Algorithm Design

JON KLEINBERG • ÉVA TARDOS
Cornell University

PEARSON
Addison Wesley

Boston  San Francisco  New York
London  Toronto  Sydney  Tokyo  Singapore  Madrid
Mexico City  Munich  Paris  Cape Town  Hong Kong  Montreal
number of iterations that is polynomial in the numbers 4 and 5, completely independently of the values of the capacities. Such an algorithm, which is polynomial in \(|V|\) and \(|E|\) only, and works with numbers having a polynomial number of bits, is called a *strongly polynomial algorithm*. In fact, there is a simple and natural implementation of the Ford-Fulkerson Algorithm that leads to such a strongly polynomial bound: each iteration chooses the augmenting path with the fewest number of edges. Dinitz, and independently Edmonds and Karp, proved that with this choice the algorithm terminates in at most \(O(mn)\) iterations. In fact, these were the first polynomial algorithms for the Maximum-Flow Problem. There has since been a huge amount of work devoted to improving the running times of maximum-flow algorithms. There are currently algorithms that achieve running times of \(O(mn \log n)\), \(O(n^3)\), and \(O(\min(n^{2/3}, m^{1/2})m \log n \log U)\), where the last bound assumes that all capacities are integral and at most \(U\). In the next section, we'll discuss a strongly polynomial maximum-flow algorithm based on a different principle.

## 7.4 The Preflow-Push Maximum-Flow Algorithm

From the very beginning, our discussion of the Maximum-Flow Problem has been centered around the idea of an augmenting path in the residual graph. However, there are some very powerful techniques for maximum flow that are not explicitly based on augmenting paths. In this section we study one such technique, the Preflow-Push Algorithm.

### Designing the Algorithm

Algorithms based on augmenting paths maintain a flow \(f\), and use the augment procedure to increase the value of the flow. By way of contrast, the Preflow-Push Algorithm will, in essence, increase the flow on an edge-by-edge basis. Changing the flow on a single edge will typically violate the conservation condition, and so the algorithm will have to maintain something less well behaved than a flow—something that does not obey conservation—as it operates.

**Preflows** We say that an \(s\)-\(t\) preflow (preflow, for short) is a function \(f\) that maps each edge \(e\) to a nonnegative real number, \(f : E \rightarrow \mathbb{R}^+\). A preflow \(f\) must satisfy the capacity conditions:

1. For each \(e \in E\), we have \(0 \leq f(e) \leq c_e\).

In place of the conservation conditions, we require only inequalities: Each node other than \(s\) must have at least as much flow entering as leaving.

2. For each node \(v\) other than the source \(s\), we have

\[
\sum_{e \text{ into } v} f(e) \geq \sum_{e \text{ out of } v} f(e).
\]
We will call the difference
\[ e_f(v) = \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) \]
the excess of the preflow at node \( v \). Notice that a preflow where all nodes other than \( s \) and \( t \) have zero excess is a flow, and the value of the flow is exactly \( e_f(t) = -e_f(s) \). We can still define the concept of a residual graph \( G_f \) for a preflow \( f \), just as we did for a flow. The algorithm will “push” flow along edges of the residual graph (using both forward and backward edges).

**Preflows and Labelings** The Preflow-Push Algorithm will maintain a preflow and work on converting the preflow into a flow. The algorithm is based on the physical intuition that flow naturally finds its way “downhill.” The “heights” for this intuition will be labels \( h(v) \) for each node \( v \) that the algorithm will define and maintain, as shown in Figure 7.7. We will push flow from nodes with higher labels to those with lower labels, following the intuition that fluid flows downhill. To make this precise, a labeling is a function \( h : V \to \mathbb{Z}_{\geq 0} \) from the nodes to the nonnegative integers. We will also refer to the labels as heights of the nodes. We will say that a labeling \( h \) and an \( s \)-\( t \) preflow \( f \) are compatible if

(i) (Source and sink conditions) \( h(t) = 0 \) and \( h(s) = n \),

(ii) (Steepness conditions) For all edges \((v, w) \in E_f\) in the residual graph, we have \( h(v) \leq h(w) + 1 \).

