TEL AVIV UNIVERSITY
Department of Computer Science
0368.4281 – Advanced topics in algorithms
Spring Semester, 2013/2014

Homework 2, March 29, 2014

Due on April 25. Please keep a copy of your homework.

1. Consider the Goldberg-Rao maximum flow algorithm. In class we "proved" that the distance from s to t in the residual network with respect to a length function which is 0 on arcs of residual capacity $\geq 3\Delta$ and 1 otherwise, increases in every iteration in which the flow is blocking. The proof was not rigorous enough. In this question you would write it precisely.

Recall that an iteration consists of the following steps:
1) Compute the distances according to the length function ℓ.
2) Change the length of special arcs to 0 (and thus making them admissible without changing distances).
3) Contract strongly connected components in the admissible subgraph.
4) Compute a blocking flow from s and t in the admissible subgraph with strongly connected components contracted.
5) Reduce the flow if needed to be of value Δ.
6) Route flow in each contracted component using incoming and outgoing trees as shown in class.

Prove that if the flow is blocking then the distance from s to t according to the length function ℓ' in the next iteration is strictly larger than the distance from s to t according to ℓ.

2. In this question we analyze an algorithm that finds a blocking flow from s to t in a directed acyclic graph $G(V, E)$ in $O(n^2)$ time where $n = |V|$. This is without dynamic trees.

We define the excess $e(v)$ of a vertex v to be the difference between the flow incoming to v and the flow outgoing of v. A preflow is an assignment of a flow for each arc that obeys the capacity constraints and $e(v) \geq 0$ for all v. The algorithm maintains a preflow f which is initialized by setting $f(s, v) = u(s, v)$ for all edges outgoing from s. We denote by $r(v, w)$ the residual capacity of an arc (v, w) with respect to the current preflow. A vertex $v \neq \{s, t\}$ with $e(v) > 0$ (more flow entering v than outgoing of v) is *active*.

Each vertex in $V \setminus \{s, t\}$ is either *blocked* or *unblocked*. Initially all vertices in $V \setminus \{s, t\}$ are unblocked.

We maintain two pointers per vertex v: 1) A *current outgoing arc pointer* which points to the next arc in the list of arcs outgoing from v. It is initialized to the first arc on this list. 2) A *current incoming arc pointer* which points to the next arc in the list of arcs incoming to v. It is initialized to the first arc on this list.

The algorithm makes use of the three operations push, pull, and block. A push operation applies to an arc (v, w) such that v is active, v and w are unblocked, and $r(v, w) > 0$. It consists of increasing $f(v, w)$ by $\min\{e(v), r(v, w)\}$. A pull operation applies to an arc (v, w) such that w is active and

blocked and $f(v,w) > 0$. It consists of decreasing $f(v,w)$ by $\min\{e(w), f(v,w)\}$. A block operation applies to an unblocked vertex v and makes it blocked.

A *discharge* operation on an active vertex v applies the following three operations to v as long as it is active. 1) If v is unblocked we perform push on the current outgoing arc (v,w) if possible, and advance the current outgoing arc pointer if the push has saturated (v,w). 2) If the current outgoing arc pointer reaches the end of the list and v is still active then we block v and stop the discharge. 3) If v is blocked we perform a pull on the current incoming arc (u,v) if possible and advance the incoming arc pointer.

We may assume that s has no incoming arcs and let T be a topological order of the graph that starts with s. Let T^r be the reverse of T. Let L be a concatenation of T^r and T (we can remove one of the two consecutive copies of s in L). The algorithm uses the list L to determine the order in which to process active vertices. A blocked active vertex will be marked in T^r and a non blocked active vertex will be marked in T.

The algorithm consists of repeating the following three steps until there are no active vertices: Select the next marked vertex in L (in a round robin fashion, when you reach the end of L start from the beginning again), say this vertex is v. Apply a discharge operation to v marking vertices as they become active in T if they are unblocked and in T^r is they are blocked.

a. Prove that the algorithm maintains the invariant that if v is blocked then there is a saturated edge on each path from v to t.

b. Prove that every $n-1$ discharge operations at least one vertex becomes blocked.

c. Prove that the algorithm finds a blocking flow in $O(n^2)$ time.

3. (a) Given a minimum cost circulation in a graph $G = (V,E)$, show how to compute node potentials $\pi(v)$, $v \in V$, such that all reduced costs with respect to π ($c^\pi(e)$) of residual edges are non-negative. What is the running time of your algorithm ?
(b) Let $\pi(v)$, $v \in V$, be potentials such that there exists a feasible circulation f whose residual edges have non-negative reduced costs with respect to π. Given π show how to find a minimum cost circulation whose residual edges have non-negative reduced cost with respect to π. What is the running time of your algorithm ?

Try to find efficient and simple algorithms.

4. Give a graph $G = (V,E)$, with non-negative costs and capacities, a source s, and a sink t, consider the following algorithm for finding a maximum flow from s to t of minimum cost. We start with the 0 flow, find an augmenting path of minimum cost in the residual network, augment as much flow along the path as possible, update the residual network, and repeat (that is find a minimum cost augmenting path in the new residual network, augment, and so on).
a) Prove that if all costs and capacities are integers then this algorithm indeed finds a max flow from s to t of minimum cost.
b) Assume the maximum cost of an edge is C and the maximum capacity of an edge is U. Suggest a concrete implementation of this algorithm and analyze its running time.

5. Consider the dynamic graph connectivity problem **on an undirected forest** where we allow only two operations:

a) *connected*(v, w) that returns "yes" if and only if v and w are in the same connected component, and
b) *delete*(v, w) that deletes the edge (v, w) from the forest.
Assume we perform k such operations on a forest with n nodes. Describe an efficient (in an amortized sense) algorithm for the problem. (Note that from the material we learned in class easily follows a solution that runs in $O(k \log n)$ time, but better solutions exist in particular if $k >> n$.)