ECS 222A: Assignment #5

Due on Tuesday,February 26, 2015

Daniel Gusfield TR 4:40pm-6:00pm

Wenhao Wu

Contents

Problem 1

Read (or skim, since we discussed it in class) the first 4 sections of the Bender et al. paper on the LCA problem. In class we derived a preprocessing time of $O(\sqrt{n} \times (\log_2 n)^3)$ to build the tables for all the binary vectors. In the Bender paper, the time stated for that is faster by a factor of $(\log_2 n)$. How do they achieve this improvement? (This is an easy problem given what we have already done in this class, but it makes sure you understand the method.)

Answer: As discussed in the class, the bottleneck in the preprocessing is, for each of the $\sqrt{n}/2$ possible binary vector of length $\log_2 n/2 - 1$ named b, where $b[i] \in \{-1, 1\}$, $i = 1, \ldots, \log_2 n/2 - 1$, compute

$$(\phi_b(u,v), \eta_b(u,v)) = \left(\min_{j\in[u,v]} \sum_{i=0}^{j-1} b[i], \arg\min_{j\in[u,v]} \sum_{i=0}^{j-1} b[i])\right), 1 \le u \le v \le \log_2 n/2 \tag{1}$$

in which we define $b[0] = 0$. In the original preprocessing method, the computation of each $(\phi_b(u,v), \eta_b(u,v))$ with a linear scan takes $O(\log_2 n)$ time. However, all $(\phi_b(u,v), \eta_b(u,v))$ can in fact be computed using dynamic programming. Specifically, define

$$\rho_b(v) = \sum_{i=0}^{v-1} b[i], \tag{2}$$

we have the recursion

$$\rho_b(v+1) = \rho_b(v) + b[v+1] \tag{3a}$$
$$\phi_b(u, v+1) = \min\{\phi_b(u,v), \rho_b(v+1)\} \tag{3b}$$
$$\eta_b(u, v+1) = \begin{cases} \eta_b(u,v) & \text{if } \phi_b(u,v) < \rho_b(v+1) \\ v+1 & \text{else.} \end{cases} \tag{3c}$$

and the base cases are

$$\rho_b(0) = 0 \tag{4a}$$
$$\phi_b(u,u) = b[u] \tag{4b}$$
$$\eta_b(u,u) = u \tag{4c}$$

In the DP, we can compute all $(\phi_b(u, u+d), \eta_b(u, u+d))$, $u = 1, \ldots, \log_2 n/2$ by increaseing d by 1 each time. In this way, the time to compute all $(\phi_b(u,v), \eta_b(u,v))$ is $O((\log_2 n)^2)$ for each b. Consequently, the overall preprocessing time is $O(\sqrt{n}(\log_2 n)^2)$, which is $\log_2 n$ faster than the linear scan method.

Problem 2

Now, read Section 5 of the LCA paper of Bender et al. In that paper, they call the Range Query problem that we discussed in class (where the list L comes from a tree) the +-1RMQ problem. In section 5, they show how to solve the general RMQ problem (where the consecutive values can differ by any amount). They show that the general RMQ problem can be solved in constant time, after $O(n)$ preprocessing, i.e., just as fast as the +-1RMQ. This is done by reducing the general RMQ problem to an LCA problem on a tree, in $O(n)$ time. Call LCA problem that the RMQ problem reduces to, the RLCA problem, and use RT to refer to the tree used in the RLCA problem.

Problem 2(a)

Is this an amazing result, or what? I mean, the fact that any LCA query can be answered in constant time, after only linear-time preprocessing, is already amazing, but probably one would (hand-waivingly) explain that it is somehow due to the fact that the adjacent L values differ by +1 or -1, which allows a 4-Russians approach. But, then to see that in fact, that condition is not needed ...

Answer: Yes. The key point is that one can transform a general RMQ problem into a LCA problem by building a Cartesian Tree with $O(n)$ amortized complexity.

Problem 2(b)

In class, we said that in some applications, only the LCAs of two leaves of a tree T are ever needed. In this true of the RT trees used to solve RLCA problems, in order to solve general RMQ problems? (This is also an easy problem, only really intended to be sure you read and understood Section 5.)

