CIS 502

Yezheng Li

September 23, 2015

1 Homework 3

Question (4-31, variant of Kruskal's algorithm) *Since Minimum Spanning Tree Problem is somehow related with shortest $u-v$ path, following variant of Kruskal's algorithm is designed:*

1. *Sort all the edges in order of increasing length (assume all edge lengths are distinct except that $d_{uv}=\infty$ once they are disconnected in certain graph);*
2. *construct a subgraph $H=(V,F)$ by considering each edge in order (from short to long);*
3. *when it comes to edge $e=(u,v)$, add e once $3l_e < d_{uv}$.*

Speaking of subgraph $H=(V,F)$ it produced:

1. *Prove that for every pair of nodes $u,v \in V$, the length of the shortest $u-v$ path in H is at most three times the length of the shortest $u-v$ path in G;*
2. *Despite its ability to approximately preserve shortest–path distances, the subgraph H cannot be too dense. Let $f(n)$ denote the maximum number of edges that can possibly be produced as the output of this algorithm, over all n– node input graphs with edge lengths. Prove that $\lim_{n\to\infty} \frac{f(n)}{n^2} = 0$.*

Answer 1. *Denote a shortest path in $G=(V,E)$ between $v_1, v_2 \in V$ by*

$$P_{12} \doteq e_{i_0 i_1} e_{i_1 i_2} \dots e_{i_{m-1} i_m} \overset{i_0=1, i_m=2}{=\!=\!=} e_{1 i_1} e_{i_1 i_2} \dots e_{i_{m-1} 2},$$

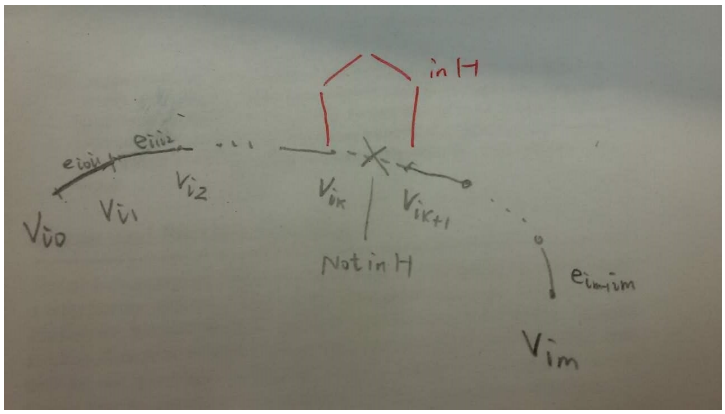

Figure 1: P_{12}, one of shortest paths between v_1, v_2 in $G = (V, E)$.

where $\{e_{i_k i_{k+1}} \doteq (v_{i_k}, v_{i_{k+1}})\}_{k=0}^{m-1} \subset E, \{v_{i_k}\}_{k=0}^{m} \subset V$ *and* $\sum_{k=0}^{m-1} l_{e_{i_k i_{k+1}}}$ *is the length of* P_{12}.

Speaking of any $e_{i_k i_{k+1}} \in E \setminus F$ *(or alternatively,*$\in G \setminus H$*) , there exists* $P'_{i_k i_{k+1}} \subset F$, *another path (red notation in figure 1 which as a matter of fact, could be shortest path between* $v_{i_k}, v_{i_{k+1}}$ *in* H*). In all, whatever* $e_{i_k i_{k+1}} \in F$ *or not, there exists a path in* H *such that its length is smaller than* $3 \sum_{k=0}^{m-1} l_{e_{i_k i_{k+1}}}$.

2. ***(refers to [3])***

Lemma 1 *Cycle in* H *has at least 5 edges.*

Proof *Suppose not and* C *is such a cycle while* $e = (u, v)$ *is the last edge added to it. Then at the moment* e *was considered, there was a* u*-*v *path* $Q_{uv} \in H = (V, F)$ *of at most three edges, on which each edge had length at most* ℓ_e. *Thus* ℓ_e *is not less than a third the length of* Q_{uv}, *and so it should not have been added.*

Lemma 2 *An n-node graph H with no cycle of length ≤ 4 must contain a node of degree at most $\sqrt{n}$.*

Proof *Suppose not and consider any node v of H. Let $S(v) \doteq \{u \in V : (u,v) \in F\}$, set of neighborhood of v and $\#S(v) = |S(v)| > \sqrt{n}$. Notice that there is no edge joining two nodes of $S(v)$, or we would have a cycle of length 3. Now let $N(S(v)) \doteq \{u \in V : S(u) \cap S(v) \neq \emptyset\}$, set of all nodes with a neighbor in $S(v)$. Since H has no cycle of length 4, each node in $N(S)$ has exactly one neighbor in $S(v)$. But $|S(v)| > \sqrt{n}$, and each node in S has $\geq \sqrt{n}$ neighbors other than v, so we would have $|N(S(v))| > n$, a contradiction.*

