COMPSCI 638: Graph Algorithms September 4, 2019

Lecture 3

Lecturer: Debmalya Panigrahi *Scribe: Kevin Sun*

1 Overview

In the last lecture, we saw that by computing a *shortest* augmenting s-t path, the Edmonds-Karp algorithm can solve the maximum flow problem in polynomial time. In this lecture, we will look at Dinitz's algorithm, which gives an even faster solution by using blocking flows. We will also look at the Push-Relabel algorithm, an even faster algorithm that follows a different strategy.

2 Dinitz's Algorithm

Before stating the algorithm, we need to establish a few concepts. The first is admissible edges: recall that these were crucial in the Edmonds-Karp algorithm from Lecture 2.

Definition 1. *Let f be a flow on a network G, and let G_f denote the corresponding residual network. An edge is* admissible *if it lies on some shortest s-t path, and an s-t path is* admissible *if all of its edges are admissible.*

Given a flow f on a network G, we can visualize the admissible network as a set of layers: layer 0 contains s, layer 1 contains the vertices whose distance (according to the number of edges in G_f) from s is 1, and so on, until the final layer, which only contains t. Every admissible edge goes from one layer to the next. Note that the admissible network is a subgraph of the residual network.

The main idea behind Dinitz's Algorithm [Din70] is the blocking flow. Intuitively, a blocking flow is a flow that cannot be "easily" augmented by sending flow along some available s-t path.

Definition 2. *A flow f is a* blocking flow *if it saturates at least one edge of every admissible s-t path.*

We are now ready to state the algorithm. Observe that it follows the same structure as the Ford-Fulkerson and Edmonds-Karp algorithms: start with an empty flow, and augment this flow until no residual s-t paths remain.

Algorithm 1 (Dinitz 1970)

```
Initialize: f(x, y) ← 0  ∀(x, y) ∈ E,   G_f ← G
while G_f has an s-t path do
    g ← a blocking flow in the admissible network
    f ← f + g
end while
```

Observe that we have not stated how a blocking flow in the admissible network is actually computed. For simplicity, we state the following result by Sleator and Tarjan [ST83] without proof.

Theorem 1. *There exists an algorithm that finds an admissible blocking flow in $O(m \log n)$ time.*

Remark: In unit-capacity graphs, a DFS-like algorithm can find a blocking flow in $O(m)$ time. In general, a slightly more complicated algorithm can find a blocking flow in $O(mn)$ time.

Now we must bound the number of iterations executed by Dinitz's algorithm. The key property of blocking flows is that augmenting along a blocking flow always increases the shortest s-t path distance $d(s,t)$ in the residual network.

Lemma 2. *In every iteration of Dinitz's algorithm, $d(s,t)$ (strictly) increases.*

Proof. Let $d(s,t)$ and $d'(s,t)$ respectively denote the shortest s-t path distance in the residual network before and after augmentation along some blocking flow g, and let Γ be a shortest admissible s-t path after some iteration of Dinitz's algorithm.

Note that Γ did not appear in the original admissible network because g saturates some edge of *every* admissible s-t path. Thus, some edge $(x,y) \in \Gamma$ is a "new" edge. In order for (x,y) to be a new edge, g must have sent flow from y to x, which implies $d(s,x) \geq d(s,y)$.

Now recall from Lecture 2 that $d(s,v)$ and $d(v,t)$ never decrease, so $d(s,x) \leq d'(s,x)$ and $d(y,t) \leq d'(y,t)$. Putting this together, we have

$$d(s,t) \leq d(s,x) + d(y,t) \leq d'(s,x) + d'(y,t),$$

where the first inequality follows from $d(s,x) \geq d(s,y)$, and

$$d'(s,y) \geq d'(s,x) + 1 + d'(y,t)$$

because when traversing Γ, we first arrive at x, then traverse (x,y), and end the path at t. □

We are now ready to bound the runtime of Dinitz's algorithm.