Answer: This is not likely to be true. In the general RMQ problem one might ask for the smallest element in the subarray $A[i, \ldots, j]$ for any pair of indices (i, j). However, in the RT tree, the nodes corresponding to indices i and j may not be leaf nodes. This indicates that solving LCA problem only for 2 leaf nodes for the RT trees constructed is not sufficient to solve general RMQ problems.

Problem 3

Read (or skim, since we have already discussed it in class) the Wagner paper on global minimum cut in an undirected graph.

Problem 3(a)

In the last sentence of Section 2, they state:

> "Notice that the starting vertex a stays the same throughout the whole algorithm. It can be selected arbitrarily in each phase instead."

Make an explicit argument for this claim.

Answer: According to Theorem 2.1. in Section 2, the generic algorithm from the lecture works if we can find any pair $s - t$ cut during each of the $n - 1$ iterations (phases) and then merge them. Since during each iteration, selecting arbitrary starting vertex eventually will result in some min $s - t$ cut, the starting point does not have to stay the same.

Problem 3(b)

It is known that in any edge-weighted undirected graph with n nodes, there is a set S of at most $n1$ cuts (bipartitions of the nodes), such that for any pair of nodes s, t (there are $\begin{pmatrix} n \\ 2 \end{pmatrix}$ pairs of nodes), S will contain a minimum st cut.
Do the $n1$ minimum cuts found by the Wagner algorithm form such a set S? Prove that they do, or find a counter-example.

Answer: Look at the example provided in Wagner's paper. The 7 cuts are

1. $\{1\}, \{2, 3, 4, 5, 6, 7, 8\}$, with $w = 5$;
2. $\{8\}, \{1, 2, 3, 4, 5, 6, 7\}$, with $w = 5$;
3. $\{7, 8\}, \{1, 2, 3, 4, 5, 6\}$, with $w = 7$;

Problem 3 [Problem 3(b)] continued on next page...

4. $\{4,7,8\}, \{1,2,3,5,6\}$, with $w = 7$;
5. $\{3,4,7,8\}, \{1,2,5,6\}$, with $w = 4$;
6. $\{1,5\}, \{2,3,4,6,7,8\}$, with $w = 7$;
7. $\{2\}, \{1,3,4,5,6,7,8\}$, with $w = 9$.

However, we can find the $5-6$ cut $\{1,2,3,4,5,7,8\}$, $\{6\}$ with $w = 6$ which is smaller than all the $5-6$ cut in the above list. This serves as a counter example indicating that the Wagner algorithm does not form such a set S.

Problem 4

Parametric P-P problem (this is probably the most time-consuming problem, but it mostly requires that you understand the Preflow-Push method.)
Recall that in HW 4, you showed that if f is some non-maximum $s-t$ flow in a graph G, and G_f is the residual flow graph with respect to f, then a maximum $s-t$ flow in G can be obtained by superimposing f with the maximum $s-t$ flow g in G_f.
Suppose one wants to compute a sequence of k maximum $s-t$ flows in a graph G, where between each flow computation the capacity on each edge out of s is either unchanged or increased, and all other capacities are left constant. We call this a parametric flow problem. There are many problems which can be solved by such a sequence of flow computations. If we consider each flow as an independent computation, the best resulting time bound is $O(kn^3)$.

Problem 4(a)

Using the result from HW 4, restated above, give a brief argument that the $(i+1)$ $s-t$ flow can be computed by changing the capacities on the edges out of s in the last residual graph used in the i-th maximum flow computation, and then finding the maximum flow in that altered residual graph. Essentially, then we are computing the $(i+1)$ $s-t$ flow, starting with the i-th flow in the $(i+1)$ $s-t$ graph.

Answer: Firstly, we note that the i-th $s-t$ flow is still a flow in the $(i+1)$-th graph, since the conservation at all nodes are still satisfied and

$$0 \le f_i(u,v) \le c_i(u,v) \le c_{i+1}(u,v) \tag{5}$$

Secondly, given $f_i(u,v)$, in the altered residual graph, the only edges whose capacities are potentially changed are

$$c_i^f(s,v) = c_i(s,v) - f_i(s,v) \longrightarrow c_{i+1}^f(s,v) = c_{i+1}(s,v) - f_i(s,v),\ v \in V \tag{6}$$

i.e. only the forward edge from s is altered in the residual graph. Consequently, according to HW4, to compute the $(i+1)-th$ max $s-t$ flow, we just need to change the capacities on the edges out of s in the last residual graph used in the i-th maximum flow computation, and then finding the maximum flow in that altered residual graph.

