Bachelor thesis
Computing Science

Radboud University

Acyclic Push-Relabel

Author:
Jelmer Firet
s1023433

First supervisor/assessor:
Prof. dr. F.W. Vaandrager
f.vaandrager@cs.ru.nl

Second assessor:
Dr. J.C. Rot
j.rot@cs.ru.nl

April 4, 2022

Abstract

There are many efficient algorithms for the maximum flow problem, but these do not guarantee that the resulting flow is acyclic. Push-relabel is one of these algorithms that is often used in practice. In this article we present a modification of the push-relabel algorithm that guarantees acyclic flow. We show that acyclic push-relabel has complexity $\mathcal{O}(V^2E)$. We also apply this modification to the maximum-height variant and prove that it runs in $\mathcal{O}(V^3)$, which we show is tight using a parametric testcase.

Contents

Chapter 1

Introduction

The maximum flow problem is an optimisation problem which asks how much can flow from a source to a sink in a graph with edge capacities. It has many applications, such as in scheduling, computer vision and transport.

Many algorithms solve the maximum flow problem. Ford and Fulkerson gave the first algorithm, which uses augmenting paths [8]. Edmonds and Karp made a deterministic version of this algorithm which had a polynomial runtime complexity [7]. Karzanov introduced the concept of a blocking flow [11, 15], which inspired Dinic's algorithm [6]. Goldberg and Tarjan introduced the push-relabel algorithm [9] that maintains a height function and a preflow and converts it into a maximum flow using push and relabel operations. Many recent algorithms build on the push-relabel algorithm [3, 12, 10, 1] and have better algorithmic complexity. HI-PR, an implementation of push-relabel, is still among the fastest maximum flow algorithms in practice.

All of these algorithms can return a maximum flow that contains cycles. As there are algorithms that turn cyclic flow into acyclic flow, there has been no interest in algorithms that guarantee acyclic flow. In competitive programming, where there is limited time to implement an algorithm, a slight change to an existing maximum flow algorithm is preferred. We found a minor modification of push-relabel that guarantees acyclic flow, and we will explore how that change affects the runtime complexity of existing algorithms.

In Chapter 2, we define the maximum flow problem and explain the push-relabel algorithm. Then we will show that the push-relabel is correct and has complexity $\mathcal{O}(V^2E)$. In Chapter 3 we describe a modification of push-relabel and show that it produces acyclic flow. We then adapt the complexity proof to show that the modified algorithm has complexity $\mathcal{O}(V^2E)$. Next, we explain the maximum-height variant of push-relabel and prove that this variant with our modification has complexity $\Theta(V^3)$.

Chapter 2

Preliminaries

In this chapter, we describe the formalisation of the maximum flow problem used by Goldberg and Tarjan[9]. Then we will show that this is equivalent to those given in textbooks. Next, we introduce the Push-Relabel algorithm, which computes a maximal flow. We close with an analysis of Push-Relabel.

2.1 Maximum Flow

2.1.1 Definitions

Definition 2.1.1 (Flow network)
A *flow network* is a graph $G = (V, E)$ with a *source* $s \in V$, a *sink* $t \in V$ and a *capacity function* $c : V \times V \to \mathbb{R}_{\geq 0}$. The *capacity* from u to v is given by $c(u, v)$. We require that $0 < c(u, v)$ if and only if $(u, v) \in E$.

Think about a flow network as a network of pipes. The pipes are the edges, and junctions between those pipes form the vertices. The capacity $c(u, v)$ tells us how many litres can flow from u to v every second.

Definition 2.1.2 (Flow function)
A *flow* is a function $f : V \times V \to \mathbb{R}$ satisfying the constraints

$$\forall_{u,v \in V} : f(u, v) = -f(v, u) \qquad (\textit{skew symmetry})$$
$$\forall_{u,v \in V} : f(u, v) \leq c(u, v) \qquad (\textit{capacity})$$
$$\forall_{v \in V \setminus \{s,t\}} : \sum_{u \in V} f(u, v) = 0 \qquad (\textit{conservation})$$

The *flow from u to v* is given by $f(u, v)$.

The flow from u to v is the volume from u that enters v every second. Skew symmetry ensures that no fluid leaks in pipes. The capacity constraint ensures junctions have no leaks. The capacity constraint prevents that more water flows through a pipe than it can handle.

For a flow we are interested in how much flows from the source to the sink.

Definition 2.1.3 (Value of flow)
The *value of a flow* is $|f| = \sum_{v \in V} f(s, v)$, the net flow out of s.

Definition 2.1.4 (Maximum flow)
A flow f is a *maximum flow* if for any flow g we have $|g| \leq |f|$.

2.1.2 Relation to textbook formalisations

The definitions we use have a few benefits over those found in textbooks[5][13]. Textbooks define flow as how much fluid moves through an edge. It then becomes hard to explain negative flow since volumes are always positive. We associated negative flow with flow in the other direction. Instead, textbooks set $f(u, v) = 0$ when $0 < f(v, u)$, which makes a distinction on flow direction necessary. The conservation constraint is then written as "flow in - flow out":

$$\sum_{u \in V} f(u, v) - \sum_{w \in V} f(v, w) = 0$$

If we include the implicit condition, we will find our definition:

$$\begin{aligned}
0 &= \sum_{\substack{u \in V \\ 0 \leq f(u,v)}} f(u, v) - \sum_{\substack{w \in V \\ 0 \leq f(v,w)}} f(v, w) \\
&= \sum_{\substack{u \in V \\ 0 \leq f(u,v)}} f(u, v) + \sum_{\substack{w \in V \\ f(w,v) \leq 0}} f(w, v) && \text{(skew symmetry)} \\
&= \sum_{u \in V} f(u, v) && \text{(remove case distinction)}
\end{aligned}$$

While we do not require a distinction between flow in and flow out, it gives an useful lemma for sets of vertices:

Lemma 2.1.1 (Flow across set boundary)
for any subset $S \subseteq V \setminus \{s, t\}$: $\sum_{u \in V \setminus S} \sum_{v \in S} f(u, v) = 0$.

Proof

$$\begin{aligned}
0 &= \sum_{u \in V} \sum_{v \in S} f(u, v) && \text{(conservation)} \\
&= \sum_{u \in V \setminus S} \sum_{v \in S} f(u, v) + \sum_{u \in S} \sum_{v \in S} f(u, v) && \text{(split on u)} \\
&= \sum_{u \in V \setminus S} \sum_{v \in S} f(u, v) && \text{(skew symmetry on edges within } S\text{)}
\end{aligned}$$

We can go a bit further and split on the direction of flow:

$$= \sum_{\substack{u \in V \setminus S \\ v \in S \\ 0 < f(u,v)}} f(u,v) - \sum_{\substack{v \in S \\ w \in V \setminus S \\ 0 < f(v,w)}} f(v,w) \quad \text{(case distinction \& skew symmetry)}$$

the final statement expresses that flow into S is the same as flow out of S.■

Unlike [13], we allow edges with capacity into the source and out of the sink. In theorem 2.1.4 we prove that these edges do not change the value of the maximum flow. For this proof, we need some lemmas about flow cycles.

Definition 2.1.5 (Flow cycle)
A *flow cycle* is a simple cycle in the *flow graph* $G_{0<f}$.
$G_{0<f} = (V, E_{0<f})$ where $E_{0<f} = \{(u,v) : 0 < f(u,v)\}$.
A cycle $v_0 \to v_1 \to \ldots \to v_n = v_0$ is simple if $v_i \neq v_j$ for all $i < j < n$.