Proof (1, for 2^{nd} part of statement; refers to [3]) *Now, if we let $g(n)$ denote the maximum number of edges in an n-node graph with no cycle of length 4, then $g(n)$ satisfies the recurrence $g(n) \leq g(n-1)+\sqrt{n}$ by deleting the lowest-degree node and lemma 2, and so we have $g(n) \leq n^{3/2} = o(n^2)$.*

Question (4-32, variant of minimum-cost arborescence algorithm) *Consider a directed graph $G = (V, E)$ with a root $r \in V$ and nonnegative costs on the edges. In this problem we consider variants of the minimum-cost arborescence (MCA) algorithm.*

1. *MCA algorithm discussed in section 4.9 of [2] works as follows. We modify the costs, consider the subgraph of zero-cost edges, look for a directed cycle in this subgraph, and contract it (if one exists). Argue briefly that instead of looking for cycles, we can instead identify and contract strong components[a] of this subgraph;*

2. *After defining y_v to be the minimum cost of an edge entering v and we modified the costs of all edges e entering node v to be $c''_e \doteq \max(0, c_e - 2y_v)$ instead of $c'_e \doteq c_e - y_v$. This new change is likely to turn more edges to 0 cost. Suppose now we find an arborescence T of 0 cost. Prove that this T has cost at most twice the cost of the MCA in the original graph;*

3. *Assume you do not find an arborescence of 0 cost. Contract all 0−cost strong components and recursively apply the same procedure on the resulting graph until an arborescence is found. Prove that this T has at most twice the cost of the MCA in the original graph.*

[a]From [1], a graph is said to be strongly connected if every vertex is reachable from every other vertex.

Answer *1. Counterpart of (4.38) of [2] needs to be presented:*

> **Lemma 3** *Let D be a strong component in G consisting of edges of cost 0, such that $r \notin D$. Then there is an optimal arborescence rooted at r that has exactly one edge entering D.*

Proof *Is that true that a strong component might contain a circle?*

2. If $e_1, \ldots, e_{n-1}$ happened to be an arborescence with $c''_{e_i} = 0, \forall i \in [n-1]$, $c_{e_i} \leq 2y_{v_i}, i \in [n-1]$ had we denote end point of e_i by v_i. This indicates

$$\sum_{i=1}^{n-1} c_{e_i} \leq 2 \sum_{i=1}^{n-1} y_{v_i} \leq 2S(\textit{(by (4.36) of [2])},$$

when S is the sum of edge lengths in a MCA[1]. Therefore proof is done;

3.

Question (5-1) *While analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values – so there are $2n$ values total – and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.*

However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two data bases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Answer (refers to [3]) *Say A and B are the two databases and a_i, b_i are i^{th} smallest elements of A, B.*

[1]Minimum-cost arborescence.

1. ***Compare the medians of the two databases.*** $k \leftarrow \lceil \frac{1}{2}n \rceil$, *then* a_k *and* b_k *are the medians of the two databases. Without loss of generality, suppose* $a_k < b_k$. *Then one can see that*

$$b_k > a_k > a_j, b_k > b_j, j = 1, \ldots, k-1,$$

Therefore b_k *is at least* $2k^{th}$ *element in the combine databases.*

   (a) *Since* $2k \geq n$, *all elements that are greater than* b_k *are greater than the median and we can eliminate* $\{b_j, j = k+1, \ldots, n\} \subset B$. $B' \leftarrow \{b_j, j = 1, \ldots, k\}$, *the half of* B.

   (b) *Similarly, the first* $\lfloor \frac{1}{2}n \rfloor$ *elements of* A *are less than* b_k, *and thus, are less than the last* $n - k + 1$ *elements of* B. *Also they are less than the last* $\lceil \frac{1}{2}n \rceil$ *elements of* A.*So they are less than at least* $n - k + 1 + \lceil \frac{1}{2}n \rceil = n + 1$ *elements of the combine database. It means that they are less than the median and we can eliminate them as well.*
   $A' \leftarrow \{a_j, j = k+1, \ldots, n\} = \{a_j, j = \lfloor \frac{1}{2}n \rfloor + 1 \ldots, n\}$, *the remaining parts of* A .

2. *Now we eliminate* $\lfloor \frac{1}{2}n \rfloor$ *elements that are less than the median, and the same number of elements that are greater than median. It is clear that the median of the remaining elements is the same as the median of the original set of elements.*

   We can find a median in the remaining set using recursion[2] *for* A', B' *instead of* A, B.

