Name: Trevor Bock

ID: 109081318

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 $n \to \infty$.

worst case = last vertex reached.

- The important thing to note here is that we are only looking at the worst case scenario. →

Next page

IF The BFS is uni-directional, that means that our worst case would be in the lowest depth of the graph. IF the BFS is bi-directional, then the worst case would be the vertex at the depth of $\left(\frac{d}{2}\right)$ or halfway down the graph. The reason that the runtimes are equal is because we <u>maintain the rate at which we look at new vertices.</u>

So in either example, we have to check $(V-1)$ vertices at $(d-1)$ levels to finally find the last vertex we are searching for.

In mathmatical terms

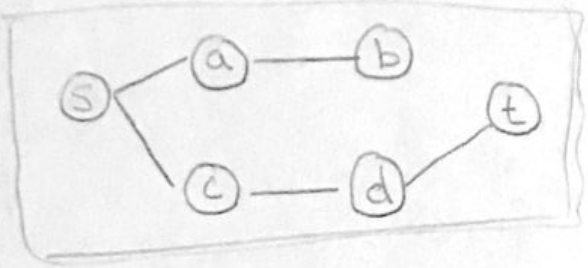
$$O(V+E) = 2O\left(\frac{V+E}{2}\right)$$

↑ Uni-directional ↑ Bidirectional

Name: Trevor Buck
ID: 109081818

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.



1st iteration - 's' searches, adds 'a' and 'c' to the visited array, pops

2nd iteration - 't' searches, adds 'd' to array, pops

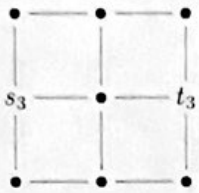
3rd iteration - 'c' searches, adds 'b' and pops.

4th iteration - 'd' searches, finds no 'non-visited' vertices, returns NO PATH ERROR

Name: Trevor Burk
ID: 109081318

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

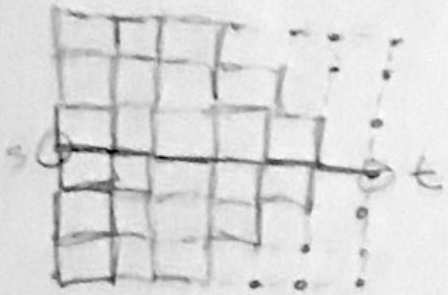
(c) Consider BFS vs. bidirectional BFS on grids. Namely, let G_n be an $n \times n$ grid, where each vertex is connected to its neighbors in the four cardinal directions (N,S,E,W). Vertices on the boundary of the grid will only have 3 neighbors, and corners will only have 2 neighbors. Let s_n be the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

In the example above, the 's' vertex is on the East side, while the 't' node is on the west. A BFS search that starts at 's' and is looking for 't' will cover most of the graph because it can only move in any direction one step at a time. This problem gets exponentially worse because at each iteration, we also have to look up and down.

In other words. The shortest direct path is directly west of the starting point. But before we take a second step west, we also have to check north and south. We end up 'growing' in all directions when the only one that matters is west.

