

Grade: Awesome

## Proof Review Exercise #2

November 2, 2017

### Approach #1

#### Background

**Definition 1** (Edge-Weighted Graphs). An edge-weighted graph  $G = (V, E, \phi)$  is a graph with a function  $\phi : E \rightarrow \mathbb{R}$  that defines edge weights. The length of a weighted path between two vertices  $u, v$  is the sum of the edge weights on this path. The distance between two vertices  $u, v$  is the shortest length weighted path between  $u, v$ .

#### WEIGHTED ALL PAIRS SHORTEST PATH BY PATHWIDTH (WEIGHTED-APSP-PW)

**Input:** An edge-weighted graph  $G = (V, E)$  and a path decomposition of width  $k$

**Parameter:**  $k$

**Question:** Compute the weighted distance between every pairs of vertices in  $f(k)n^2$  time

**Lemma 1.** A graph with a path decomposition of width  $k$  has at most  $kn$  edges.

*Proof Sketch.* Suppose we have a nice path decomposition of width  $k$ . By definition, this decomposition has  $n$  **introduce** bags. We can argue that each **introduce** bag can “introduce” at most  $k$  edges from the original graph, and that this is the only way to introduce edges. Therefore the graph has at most  $kn$  edges.

(Note: This can also be shown by noting that a graph with pathwidth  $k$  has degeneracy at most  $k$ , which provides the same edge bound.) ✓ □

#### Main Result

The theorem 1 should be changed to “There is a  $O(kn^2)$  algorithm for WEIGHTED-ASAP-PW, in which each edge of the graph has equivalent weight”, under the condition of this proof.

**Theorem 1.** There is a  $O(kn^2)$  algorithm for WEIGHTED-ASAP-PW.

*Proof.* WLOG, assume our graph is connected, otherwise we can run this algorithm on each component. Denote the graph as  $G = (V, E)$ , let  $n = |V|$  and  $m = |E|$ , and let  $k$  be the width of the provided path decomposition.

*Algorithm description:* Run BREADTH-FIRST SEARCH (BFS) from each vertex. When  $v \in V$  appears at level  $\ell$  for a BFS rooted at  $u \in V$ , then store  $dist(u, v) = \ell$ .

✓

This algorithm implicitly assumes that each edge in the graph has the same edge weight. Furthermore, In fact, the algorithm itself doesn't consider any edge weight, because it just set  $dist(u, v)$  as the level number. Therefore, the algorithm is totally incorrect if not all edge has the same weight. We give a counterexample in the next page.

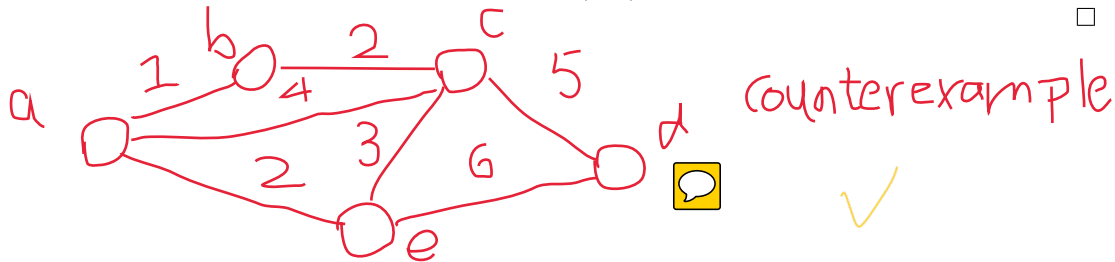
*Run time:* BFS has a worst case run time of  $O(n + m)$ , which is at most  $O(n + kn) = O(kn)$  by Lemma 1. Since BFS is run exactly  $n$  times, the total run time is  $O(kn^2)$ .

*Correctness:* To prove that all distances are computed correctly, we need to prove that  $dist(u, v) = \ell$  for an arbitrary  $u, v \in V$ ; we will prove this by contradiction. Assume that  $dist(u, v) \neq \ell$ , there are two cases to consider:

I would say "...prove that  $dist(u, v) = \ell$  times edgeweight...", but it's ok...

- Assume that  $dist(u, v) > \ell$ . This must be a contradiction because BFS produced a path from  $u$  to  $v$  that is only  $\ell$  edges long.
- Assume that  $dist(u, v) < \ell$ . Suppose the length of this path is  $d$ . Let  $u = v_0, v_1, \dots, v_d = v$  be the actual shortest path. Then  $v_1$  must have appeared in the first BFS level set,  $v_2$  in the second, etc., until  $v_d = v$  occurs in the  $d^{\text{th}}$  level set. But, by definition of  $\ell$ , this is also the  $\ell^{\text{th}}$  level set, which contradicts that  $d < \ell$ .