Theorem 3. *There exists an implementation of Dinitz's algorithm that runs in $O(mn \log n)$ time.*

Proof. From Theorem 1, we know that a blocking flow can be computed in $O(m \log n)$ time, so this bounds the runtime of each iteration. The number of iterations is at most $n-1$, because $d(s,t)$ is at most $n-1$ and Lemma 2 implies that its value increases by at least one in each iteration. Thus, the total runtime is $O(mn \log n)$, as desired. □

3 The Push-Relabel Algorithm

Until now, we have focused on maximum flow algorithms that use the following strategy:

1. Maintain a feasible flow and augment along residual s-t paths.
2. Terminate when there are no more residual s-t paths, i.e., the flow value is maximum.

We now turn our attention to an opposite strategy:

1. Maintain the non-existence of a residual s-t path (so the flow value is at least the maximum).
2. Terminate when the flow becomes feasible.

The idea is the following: if the value of our flow is always at least the maximum flow value, and at some point the flow becomes feasible, then the value of this feasible flow must be the maximum flow value. We now define the kind of infeasible flow that this strategy maintains.

Definition 3. *A raw flow $r : E \to \mathbb{R}$ on a network G with edge capacities $u : E \to \mathbb{R}_{\geq 0}$ is a* preflow *if it satisfies the following:*

1. *Nonnegativity:* $r(e) \geq 0 \quad \forall e \in E$.
2. *Capacity constraints:* $r(e) \leq u(e) \quad \forall e \in E$.
3. *Nonnegative excess:* $r_{in}(v) - r_{out}(v) \geq 0 \quad \forall v \in V \setminus \{s, t\}$.

The value *of a preflow r is the amount of flow leaving s, that is, $|r| = \sum_v r(s, v)$.*

For each vertex v, we let $e(v) = r_{in}(v) - r_{out}(v)$ denote the *excess* flow at vertex v. A preflow becomes a feasible flow once we have $e(v) = 0$ for every $v \in V \setminus \{s, t\}$.

We also need the concept of a level function ℓ, which changes over time. Initially, we have $\ell(s) = n$ and $\ell(v) = 0$ for all other vertices $v \in V$. An edge (x, y) is *downhill* if $\ell(x) > \ell(y)$. As we will see, the values of $\ell(s)$ and $\ell(t)$ will never change, but the other level values may increase. Now we are ready to describe the two operations behind the Push-Relabel algorithm.

PUSH(x, y): This operation sends some excess flow from x to y along a downhill edge (x, y). More specifically, it increases the flow value in f along (x, y) by the largest amount possible (i.e., $\max\{u_f(x, y), e(x)\}$). If the residual capacity of (x, y) reduces to zero then the push is *saturating*, otherwise it is *non-saturating*.

RELABEL(x): Since PUSH is only applicable to downhill edges, we need a way to adjust the level function ℓ. The operation RELABEL(x) increases $\ell(x)$ until x has at least one admissible outgoing edge, that is, it sets $\ell(x)$ as $\min_{y:(x,y)\in G_f} \ell(y) + 1$.

We are now ready to state the Push-Relabel algorithm. The initial preflow is constructed by saturating every edge leaving s. We then push excess flow, relabeling levels as necessary, until no vertices in $V \setminus \{s, t\}$ have positive excess.

Algorithm 2 (Push-Relabel (Goldberg and Tarjan 1972))

```
f(x,y) ← 0  ∀(x,y) ∈ E,   G_f ← G
ℓ(s) ← n,   ℓ(x) ← 0  ∀x ∈ V \ {s}
for all (s,v) ∈ E do
    f(s,v) ← u(s,v)
end for
while ∃x ∈ V \ {s,t} s.t. e(x) > 0 do
    if ∃(x,y) ∈ E s.t. ℓ(x) > ℓ(y) then
        PUSH(x,y)
    else
        RELABEL(x)
    end if
end while
```

To prove the correctness of the Push-Relabel algorithm, we will first prove that the level function ℓ satisfies two key properties.

Lemma 4. *Throughout the Push-Relabel algorithm, the level function ℓ satisfies the following:*

1. *The values of $\ell(s)$ and $\ell(t)$ do not change, i.e., $\ell(s) = n, \ell(t) = 0$.*
2. *There are no "steep" downhill residual edges: $\ell(x) \leq \ell(y) + 1$ for every residual edge (x, y).*

Proof. Our proof proceeds inductively. Initially, we have $\ell(s) = n$ and $\ell(t) = 0$. Also, the only vertex with positive height is s, but every edge leaving s is saturated. Thus, no residual edges leave s, so there are no downhill edges at all.

Suppose we apply PUSH(x, y). This doesn't affect ℓ, but it might create a new residual edge (y, x). However, since (x, y) is downhill, the new edge (y, x) cannot be downhill as well.

Now suppose we apply RELABEL(x), which means x has no downhill edges before the relabel. Since the RELABEL operation increases $\ell(x)$ by the smallest amount to create a downhill edge leaving x, it does not create any "steep" downhill edges. □

We now prove a lemma that will allow us to bound the runtime of the Push-Relabel algorithm.

Lemma 5. *Any vertex x with excess $e(x) > 0$ has a residual path to s.*

Proof. We also prove this inductively. Initially, if $e(x) > 0$ then (x, s) is a residual edge, i.e., a residual path to s. When we apply PUSH(x, y), vertex y may have positive excess, but (y, x) is now a residual edge so one y-s path follows this edge and then the x-s path which exists by induction. Relabeling doesn't change any excess values, so we are done. □

Lemma 6. *Each vertex is relabeled at most $2n - 1$ times, which implies the total number of* RELABEL *operations is at most $2n^2$.*

Proof. Let x be a vertex and consider the maximum possible value of $\ell(x)$. We can only run RELABEL(x) if $e(x) > 0$, which by Lemma 5, implies there exists an x-s residual path. Such a path contains at most $n - 1$ edges, so by Lemma 4, we have $\ell(x) \leq \ell(s) + n - 1 = 2n - 1$. Thus, the total number of times we run RELABEL(x) is at most $2n - 1$. □

Lemma 7. *The total number of* PUSH *operations is $O(mn^2)$.*

Proof. For brevity, we only give a sketch of this proof. We first bound the number of saturating pushes: suppose an edge (x, y) is saturated twice. After the first saturation, (x, y) is removed from the residual network. For (x, y) to be saturated again, we must apply RELABEL(x) and RELABEL(y). So by Lemma 6, the total number of times saturating pushes on (x, y) is $O(n)$, for a total of $O(mn)$ saturating pushes. Bounding the number of non-saturating pushes by $O(mn^2)$ is more involved, so we omit the proof. □

We conclude this section by combining the above lemmas to prove the correctness and bound the overall runtime of the Push-Relabel algorithm.

Theorem 8. *The Push-Relabel algorithm terminates with a maximum s-t flow in $O(mn^2)$ time.*

Proof. When the algorithm terminates, the flow is feasible because no vertex in $V \setminus \{s, t\}$ has excess flow. Furthermore, we claim that no residual s-t path is ever created by the Push-Relabel algorithm. This follows from the two properties in Lemma 4 and the fact that any shortest path in a graph with n vertices has at most $n - 1$ edges. Since no residual s-t path is ever created, the terminating flow value is maximum (by the max-flow min-cut theorem).

From Lemmas 6 and 7, we see that the total number of push/relabel operations is $O(mn^2)$, and each operation can be implemented to run in constant time; this proves the overall runtime. □

Remark: From the above discussion, we see that the bottleneck operation of the Push-Relabel algorithm is the total number of non-saturating pushes. Indeed, Ahuja and Orlin [AO89] use the idea of *scaling*, introduced by Edmonds and Karp [EK72] to reduce the number of non-saturating pushes to $O(n^2 \log U)$ where U is the largest edge capacity. This yields an algorithm for maximum flow that runs in $O(mn + n^2 \log U)$ time.

4 Summary

In this section, we saw the use of blocking flows in Dinitz's algorithm, which allowed us to further improve the runtime on the Ford-Fulkerson and Edmonds-Karp algorithms. We also saw a maximum flow algorithm known as Push-Relabel, which modifies a preflow until it is feasible.

References

[AO89] Ravindra K Ahuja and James B Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Doklady Akademii Nauk SSSR*, 194(4):1277–1280, 1970.

[EK72] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[ST83] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.