Paper Report

Siddhant Chaudhary

Abstract

This is a report on the paper "*A fast and simple algorithm for the maximum flow problem*" by James B. Orlin and R. K. Ahuja (1989). This paper was presented as a part of a course on *Matching and Flow Algorithms* at the Chennai Mathematical Institute.

Contents

0.1 Overview and Notation

The paper's main result is a new algorithm for computing the maximum flow in a flow network, motivated by the celebrated *push-relabel* algorithm for the same problem. The authors first review the push-relabel algorithm for the maximum flow problem; then they identify the bottleneck in the algorithm and try to find a way around it. In this section, we will introduce the relevant definitions and notation.

0.1.1 A review of flows and preflows. Throughout this report, we will be working with a directed network $G = (N, A)$, where N is the set of nodes, and A the set of arcs in the network. For every arc $(i, j) \in A$, we will use the symbol u_{ij} to denote it's (integral) capacity. Also, we define $n := |N|$ and $m := |A|$. The *source* s and *sink* t will be two distinguished nodes in the network. We will assume that if $(i, j) \in A$, then $(j, i) \in A$ (with (j, i) possibly having zero capacity).

We will use the symbol U to denote the *maximum* capacity of any arc emanating from the source vertex s, i.e

$$U := \max_{(s,j)\in A} \{u_{sj}\}$$

A *flow* is a function $f : A \to \mathbb{R}$ that satisfies the following.

- For all $i \in N - \{s, t\}$, we have

$$\sum_{\{j:(j,i)\in A\}} f_{ji} - \sum_{\{j:(i,j)\in A\}} f_{ij} = 0$$

This is also called the *flow conservation law.*

- For each arc $(i, j) \in A$, we must have

$$0 \leq f_{ij} \leq u_{ij}$$

This is called the *capacity constraint.*

The *maximum flow problem* tries to compute a flow f such that the following quantity, called the *size* of the flow f, is maximized.

$$\sum_{\{j:(j,t)\in A\}} f_{jt}$$

Next, we quickly discuss *preflows.* A *preflow* f is a function $f : A \to \mathbb{R}$ that satisfies all capacity constraints, and satisfies the following relaxation of the flow conservation law for all $i \in N - \{s, t\}$.

$$\sum_{\{j:(j,i)\in A\}} f_{ji} - \sum_{\{j:(i,j)\in A\}} f_{ij} \geq 0$$

For a node i, the above quantity is called the *excess* at node i, and is denoted by the symbol e_i. So, for a preflow, all the excesses must be non-negative.

A node $i \in N - \{s, t\}$ with positive excess is said to be *active.* The *residual capacity* of an arc $(i, j) \in A$, with respect to a preflow f, is defined as

$$r_{ij} = u_{ij} - f_{ij} + f_{ji}$$

The *residual network* for a preflow f is defined to be the network containing only those edges which have a positive residual capacity.

Finally, for each vertex i, we define the *arc adjacency list* $A(i)$ as the set

$$A(i) := \{(i, k) \in A : k \in N\}$$

0.1.2 Valid distance functions. A *valid distance function* is a function $d : N \to \mathbb{Z}^+$ for a preflow f that satisfies the following two conditions.

- $d(t) = 0$.
- $d(i) \leq d(j) + 1$ for every arc $(i, j) \in A$ with $r_{ij} > 0$.

An arc $(i, j) \in A$ in the residual network is called *admissible* if $d(i) = d(j) + 1$; otherwise it's called an *inadmissible arc.*

0.2 The PUSH-RELABEL Algorithm

In this section, we will review the PUSH-RELABEL algorithm for the max flow problem. The algorithm consists of the following three subroutines.

(1) PREPROCESS. In this subroutine, we initialize a preflow and a valid distance function. For each arc $(s, j) \in A(s)$, send u_{sj} units of flow. Also, initialize $d(s) = n$, $d(t) = 0$ and $d(i) = 1$ for all $i \neq s, t$. It can be easily verified that this is a valid distance function.