Lemma 2.1.2 (Remove a single flow cycle)
Let f be a flow and $v_0 \to v_1 \to \ldots \to v_n = v_0$ a flow cycle.
Then we can construct a flow f' such that $|f| = |f'|$ and $\left|E_{0<f'}\right| < |E_{0<f}|$.

Proof Let $\delta = \min\{f(v_i, v_{i+1}) \mid 0 \leq i < n\}$. Take $f' = f$ and then decrease $f'(v_i, v_{i+1})$ by δ for each $i < n$. We also increase $f'(v_{i+1}, v_i)$ by δ for skew symmetry. f' satisfies the capacity constraint:

$$\begin{aligned} f'(v_i, v_{i+1}) &= f(v_i, v_{i+1}) - \delta < f(v_i, v_{i+1}) \leq c(v_i, v_{i+1}) \\ f'(v_{i+1}, v_i) &= f(v_{i+1}, v_i) + \delta \leq f(v_{i+1}, v_i) + f(v_i, v_{i+1}) = 0 \leq c(v_{i+1}, v_i) \end{aligned}$$

The last line also shows that (v_{i+1}, v_i) did not become a new edge in $G_{0<f'}$.
However, the edge that determined δ is deleted from $G_{0<f}$.
Finally observe that for each vertex the net flow remained the same:

$$\begin{aligned} \sum_{u \in V} f'(u, v_i) &= f'(v_{i-1}, v_i) \qquad + f'(v_{i+1}, v_i) \qquad + \ldots \\ &= f(v_{i-1}, v_i) - \delta \quad + f(v_{i+1}, v_i) + \delta \quad + \ldots \quad = \sum_{u \in V} f(u, v_i) \end{aligned}$$

This gives us that the conservation constraint is satisfied and $|f| = |f'|$. ■

Theorem 2.1.3 (Remove all flow cycles)
Given a flow f we can make a flow f' with $G_{0<f'}$ acyclic and $|f| = |f'|$.

Proof We will find f' by a sequence of flows. First set $f_0 = f$. If f_i contains a cycle we apply Lemma 2.1.2 to produce the flow f_{i+1}. This sequence of flows must be finite, as $|E_{0<f_i}|$ decreases each step. Let $f' = f_n$, the last flow in the sequence. Then f' is acyclic and $|f| = |f_0| = \ldots = |f_n| = |f'|$.■

Theorem 2.1.4 (Edges into source / out of sink are irrelevant)
Let G be a flow network. Define G' as G without edges into the source or out of the sink. Then G and G' have the same maximum flow value.

Proof Let $|G|$ be the value of the maximum flow in G. We will prove $|G| = |G'|$ by showing that $|G| \leq |G'|$ and $|G'| \leq |G|$. The latter follows directly from $G' \subseteq G$ since a maximum flow of G' is a flow of G.

Let f be a maximum flow of G. Construct f' by applying Theorem 2.1.3. Assume that $0 < f'(u, s)$. Let S be the vertices with a path to u in $G_{0<f'}$. Note that $s \notin S$, otherwise there is a cycle $s \rightsquigarrow u \to s$ but f' is acyclic. Also $t \notin S$, otherwise we can increase $|f'|$ by sending flow along $t \rightsquigarrow u \to s$. Now we can use Lemma 2.1.1 to find "flow into S = flow out of S". Since all vertices with flow to S are part of S, the flow into S must be zero. But the flow out of S is strictly positive because $0 < f'(u, s)$. We get a contradiction, so we conclude that f' has no flow to the source.
A similar argument shows that f' has no flow out of t. As f' does not use the edges in $E \setminus E'$ it is also a flow for G'. Thus $|G| = |f| = |f'| \leq |G'|$. ■

2.1.3 Augmenting path

We shall end this section with a sufficient condition for a flow to be maximal.

Definition 2.1.6 (Residual graph)
Define the *residual capacity* of an edge as $r(u, v) = c(u, v) - f(u, v)$.
The *residual graph* is the induced subgraph of edges with residual capacity: $G_{f<c} = (V, E_{f<c})$ where $E_{f<c} = \{(u, v) : f(u, v) < c(u, v)\}$.

Then we have the following theorem:

Theorem 2.1.5 (Maximum flow has no augmenting paths)
A flow f is a maximum flow iff $G_{f<c}$ does not contain a path from s to t.

Proof Let f be a maximum flow, and assume $s = v_0 \to \ldots \to v_n = t$ is a simple path in $G_{f<c}$. Compute $\delta = \min\{r(v_i, v_{i+1}) \mid 0 \leq i < n\} > 0$. Define f' with $f'(v_i, v_{i+1}) = f(v_i, v_{i+1}) + \delta$. Set $f'(v_{i+1}, v_i) = f(v_{i+1}, v_i) - \delta$ to satisfy skew symmetry. Let $f'(u, v) = f(u, v)$ for the remaining edges. We need to check if edges with more flow satisfy the capacity constraint:

$$f'(u, v) = f(u, v) + \delta \leq f(u, v) + c(u, v) - f(u, v) = c(u, v)$$

We also check if vertices on the cycle still satisfy the conservation constraint:

$$\begin{aligned}\sum_{u \in V} f'(u, v_i) &= f'(v_{i-1}, v_i) \qquad + f'(v_{i+1}, v_i) \qquad + \ldots \\ &= f(v_{i-1}, v_i) + \delta \quad + f(v_{i+1}, v_i) - \delta \quad + \ldots \quad = \sum_{u \in V} f(u, v_i) = 0\end{aligned}$$

Thus f' is a flow with $|f'| = |f| + \delta > |f|$, contradicting f being maximal.

Now the reverse statement, suppose $G_{f<c}$ contains no paths from s to t. Define S as the vertices reachable from s in $G_{f<c}$. Since $t \notin S$ we can use Lemma 2.1.1 and the definition of flow value to show that for all flows: flow out of S - flow into S = value of flow. For f, the flow into S is zero. Otherwise take $u \in V \setminus S, v \in S, 0 < f(u,v)$. But then $f(v,u) < 0 \leq c(v,u)$ and so $u \in S$ giving a contradiction. Now let f' be a (different) flow. Observe that for every edge out of S we have $f'(u,v) \leq c(u,v) = f(u,v)$. We find that f' can not have more flow out of S than f. Less flow into S is also impossible since f already has no flow into S. Therefore $|f'| \leq |f|$. ■

2.2 Push-Relabel

Push-Relabel is a non-deterministic algorithm by Goldberg and Tarjan that solves the maximum flow problem[9]. Many explanations of Push-Relabel exist in textbooks [13, 17], lectures [14] and on Wikipedia [16]. We have based our explanation on those sources.

2.2.1 Description of Push-Relabel

Instead of maintaining a flow, Push-Relabel works on preflows. A preflow is like a flow but has a less restrictive version of the conservation constraint.

Definition 2.2.1 (Preflow)
A *preflow* on network G is a function $f : V \times V \to \mathbb{R}$ that satisfies skew symmetry, the capacity constraint and (instead of conservation):

$$\forall_{v \in V \setminus \{s\}} : 0 \leq \sum_{u \in V} f(u,v) \qquad (excess)$$

Our intuition for a preflow is similar to that of a flow. But now we imagine a vertex as a bucket. A bucket is open on top, so it can overflow when more flows in than out. The excess constraint prevents the opposite, more flow out of a bucket than into it. Then the bucket would dry up, so the flow out can not continue forever.

Definition 2.2.2 (Excess)
The sum in the excess constraint is called the *excess* of vertex v, written as $x_f(v)$. We define $x_f(s) = \infty$ to indicate that the source can create flow.
A vertex is called **active** when it has excess, $0 < x_f(u)$.

