (f,e)$

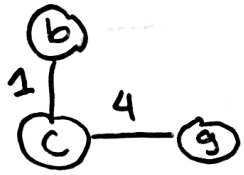
$= 5+3+3+4+1+5 = 18$

So MST minimum weight is 18

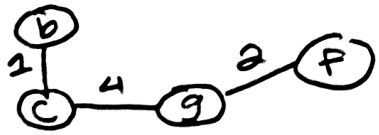
Kruskal's algorithm

choose smallest edge

Choose next smallest edges from either vertice

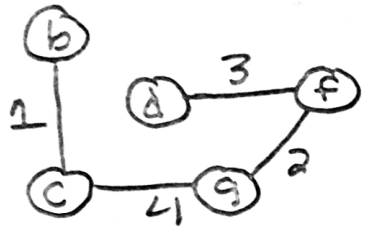


Choose nex smallest edge From b, c, or g

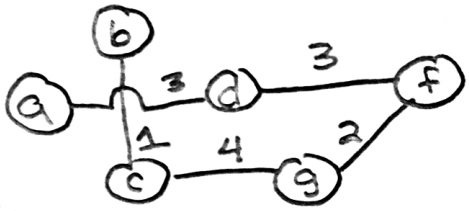


3 cont...)

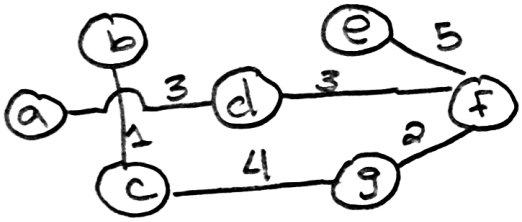
Choose overall smallest edge again



Choose smallest edge



Next smallest



$\therefore (c,b) \rightarrow (c,g) \rightarrow (g,F) \rightarrow (F,d) \rightarrow (d,a) \rightarrow (f,e)$

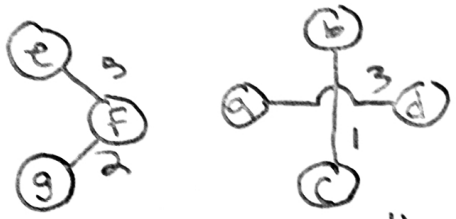
$= 1+4+2+3+3+5 = 18$

Boruka's algorithm

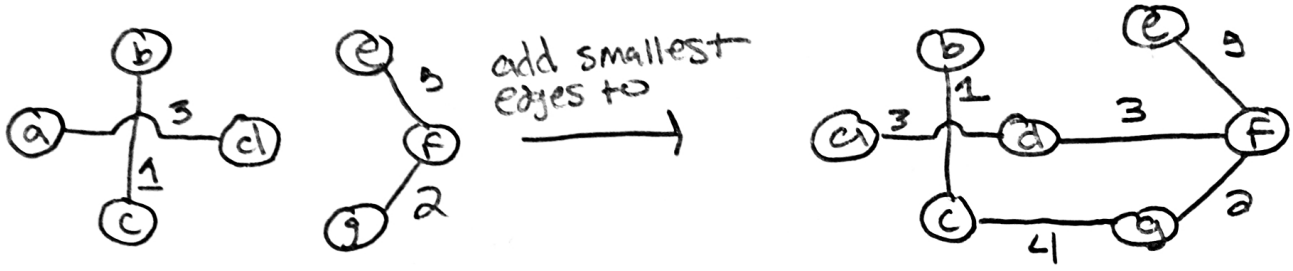
vertex	smallest edge	weight
a	a-b	2
b	b-a	2
c	c-a	5
d	d-f	3
e	e-g	1
f	f-d	3
g	g-e	1

3 Cont...)

MST = (F-g), (e-F), (a-d), (d-a), (b-c)



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



Vertices	Smallest edge	Weight
(f, g, e)	(F, g)	3
(d, a)	(g, F)	3
(b, c)	(C, g)	4

MST = (f, g, e) → (d, a) → (b, c)

= 2 + 3 + 5 + 3 + 4 + 1 = 18