(2) PUSH(i). Here, we select an admissible arc (i, j) in $A(i)$, and send $\delta = \min\{e_i, r_{ij}\}$ units of flow from node i to j.

A push of flow on arc (i, j) is said to be *saturating* if $\delta = r_{ij}$, and non-saturating otherwise.

(3) RELABEL(i). Here, we just replace $d(i)$ by $\min\{d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0\}$. This is called a *relabel step.*

The PUSH-RELABEl algorithm can be described by the following pseudocode.

Algorithm 1 PUSH-RELABEL Algorithm

```
PREPROCESS.
while there is an active node do
    Select an active node i.
    if there is an admissible arc in A(i) then
        PUSH(i).
    else
        RELABEL(i).
    end if
end while
```

Proposition 0.1. *The* PUSH-RELABEL *algorithm satisfies the following properties.*

(1) It maintains valid distance labels at each step. The distance labels only increase over the course of the algorithm, and at each step, the distance label of some node strictly increases.
(2) For each node $i \in N$, $d(i) < 2n$.
(3) The number of relabel steps is less than $2n^2$.
(4) The number of saturating pushes is atmost nm.
(5) The number of non-saturating pushes is atmost $2n^2m$.

From this proposition, one can easily conclude that the time complexity of the PUSH-RELABEL algorithm is $O(nm + n^2m)$.

0.3 The bottleneck, and the Excess Scaling Algorithm

The authors remark that the bottleneck operation in many preflow-based algorithms is the number of non-saturating pushes. Intuitively, each saturating push changes the structure of the residual network (by removing an edge from the network). However, a non-saturating push doesn't change this structure, and hence it becomes difficult to bound the total number of non-saturating pushes. The main result of the paper, namely the *Excess Scaling Algorithm*, relies on a good upper bound on the number of non-saturating pushes.

The authors have shown that their algorithm reduces the number of non-saturating pushes from $O(n^2m)$ to $O(n^2 \log U)$. In the upcoming sections, we will describe the algorithm.

0.3.1 Description of the Excess Scaling Algorithm. The algorithm consists of a number of iterations, called *scaling iterations*. Each scaling iteration roughly does the following.

(1) For each scaling iteration, we define the *excess dominator* to be the least integer Δ that is a power of 2 and which satisfies $e_i \leq \Delta$ for all $i \in V$.
(2) After each scaling iteration, we ensure that Δ decreases by a factor of 2.
(3) Note that when we initialize our preflow in the PREPROCESS subroutine, the maximum possible excess on any vertex is precisely $\max_{(s,j)\in A}\{u_{sj}\}$, which we denoted by U. So naturally, we will start off with $\Delta = 2^{\lceil \log U \rceil}$.
(4) Since each scaling iteration decreases Δ by a factor of 2, we see that there will be $1 + \lceil \log U \rceil$ scaling iterations.

Suppose Δ is the excess dominator at start of a scaling iteration. We will somehow guarantee that each *non-saturating* push during this pushes atleast $\Delta/2$ units of flow; this will help us in bounding the number of non-saturating pushes better. Moreover, we also have to ensure that during a scaling iteration, the excess dominator *never increases*.

To ensure that each non-saturating push has a value of atleast $\Delta/2$, we do the following.

(1) In a scaling iteration, we only push flow from vertices with an excess more than $\Delta/2$.
(2) Moreover, among all vertices with excess more than $\Delta/2$, we push flow from the vertex with the minimum distance label. Since we push flow on only admissible edges, this choice will ensure that flow is being pushed to a vertex with excess atmost $\Delta/2$.

Next, we describe the data structures that are maintained throughout the algorithm.

