15-451/651: Design & Analysis of Algorithms October 10, 2013
Lecture #14 last changed: October 21, 2013

Today's highlights:

- Flows and preflows
- The Push-Relabel Algorithm
- An improved algorithm using the first-in-first-out policy
- Minimum-Cost Matchings and Min-Cost Max-Flow

1 Introduction

In the previous lecture we talked about augmenting path based approaches to finding maximum flows in graphs: there was Ford-Fulkerson, and then refinements based on choosing the augmenting paths carefully (the two Edmonds-Karp algorithms, fattest augmenting path and shortest augmenting path), and finally the blocking-flows based algorithms (Dinic/MPM).

Today we'll see a fairly different approach to finding maximum flows in graphs: it is based on pushing as much flow as possible out of the source, and trying to get it to the sink. Of course, it may be possible to push a lot of flow out of the source only to find a small cut further down the graph, so we then have to push the "excess" flow back to the source. This is called the push-relabel approach, and is due to Andrew Goldberg and Bob Tarjan.

2 The Push-Relabel Algorithm

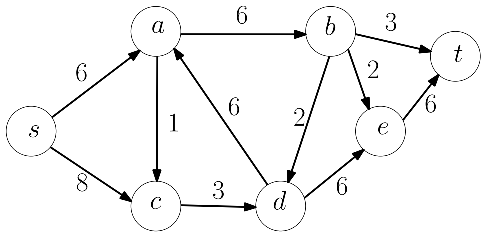
Think of the push-relabel algorithm as s impatiently sending flow to nodes "downstream" from it, which in turn try to send flow to nodes "downstream" from them, until some of the nodes cannot send any more. They re-evaluate the situation. In particular, they re-evaluate what "downstream" means. Eventually they send the flow back to s. Let's see how.

We are given a directed capacitated graph $G = (V, E)$ with source s and sink t, and want to find a max s-t flow. Let's imagine all nodes are reachable from s, and can reach t. (All other nodes can be deleted.) For brevity, we denote the directed edges $(u, v) \in E$ merely as uv.

2.1 Preflows

The algorithm will send flow along edges. But in contrast to the previous algorithms, we will not maintain a flow at each point in time — we will only maintain a "preflow". What's a preflow? It's what we get when we don't have flow conservation at even the non-s-t nodes: in a preflow the flow entering the node is allowed to be more than the flow leaving it. (It is not allowed to be less.) So if f is a preflow, then for all $v \in V \setminus \{s, t\}$, $\sum_{uv\in E} f(uv) \geq \sum_{vw\in E} f(vw)$.

Consider this s-t flow network:



Here's an example of a preflow.

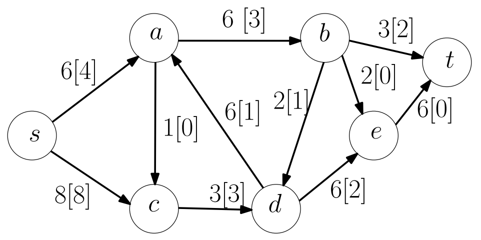
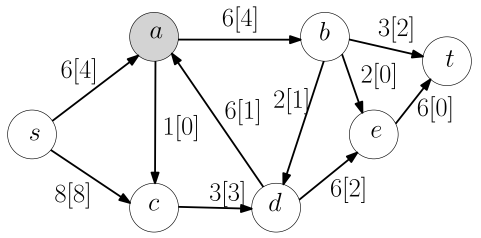


Figure 1: The running preflow example.

Every flow is trivially a preflow. And here is an example of something that is *not* a preflow, since node a has more flow leaving it than entering it.