In both cases we found a contradiction, therefore  $dist(u, v)$  was computed correctly. □



## Approach #2

### Background

ALL PAIRS SHORTEST PATH BY PATHWIDTH (APSP-PW)

**Input:** A graph  $G = (V, E)$  and a path decomposition of width  $k$

**Parameter:**  $k$

**Question:** Compute the distance between every pairs of vertices in  $f(k)n^2$  time

### Main Result

Is it a typo here? In the proof the running time is  $O(kn^2)$ .

**Theorem 2.** There is a  $O(k^2n^2)$  algorithm for APSP-PW.

*Proof.*

We proceed with dynamic programming over the path decomposition. Maintain two data structures:

**visited** = {} // A set of vertices

**APSP** = {} // A hash table that maps a pair of vertices to a distance

Additionally, initialize **visited** to be empty, and **APSP** with a distance of  $\infty$  between every pair of vertices. Iterate over the bags as follows. Note that no actions are required for any bags other than **introduce** bags. Let  $X_{i+1}$  be the next **introduce** bag we encounter during the course of the algorithm, and let  $X_i$  be the previous bag so that  $X_{i+1} \equiv X_i \cup \{v_{i+1}\}$  then update the data structures as follows:

For the iteration, we suggest adding for  $i=0, \dots, n-1$

1. For each  $u \in X_{i+1}$  that isn't  $v_{i+1}$ , set  $\text{APSP}[(v_{i+1}, u)] = 1$
2. For each  $y \in \text{visited}$ , if  $y \notin X_{i+1}$  then set  $\text{APSP}[(v_{i+1}, y)] = 1 + \min_{u \in X_i} \text{APSP}[(u, y)]$
3. Add  $v_{i+1}$  to **visited**

*Proof of runtime:*

Only the **introduce** bags require operations; there are exactly  $n$  of these (because each vertex gets introduced exactly once), so there are  $n$  total iterations. Each iteration consists of computing the distance of the new vertex to the other vertices in the bag, which requires  $O(k)$  work, then computing the distance from the new vertex to all previously visited vertices, which takes  $O(k)$  per vertex in **visited**. There are never more than  $n$  vertices in **visited**, so the work within an iteration is bounded by  $O(kn)$ . Thus, the whole algorithm is  $O(kn^2)$ .


*Proof of correctness:*

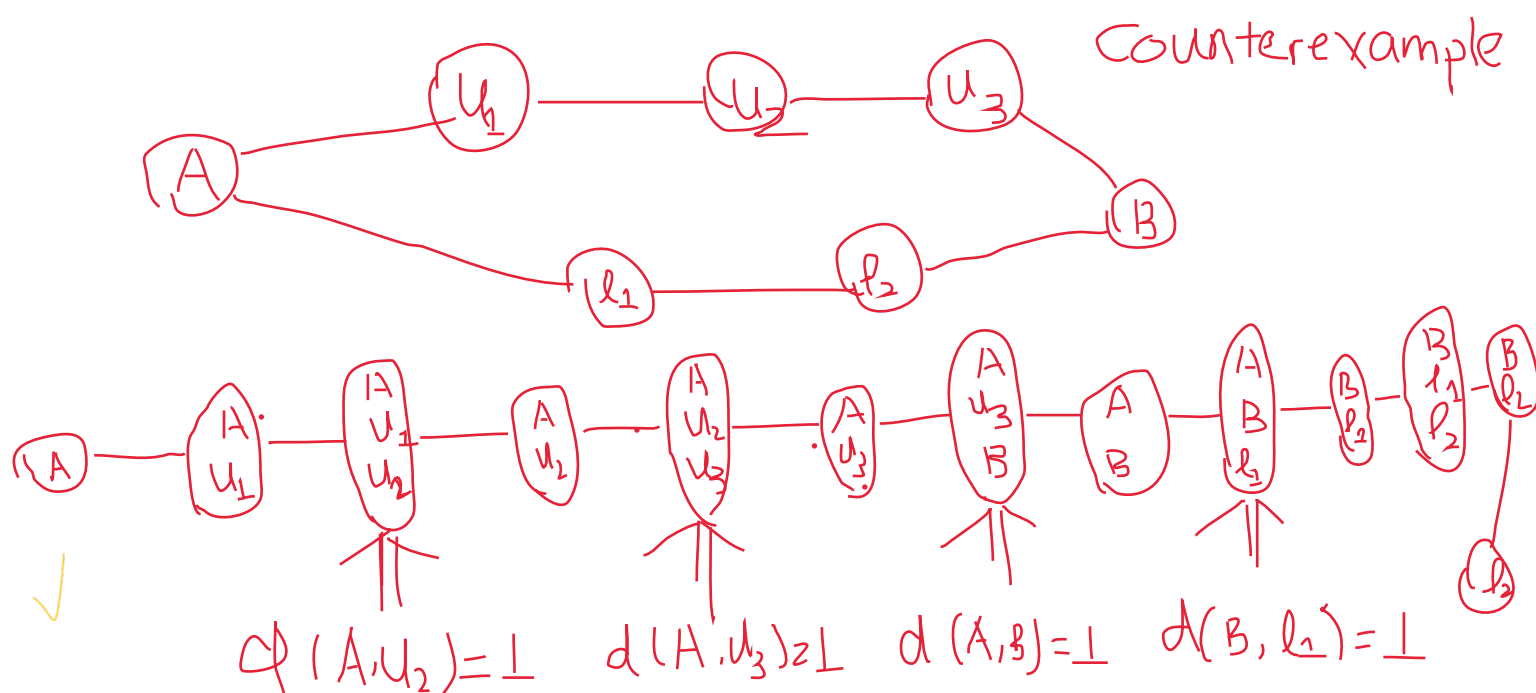
We will show that each vertex  $v$  has the correct distance computed to each vertex  $u$  introduced after it. This suffices to prove correctness.