(1) For each $r \in \{1, 2, ..., 2n-1\}$, we maintain a list denoted by LIST(r); this list will just be the set

$$\left\{ i \in V : e_i > \frac{\delta}{2}, d(i) = r \right\}$$

Note that since d is a valid distance function, $d(i) < 2n$ for all i at all times in the algorithm. Hence, we only need to maintain these lists for values of r atmost $2n-1$. In practice, LIST(r) for each r will be implemented as a linked list, in which we can add and remove elements in $O(1)$ time.
(2) We will maintain a variable called *level*, which will represent the smallest index r for which LIST(r) is non-empty.
(3) For each vertex i, an edge adjacency list $A(i)$ will be maintained. Moreover, for each i, we will maintain a *current edge*, which will be an edge in $A(i)$ which is a potential candidate for pushing flow out of i.

0.3.2 Pseudocode. The algorithm can be described precisely by the given pseudocode. It consists of the subroutine PUSH/RELABEL(i), for which we've also provided the pseudocode.

Algorithm 2 Excess Scaling Algorithm

```
PREPROCESS.
K := 1 + ⌈log U⌉                          ▷ The number of scaling iterations
for k = 1 to K do
    Δ = 2^{K−k}                           ▷ The excess dominator
    for each vertex i do
        if e_i > Δ/2 then
            Add i to LIST(d(i))
        end if
    end for
    level := 1.
    while level < 2n do
        If LIST(level) = φ, then level := level + 1.
        Otherwise, select a vertex i from LIST(level), and do PUSH/RELABEL(i).
    end while
end for
```

Algorithm 3 PUSH/RELABEL(i)

1: Starting from the current edge of i, find an admissible edge $(i,j) \in A(i)$ with $r_{ij} > 0$, incrementing the current edge pointer if necessary.
2: **if** an admissible edge (i,j) has been found **then**
3: Push $\min\{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i,j).
4: Update residual capacity r_{ij} and excesses e_i and e_j.
5: If (updated) $e_i \leq \Delta/2$, delete i from LIST($d(i)$).
6: If $j \neq s,t$ and if the (updated) $e_j > \Delta/2$, then add node j to LIST($d(j)$); and set $level := level - 1$.
7: **else** ▷ No admissible edge found in the previous step
8: Delete i from LIST($d(i)$).
9: Update label $d(i)$ as usual.
10: Add i to LIST($d(i)$), and set the *current edge* of i to the first edge of $A(i)$.
11: **end if**

Lines 11-13 of the main algorithm do the following: among all vertices with excess more than $\Delta/2$, select the vertex with the minimum distance label, and try to push flow out of it; if not, relabel it. Line 3 of the PUSH/RELABEL(i) subroutine makes sure that, if the push is non-saturating, atleast $\Delta/2$ units of flow is pushed. This is quite easy to see: if the push is non-saturating, then the minimum of the three quantities is actually the minimum of $\{e_i, \Delta - e_j\}$; we know that $e_i > \Delta/2$ and $e_j \leq \Delta/2$ (since (i,j) is an admissible edge, and i is the vertex with minimum distance label among all vertices with excess more than $\Delta/2$); hence, the minimum of these two quantities is atleast $\Delta/2$.

Proposition 0.2. *The Excess scaling algorithm satisfies the following.*
(1) Each non-saturating push from vertex i to vertex j sends atleast $\Delta/2$ units of flow.
(2) No excess ever increases above Δ withing a scaling iteration.
(3) The number of non-saturating pushes per scaling iteration is at most $8n^2$.

The above three claims are not hard to prove. The third claim is proved by defining a potential function F for a scaling iteration by

$$F = \sum_{i \in V} \frac{e_i d(i)}{\Delta}$$

The idea is to track how much F increases/decreases over a scaling iteration. It can be shown that whenever a push is done, F decreases; using this along with the fact that each non-saturating push has a value of atleast $\Delta/2$, it can be shown that the number of non-saturating pushes is atmost $8n^2$.

0.3.3 Complexity of the algorithm. Finally, with a little bit of additional work and using the facts mentioned in the previous section, one can show that the complexity of the Excess Scaling algorithm is $O(nm + n^2 \log U)$. Here, the term nm comes from an upper bound on the number of saturating pushes, and the term $n^2 \log U$ comes from the upper bound on the number of non-saturating pushes. As before, the number of relabel operations is again $O(n^2)$.