In our intuition, an active vertex corresponds to an overflowing bucket.
The excess is how many litres flow over the edge every second.

Theorems for flows also have counterparts for preflows:

Lemma 2.2.1 (Flow across set boundary (preflow))
for any subset $S \subseteq V \setminus \{s\}$: $\sum_{u \in V \setminus S} \sum_{v \in S} f(u,v) = \sum_{v \in S} x_f(v)$. The difference between flow in and out of a set is the sum of the excesses.

Proof Similar to the proof of Lemma 2.1.1 ■

Theorem 2.2.2 (Remove all flow cycles in a preflow)
Given a preflow f we can make a preflow f' with $G_{0<f'}$ acyclic and where the excesses are the same: $\forall_{v \in V} : x_f(v) = x_{f'}(v)$.

Note The excess constraint replaces both $|f| = |f'|$ and conservation.

Proof Similar to the proof of Theorem 2.1.3 ■

A preflow is a flow where all the excesses are zero (no bucket is overflowing). This state will be the aim of the Push-Relabel algorithm. When this happens, we want the flow to be maximal. Since there is no notion of a "maximum preflow" we will instead use the condition that there is no augmenting path (see Theorem 2.1.5). We will use a height function to guarantee this:

Definition 2.2.3 (Height function)
A *height function* for a flow f on network G is a function $h : V \to \mathbb{R}_{\geq 0}$ that satisfies the following constraints:

$$h(s) = |V| \quad \text{(source)}$$
$$h(t) = 0 \quad \text{(sink)}$$
$$h(u) \leq h(v) + 1 \text{ if } f(u,v) < c(u,v) \quad \text{(residue)}$$

The *height* of a vertex u is $h(u)$.

In our buckets-with-pipes intuition, we imagine each bucket attached to a rope and hoisted $h(u)$ meters up. The source bucket is high up while the sink is on the ground. The last constraint states that when a bucket is more than one meter above another, the pipe between them is saturated.

Note We allow flow to go up one meter through each pipe. Imagine that we achieve this with a small pump. With our modification this is not necessary.

The existence of a height function ensures that there is no augmenting path:

Theorem 2.2.3 (Height function prevents augmenting path)
If preflow f has a height function h, then $G_{f<c}$ has no path from s to t.

Proof Assume that $s = v_0 \to \ldots \to v_n = t$ is a simple path in $G_{f<c}$. By the definition of a height function we have $h(v_i) \leq h(v_{i+1}) + 1$ since $0 < r(v_i, v_{i+1})$ implies $f(v_i, v_{i+1}) < c(v_i, v_{i+1})$. Then we can derive

$$|V| = h(v_0) \leq h(v_1) + 1 \leq h(v_2) + 2 \leq \ldots \leq h(v_n) + n = n$$

On the other hand $n < |V|$ since $v_0 \to \ldots \to v_n$ is a simple path. We must conclude that a path from s to t in $G_{f<c}$ can not exist. ■

All Push-Relabel algorithms follow the same scheme. They initialise the pre-flow and height function:

$$f(u,v) = \begin{cases} c(u,v) & \text{if } u = s \\ -c(v,u) & \text{if } v = s \\ 0 & \text{otherwise} \end{cases} \qquad h(u) = \begin{cases} |V| & \text{if } u = s \\ 0 & \text{otherwise} \end{cases}$$

Then they perform pushes and relabels to turn the preflow into a max flow.

Definition 2.2.4 (Push)
A *push* from u to v increases $f(u,v)$ by δ and decreases $f(v,u)$ by δ where $\delta = \min(x_f(u), c(u,v) - f(u,v))$. We only push when $0 < \delta$ and $h(v) < h(u)$.

Definition 2.2.5 (Relabel)
A *relabel* of u increases $h(u)$ by one. We only relabel vertices with excess that are not the source, the sink, or a vertex that can push to other vertices.

Note In [9] the term 'label' was used for the height of a vertex. While using 'height' and 'relabel' is confusing, there is no good alternative. Since the algorithm is called push-relabel, we must use the term relabel. The term 'height' gives intuition for the algorithm that 'label' does not.

When we push from bucket u to bucket v, we increase the flow through the pipe. We cannot exceed the capacity, so the increase is limited to the residual capacity. We must also prevent bucket u drying up, so we can only use the water that previously went over the edge of bucket u. The formula for δ encodes these restrictions. The other conditions on pushes and relabels guarantee that h remains a height function, which we proof in Lemmas 2.2.5 and 2.2.6. But first, we give an example of how push-relabel works.

2.2.2 Example execution of Push-Relabel

We will consider the network depicted in Figure 2.1. First we initialise the height and flow functions (2.1a) . Then we relabel A because there are no valid pushes (2.1b), after which we push from A to C (2.1c). As there are no valid pushes we relabel C (2.1d). Now we could push from C to T, instead we choose to push from from C to B (2.1e). Then we can relabel B (2.1f) and push from C to T (2.1g). The algorithm will continue pushing between A, B and C while relabelling those vertices. Eventually $h(A) = 6$ and a final push from A to S causes the algorithm to terminate (2.1h).

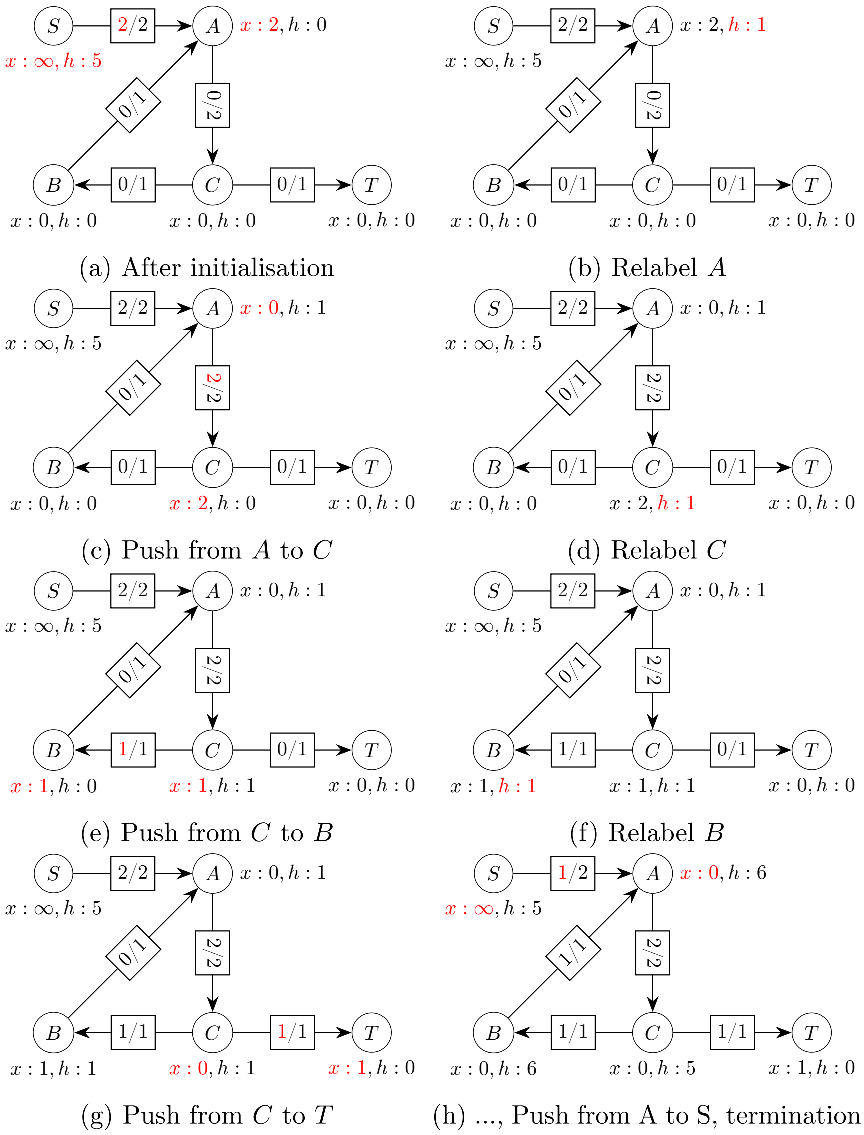


Figure 2.1: Example of Push-Relabel

2.2.3 Correctness of Push-Relabel

While describing Push-Relabel, we already covered most of the argument on why it is correct. The correctness proof uses the invariant that h is a height function for the preflow f. We must show that this is indeed an invariant. In particular, that it holds after initialisation and after each operation.