Problem 4(b)

Now we explore how the variant of the Preflow-Push algorithm which always picks the active node with largest d value works in this parametric environment.
Argue that the following method correctly computes the $i+1$ $s-t$ maximum flow in the sequence.

1. In the last residual graph used by the Preflow-Push algorithm in the i-th flow computation, remove all edges into s, and remove every node, other than s, whose d label is n or greater. Leave all of the node labels d on the remaining nodes unchanged. Call this graph G'.

2. Increase the capacities (as given by the new capacity data) of every edge of G' out of s to a remaining node in G'. This might create new forward edges. Call this graph G''.

3. Saturate all the edges in G'' out of s.

4. With this preflow, and the existing d labels, continue with the preflow-push algorithm, with the maximum d variant, on G''. When it stops, superimpose its flow with the flow for the i-th problem in the sequence. We claim that this gives a maximum flow in the $i+1$ $s-t$ graph in the sequence.

The key to proving that this is correct is to first show that Step 3 does indeed create a preflow in G'', and the node labels are valid for that preflow. After that, you only need to briefly argue that the overall algorithm is correct.

Answer:

Lemma 4.1 *In the above algorithm, Step 3 does indeed create a preflow in G'' and the node labels are valid fo that preflow.*

Proof Firstly we show that Step 3 creates a preflow in G''. Denote the node set in G' and G'' as V' the edge set in G'' as E''_f, and the capacity of edge (u,v) in G'' as $c''_f(u,v)$, then

- For each node v in $\{v|(s,v) \in E''_f, v \in V'\backslash\{s\}\}$, after Step 3, we have $f(v,u) = 0$ for all $u \in V'$, $f(s,v) = c''_f(s,v)$ and $f(u,v) = 0$ for $u \neq s$. Consequently, the excess of v is

$$\begin{aligned} e(v) &= \sum_{v \in V'} f(v,u) - \sum_{v \in V'} f(u,v) \\ &= c''_f(s,v) \end{aligned} \tag{7}$$

- For each node v in $\{v|(s,v) \notin E''_f, v \in V'\backslash\{s\}\}$, i.e. there is no edge from s to v in G'', we have $f(v,u) = f(u,v) = 0$ all $u \in V'$. Consequently, the excess of v is

$$\begin{aligned} e(v) &= \sum_{v \in V'} f(v,u) - \sum_{v \in V'} f(u,v) \\ &= 0 \end{aligned} \tag{8}$$

In summary, for any $u, v \in V'$, we have $0 \leq f(u,v) \leq c''_f(u,v)$ and for all $v \in V'\backslash\{s\}$ we have $e(v) \geq 0$, therefore after Step 3 there is indeed a preflow in G''.
Next we show that after Step 3 the node labels are valid. For all the remaining nodes in G'' their node labels are the same as in G_f the original residual graph used by the Preflow-Push algorithm. Denote the capacity of edge (u,v) in G' as $c'_f(u,v)$. In graph G'', for all $c''_f(u,v) > 0$, we must have $u \neq s$ since after Step 3 all edges (s,v) must have been satisfied. Since $u \neq s$, from Step 1 we know that $v \neq s$, concequently, edge (u,v) must be in the original residual graph. Thus

$$d(v) \leq d(u) + 1 \tag{9}$$

in the original residual graph. Since in G'' the labels of the remaining nodes are not changed, the labels are also valid in G''

Problem 4 [Problem 4(b)] continued on next page. . .

Since the flow from running the max-d algorithm on G'' serves as a maximum flow in te $i+1$-th altered residual graph. According to Problem 4(a), the overall algorithm results in the maximum $s-t$ flow in the $(i+1)$-th graph.

So far, we havent made much progress, since the time for each running of the above algorithm is bounded only by $O(n^3)$. But, I believe that the entire sequence of k flows can be computed in $O(n^3+kn)$ time, which is a dramatic improvement over the method that starts each flow from scratch. We will explore this in the next homework (or you can get started now).