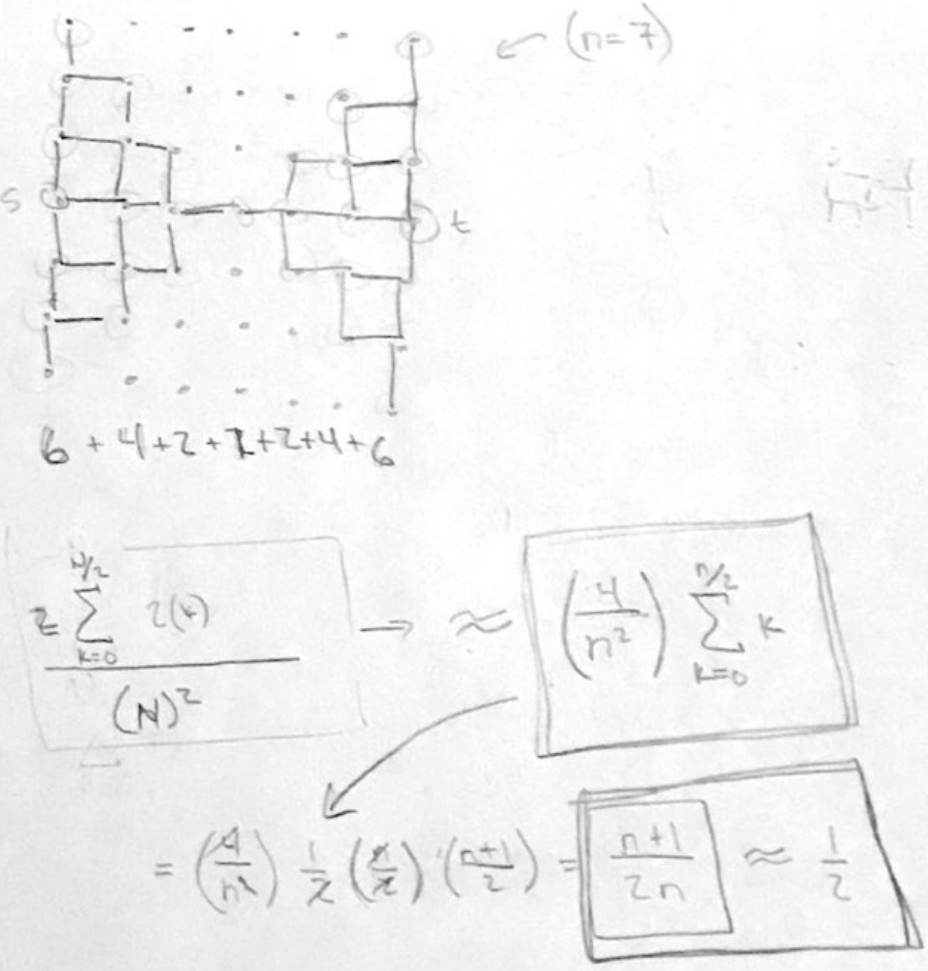


← Here is an example of what an expand version of the search might look like (n=7)

Name: Trevor Buck
ID: 109081318

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

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.



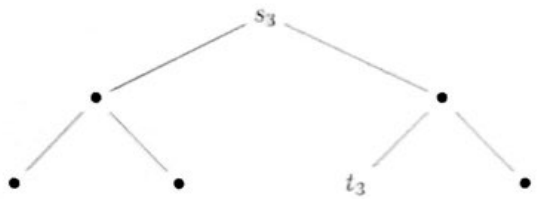
Name: Trevor Buck

ID: 109653158

CSCI 3104 **Profs. Grochow & Layer**

Problem Set 9 **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.

From class, we know that the runtime of any graph is $O(V+E)$. A tree is just a certain type of graph meaning that our equation will work

$$V = E = \sum_{k=0}^{n} 2^{k-1}$$

(as $n \to \infty$)

$$= 2^n$$

$$\rightarrow (V+E) = 2(V) = 2(2^n) = 2^n$$

$$\Rightarrow O(2^n)$$

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

For every step we go down T_n, the runtime is the same. However, going up should be a constant because our node only has one parent. We know that they must also meet in the middle.

$$\Rightarrow O\left(2^{\frac{n}{2}} + 1\right) = \boxed{O\left(2^{n/2}\right)}$$

Name: Trevor Bauer
ID: 109085318

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 9 **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'.

Let $w(T) = \sum_{(x,y)\in T} w(x,y)$.

and we know that $\underline{w'(T) = w(T) - k}$

Now lets consider any other spanning tree (T') such that $w(T) \leq w(T')$:

→ look at each vertex

① if $(x,y) \notin T'$ then → $w'(T') = w(T')$
and $[w(T') \geq w(T) \geq w'(T)]$
$\Rightarrow w'(T') \geq w'(T)$

② if $(x,y) \in T'$ then → $w'(T') = w(T') - k \geq w(T) - k \geq w'(T)$
$\Rightarrow w'(T') \geq w'(T)$

→ Either way, $\boxed{w'(T) \leq w'(T')}$ ← This proves that T is a MST of G ∎

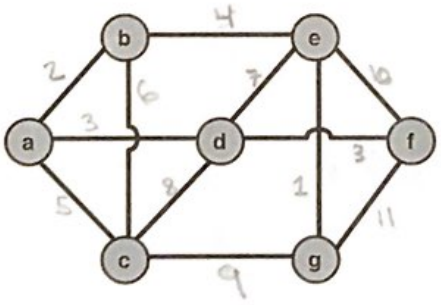
Name: Trevor Buck
ID: 109081318

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

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.
   - Boruvka's algorithm takes at least two rounds to construct the MST.

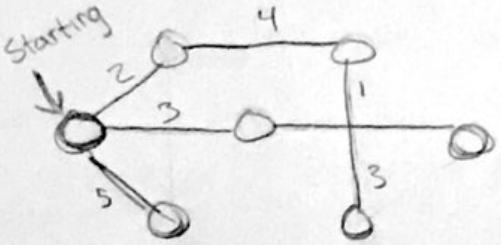
   Justify your solution by (i) giving the edges weights, (ii) showing the corresponding MST and all the SSSP trees, and (iii) giving the order in which edges are added by each of the three algorithms. (For Boruvka's algorithm, be sure to denote which edges are added simultaneously in a single round.)