Lemma 2.2.4 (Height function invariant - initialisation)
After initialisation the function h is a height function for the preflow f.

Proof During initialisation we set $h(s) = |V|$, so the source constraint holds. The sink constraint also holds since $s \neq t$ and therefore $h(t) = 0$. We consider the residue constraint in two parts: $u = s$ and $u \neq s$. If $u = s$ then $f(s, v) = c(s, v)$ and the residue constraint vacuously holds. If $u \neq s$ then $h(u) = 0$ and we have $h(u) = 0 \leq h(v) < h(v) + 1$. ■

Lemma 2.2.5 (Height function invariant - push)
If $h(v) < h(u)$ then a push from u to v maintains the invariant.

Proof The source and sink constraints are not affected by a push. The residue constraint may be broken by a new edge $v \to u$ in $G_{f<c}$. But $h(v) < h(u) < h(u) + 1$, so this does not happen. ■

Lemma 2.2.6 (Height function invariant - relabel)
If u has excess and is not the source, the sink or a vertex that can push; then relabelling u maintains the invariant.

Proof Since $u \neq s$ and $u \neq t$, the source and sink constraints still hold. The residue constraints could break when $h(u)+1 > h(v)+1$. Then $h(v) < h(u)$, but there is no valid push from u to v. Since u has excess we must have $f(u, v) = c(u, v)$. Therefore the residue constraint will hold vacuously. ■

The algorithm finishes when there are no valid operations to perform.

Theorem 2.2.7 (Push-Relabel finishes with flow)
When Push-Relabel finishes the preflow f will be a flow.

Proof Recall the earlier observation that f is a flow when all excesses are zero (except source and sink). Suppose that vertex u still has excess when the algorithm finishes. Since the algorithm terminated, there are no valid pushes from u. But then we can relabel u, so the algorithm could not have finished. ■

Because h is a height function of f, the flow the algorithm produces must be maximal (Theorem 2.2.3). However, we have not yet proven that the algorithm terminates. The complexity analysis in the next section shows that the algorithm does finish.

2.2.4 Complexity of Push-Relabel

Multiple factors determine the complexity of Push-Relabel:

1. the complexity of an operation,
2. the complexity of selecting the next operation,
3. the number of operations before the algorithm finishes.

It is possible to implement `push` and `relabel` to run in constant time. The other factors depend on the specific variant used. The complexity of selecting an operation is often constant. We are going to bound the number of operations performed by generic Push-Relabel.

The simplest operations to count are relabels.

Lemma 2.2.8 (Flow path to node with excess)
If a vertex v has excess, there exists a path $s \rightsquigarrow v$ in $G_{0<f}$

Proof Define $S = \{u \mid \text{there exists a path } u \rightsquigarrow v \text{ in } G_{0<f}\}$.
Suppose $s \notin S$, then using Lemma 2.2.1 we find
$0 < x_f(v) \leq \sum_{w \in S} x_f(w) = \sum_{u \in V \setminus S} \sum_{w \in S} f(u, w)$.
Since the final sum is positive, we have a term $0 < f(u, w)$.
Then $u \to w \rightsquigarrow v$ is a path in $G_{0<f}$, but $u \notin S$. We must conclude $s \in S$.■

Lemma 2.2.9 (Height of vertex at most $2V$)
A vertex will never have $2V \leq h(u)$.

Proof Assume we can relabel u to height $2V$. Then u must contain excess. By Lemma 2.2.8, there is a path $s = v_0 \to \ldots \to v_n = u$ in $G_{0<f}$. For each i we have $f(v_{i+1}, v_i) < 0 \leq c(v_{i+1}, v_i)$ since $0 < f(v_i, v_{i+1})$. From the residue constraint we then get $h(v_{i+1}) \leq h(v_i)+1$. For the entire path we then must have $h(u) \leq h(s) + n$. Since $h(u) = 2V - 1 = h(s) + V - 1$ we know that $V - 1 \leq n$. Thus the path visits all vertices and we have $h(v_{i+1}) = h(v_i)+1$s. In particular $h(v_{n-1}) < h(u)$ and $f(u, v_{n-1}) < 0 \leq c(u, v_{n-1})$. As we can push from u to v_{n-1}, we may not relabel u after all. ■

Lemma 2.2.10 (At most $\mathcal{O}(V^2)$ relabels)
During execution of the algorithm, there are at most $\mathcal{O}(V^2)$ relabels.

Proof Each vertex starts with $0 \leq h(u)$ and will never exceed $h(u) < 2V$. Each relabel increases $h(u)$ by one, so each vertex can be relabelled at most $2V - 1$ times. The total number of relabels is at most $V(2V - 1) \in \mathcal{O}(V^2)$■

Next, we count the pushes in two parts: saturating and non-saturating. We make this distinction because we can bound saturating pushes. Then we use that bound to restrict the number of non-saturating pushes.

Definition 2.2.6 (Saturating push)
A push $u \to v$ is *saturating* if $f(u, v) = c(u, v)$ after the push.

Lemma 2.2.11 (At most $\mathcal{O}(VE)$ saturating pushes)
During execution of the algorithm, there are $\mathcal{O}(VE)$ saturating pushes.

Proof Look at a single edge $u \to v$. After a saturating push from u to v we have $f(u,v) = c(u,v)$. Since $\delta \leq c(u,v) - f(u,v) = 0$, we may not push from u to v again. First, we need a push from v to u, so we need to relabel v. We can relabel $\mathcal{O}(V)$ times, so there are $\mathcal{O}(V)$ saturating pushes per edge. Therefore we perform $\mathcal{O}(VE)$ saturating pushes. ■

Finally, we will bound the non-saturating pushes with the potential method.

Definition 2.2.7 (Activating push)
A push $u \to v$ *activating* if $x_f(v) = 0$ before the push $0 < x_f(v)$ and after.

Definition 2.2.8 (Emptying push)
A push $u \to v$ is *emptying* if $0 < x_f(u)$ before the push $x_f(u) = 0$ and after.

Lemma 2.2.12 (At most $\mathcal{O}(V^2E)$ non-saturating pushes)
Define the potential P as the sum of the heights of active vertices.

After initialisation each vertex has $h(u) = 0$ except for the start vertex. Therefore $P = h(s) = V$. The potential is always non-negative, so P decreases at most V over the entire algorithm.

Now consider what happens if we relabel u. Based on the conditions of a relabel, we know that u is active and $h(u)$ increases by one. Thus each relabel increases the potential by one.

Then we look at a saturating push from u to v. We are only interested in the maximum increase of P, which happens for an activating non-emptying push. Then the potential increases by $h(v) \leq V$.