Let  $v$  be the first vertex introduced. Then certainly the algorithm correctly computes the distance from  $v$  to every vertex that is ever in the same bag as  $v$  ( $\text{dist} = 1$ ). What about vertices that are never in the same bag as  $v$ ? If no such vertex exists, then  $v$  is an apex vertex and all its distances are correctly computed as 1.

If there exists a vertex in the bag that doesn't have connection to  $v$ , then the distance between the vertex and  $v$  is set to be 1 wrongly.

Suppose  $v$  is forgotten, and assume, by way of induction, that the algorithm has computed the correct distance from  $v$  to all visited vertices up to and including the vertices in the current bag being iterated over. (Note in the “base case” that in the bag immediately after  $v$  is forgotten this inductive assumption holds, i.e.,  $v$  has distance 1 to all vertices in that bag.) Now let  $u$  be the next vertex introduced. The algorithm correctly computes distance = 1 from  $u$  to each vertex in the current bag  $X_i$  (since there is an edge from  $u$  to every other vertex in  $X_i$ ). Then, by induction we have the correct distances from  $v$  to each vertex in  $X_i$ , and thus we can compute the correct minimum distance from  $v$  to  $u$  via the vertices in  $X_i$ . No other possible path from  $v$  to  $u$  is possible other than a path going through a vertex in  $X_{i-1} = X_i - \{u\}$  because  $X_{i-1}$  is a vertex-cut separating  $v$  from  $u$ .

This proves the algorithm gives the correct distances from the first vertex,  $v$ , to all vertices introduced after  $v$ . This same proof applies to any vertex  $v_j$ ; that is, the correct distance is computed from  $v_j$  to all vertices introduced after  $v_j$  is.   $\square$



The above is a counterexample by giving a graph and its nice path decomposition.

This is the same counterexample given by Kyle in class. This also works for this problem. The most problematic point is that in bags in the path decomposition above, there exists pairs of vertices  $(u, v)$  in bag  $X_{i+1}$  such that  $\text{dist}(u, v) = \infty$ , that is, there is no connection between  $u$  and  $v$ , where one of  $u$  and  $v$  is an introduced vertex. However, the algorithm immediately set the distance between  $u$  and  $v$  to be 1. For example, in the above path decomposition, there is no connection between node  $A$  and node  $B$ , but the algorithm set  $\text{dist}(A, B) = 1$  immediately in the corresponding iteration. Also, this problem leads to the incorrecion in the second step in some iterations.

Therefore, to fix this, we need to add an assumption that each bag must contain a clique. Namely, each two vertices in one bag should be adjacent. I think that in order to make it more maximal, we can assume a more restrict assumption. That is, we can assume that for each introduced vertex in every introduced bag, it must have connection to every other vertex in the bag.