![Figure 7.7 A residual graph and a compatible labeling. No edge in the residual graph can be too “steep”—its tail can be at most one unit above its head in height. The source node \( s \) must have \( h(s) = n \) and is not drawn in the figure.](image)
Intuitively, the height difference \( n \) between the source and the sink is meant to ensure that the flow starts high enough to flow from \( s \) toward the sink \( t \), while the steepness condition will help by making the descent of the flow gradual enough to make it to the sink.

The key property of a compatible preflow and labeling is that there can be no \( s-t \) path in the residual graph.

\[ (7.21) \] If \( s-t \) preflow \( f \) is compatible with a labeling \( h \), then there is no \( s-t \) path in the residual graph \( G_f \).

**Proof.** We prove the statement by contradiction. Let \( P \) be a simple \( s-t \) path in the residual graph \( G \). Assume that the nodes along \( P \) are \( s, v_1, \ldots, v_k = t \). By definition of a labeling compatible with preflow \( f \), we have that \( h(s) = n \). The edge \((s, v_i)\) is in the residual graph, and hence \( h(v_i) \geq h(s) - 1 = n - 1 \). Using induction on \( i \) and the steepness condition for the edge \((v_{i-1}, v_i)\), we get that for all nodes \( v_i \) in path \( P \) the height is at least \( h(v_i) \geq n - i \). Notice that the last node of the path is \( v_k = t \); hence we get that \( h(t) \geq n - k \). However, \( h(t) = 0 \) by definition; and \( k < n \) as the path \( P \) is simple. This contradiction proves the claim.

Recall from (7.9) that if there is no \( s-t \) path in the residual graph \( G_f \) of a flow \( f \), then the flow has maximum value. This implies the following corollary.

\[ (7.22) \] If \( s-t \) flow \( f \) is compatible with a labeling \( h \), then \( f \) is a flow of maximum value.

Note that (7.21) applies to preflows, while (7.22) is more restrictive in that it applies only to flows. Thus the Preflow-Push Algorithm will maintain a preflow \( f \) and a labeling \( h \) compatible with \( f \), and it will work on modifying \( f \) and \( h \) so as to move \( f \) toward being a flow. Once \( f \) actually becomes a flow, we can invoke (7.22) to conclude that it is a maximum flow. In light of this, we can view the Preflow-Push Algorithm as being in a way orthogonal to the Ford-Fulkerson Algorithm. The Ford-Fulkerson Algorithm maintains a feasible flow while changing it gradually toward optimality. The Preflow-Push Algorithm, on the other hand, maintains a condition that would imply the optimality of a preflow \( f \), if it were to be a feasible flow, and the algorithm gradually transforms the preflow \( f \) into a flow.

To start the algorithm, we will need to define an initial preflow \( f \) and labeling \( h \) that are compatible. We will use \( h(v) = 0 \) for all \( v \neq s \), and \( h(s) = n \), as our initial labeling. To make a preflow \( f \) compatible with this labeling, we need to make sure that no edges leaving \( s \) are in the residual graph (as these edges do not satisfy the steepness condition). To this end, we define the initial
preflow as $f(e) = c_e$ for all edges $e = (s, v)$ leaving the source, and $f(e) = 0$ for all other edges.

**7.23** The initial preflow $f$ and labeling $h$ are compatible.

**Pushing and Relabeling** Next we will discuss the steps the algorithm makes toward turning the preflow $f$ into a feasible flow, while keeping it compatible with some labeling $h$. Consider any node $v$ that has excess—that is, $e_f(v) > 0$. If there is any edge $e$ in the residual graph $G_f$ that leaves $v$ and goes to a node $w$ at a lower height (note that $h(w)$ is at most 1 less than $h(v)$ due to the steepness condition), then we can modify $f$ by pushing some of the excess flow from $v$ to $w$. We will call this a push operation.

```plaintext
push(f, h, v, w)
Applicable if $e_f(v) > 0$, $h(w) < h(v)$ and $(v, w) \in E_f$
If $e = (v, w)$ is a forward edge then
   let $\delta = \min(e_f(v), c_e - f(e))$ and
   increase $f(e)$ by $\delta$
If $(v, w)$ is a backward edge then
   let $e = (w, v)$, $\delta = \min(e_f(v), f(e))$ and
   decrease $f(e)$ by $\delta$
Return($f, h$)
```