Finally, we look at non-saturating pushes. Since we want to bound this operation, we want to know the minimum decrease of P. This happens for a non-activating emptying push when it decreases P by $h(u) - h(v) \geq 1$.

Because the potential increases at most $\mathcal{O}(V^2) \cdot 1 + \mathcal{O}(VE) \cdot V = \mathcal{O}(V^2E)$ and decreases by at least one for each non-saturating push we can bound the number of non-saturating pushes by $\frac{\mathcal{O}(V^2E)+V}{1} = \mathcal{O}(V^2E)$.

Chapter 3

Research

In this chapter, we will describe ways to compute acyclic maximum flow. We introduce an algorithm based on push-relabel and show that it has the same complexity. Then we look at the maximum-height variant of push-relabel and modify it too. We also prove its complexity and show that this bound is tight by giving a parametric testcase.

3.1 Preventing cyclic flow

If we consider again at the example in Section 2.2.2, we see that after running Push-Relabel the flow looks as follows:

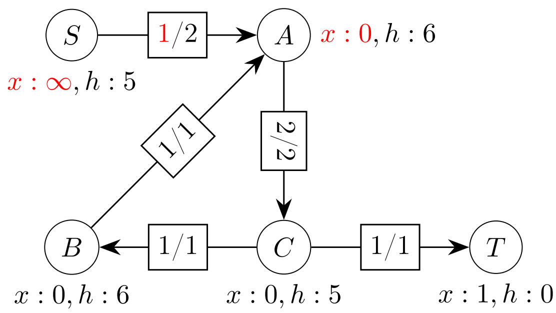


Figure 3.1: Final flow after running Push-Relabel

This network contains a flow cycle $A \to C \to B \to A$, which may be undesired. There are a couple of methods to guarantee acyclic flow. First, we can run Push-Relabel and then remove the flow cycles afterwards with a depth-first search. This approach is likely used in practice if acyclic flow is required.

We can also modify Push-Relabel to forbid pushes that form a flow cycle. To check this requires a depth-first or breadth-first search for each push. But that increases the complexity as pushes are the most common operation.

In this thesis, we will take a third approach: constrain the height function so that a valid push can never form a cycle. A push from u to v is only valid when $h(v) < h(u)$. This push forms a cycle if there is a path $v \rightsquigarrow u$ in $G_{0<f}$. so we want the following constraint:

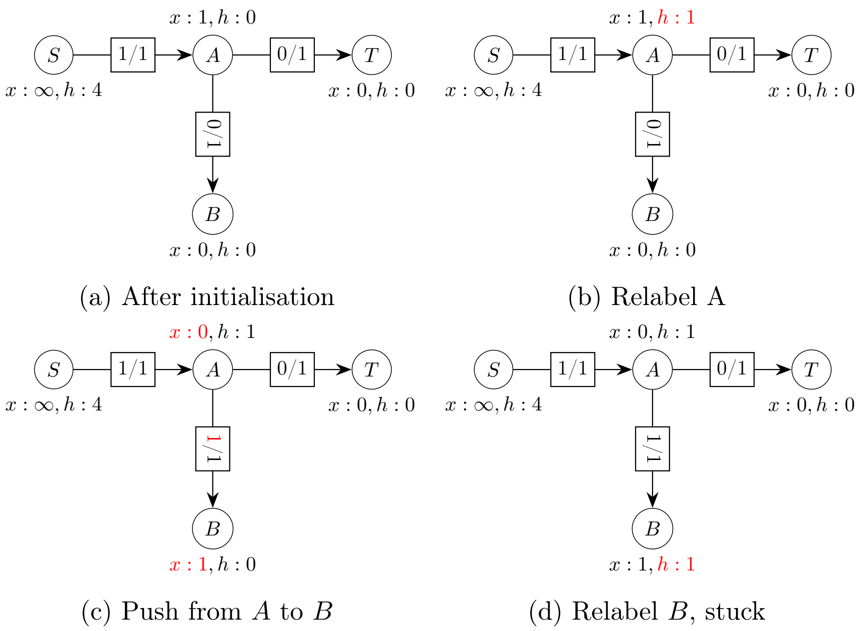
Definition 3.1.1 (Downhill constraint)
All paths $v \rightsquigarrow u$ in $G_{0<f}$ satisfy $h(u) \leq h(v)$, or equivalently
All edges $v \to u$ in $G_{0<f}$ satisfy $h(u) \leq h(v)$.

This constraint forbids flow from a vertex to a higher vertex.

Lemma 3.1.1 (Equivalent definitions downhill constraints)
Both statements of the downhill constraint are equivalent

Proof That (b) follows from (a) is trivial, a single edge is also a path.
For the other direction consider the path $u = u_0 \to u_1 \to \ldots \to u_k = v$.
Then we have $h(u) = h(u_0) \leq h(u_1) \leq \ldots \leq h(u_k) = h(v)$. ∎

We want to change Push-Relabel to ensure that the downhill constraint always holds. As the constraint is invariant under pushes, we should change when relabelling is allowed. Currently, we may only relabel u when there is no valid push out of u. We can forbid relabels that break the downhill constraint. However, this gives a new problem:



(a) After initialisation

(b) Relabel A

(c) Push from A to B

(d) Relabel B, stuck

Figure 3.2: Push-Relabel gets stuck with naive change to valid relabels

Consider the network in Figure 3.2. In three steps we get into a situation where we are stuck. Essentially the push from A to B was a mistake, but we could not have foreseen that. Push-Relabel can fix this mistake by relabelling B and pushing back from B to A. We are stuck because the downhill constraint forbids relabelling B.

We introduce a new operation to solve this problem. A *flat push* is a push to a vertex at the same height. However, we assumed that all pushes go to a lower layer to guarantee acyclicity. Instead, we must ensure that flat pushes don't introduce a new edge in $G_{0<f}$. We get the following definition of a flat push:

Definition 3.1.2 (Flat push)
A *flat push* from u to v increases $f(u,v)$ and decreases $f(v,u)$ by δ, which is valid if $0 < \delta = \min(x_f(u), -f(u,v))$ and $h(u) \mathbf{=} h(v)$.
A flat push is saturating when $\delta = -f(u,v)$.

From now on, we will call the original push a "down push".

Definition 3.1.3 (Down push)
A *down push* from u to v increases $f(u,v)$ and decreases $f(v,u)$ by δ, which is valid if $0 < \delta = \min(x_f(u), \mathbf{c(u,v)} - f(u,v))$ and $h(v) \mathbf{<} h(u)$.
A down push is saturating when $\delta = \mathbf{c(u,v)} - f(u,v)$.

There are two observations for implementing Acyclic Push-Relabel:

1. The definitions of a flat push and a down push are similar. We have highlighted the differences between the in bold. This similarity makes it convenient to implement both types of push in a single function.
2. We don't need to change the conditions for relabelling. The introduction of flat pushes covers the extra restrictions on relabels that we need to guarantee acyclicity. If the edge $u \to v$ has flow, then there is a valid flat push from v to u when $h(u) = h(v)$. Since a relabel of v is only valid when there is no valid push out of v, we cannot relabel v above $h(u)$ until we removed $u \to v$ from $G_{0<f}$. This makes sure that $h(v) \leq h(u)$ for every edge $u \to v$ in $G_{0<f}$.

3.2 Complexity of Acyclic Push-Relabel

When you change an algorithm, you need to consider correctness and complexity again. Old proofs may depend on specific aspects of an algorithm that may not hold anymore. Luckily, the correctness argument for Push-Relabel still works for Acyclic Push-Relabel. However, we need to adapt the complexity analysis.