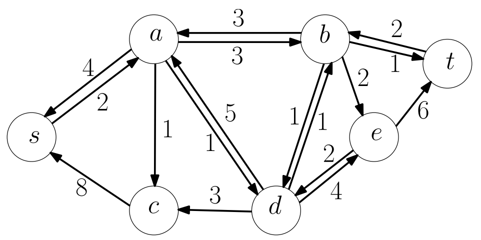


Given a preflow f, a node v has some *excess* value $e_f(v) \geq 0$, this is just the difference between the flow entering the node and the flow leaving it. Namely,

$$e_f(v) := \sum_{uv \in E} f(uv) - \sum_{vw \in E} f(vw).$$

A node v is *active* if $v \notin \{s,t\}$ and $e_f(v) > 0$. (In the example from Figure 1, the nodes a, c, d, e, are all active, with excesses of $2, 5, 1$ and 2 respectively.)

Let E_f denote the edges of the residual graph with respect to the preflow (and let G_f denote the residual graph itself). As before, when you send flow f along an edge uv with capacity c, in the residual graph you get an edge uv with capacity $c - f$, and you also get the reverse edge vu with capacity f. If $c = f$ then we drop the edge uv from E_f (since it has residual capacity $c - f = 0$), so E_f only contains the arcs with non-zero residual capacity. The residual graph corresponding to the preflow in Figure 1 is this:



2.2 Labelings

Each node v is given a *label* $d(v) \in \mathbb{Z}_{\geq 0}$. A labeling is *valid for preflow* f if for all edges uv in E_f, we have

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

We will maintain the invariant that our current labeling is valid for the current preflow. Call an arc uv *admissible* if $uv \in E_f$ and

$$d(u) \geq d(v) + 1. \tag{2}$$

Since we always have a valid labeling, we also have the inequality (1). Putting the two together, admissible edges uv in fact satisfy the *equality*

$$d(u) = d(v) + 1. \tag{3}$$

These admissible edges will be the ones we will try to advance flow on.

3 The Push-Relabel Algorithm

Initialization. We start off by saturating all the edges sv coming out the source s. In other words, we send $c(sv)$ flow on each such edge $sv \in E$. This gives a valid preflow f_0. Moreover, the residual graph with respect to this preflow f_0 is almost the same as the original flow network G, except all the arcs $sv \in E$ have been replaced by arcs vs. For each node v with $sv \in E$ (and hence $vs \in G_{f_0}$), we have excess $e_{f_0}(v) = c(sv)$.

Also, we set the original labeling $d(s) = n$, $d(v) = 0$ for all $v \in V, v \neq s$. Note that this is a valid labeling with respect to f_0. In other words, every edge uv in the residual graph G_{f_0} satisfies $d(u) \leq d(v) + 1$. (Exercise: Check!)

The Main Loop. While there is an active node u, do one of the following for u.

> **Push**(u). If $\exists v$ with admissible arc $uv \in E_f$, then send flow $\delta := \min(c_f(uv), e_f(u))$ from u to v. Note that this causes excess $e_f(u)$ to fall by δ, and excess $e_f(v)$ to increase by δ. If $\delta = c_f(uv)$, this is called a *saturating* push, else it is an *non-saturating* push.
>
> **Relabel**(u). If for all arcs uv out of u, either uv is not in E_f or $d(v) \geq d(u)$, then set
>
> $$d(u) = 1 + \min_{v:uv \in E_f} d(v). \tag{4}$$

3.1 The Analysis

Claim 1 *The labels d remain valid with respect to the current preflow f.*

Proof: The proof is by induction. Initially d is valid with respect to f_0. When we send flow along an edge uv, this adds the residual arc vu. But since we just sent flow on the admissible arc uv, we know that $d(u) = d(v) + 1$, so the new arc $vu \in E_f$ satisfies $d(v) = d(u) - 1 \leq d(u) + 1$, and the labels remain valid.

When we relabel a node, the new label set as in (4) is the largest it can be while remaining valid. This proves the claim. ■

Claim 2 *If f is a preflow and v has excess $e_f(v) > 0$, then v has a path in E_f to source s.*

Proof: By induction. Initially, in flow f_0 only the nodes v such that $sv \in E$ have excess, but we now have $vs \in E_{f_0}$. If we send flow from active node u (which had excess and hence inductively had a path P to s) to v, then we add in edge vu to E_f. Hence now v has excess, but $vu \circ P$ is a path from v to s in the new residual graph G_f. Relabeling does not change this at all. ■

Claim 3 *Suppose the algorithm terminates, then the final preflow is a maximum s-t flow.*

Proof: If the algorithm terminates with flow f, then none of the nodes in $V \setminus \{s, t\}$ is active, and these nodes have no excess. There is flow-conservation at all nodes except s and t. Hence f is a flow.

Now suppose this is a not a maximum flow. Then we know there must be an augmenting s-t path — i.e., a path from s to t in the residual graph G_f. Let this path be $v_0 = s, v_1, v_2, \ldots, v_k = t$.

Since the final labeling d must be valid (by Claim 1), we know that each $d(v_i) \leq d(v_{i+1}) + 1$. Hence

$$d(s) = d(v_0) \leq d(v_k) = d(t) + k.$$

But the path can have at most $n - 1$ edges, so $k \leq n - 1$, and $d(s) \leq d(t) + n - 1$. But this contradicts the fact that $d(s) = n$ and $d(t) = 0$. ■

Claim 4 *At any time, the node labels are at most $d(v) \leq 2n - 1$.*

Proof: The only nodes that are relabeled are active nodes. By Claim 2 such a node u has a path P to s, say $u = v_0, v_1, v_2, \ldots, v_k = s$. Again, the valid labels mean that $d(u) \leq d(s) + k$. The path has at most $n - 1$ edges, so $d(u) \leq d(s) + n - 1 = n + (n - 1) = 2n - 1$. ■

3.1.1 Bounding the Number of Operations

Lemma 5 (Relabels) *The number of relabel operations is at most $2n^2$.*

Proof: Each relabel increases the label of some node in $V \setminus \{s, t\}$ by at most 1. The maximum label is $2n - 1$ by Claim 4, and there are $n - 2$ nodes. This gives a bound of at most $(n-2)(2n-1) \leq 2n^2$ on the number of relabel operations. ■

Lemma 6 (Saturating Pushes) *The number of saturating push operations is at most mn.*

Proof: Consider an edge uv and suppose consider two consecutive saturating pushes that it participates in. Suppose the earlier one happens when the labels are $d_1(u) = d_1(v) + 1$. Now because this is a saturating push operation, the arc uv disappears from the residual graph. For this to reappear, we must push flow in the opposite direction, from v to u. This happens when $d_2(v) = d_2(u) + 1$. But since the labels only increase, this happens when $d_2(v) \geq d_1(u) + 1 = d_1(v) + 2$. Finally, the later saturating push on uv again happens when $d_3(u) = d_3(v) + 1 \geq d_2(v) + 1 \geq d_1(u) + 2$.

So every subsequent saturating push on an arc uv (except the very first one) only happens the label $d(u)$ increases by 2. This means the number of saturating pushes on any arc uv is at most $1 + \lfloor (2n - 1)/2 \rfloor = n$. There are m arcs, and hence a total of at most mn saturating pushes. ■

Lemma 7 (Non-Saturating Pushes) *The number of non-saturating push operations is $O(mn^2)$.*

Proof: For this part we will use a potential function

$$\Phi = \sum_{\text{active vertices } v} d(v).$$

Note that the initial potential Φ_{init} is zero, and the potential remains positive at all times (and hence the final value $\Phi_{final} \geq 0$). How does the potential change over the course of the algorithm?

- **Relabelings.** The label for each node only increases. If we relabel a node (which must be active), the potential increases by exactly the increase in its label. Hence the total increase in potential due to relabelings is at most $2n^2$.

- **Saturating Pushes.** What happens in a saturating push? Say we pushed along uv, and we sent $c_f(u,v)$ amounts of flow. It may be the case that v now has an excess (and $v \neq s, t$) and so v is active. Moreover, it may be that u still has an excess. So

  $$\text{change in } \Phi \leq d(v) \leq 2n - 1.$$

  Hence the total increase in Φ due to saturating pushes is at most $mn \times (2n-1) \leq 2mn^2$.

Good. In total the relabels and saturating pushes can only push the potential up by at most $2mn^2 + 2n^2 \leq 4mn^2$. Now to show that each non-saturating push only decreases the potential.

- **Non-Saturating Pushes.** Suppose we push along uv. Since this is non-saturating, we must have sent $e_f(u)$ along the edge. After the push, u has no more excess, it is not active. So it does not contribute to Φ any more. On the other hand, v is possibly active (unless it is s or t), so it may have started contributing to Φ. But in any case, the potential decreases by at least

  $$d(u) - d(v) = 1.$$

  (The equality holds because uv was an admissible edge.)

In other words, each non-saturating push decreases the potential by at least 1. But the total increase is at most $4mn^2$, so the number of non-saturating pushes is at most $4mn^2$. ■

We can implement the algorithm so that each step (relabel or push) takes constant time. So the total runtime is $O(mn^2)$. Not yet faster than the other algorithms we've seen, but it's a very different algorithm from the augmenting paths algorithms we've seen so far. And we can improve the runtime further.

4 Improving on the Basic Push-Relabel

One simple way to improve things a little is to not start off with the simple-minded initial labeling d_0. We could start off with the "highest" valid labeling. We could set $d(s) = n$, and for all other nodes $v \in V \setminus \{s\}$, set $d(v)$ to the shortest path (in terms of the number of arcs) in G_{f_0} from v to the sink t. For this we can just run Dijkstra's algorithm, which has an impementation in $O(m + n \log n)$ time using a fancy data strcuture called Fibonacci heaps (due to our own Danny Sleator, and Bob Tarjan). But this does not change the Theta bound.

On the other hand, there is a lot more freedom in our choice of which active node to work on. Here are some options:

- *First-in-first-out (FIFO).* Keep a queue, add newly active vertices to the end of the queue. When you need an active vertex, take the node at the front of the queue.

  We can show that this takes $O(n^3)$ time, the same as the blocking flow algorithm of Dinitz.

- *Highest Label.* The idea behind this is that the highest labels are the ones furthest away from the sink t. (Especially if you chose the "smarter" starting labeling.) The hope is that working on those nodes will move the flow towards the sink, and aggregate them along the way. This can be implemented in $O(n^2\sqrt{m})$ time, which is never worse than $O(n^3)$ and often better when the network is sparse.

- *Excess Scaling.* In this algorithm, you start off a "phase" with the non-s-t node excesses being at most some Δ, and end it when the excesses are $\Delta/2$. To do this, we don't send $\delta := \min(c_f(uv), e_f(u))$ flow in every push, but often push some (integer) amount less than that. This is done in a way to take $O(mn)$ time per phase. If $C := \max_{sv \in E} c(s,v)$ is the initial maximum excess, this will require $\lceil \log_2 C \rceil$ phases before the excesses fall below 1, and hence are zero, giving a flow. So this takes $O(mn \log C)$ time.

Finally, if we use the clever *dynamic tree* data structure (again due to Sleator and Tarjan), the running time of FIFO can be made $O(mn \log(n^2/m))$.

4.1 First-In-First-Out

What's the idea? Just look at the basic analysis, the bottleneck is the bounding the number of non-saturating pushes (using the potential). Can we bound that number in some other way when we have FIFO?

The idea is this: Why not keep pushing and relabeling u until it becomes inactive? Call this set of pushes and relabels a "discharge" on u. Note that while the last push in a discharge on u could have been a non-saturating push, all other pushes must have been saturating. So if we could bound the number of discharges by D, the number of non-saturating pushes is also at most D.

In FIFO, we have a queue. The nodes initially put in the queue are the neighbors of s. Call them phase 1 nodes. Now discharging these phase-1 nodes causes some nodes to be added to the queue, call these phase-2 nodes. Similarly, discharging phase-i nodes causes some nodes to be added to the queue, call these phase-$(i+1)$ nodes. Of course, nodes may be added in many of the phases — node v may get discharged in phase i, but get added again (as a phase-$(i+1)$ node) when it receives flow from during w's discharge.

Claim 8 *There are at most $4n^2$ phases.*

Proof: Again, this will be a potential-function-based proof. The potential is now

$$\max_{\text{active vertices } v} d(v).$$

(If there are no active vertices, we define the empty maximum to be 0.)

Let's see how this potential behaves. Call a phase *good* if no vertex is relabeled during the phase. If a phase is good, it must only have had pushes. We claim that the potential Φ at the end of a good phase must be strictly smaller than at the beginning. Hence each good phase decreases Φ by at least 1. Since the potential stays non-negative, this means the number of good phases can be bounded by the total increase in Φ over the course of the algorithm. But Φ can only increase when vertices get relabeled. And the total increase in Φ is at most $2n^2$. So the number of good phases is at most $2n^2$.

How many non-good phases? Each non-good phase had at least one relabel, so the number of these phases is at most the number of relabels, a further $2n^2$. Summing up the two completes the proof. ■

Corollary 9 *The total number of non-saturating pushes is $O(n^3)$.*

Proof: Each node is discharged at most once in each phase. There are $O(n^2)$ phases by Claim 8. So there are $O(n^3)$ discharges. And each non-saturating push belongs to some discharge. ■

5 Min-cost Matchings, Min-cost Max Flows

We talked about the problem of assigning groups to time-slots where each group had a list of acceptable versus unacceptable slots. A natural generalization is to ask: what about preferences? E.g, maybe group A prefers slot 1 so it costs only \$1 to match to there, their second choice is slot

2 so it costs us \$2 to match the group here, and it can't make slot 3 so it costs \$infinity to match the group to there. And, so on with the other groups. Then we could ask for the *minimum cost* perfect matching. This is a perfect matching that, out of all perfect matchings, has the least total cost.

The generalization of this problem to flows is called the *min-cost max flow* problem. Formally, the min-cost max flow problem is defined as follows. We are given a graph G where each edge has a *cost* $w(e)$ in as well as a capacity $c(e)$. The cost of a flow is the sum over all edges of the positive flow on that edge times the cost of the edge. That is,

$$\mathit{cost}(f) = \sum_e w(e) f^+(e),$$

where $f^+(e) = \max[f(e), 0]$. Our goal is to find, out of all possible maximum flows, the one with the least total cost. We can have negative costs (or benefits) on edges too, but let's assume just for simplicity that the graph has no negative-cost cycles. (Otherwise, the min-cost max flow will have little disconnected cycles in it; one can solve the problem without this assumption, but it's conceptually easier with it.)

Min-cost max flow is more general than plain max flow so it can model more things. For example, it can model the min-cost matching problem described above.

There are several ways to solve the min-cost max-flow problem. One way is we can run Ford-Fulkerson, where each time we choose the *least cost* path from s to t. In other words, we find the shortest path but using the costs as distances. To do this correctly, when we add a back-edge to some edge e into the residual graph, we give it a cost of $-w(e)$, representing that we get our money back if we undo the flow on it. So, this procedure will create residual graphs with negative-weight edges, but we can still find shortest paths in them using the Bellman-Ford algorithm.

We can argue correctness for this procedure as follows. Remember that we assumed the initial graph G had no negative-cost cycles. Now, even though each augmenting path might add new negative-cost *edges*, it will never create a negative-cost *cycle* in the residual graph. That is because if it did, say containing some back-edge (v, u), that means the path using (u, v) wasn't really shortest since a better way to get from u to v would have been to travel around the cycle instead. So, this implies that the final flow f found has the property that G_f has no negative-cost cycles either. This means that f is optimal: if g was a maximum flow of lower cost, then $g - f$ is a legal circulation in G_f (a flow satisfying flow-in = flow-out at *all* nodes including s and t) with negative cost, which means it would have to contain a negative-cost cycle, a contradiction.

The running time for this algorithm is similar to Ford-Fulkerson, except using Bellman-Ford instead of Dijkstra. It is possible to speed it up, but we won't discuss that in this course.