If we cannot push the excess of $v$ along any edge leaving $v$, then we will need to raise $v$'s height. We will call this a relabel operation.

```plaintext
relabel(f, h, v)
Applicable if $e_f(v) > 0$, and
   for all edges $(v, w) \in E_f$ we have $h(w) \geq h(v)$
Increase $h(v)$ by 1
Return($f, h$)
```

**The Full Preflow-Push Algorithm** So, in summary, the Preflow-Push Algorithm is as follows.

**Preflow-Push**
Initially $h(v) = 0$ for all $v \neq s$ and $h(s) = n$ and
$f(e) = c_e$ for all $e = (s, v)$ and $f(e) = 0$ for all other edges
While there is a node $v \neq t$ with excess $e_f(v) > 0$
   Let $v$ be a node with excess
   If there is $w$ such that push($f, h, v, w$) can be applied then
      push($f, h, v, w$)
Else
    relabel\(f, h, v\)
Endwhile
Return\(f\)

---

Analyzing the Algorithm

As usual, this algorithm is somewhat underspecified. For an implementation of the algorithm, we will have to specify which node with excess to choose, and how to efficiently select an edge on which to push. However, it is clear that each iteration of this algorithm can be implemented in polynomial time. (We’ll discuss later how to implement it reasonably efficiently.) Further, it is not hard to see that the preflow \(f\) and the labeling \(h\) are compatible throughout the algorithm. If the algorithm terminates—something that is far from obvious based on its description—then there are no nodes other than \(s\) with positive excess, and hence the preflow \(f\) is in fact a flow. It then follows from (7.22) that \(f\) would be a maximum flow at termination.

We summarize a few simple observations about the algorithm.

(7.24) Throughout the Preflow-Push Algorithm:

(i) the labels are nonnegative integers;

(ii) \(f\) is a preflow, and if the capacities are integral, then the preflow \(f\) is integral; and

(iii) the preflow \(f\) and labeling \(h\) are compatible.

If the algorithm returns a preflow \(f\), then \(f\) is a flow of maximum value.

Proof. By (7.23) the initial preflow \(f\) and labeling \(h\) are compatible. We will show using induction on the number of push and relabel operations that \(f\) and \(h\) satisfy the properties of the statement. The push operation modifies the preflow \(f\), but the bounds on \(\delta\) guarantee that the \(f\) returned satisfies the capacity constraints, and that excesses all remain nonnegative, so \(f\) is a preflow. To see that the preflow \(f\) and the labeling \(h\) are compatible, note that push\((f, h, v, w)\) can add one edge to the residual graph, the reverse edge \((v, w)\), and this edge does satisfy the steepness condition. The relabel operation increases the label of \(v\), and hence increases the steepness of all edges leaving \(v\). However, it only applies when no edge leaving \(v\) in the residual graph is going downward, and hence the preflow \(f\) and the labeling \(h\) are compatible after relabeling.

The algorithm terminates if no node other than \(s\) or \(t\) has excess. In this case, \(f\) is a flow by definition; and since the preflow \(f\) and the labeling \(h\)
remain compatible throughout the algorithm, (7.22) implies that $f$ is a flow of maximum value. ■

Next we will consider the number of push and relabel operations. First we will prove a limit on the relabel operations, and this will help prove a limit on the maximum number of push operations possible. The algorithm never changes the label of $s$ (as the source never has positive excess). Each other node $v$ starts with $h(v) = 0$, and its label increases by 1 every time it changes. So we simply need to give a limit on how high a label can get. We only consider a node $v$ for relabel when $v$ has excess. The only source of flow in the network is the source $s$; hence, intuitively, the excess at $v$ must have originated at $s$. The following consequence of this fact will be key to bounding the labels.

(7.25) Let $f$ be a preflow. If the node $v$ has excess, then there is a path in $G_f$ from $v$ to the source $s$.

Proof. Let $A$ denote all the nodes $w$ such that there is a path from $w$ to $s$ in the residual graph $G_f$, and let $B = V - A$. We need to show that all nodes with excess are in $A$.

Notice that $s \in A$. Further, no edges $e = (x, y)$ leaving $A$ can have positive flow, as an edge with $f(e) > 0$ would give rise to a reverse edge $(y, x)$ in the residual graph, and then $y$ would have been in $A$. Now consider the sum of excesses in the set $B$, and recall that each node in $B$ has nonnegative excess, as $s \not\in B$.

$$0 \leq \sum_{v \in B} e_f(v) = \sum_{v \in B} (f^\text{in}(v) - f^\text{out}(v))$$

Let's rewrite the sum on the right as follows. If an edge $e$ has both ends in $B$, then $f(e)$ appears once in the sum with a “+” and once with a “-”, and hence these two terms cancel out. If $e$ has only its head in $B$, then $e$ leaves $A$, and we saw above that all edges leaving $A$ have $f(e) = 0$. If $e$ has only its tail in $B$, then $f(e)$ appears just once in the sum, with a “-”. So we get

$$0 \leq \sum_{v \in B} e_f(v) = -f^\text{out}(B).$$

Since flows are nonnegative, we see that the sum of the excesses in $B$ is zero; since each individual excess in $B$ is nonnegative, they must therefore all be 0. ■

Now we are ready to prove that the labels do not change too much. Recall that $n$ denotes the number of nodes in $V$. 

(7.26) Throughout the algorithm, all nodes have $h(v) \leq 2n - 1$.

**Proof.** The initial labels $h(t) = 0$ and $h(s) = n$ do not change during the algorithm. Consider some other node $v \neq s, t$. The algorithm changes $v$'s label only when applying the `relabel` operation, so let $f$ and $h$ be the preflow and labeling returned by a `relabel(f, h, v)` operation. By (7.25) there is a path $P$ in the residual graph $G_f$ from $v$ to $s$. Let $|P|$ denote the number of edges in $P$, and note that $|P| \leq n - 1$. The steepness condition implies that heights of the nodes can decrease by at most 1 along each edge in $P$, and hence $h(v) - h(s) \leq |P|$, which proves the statement. ■

Labels are monotone increasing throughout the algorithm, so this statement immediately implies a limit on the number of relabeling operations.

(7.27) Throughout the algorithm, each node is relabeled at most $2n - 1$ times, and the total number of relabeling operations is less than $2n^2$.

Next we will bound the number of push operations. We will distinguish two kinds of push operations. A `push(f, h, v, w)` operation is saturating if either $e = (v, w)$ is a forward edge in $E_f$ and $\delta = c_e - f(e)$, or $(v, w)$ is a backward edge with $e = (w, v)$ and $\delta = f(e)$. In other words, the push is saturating if, after the push, the edge $(v, w)$ is no longer in the residual graph. All other push operations will be referred to as nonsaturating.

(7.28) Throughout the algorithm, the number of saturating push operations is at most $2n$.  

**Proof.** Consider an edge $(v, w)$ in the residual graph. After a saturating `push(f, h, v, w)` operation, we have $h(v) = h(w) + 1$, and the edge $(v, w)$ is no longer in the residual graph $G_f$, as shown in Figure 7.8. Before we can push again along this edge, first we have to push from $w$ to $v$ to make the edge $(v, w)$ appear in the residual graph. However, in order to push from $w$ to $v$, we first need for $w$'s label to increase by at least 2 (so that $w$ is above $v$). The label of $w$ can increase by 2 at most $n - 1$ times, so a saturating push from $v$ to $w$ can occur at most $n$ times. Each edge $e \in E$ can give rise to two edges in the residual graph, so overall we can have at most $2nm$ saturating pushes. ■

The hardest part of the analysis is proving a bound on the number of nonsaturating pushes, and this also will be the bottleneck for the theoretical bound on the running time.

(7.29) Throughout the algorithm, the number of nonsaturating push operations is at most $2n^2m$. 

The height of node $w$ has to increase by 2 before it can push flow back to node $v$.

![Diagram showing heights and nodes](image)

Figure 7.8 After a saturating push($f,h,v,w$), the height of $v$ exceeds the height of $w$ by 1.