The $\mathcal{O}(V^2)$ bound on relabels, proven in Lemma 2.2.10, remains valid.
It is possible to tighten the upper bound on $h(u)$ from $h(u) < 2V$ to $h(u) \leq V$. Since we may only relabel a vertex u when it has excess there must be a path $s \rightsquigarrow u$ in $G_{0<f}$, which means that $h(u) \leq h(s) = V$. Since constant factors are ignored, the complexity is still $\mathcal{O}(V^2)$.

The $\mathcal{O}(VE)$ bound on saturating pushes (Lemma 2.2.11) also remains valid. When we saturate an edge $u \to v$, we need a push back before the next saturating push from u to v. For that, we need to relabel v at least once. We can relabel v at most V times, so each edge has at most V saturating pushes in each direction. The $\mathcal{O}(VE)$ bound follows directly.

The $\mathcal{O}(V^2E)$ bound on non-saturating pushes needs a new proof. With standard Push-Relabel, the summed height of active vertices decreased for every non-saturating push. But this does not hold for flat pushes. They can even increase that sum, which means the complexity argument also breaks for non-saturating down pushes.

We will give two new proofs of the $\mathcal{O}(V^2E)$ bound on non-saturating pushes in Acyclic Push-Relabel. The first uses the potential method like the original proof. The second gives a direct relation between non-saturating pushes and other operations.

3.2.1 Fixing the complexity proof

To fix the proof, we need a new potential function that uses an order $<$ on the vertices. With respect to this order, we want every push to go to a smaller vertex.

Definition 3.2.1 (Order of vertices)
For two vertices u, v we say that $u < v$ iff

- $h(u) < h(v)$ or
- $h(u) = h(v)$ and u was relabelled to $h(u)$ before v.

When $h(u) = h(v) = 0$ we can assign an arbitrary order.

Lemma 3.2.1 (Flat pushes are decreasing)
For every push $u \to v$ we have $v < u$.

Proof For a down push $u \to v$ we have $h(v) < h(u)$, so $v < u$ by definition. Now consider a flat push from u to v, which is only valid when $0 < f(v, u)$. Look back to the first moment where both vertices are at layer $h(u) = h(v)$. Since then there were only flat pushes between u and v, which can not reverse the inequality $0 < f(v, u)$. If v was relabelled last, there must have been a moment where $h(v) < h(u)$ and $0 < f(v, u)$, which contradicts the downhill constraint. Thus v was relabelled before u and we have $v < u$. ■

Now define the index $i(u)$ of a vertex u as the number of v with $v < u$. Let P, the potential function, be the sum of i for the active vertices.

The potential is at most $0 + 1 + \ldots + V - 1 = \frac{V(V-1)}{2}$ after initialisation. When the algorithm terminates, the potential is zero. Therefore the total increase and decrease of the potential differ at most $\mathcal{O}(V^2)$.

Consider what happens to the potential when we relabel a vertex u. Because we increase the height of u, the position of u in the order can change. $i(u)$ increases at most $V-1$, which would happen if u is the smallest vertex and becomes the largest. The order of the other vertices remains the same, so their indices cannot increase. Therefore the potential increases by at most $V-1$ for each relabel.

index	0	1	2	3	4
before	v	u	t	w	s
after	v	t	w	u	s

(a) Relabbeling u changes its place in the order. The potential increases from $1+3=4$ to $2+3=5$

index	0	1	2	3	4
before	v	u	t	w	s
after	v	u	t	w	s

(b) A sat. non-empt. act. push from w to u increases the potential from $0+3=3$ to $0+1+3=4$

Now look at a saturating push from u to v. If v becomes active we find the largest increase of the potential: P increases by $i(v) < V$.

Finally consider the non-saturating pushes. A non-saturating push is always emptying, which decreases the potential by $i(u)$. If this push is also activating it increases by $i(v)$. Since we push from u to v we know $i(v) < i(u)$, so $i(u) - i(v) \geq 1$. So each non-saturating push decreases P by at least one.

If we combine all operations we see that the potential increases at most $\mathcal{O}(V) \cdot \mathcal{O}(V^2) + \mathcal{O}(V) \cdot \mathcal{O}(VE) = \mathcal{O}(V^2E)$. The number of non-saturating pushes is then $(\mathcal{O}(V^2E) + \mathcal{O}(V^2))/1 = \mathcal{O}(V^2E)$, where the $\mathcal{O}(V^2)$ term comes from the initialisation. From this it follows that the generic Acyclic Push-Relabel has runtime complexity $\mathcal{O}(V^2E)$.

3.2.2 Complexity proof without potential function

The proof we described shows that the algorithm has complexity $\mathcal{O}(V^2E)$ but does not give much intuition for this bound. Now we will give a direct connection between non-saturating pushes and other operations. That requires a better understanding of the relationship between the terms 'saturating', 'emptying' and 'activating'.

Recall that for a down push we send $\delta = \min(x_f(u), c(u,v) - f(u,v))$. We send enough to either saturate the edge or empty vertex u. Therefore every non-saturating push is emptying, and every non-emptying push is saturating. The latter observation allows us to apply the $\mathcal{O}(VE)$ bound of saturating pushes to non-emptying pushes.

Next, we will compare the emptying pushes and activating pushes. Each activating push increases the number of active vertices by one. Every emptying push removes all excess from a vertex, decreasing the number of active vertices by one. We have between 0 and V active vertices after initialisation and none when the algorithm terminates. Therefore the number of activating and emptying pushes differs at most V. If we take the complement, we find that the difference between the number of non-activating pushes and non-emptying pushes is the same. So we can bound the non-activating pushes by $\mathcal{O}(VE)$.

The activating emptying pushes still need an upper bound. These pushes transfer all excess from one vertex to another. We will combine these pushes into chains:

Definition 3.2.2 (Chain of pushes)
Consider a chronological list of pushes during the execution of Push-Relabel.
Each chain consists of a subset of pushes from this list.
Start a new chain for every non-emptying push.
Then build each chain by repeating the following steps:

- Look at the last push $u \to v$ in the chain.
- If this push is not activating, the chain is finished.
- Otherwise, find the next emptying push out of v and append it.

Note Since the initialisation step gives vertices excess, it should also start chains. We can apply the definition above if we look at initialisation from a different angle. Recall that initialisation sets

$$f(u,v) = \begin{cases} c(u,v) & \text{if } u = s \\ -c(v,u) & \text{if } v = s \\ 0 & \text{otherwise} \end{cases} \qquad h(u) = \begin{cases} |V| & \text{if } u = s \\ 0 & \text{otherwise} \end{cases}$$

This was needed for the correctness argument. Instead we will start with $f(u,v) = 0$ $h(u) = 0$. Then we relabel s once, push from s to all other vertices and then relabel s until $h(s) = V$. This sequence of operations results in the same initial values, so the algorithm still works. We can also include these individual operations in the chains.

We will illustrate the definition of chains with the following example.
Suppose that during Push-Relabel the following pushes occur:

nr.	from	to	emptying	activating	chain		
					A	B	C
1	s	a	no	yes	start		
2	s	b	no	yes		start	
3	a	c	yes	yes			
4	c	b	yes	no	end		
5	b	c	yes	yes			
6	c	t	no	no			start
end							
7	c	a	yes	yes			
8	a	s	yes	no		end	

These pushes form three chains. Chain A starts with a non-emptying push from s to a. The next push emptying push from a is push 3 to c. In push 4, c is emptied into b. This push is not activating, so the chain ends. Similarly, chains B and C consist of pushes [2,5,7,8] and [6].

We will divide each chain into chain segments based on the relabels. So suppose that a chain contains a push from a to b and then a push from b to c. We split the chain between those pushes if b relabels in between.
Recall that every push goes to a vertex with a smaller index. For a chain segment with k pushes, the index of the first and last vertex differ more than k. Since indices range from 0 to $V-1$, a chain segment can have at most $V-1$ pushes.

Now observe that each chain segment begins with a non-emptying push or the first emptying push after a relabel. Thus there are at most $\mathcal{O}(VE)+\mathcal{O}(V^2) = \mathcal{O}(VE)$ chain segments. Each chain segment contains $\mathcal{O}(V)$ pushes, so there are at most $\mathcal{O}(V^2E)$ pushes during Acyclic Push-Relabel.

3.3 Variants of Push-Relabel

3.3.1 Maximum height Push-Relabel

Push-Relabel is a non-deterministic algorithm that solves the maximum flow problem. It defines two operations and when you can apply them, but not in what order to execute these operations. A variant of push-relabel is an algorithm that uses push-relabel and defines how to choose operations. Variants can have a lower time complexity than generic Push-Relabel if they choose good actions to perform. In this section, we will look at the maximum-height variant. An implementation of this variant, with some additional heuristics, has long been the benchmark for maximum flow algorithms[1].

The maximum-height variant uses the following meta-operation:

Definition 3.3.1 (Discharge)
To *discharge* vertex u, we repeat these steps while u has excess:
- For all neighbours v: push from u to v if that is allowed.
- Relabel u if there are no valid pushes out of u.

As the name implies, the maximum-height variant repeatedly discharges the highest active vertex. This variant originates from the first article about Push-Relabel written by Goldberg and Tarjan [9]. They claimed that this variant has a complexity of $\mathcal{O}(V^3)$. Cheriyan and Maheshwari [2] showed that the algorithm runs in $\mathcal{O}(V^2\sqrt{E})$ and that this bound is tight. Later Cheriyan and Melhorn [4] gave a new argument for the same bound. We will describe their proof and then adapt it to show that the acyclic version of this variant runs in $\mathcal{O}(V^3)$. Finally, we will give a network where the algorithm finds the maximum flow in $\Omega(V^3)$, proving that the worst-case bound is tight.

Theorem 3.3.1 (Maximum-height Push-Relabel runs in $\mathcal{O}(V^2\sqrt{E})$)
The maximum-height Push-Relabel algorithm has $\mathcal{O}(V^2\sqrt{E})$ operations.

Proof (Cheriyan and Melhorn [4]) Let h^* be the height of the highest active vertex. h^* starts and ends at 0. It can increase if a vertex relabels, so the total increase of h^* is in $\mathcal{O}(V^2)$. Thus we find that h^* decreases at most $\mathcal{O}(V^2)$ times. Define a phase as a sequence of pushes for which h^* remains the same. We have just shown there can be at most $\mathcal{O}(V^2)$ phases. Define $\mathcal{N}_{\leq}(u) = \#\{v \mid h(v) \leq h(u)\}$ and the potential $P = \sum_{u \text{ active}} \mathcal{N}_{\leq}(u)$. We have $P \leq V^2$ initially, and $P = 0$ at termination.
Each relabel and saturating push can increase the potential by V.

Now consider the non-saturating pushes during a phase. A phase is short if it has at most $\sqrt{E}$ non-saturating pushes. Because there are $\mathcal{O}(V^2)$ phases, we perform $\mathcal{O}(V^2\sqrt{E})$ non-saturating pushes in short phases.

Now consider the non-saturating pushes in a long phase. Each push goes to a lower layer, so decreases the potential by $\mathcal{N}_{\leq}(h^*) - \mathcal{N}_{\leq}(h^* - 1)$.
This is the number of vertices at layer h^*, which is at least $\sqrt{E}$.
We find that there are $(V^2 + V \cdot \mathcal{O}(V^2) + V \cdot \mathcal{O}(VE))/\sqrt{E} = \mathcal{O}(V^2\sqrt{E})$ non-saturating pushes in long phases.

Therefore the maximum height variant performs $\mathcal{O}(V^2\sqrt{E})$ operations. ■

3.3.2 Maximum index Push-Relabel

If we want to add flat pushes to the maximum-height Push-Relabel, we should change the selection rule. Instead of discharging the vertex with maximal height, we empty the vertex with the highest index. A flat push can not reactivate a vertex with a greater index. Thus each vertex is discharged once per phase. We will name this the maximum-index variant and show that it has complexity $\Theta(V^3)$.

Theorem 3.3.2 (Maximum-index Push-Relabel runs in $\mathcal{O}(V^3)$) The maximum-index Push-Relabel performs at most $\mathcal{O}(V^3)$ operations.

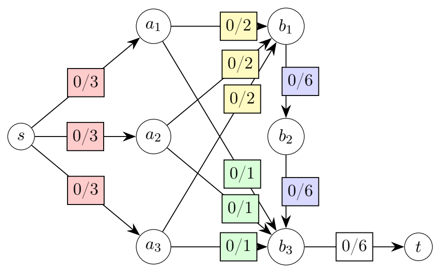
Proof Take the definition of a phase from the proof of Theorem 3.3.1. Each phase has at most V discharges, as each empties a vertex and only activates vertices with a lower index. A discharge has at most one non-saturating push because such a push leaves no excess. Therefore there are at most V non-saturating pushes per phase. So we perform $\mathcal{O}(V^3)$ non-saturating pushes during the entire algorithm. ■

This bound is worse than the $\mathcal{O}(V^2\sqrt{E})$ bound for maximum-height push-relabel. We can use the chain-based argument for generic push-relabel to explain this difference. It is hard to make long chains with the maximum-height variant. The longer a chain is, the lower the last vertex. As we always discharge the highest vertex, it is hard to extend the chain. The same reasoning works for the maximum-index variant, which explains why it has better complexity than the generic acyclic push-relabel. But with the maximum-index variant, a single layer can contain an entire chain. So the end of the chain is, in a sense, closer to the vertex with the highest index. Therefore the chain is easier to extend.
We will now construct a parametric testcase where maximum-index push-relabel performs $\Omega(V^3)$ operations. First, we describe the structure of this testcase. Then we describe how the algorithm executes.

The testcase contains four types of vertices: the source s, the sink t, storage nodes $a_1, \dots, a_n$ and conveyor nodes $b_1, \dots, b_n$. Between these vertices we add the following edges:

from	to	capacity
s	a_i	n
a_i	b_1	$n-1$
a_i	b_n	1
b_i	b_{i+1}	$n(n-1)$
b_n	t	$n(n-1)$

(a) Overview of edges in the testcase



(b) The testcase for $n = 3$

We can distinguish two stages when we execute maximum-index Push-Relabel on this network. The second stage will have $\Omega(n^3)$ non-saturating pushes, which gives us the desired lower bound.

The order of discharges depends on the initial ordering of the vertices. We will assume that $s > a_1 > \ldots > a_n > b_1 > \ldots > b_n > t$, so that all edges in the network go to a smaller vertex. We will also assume that each discharge first pushes to the vertex with the smallest index.

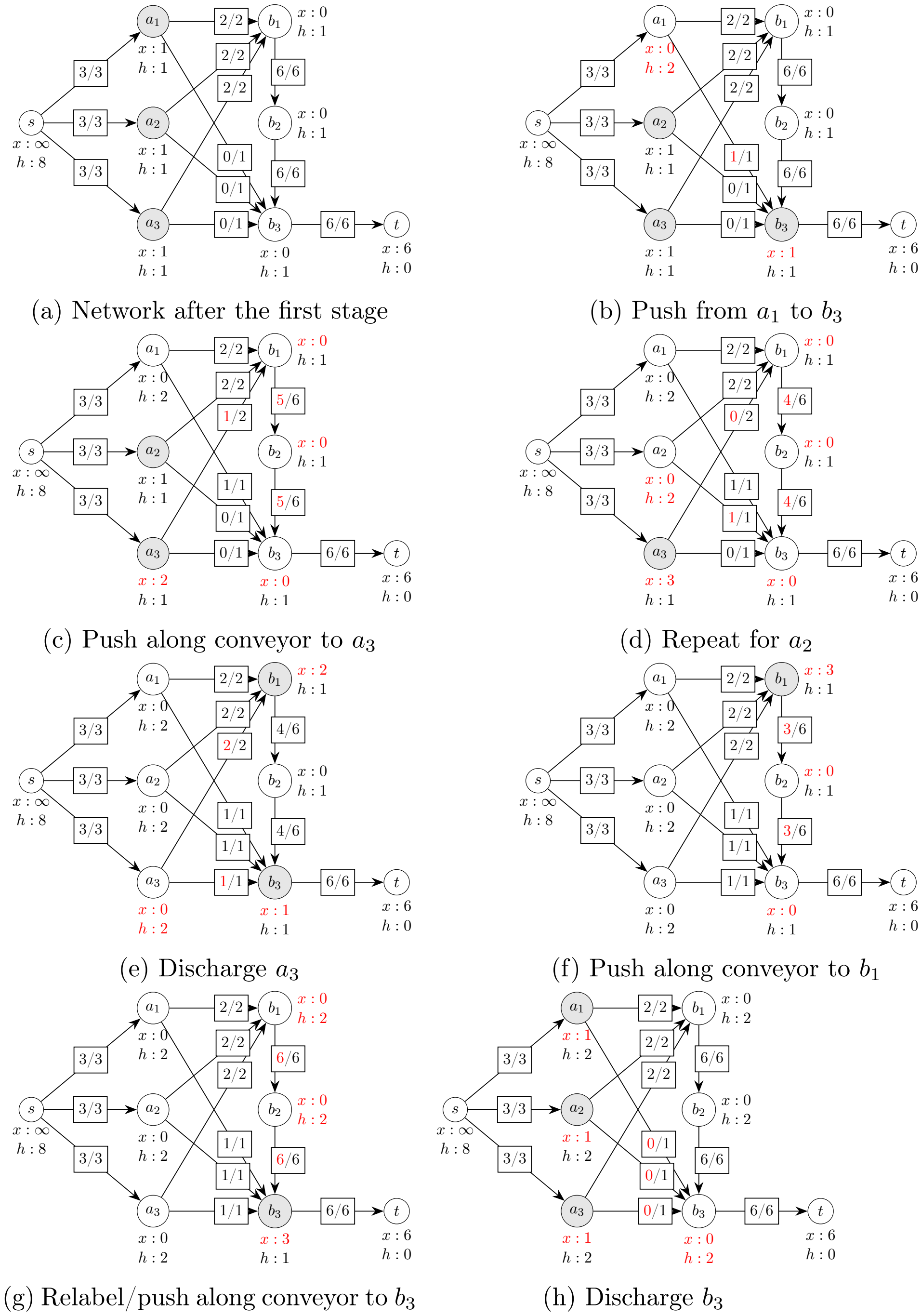
The first stage saturates all edges and moves as much flow to the sink. During initialisation, s saturates the edges to all a_i. Then a_n is relabelled and pushes to b_1 and b_n. All other storage nodes do the same. Next, b_1 relabels and pushes everything to b_2. Other conveyor nodes do the same: relabel and push to the next conveyor node. Finally, b_n pushes everything it can to the sink and the remaining n units back to the storage nodes. Figure 3.5a shows the state after the first stage.

The second stage starts with a_1 relabelling and pushing one unit to b_n (3.5b). Then the excess at b_n will be pushed along the conveyor to b_1. b_1 will push this excess to a_n (3.5c). Next a_2 relabels, and repeats this pattern (3.5d). The other a_i will follow. Finally a_n relabels, pushes the excess it accumulated to b_1, and the one remaining unit to b_n (3.5e). This unit moves along the conveyor to b_1 (3.5f). Then b_1 relabels and pushes n units to b_2. The other conveyor nodes do the same, and the excess arrives at b_n (3.5g). b_n relabels and pushes one unit back to every storage node (3.5h).

Now we have the same preflow as the one at the start of the second stage. Furthermore, the height of every vertex increased by one. The order of the vertices is still $s > a_1 \ldots a_n > b_1 > \ldots > b_n > t$, only the order between storage nodes is different. Because the current state is (almost) identical, we will see the same pattern at each layer. The order between storage nodes is irrelevant since those nodes have identical connections.

The algorithm finishes when b_n relabels to height V. It pushes a unit of excess back to each storage node, which pushes it to the source.

In summary: for each layer, all the storage nodes push one unit of excess to b_n, which moves through all the conveyor nodes. Moving the excess through the conveyor nodes takes $n - 1$ flat pushes. These pushes happen for all n storage nodes and $2n + 1$ layers.Therefore the algorithm performs $\Omega((n-1)n(2n+1)) = \Omega(n^3)$ flat pushes. As $V = \Theta(n)$ we find that the algorithm performs $\Omega(V^3)$ operations in the worst case.



(a) Network after the first stage

(b) Push from a_1 to b_3

(c) Push along conveyor to a_3

(d) Repeat for a_2

(e) Discharge a_3

(f) Push along conveyor to b_1

(g) Relabel/push along conveyor to b_3

(h) Discharge b_3

Figure 3.5: Execution of maximum-index Push-Relabel

Chapter 4

Conclusions

Our introduction of flat pushes is a minor modification of push-relabel that guarantees acyclic flow. The resulting algorithm has the same complexity as generic push-relabel, namely $\mathcal{O}(V^2E)$. We give two proofs of this complexity. The second proof provides a direct correspondence between non-saturating pushes and other operations. In our opinion, this proof is more enlightening than proofs that use the potential method. This proof even works for standard push-relabel. We could modify the maximum-height variant to guarantee acyclic flow. We have proven that the maximum-index variant executes $\Theta(V^3)$ operations, which is not as good as the maximum-height variant.

Future work is to investigate how the modification affects the complexity of other variants. Especially for the dynamic-trees variant since it has the lowest complexity of all push-relabel variants. In our experience, there are often faster algorithms for acyclic graphs than general graphs. We believe that a push-relabel algorithm could use the topological order in choosing operations. Finally, we could look at how the flat push modification applies to algorithms that are based on push-relabel. We think it could improve the partial-augment algorithm of Goldberg [1]. The current fastest push-relabel algorithm (King-Rao-Tarjan [12]) is also based on push-relabel but might benefit less from flat pushes.

Bibliography

[1] Andrew V. Goldberg. "The Partial Augment–Relabel Algorithm for the Maximum Flow Problem". In: *Algorithms - ESA 2008*. Ed. by Dan Halperin and Kurt Mehlhorn. Red. by David Hutchison et al. Vol. 5193. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 466–477. ISBN: 978-3-540-87743-1 978-3-540-87744-8. DOI: 10.1007/978-3-540-87744-8_39.

[2] J. Cheriyan and S. N. Maheshwari. "Analysis of Preflow Push Algorithms for Maximum Network Flow". In: *SIAM J. Comput.* 18.6 (Dec. 1989), pp. 1057–1086. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0218072.

[3] Joseph Cheriyan, