Optimization II **Winter 2009/10**

Lecture 4 — November 2

Lecturer: Julián Mestre

4.1 Implementation of generic push-relabel

To obtain an efficient implementation of the push-relabel we need two new ingredients. First, we need to be able to select an active node u in each iteration; that is, a node with $e_f(u) > 0$. To that end, we keep a list of all the active nodes. In each iteration we look a the vertex at the front of the list. A push from u to v may make u inactive or v active. If u becomes inactive, we just remove u from the list. If v becomes active, we add it to the end of the list. Clearly, this can be done in $O(1)$ per push operation.

Second, once u is chosen we need to find an edge $(u, v) \in G_f$ such that $h(v) < h(u)$. To do this, we start scanning the edges in $\delta^{in}(u) \cup \delta^{out}(u)$ in some fixed but arbitrary order. If no appropriate edge is found, we relabel u and start over. Otherwise, we do a push and remember the edge e we pushed along so that if in a later iteration we need to find another edge out of u, we start scanning from e and not from the beginning of the list.

Theorem 4.1. *The generic* PUSH-RELABEL *algorithm can be implemented to run in $O(n^2m)$ time.*

Proof: Let $T(n, m)$ be an upperbound on the number of iterations of the generic algorithm. A node can scan the list of adjacent nodes at most as many times as it is relabeled. Thus, the algorithm runs in $O(nm + T(n, m))$. Recall that $T(n, m) = O(n^2m)$, which was the number of non-saturating push operations. $\square$

In order to improve the running time we need reduce the number of non-saturating push operations. We can achieve this by being more careful when choosing the active node.

4.2 Highest-label rule

In this section we analyze the following refinement of the basic PUSH-RELABEL algorithm. In each iteration, instead of choosing an arbitrary node, we select the one having the highest label.

The implementation details are as before. The only difference is that instead of keeping a list of active nodes, we keep a list of the currently used label values. For each value we keep track of the nodes having the given value. Furthermore, we split these into active and inactive node. The updates to maintain this structure are $O(1)$ per push or relabel.

Lemma 4.2. *The number of non-saturating push operation done by the* PUSH-RELABEL *algorithm using the highest-label rule is $O(n^2\sqrt{m})$.*

Proof: Use the potential function $\Phi = \sum_{u \text{ active}} \frac{|\{v \mid h(v) \leq h(u)\}|}{\sqrt{m}}$. Initially, $\Phi \leq \frac{n^2}{\sqrt{m}}$ and always $\Phi \geq 0$. Relabel increases Φ at most $\frac{n}{\sqrt{m}}$. Saturating push increases Φ at most $\frac{n}{\sqrt{m}}$. A non-saturating push does not increase Φ.

We split the execution time into phases. A phase consists of all pushes between two consecutive changes of $\max_{u \text{ active}} h(u)$. A phase is *cheap* if it does less than $\sqrt{m}$ non-saturating pushes, and *expensive* otherwise.

The number of phases is at most $4n^2$—at most twice the number of relabel operations. Thus, we only need to bound the number of non-saturating pushes in expensive phases. Each non-saturating push in an expensive phase decreases Φ by at least 1. Thus, the number of non-saturating push operation in expensive phases is at most $O(n^2\sqrt{m})$. $\square$

Theorem 4.3. *The* PUSH-RELABEL *algorithm with the highest-label rule can be implemented to run in $O(n^2\sqrt{m})$ time.*

Proof: Let $H(n, m)$ be an upperbound on the number of iterations of the algorithm when using the highest-label rule. Again, a node can scan the list of adjacent nodes at most as many times as it is relabeled. Thus, the algorithm runs in $O(nm + H(n, m))$. Now a quick calculation shows that the bottleneck is still the number of non-saturating push operations, so $H(n, m) = O(n^2\sqrt{m})$. $\square$

4.3 Excess scaling

We conclude our treatment of the push-relabel method with a variant of the basic algorithm whose aim is to minimize the maximum excess. The idea is to work in iterations; halving the excess in each iteration. Initially, $\max_{u \in V} e^f(u)$ may be as high as $U = \max_{e \in G} c(e)$, so we may end up having $\log U$ iterations.

Algorithm 1 EXCESS-SCALING(V, E, c, s, t)

```
1. initialize $h$ and $f$
2. $\Delta = 2^{\lceil \log U \rceil}$.
3. while $\Delta \geq 1$ do
4.    while $\exists\, u$ with $e^f(u) > \Delta/2$ do
5.       let $u$ be such node with smallest label
6.       do push/relabel out of $u$ making sure $\max_{v \in V} e^f(v) \leq \Delta$
7.    $\Delta \leftarrow \Delta/2$.
```

In each iteration we will maintain the invariant that $\Delta/2 < \max_{u \in V} e^f(u) \leq \Delta$. To that end we make sure that when pushing flow from u to v we never make the excess of v larger than Δ. In other words, if the residual capacity of the edge $(u, v) \in G_f$ is $r(u, v)$ then we push at most $\min\left\{e^f(u), r(u, v), \Delta - e^f(v)\right\}$ units of flow along (u, v).

Lemma 4.4. *Each non-saturating push transfers at least $\Delta/2$ units of flow. Also, no vertex excess ever exceeds Δ.*

Lemma 4.5. *The* EXCESS-SCALING *algorithm performs $O(n^2)$ non-saturating push operations per scaling phase.*

Proof: Use the potential function $\Phi = \sum_{u \in V} \frac{e^f(u)h(u)}{\Delta}$. Each relabel increases Φ by at most 1. Thus, the total increase due to relabeling of nodes is no more than $2n^2$. Push operations decrease Φ and non-saturating pushes decrease Φ by at least 1/2 $\square$

Theorem 4.6. *The* EXCESS-SCALING *algorithm can be implemented to run in* $O(nm + n^2 \log U)$ *time.*