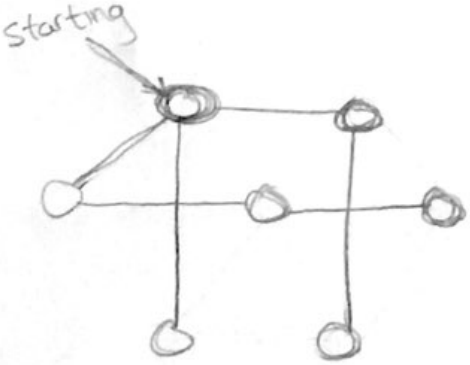


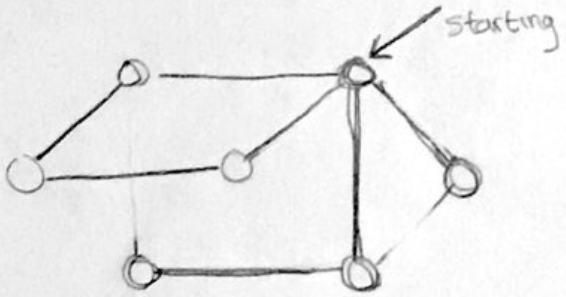
① Edge Weights

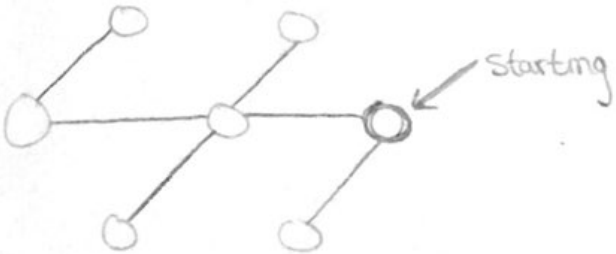
(a-b) → 2
(a-c) → 5
(b-c) → 6
(b-e) → 4
(c-d) → 8
(c-g) → 9
(d-e) → 7
(d-f) → 3
(e-f) → 10
(e-g) → 1
(g-f) → 11
(a-d) → 3

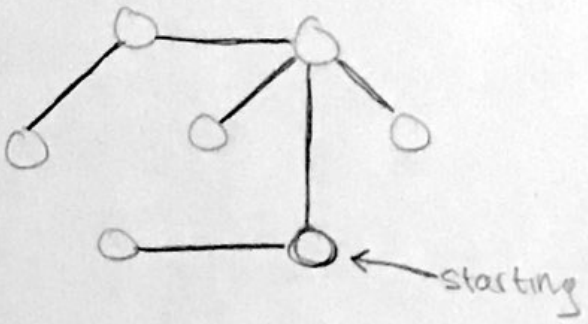
Single Source Spanning Tree

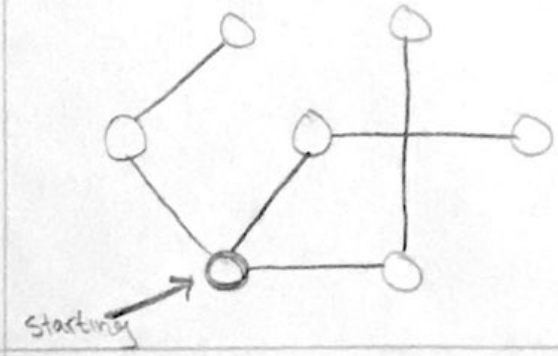


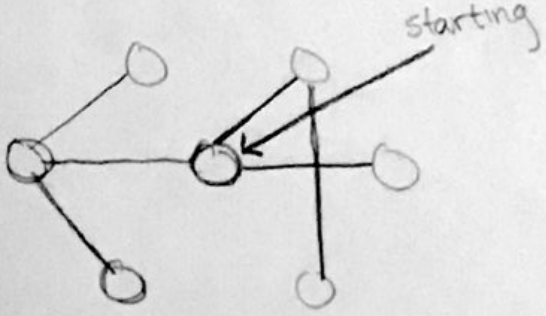




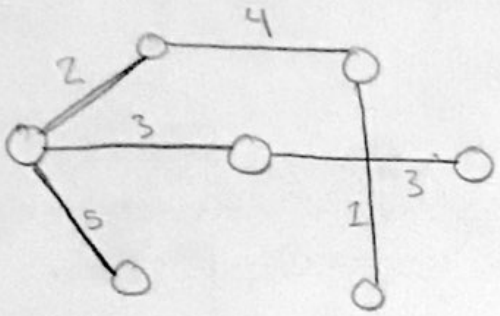








MST



total = 10 + 8 = **18**

Order

Prim:

(a - b) → 2
(a - d) → 3
(d - f) → 3
(b - e) → 4
(f - g) → 1
(a - c) → 5

= 18 ✓

KRUSKAL:

(e-g) → 1
(b-e) → 4
(a-b) → 2
(a-d) → 3
(d-f) → 3
(a-c) → 5

= 18 ✓

Boruvka:

Round 1
(a-b) → 2
(e-g) → 1
(a-c) → 5
(d-f) → 3

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

Round 2
(a→d) → 3
(b→e) → 4

Combine:
(a,b,c,d,f,e,g)
done ✓

11 + 7 = 18 ✓