Lecture Notes for Advanced Algorithms

Lecture of November 1, 2012

The Push-Relabel (or Preflow-Push) Algorithm

This algorithm, devised by Andrew Goldberg in his PhD dissertation (his advisor was, unsurprisingly, Robert Tarjan), focuses on vertices rather than edges. In that, it bears some similarity to a classic algorithm of Karzanov's, which blocks vertices rather than edges and achieves a running time in $O(|V|^3)$. However, it differs from it in that it never maintains a valid flow, only what we will call a *preflow*—only at the conclusion of the last operation will the preflow have turned into a flow. The running time of the naïve version is $O(|E| \cdot |V|^2)$, but it is easily improved to $O(|V|^3)$ and beyond that, with increasing complications, to $O(|E| \cdot |V| \cdot \log(|V|^2/|E|))$.

Flow conservation requires that, for every vertex $v \in V \setminus \{s, t\}$, the flow coming in v equals that leaving v, or $\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$. A preflow is a relaxation of flow: we now only require $\sum_{u \in V} f(u, v) \geq \sum_{w \in V} f(v, w)$. If the flow coming into v exceeds that leaving v, we say that v has *excess capacity*; thus the excess flow at v is given by $e(v) = \sum_{u \in V} f(u, v) - \sum_{w \in V} f(v, w)$. We define v as an active vertex if $e(v) > 0$.

The algorithm itself can be viewed as a gravity-assisted procedure. Picture the network held up by the source node, with the edges and other vertices dangling below, and the sink at the bottom. Gravity will move liquid down from the source to the sink, but only across edges (pipes) that angle down. Thus much of the algorithm will be concerned with the relative heights of vertices in order to controlled the direction of flow across edges. To start, push as much flow into the network as seems possible, which is simply as much as the edges from the source can carry. This quantity may well be too much, but it is all we can tell in a first step. Putting this flow into the edges incident upon the source creates excess flow at each of the neighbors of the source. The main job of the algorithm is then to remove this excess flow, preferably by moving it down the network towards the sink, but, if necessary (in case the initial amount was too high) back towards the source; the algorithm achieves this by controlling the relative heights of the vertices. Hence the algorithm takes two distinct, complementary actions: change the height of a vertex, or push as much excess flow as possible to a connected node of lower height. All operations will take place on the residual graph, so we use arcs henceforth rather than edges.

Let us begin by formalizing the notion of height. We will simply say that each vertex carries a *label*, $l \colon V \to \{0, 1, \ldots, |V|, |V| + 1, \ldots, 2|V| - 1\}$. The label of the source will always be $|V|$ and that of the sink 0; these two labels will remain unaffected throughout the algorithm and serve as reference values. Other nodes will be assigned an initial label that can then be altered through a *Relabel* operation. The labelling of any node (other than source and sink) always obeys one condition: if (u, v) is an arc of the residual graph, then we have $l(u) \leq l(v) + 1$. If vertex v has $0 \leq l(v) < |V|$, then the algorithm will be working to push any excess flow at v towards the sink; but if $l(v)$ is at least as large as $|V|$, then the algorithm will have reached a stage at which the excess flow at v must be pushed back towards the source and removed from the network. In the first case, we can think of $l(v)$ as the "height" of v above the sink, whereas in the second case we can think of $|V| - l(v)$ as the depth of v below the source.

Remember the gravity analogy: liquid will flow through an edge only if that edge slopes downwards. Thus, if the algorithm finds a vertex u with excess flow, two cases can happen. If there is an arc from vertex u to vertex v and v has a lower label than u (so that vertex v is "below" vertex u), then the algorithm can push some excess flow from u into v. This is a `Push` operation. However, if every arc out of u leads to a vertex with a label no smaller than that of u itself, u is "too low" and needs to be "raised" in order for gravity to assist in getting rid of the excess flow; thus we increase the label of u to be one more than the value of the smallest label of its neighbors: $l(u) \leftarrow 1+\min\{l(v) \mid (u.v)\text{in residual graph}\}$. This assignment ensures that the new label still obeys the labelling condition and also "raises" u to the minimum height at which gravity can become operational again. This is a `Relabel` operation. Note that both operations are mutually exclusive, so that, at any time, a vertex with excess flow can be either pushed or relabelled, but not both—there is no ambiguity as to the choice of operations.

Let us now formalize these two operations. In the definitions below, we use $f(u, v)$ to denote the preflow from u to v and $c(u, v)$ be the residual capacity of the edge from u to v in the residual graph.

Definition 1 (The `Push` operation) *The goal of `Push`(u, v) is to push some flow from u to v when the label of v is smaller than the label of u.*

When do we do a `Push`(u, v)? *whenever (u, v) is in the residual graph and we have $e(u) > 0$, $c(u, v) > 0$, and $l(u) > l(v)$*

What does a `Push`(u) do? *push $\min\{e(u), c(u, v)\}$ flow from u to v*

Fig. 1 illustrates the `Push`(u, v) operation.