[3] formally address the algorithm by writing recursive function:
`median(`n`,`$StartA$`,`$StartB$`)`*:*

1. *input: take integers* n, $StartA$ *and* $StartB$;

2. *output: return the median of the union of the two segments* $A[StartA+1; StartA+n]$ *and* $B[StartB+1; StartB+n]$;

3. *codes:* `median`*(*n, $StartA$, $StartB$*)*

[2]Note that we can't delete elements from the databases. However, we can access a'_i, b'_i, i^{th} smallest elements of A' and B' separately since $a'_i = a_{i+\lfloor \frac{1}{2}n \rfloor}, b'_i = b_{i+\lfloor \frac{1}{2}n \rfloor}$.

Code 1 *(a) if* $1 == n$ *then return* $\min(a_{StartA+k}, b_{StartB+k})$*;*
(b) $k \leftarrow \lceil \frac{1}{2}n \rceil$*;*
(c) if $a_{StartA+k} < b_{StartB+k}$ *then*
return `median(`k`,` $StartA + \lfloor \frac{1}{2}n \rfloor$ `,`$StartB$`);`
else return `median(`k`,` $StartA$`,` $StartB + \lfloor \frac{1}{2}n \rfloor$`).`

To find median in the whole set of elements we evaluate `median(`n`,0,0)`*.*
Let $T(n)$ *be the number of queries asked by our algorithm to evaluate* `median(`n`,`$StartA$`,`$StartB$`)`*. Then it is clear that* $T(n) = T(\lceil \frac{1}{2}n \rceil) + 2$*. Therefore* $T(n) = 2\lceil \log n \rceil = O(\log n)$*.*

Question (5-2) *Recall the problem of finding the number of inversions. As in the text, we are given a sequence of* n *numbers* $a_1, \ldots, a_n$ *which we assume are all distinct, and we define an inversion to be a pair* $i < j$ *such that* $a_i > a_j$*.*

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a significant inversion if $i < j$ *and* $a_i > 2a_j$*. Give an* $O(n \log n)$ *algorithm to count the number of significant inversions between two orderings.*

Answer *functions* `Merge-and-Count`*,* `Merge-and-Count2`*,* `Sort-and-Count` *are involved while*

- *function* `Merge-and-Count` *is the classic function for classic "counting-inversion" from page 324 of [2];*
- `Merge-and-Count` *is the actual merging process for the problem, calling function* `Merge-and-Count`*;*
- *function* `Sort-and-Count2` *differs from* `Sort-and-Count` *on page 325 of [2] only on one line – that is it calls* `Merge-and-Count2` *instead of* `Sort-and-Count`*.*

[r,L]=`Merge-and-Count`*(A,B) (classic "*`Merge-and-Count`*" function for classic "counting-inversion", from page 324 of [2]):*

Code 2 *1. Maintain a current pointer into each list, initialized to point to the front elements;*

2. Maintain a variable Count for the number of inversions, initialized

to 0*;*

3. *While both lists are nonempty:*
   (a) *Let* a_i *and* b_j *be the elements pointed to by the* Current *pointer;*
   (b) *Append the smaller of these two to the output list;*
   (c) *If* b_j *is the smaller element then: increment* Count *by the number of elements remaining in* A*;*
   (d) *Advance the* Current *pointer in the list from which the smaller element was selected.*
4. *Once one list is empty, append the remainder of the other list to the output;*
5. *Return* Count *and the merged list.*

[r,L]=`Merge-and-Count2`*(A,B) (actual "*`Merge-and-Count`*" function for the problem):*

Code 3
1. $[r, L1]$ =`Merge-and-count`*(2A,B) where* $2A \doteq \{2a \in \mathbf{R} : a \in A\}$ *(actually only need to find r);*
2. $[r1, L]$ =`Merge-and-count`*(A,B) (actually only need to merge two lists);*
3. *return r,L.*

[r,(sorted) L]=`Sort-and-Count`*(L) (different from the one in classic "counting-inversion" algorithm)*

Code 4
1. *If the list has one element then: return* $0, L$*;*
2. *divide the list into two halves: A contains the first* $\lfloor \frac{1}{2}n \rfloor$ *elements, B contains the remaining elements;*
3. $[r_A, A]$=`Sort-and-Count2`*(A);*
4. $[r_B, B]$=`Sort-and-Count2`*(B);*
5. $[r, L]$=`Merge-and-Count2`*(A,B); (the only major as classic "counting-inversion" algorithm)*

6. Return $r = r_A + r_B + r$ and the sorted list L.

Time cost: *Since time cost of* `Merge-and-Sort2` *is also $O(n)$, (5.1) and (5.2) of [2] also infer the whole algoritm to be $O(n \log n)$ with respect to time cost.*

References

[1] en.wikipedia.org.

[2] Jon Kleinberg, va Tardos, *Algorithm Design*, Addison-Wesley: 2005-3-26, 864 pages

[3] solcornell.tex.