**Proof.** For this proof, we will use a so-called potential function method. For a preflow $f$ and a compatible labeling $h$, we define

$$\Phi(f,h) = \sum_{v:r_f(v) > 0} h(v)$$

to be the sum of the heights of all nodes with positive excess. ($\Phi$ is often called a potential since it resembles the “potential energy” of all nodes with positive excess.)

In the initial preflow and labeling, all nodes with positive excess are at height 0, so $\Phi(f,h) = 0$. $\Phi(f,h)$ remains nonnegative throughout the algorithm. A nonsaturating push($f,h,v,w$) operation decreases $\Phi(f,h)$ by at least 1, since after the push the node $v$ will have no excess, and $w$, the only node that gets new excess from the operation, is at a height 1 less than $v$. However, each saturating push and each relabel operation can increase $\Phi(f,h)$. A relabel operation increases $\Phi(f,h)$ by exactly 1. There are at most $2n^2$ relabel operations, so the total increase in $\Phi(f,h)$ due to relabel operations is $2n^2$. A saturating push($f,h,v,w$) operation does not change labels, but it can increase $\Phi(f,h)$, since the node $w$ may suddenly acquire positive excess after the push. This would increase $\Phi(f,h)$ by the height of $w$, which is at most $2n - 1$. There are at most $2nm$ saturating push operations, so the total increase in $\Phi(f,h)$ due to push operations is at most $2mn(2n - 1)$. So, between the two causes, $\Phi(f,h)$ can increase by at most $4mn^2$ during the algorithm.
But since $\Phi$ remains nonnegative throughout, and it decreases by at least 1 on each nonsaturating push operation, it follows that there can be at most $4mn^2$ nonsaturating push operations.

**Extensions: An Improved Version of the Algorithm**

There has been a lot of work devoted to choosing node selection rules for the Preflow-Push Algorithm to improve the worst-case running time. Here we consider a simple rule that leads to an improved $O(n^3)$ bound on the number of nonsaturating push operations.

(7.30) If at each step we choose the node with excess at maximum height, then the number of nonsaturating push operations throughout the algorithm is at most $4n^3$.

**Proof.** Consider the maximum height $H = \max_{v \in \{x \mid \delta^*(v) > 0\}} h(v)$ of any node with excess as the algorithm proceeds. The analysis will use this maximum height $H$ in place of the potential function $\Phi$ in the previous $O(n^2m)$ bound.

This maximum height $H$ can only increase due to relabeling (as flow is always pushed to nodes at lower height), and so the total increase in $H$ throughout the algorithm is at most $2n^2$ by (7.26). $H$ starts out 0 and remains nonnegative, so the number of times $H$ changes is at most $4n^2$.

Now consider the behavior of the algorithm over a phase of time in which $H$ remains constant. We claim that each node can have at most one nonsaturating push operation during this phase. Indeed, during this phase, flow is being pushed from nodes at height $H$ to nodes at height $H - 1$; and after a nonsaturating push operation from $v$, it must receive flow from a node at height $H + 1$ before we can push from it again.

Since there are at most $n$ nonsaturating push operations between each change to $H$, and $H$ changes at most $4n^2$ times, the total number of nonsaturating push operations is at most $4n^3$.

As a follow-up to (7.30), it is interesting to note that experimentally the computational bottleneck of the method is the number of relabeling operations, and a better experimental running time is obtained by variants that work on increasing labels faster than one by one. This is a point that we pursue further in some of the exercises.

**Implementing the Preflow-Push Algorithm**

Finally, we need to briefly discuss how to implement this algorithm efficiently. Maintaining a few simple data structures will allow us to effectively implement
the operations of the algorithm in constant time each, and overall to implement the algorithm in time $O(mn)$ plus the number of nonsaturating push operations. Hence the generic algorithm will run in $O(mn^2)$ time, while the version that always selects the node at maximum height will run in $O(n^3)$ time.

We can maintain all nodes with excess on a simple list, and so we will be able to select a node with excess in constant time. One has to be a bit more careful to be able to select a node with maximum height $H$ in constant time. In order to do this, we will maintain a linked list of all nodes with excess at every possible height. Note that whenever a node $v$ gets relabeled, or continues to have positive excess after a push, it remains a node with maximum height $H$. Thus we only have to select a new node after a push when the current node $v$ no longer has positive excess. If node $v$ was at height $H$, then the new node at maximum height will also be at height $H$ or, if no node at height $H$ has excess, then the maximum height will be $H - 1$, since the previous push operation out of $v$ pushed flow to a node at height $H - 1$.

Now assume we have selected a node $v$, and we need to select an edge $(v, w)$ on which to apply push$(f, h, v, w)$ (or relabel$(f, h, v)$ if no such $w$ exists). To be able to select an edge quickly, we will use the adjacency list representation of the graph. More precisely, we will maintain, for each node $v$, all possible edges leaving $v$ in the residual graph (both forward and backward edges) in a linked list, and with each edge we keep its capacity and flow value. Note that this way we have two copies of each edge in our data structure: a forward and a backward copy. These two copies will have pointers to each other, so that updates done at one copy can be carried over to the other one in $O(1)$ time. We will select edges leaving a node $v$ for push operations in the order they appear on node $v$'s list. To facilitate this selection, we will maintain a pointer current$(v)$ for each node $v$ to the last edge on the list that has been considered for a push operation. So, if node $v$ no longer has excess after a nonsaturating push operation out of node $v$, the pointer current$(v)$ will stay at this edge, and we will use the same edge for the next push operation out of $v$. After a saturating push operation out of node $v$, we advance current$(v)$ to the next edge on the list.

The key observation is that, after advancing the pointer current$(v)$ from an edge $(v, w)$, we will not want to apply push to this edge again until we relabel $v$.

(7.31) After the current$(v)$ pointer is advanced from an edge $(v, w)$, we cannot apply push to this edge until $v$ gets relabeled.

**Proof.** At the moment current$(v)$ is advanced from the edge $(v, w)$, there is some reason push cannot be applied to this edge. Either $h(w) \geq h(v)$, or the
edge is not in the residual graph. In the first case, we clearly need to relabel \( v \) before applying a push on this edge. In the latter case, one needs to apply push to the reverse edge \((w, v)\) to make \((v, w)\) reenter the residual graph. However, when we apply push to edge \((w, v)\), then \( w \) is above \( v \), and so \( v \) needs to be relabeled before one can push flow from \( v \) to \( w \) again.

Since edges do not have to be considered again for push before relabeling, we get the following.

(7.32) When the current(\( v \)) pointer reaches the end of the edge list for \( v \), the relabel operation can be applied to node \( v \).

After relabeling node \( v \), we reset current(\( v \)) to the first edge on the list and start considering edges again in the order they appear on \( v \)'s list.

(7.33) The running time of the Preflow-Push Algorithm, implemented using the above data structures, is \( O(mn) \) plus \( O(1) \) for each nonsaturating push operation. In particular, the generic Preflow-Push Algorithm runs in \( O(n^2m) \) time, while the version where we always select the node at maximum height runs in \( O(n^3) \) time.

**Proof.** The initial flow and relabeling is set up in \( O(m) \) time. Both push and relabel operations can be implemented in \( O(1) \) time, once the operation has been selected. Consider a node \( v \). We know that \( v \) can be relabeled at most \( 2n \) times throughout the algorithm. We will consider the total time the algorithm spends on finding the right edge on which to push flow out of node \( v \), between two times that node \( v \) gets relabeled. If node \( v \) has \( d_v \) adjacent edges, then by (7.32) we spend \( O(d_v) \) time on advancing the current(\( v \)) pointer between consecutive relabelings of \( v \). Thus the total time spent on advancing the current pointers throughout the algorithm is \( O(\sum_{v \in V} nd_v) = O(mn) \), as claimed.

### 7.5 A First Application: The Bipartite Matching Problem

Having developed a set of powerful algorithms for the Maximum-Flow Problem, we now turn to the task of developing applications of maximum flows and minimum cuts in graphs. We begin with two very basic applications. First, in this section, we discuss the Bipartite Matching Problem mentioned at the beginning of this chapter. In the next section, we discuss the more general Disjoint Paths Problem.