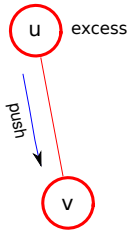

Figure 1: *`Push` from a vertex u to a vertex v.*

Definition 2 (The `Relabel` operation) *The goal of `Relabel`(u) is to change the label of vertex u so that it will be above the lowest of its neighbors.*

When do we do a `Relabel`(u)? *whenever we have $e(u) > 0$ and, for all v such that arc (u, v) is in the residual network, $l(u) \leq l(v)$*

What does a `Relabel`(u) do? *change the label of vertex u according to $l(u) = l(u) + 1$(until $l(u) = \min\{l(v) \mid (u, v) \in$ residual network$\} + 1$)*

Fig. 1 illustrates the `Relabel`(u, v) operation.

The preflow-push is an iterative algorithm that updates the preflow in the residual network. At the end of the computation, we will obtain a valid flow in the network and moreover, this flow will be the maximum flow in the network.

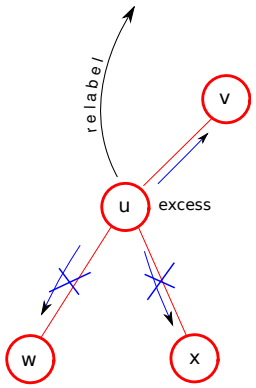

Figure 2: *`Relabel` on a vertex u.*

Correctness of the preflow-push algorithm

In order to prove that the preflow-push algorithm is correct, we must prove two results:

1. each iteration preserves a preflow, preserves a valid labelling.
2. at termination, the algorithm converges to a maximum flow.
3. the number of iterations is bounded.

Proof of 1

Lemma 1 *At every iteration, if there is an arc (u,v) with positive capacity in the residual network,* $h(u) \leq h(v) + 1$.

Proof: We show that the algorithm always keep this invariant at each iteration. At the beginning, we push enough flow to saturate all the outgoing edges from s, and label all the vertices other than s to be height 0, and the invariant holds. Now we prove that this invariant has been preserved through iterations of the algorithm. If we do a relabel on vertex v, i.e., $l'(v) = l(v) + 1$, for all arcs (u, v) with positive capacity, $l(u) \leq l(v) + 1 \leq l'(v) + 1$; for all arcs (v, w) with positive capacity, $l(v) = \min\{l(u) \mid (v, u) \in \text{residual network}\} + 1 \leq l(w) + 1$. If we do a push from vertex v to u, we have $l(v) = l(u) + 1$, and this operation may create an arc from u to v, and we still have $l(u) \leq l(v) + 1$. Q.E.D.

Proof of 2

One striking difference between this preflow-push algorithm and augmentation-based methods is that, with this method, the sink is never reachable from the source in the residual graph. This invariant may seem somewhat surprising, but it really only affirms that this algorithm always maintains as much flow out of the source as it possibly can.

Lemma 2 *If f is a preflow and d a valid labelling, then the sink is not reachable from the source in the residual graph.*

Proof: Since $l(s)$ equals $|V|$, if a path from s to t exists in the residual graph, the path has to have length not less than $|V|$. But a shortest such path can

have length at most $|V| - 1$. Q.E.D.

Now, we can prove that at termination, the preflow-push algorithm has constructed a maximum flow. The algorithm only terminates when neither operation applies at any vertex; since one of the operations is always applicable at any active vertex, it follows that, when the algorithm terminates, no vertex has excess flow. Thus flow must be conserved, so the final preflow is in fact a flow. Since the sink is not reachable from the source in the final residual graph, a residual graph constructed with respect to a true flow, there is no augmenting path in the graph with respect to this final flow. From the max-flow min-cut theorem, we conclude that this final flow is a maximum flow.

Proof of 3

This part is a bit more challenging, so we begin with two technical lemmas.

Lemma 3 *If there is a path to the source s from some vertex v in the residual graph, $2|V| - 1$ is an upper bound on $l(v)$.*

Proof: If there is a path to the source s from some vertex v, let the shortest one be $v \to x_1 \to x_2 \to \ldots \to x_k \to s$. From the way we constructed the labeling, $l(s) = |V|$, and while going along the path towards v, the label can increase by at most 1 each time. Therefore, we have $l(v) \leq |V| + k$, where k is the number of vertices along the shortest path. The number of vertices along the shortest path from v to s is bounded by $|V| - 1$, and thus $k \leq |V| - 1$ and $l(v) \leq |V| + k$ yield the desired result.

Q.E.D.

Lemma 4 *Any active vertex (that is, any vertex with nonzero excess flow) has a path to the source.*

Proof: We prove the result by contradiction. Assume that the source is not reachable from some active vertex v. Let S be the set of vertices reachable from v, and assume that s is not in S. By adding the excess flow at each of the vertices in S, we get:

$$\sum_{w \in S} e(w) = \sum_{x \in V,\, w \in S} f(x,w) - \sum_{w \in S,\, y \in V} f(w,y)$$

$$= \sum_{x \in \bar{S},\, w \in S} f(x,w) - \sum_{w \in S,\, y \in \bar{S}} f(w,y) + \sum_{x \in S,\, w \in S} f(x,w) - \sum_{w \in S,\, y \in S} f(w,y)$$

In the previous equation, $\sum_{x \in S,\, w \in S} f(x,w) - \sum_{w \in S,\, y \in S} f(w,y)$ is actually 0, because, for each $f(u,w)$ term in the sum there will also be its corresponding $f(w,u)$, and the two cancel out. Therefore we can write

$$\sum_{w \in S} e(w) = \sum_{x \in \bar{S},\, w \in S} f(x,w) - \sum_{w \in S,\, y \in \bar{S}} f(w,y)$$

In order to determine what this sum means, we inspect Figure 3. Because s and w are in S, there is a path between them, while there is no path from w to u in the residual graph. Therefore, the flow between w and u must be from u to w, so that we have $\sum_{x \in \bar{S},\, w \in S} f(x,w) = 0$ and $\sum_{w \in S,\, y \in S} f(w,y) \geq 0$ which leads us to contradiction. The sum should have been strictly positive, since the $e(w)$

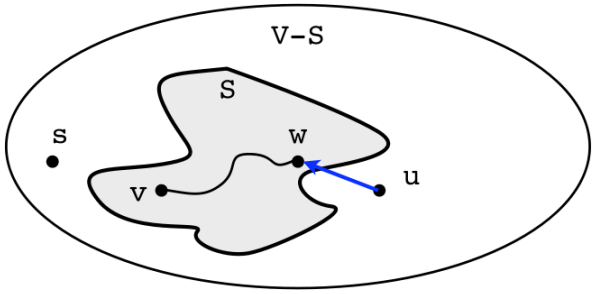

Figure 3: A graph with a vertex v and its associated set S of reachable vertices. The black path from v to w is composed of edges in the residual graph, while the blue edge is in the flow graph.

are positive and since we assumed that v was an active vertex. Therefore, our assumption that s is not in S was wrong and the proof is complete. Q.E.D.

Now we have the results we need in order to be able to bound the number of iterations of the Preflow-Push algorithm. We begin by verifying that the Push and Relabel operations keep the label values within their legal limits

$$l\colon V \to \{0, 1, \ldots, 2|V| - 1\}$$

. Since only active vertices may get new labels, and the label of any active vertex cannot grow beyond $2|V| - 1$, which implies that the total number of relabelings is $O(|V|^2)$. Relabeling takes a vertex that has excess flow but is too low to push that flow elsewhere, inspects all its neighbors, and raises it by assigning it a new label just above that of the lowest neighbor. This operation thus takes time proportional to the degree of the vertex being relabeled, which we can bound (roughly) in $O(|V|)$. Thus the total time taken by all relabelings is in $O(|V|^3)$.

For push operations, we distinguish between saturating and nonsaturating pushes. A saturating push along an arc uses up the entire residual capacity of this arc and thus removes it from the residual graph, whereas a nonsaturating push uses up the entire excess flow, but still leaves some residual capacity on the arc.

- Consider a particular arc from u to v—see Figure 4. After a saturating Push operation, no forward capacity remains, but we have an arc in the opposite direction. We must have had $l(u) = l(v)+1$ because we were able to do the Push. For the next saturating push between these two vertices, the push must go in the other direction and thus the label of v must go up by 2. Since such an increase cannot take place more often than $O(|V|)$ times, the number of saturating pushes is in $O(|V| \cdot |E|)$. Note that a saturating push can be implemented in constant time—the first arc in the adjacency list of the active vertex is used and suitable adjustments made to the residual graph and the active status of vertices.
- For nonsaturating pushes, we will use amortized analysis. Our potential function will be the sum of the labels of the active vertices:

$$\Phi = \sum_{v \text{ is active}} l(v)$$

Figure 4: The transformation of the residual capacity between two vertices, after a Push operation.

> Note that we have $\Phi_{\text{final}} = 0$, while the change in potential caused by nonsaturating pushes is non-positive (to be verified below). Therefore, the total increase in the potential (which can be much larger than the maximum value of the potential) is also an upper bound on the number of nonsaturating pushes. A non-saturating push has the same running time as a saturating push.

How big could the potential get? We have two ways to get at an answer. The simple way is to note that the maximum value of a label is $2|V| - 1$ and there are $|V|$ labels, so that the sum of all labels is $O(|V|^2)$. This is helpful, but not sufficient, as we need to know the sum of all increases in the potential. So we look at the contribution of each operation; Φ_{max} is affected by these operations as follows.

- Relabeling always increases the potential because it does not affect the active status of any vertex and only increases labels. We saw that the total number of relabelings is in $O(|V|^2)$, since that is the maximum value of all labels and since the increase could be by just one every time. This is also a bound on the sum of all increases, because the sum can only increase. (Clearly, with additional knowledge about the choices made by the algorithm, it could be bounded more tightly.)
- We argued that the number of saturating pushes is in $O(|E| \cdot |V|)$. A saturating push does not alter labels, but it can alter the active status of the endpoints of the arc used in the push: if the push was from u to v, then u was active before the push and could be active or not afterwards (it loses active status if all excess flow was pushed to v), while v may or may not have been active before the push, but is definitely active after it. Hence a saturating push can increase, decrease, or leave unchanged the potential. However, since the increase in potential caused by a saturating push is clearly in $O(|V|)$, the sum all increases contributed by all saturating pushes is in $O(|E| \cdot |V|^2)$.

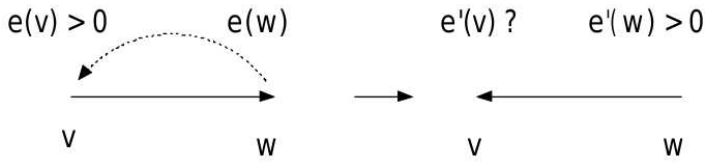

Figure 5: Saturating push: $e(v) > e'(v) > 0$, $e'(w) > 0$

- Nonsaturating pushes could presumably be more numerous than saturating ones, since they do not require intermediate relabelings. Like saturating pushes, they do not alter labelings and may alter the active status of the destination vertex (active afterwards, of either status beforehand); on

the other hand, they are sure to disactivate the originating vertex, since they use up its entire excess flow. Hence, even if the destination vertex was not active before the operation (in which case its activation adds a new term in the potential), the overall effect on the potential can only be a decrease, since the label of the originating vertex must be larger than that of the destination vertex (or the push would not have taken place).

We thus get a total increase in potential in $O(|E| \cdot |V|^2)$. Assuming that only nonsaturating pushes decrease the potential and that each nonsaturating push decreases it by just 1, we see that the total number of nonsaturating pushes is in $O(|E| \cdot |V|^2)$. Pushes take just constant time and so the nonsaturating pushes dominate the overall running time, which is in $O(|E| \cdot |V|^2)$. Whether this bound is actually attainable is unclear, but we have not said anything about initialization (it could be, for instance, settings the label of every vertex except the source to 0) nor about how to pick the next active vertex to work on.

Improved running time

We can improve the algorithm to make it faster by some simple and some not-so-simple measures.

- *easy:* $(|V|^3)$. Put the active vertices in a queue and "schedule" them in a FIFO discipline. An active vertex may have several outgoing residual arcs. If we select such a vertex for Push, we continue running Push operations on it until such is no longer possible (not switching from this vertex to another one, either for a Push or for a Relabel). We put the initial active vertices in a queue, and every time dequeue the first vertex, remove as much excess flow as possible from it by using as many residual arcs as needed and available, then, if excess flow remains, put this vertex at the end of the queue.
- *harder:* $O(|V|^2\sqrt{|E|})$. The idea here is to choose the next vertex to handle by local optimization, and to combine it with a smart initialization.
  - For the initial labeling, use breadth-first search from t to s and label each vertex by its distance to the sink. This is the largest legal label we can give to a vertex.
  - For selecting an active vertex: always pick next the active vertex with the largest label.

  The behavior remains in $O(|E| \cdot |V|)$ for dense networks, but is now in $O(|V|^{5/2})$ for sparse networks.

  The choice of a vertex of largest label is reasonably intuitive. Recall that our labels are correlated with the height of each vertex above the sink. The overall algorithm attempts to get rid of the excess flow by moving it (at least at first) downward, so starting at the bottom would make local fixes that would immediately get undone by a push at a higher level and so would waste running time. In the top-down approach described here, once a vertex is "fixed," it remains fixed, unless some of the excess flow must come back up.
- *very hard:* $O(|E| \cdot |V| \cdot \log(|V|^2/|E|))$ The same algorithm as above, but with tricked-out data structures (due, naturally, to Tarjan) to handle whole paths and move excess flow along the path rather than just to a neighbor.

In summary, iterative algorithms offer optimal solutions to two classes of problems of major importance in applications, matching and flow. They also provide a useful general strategy for heuristics and (rarely) approximation algorithms. In contrast to greedy algorithms, there is next to no result about general conditions under which iterative algorithms return optimal solutions.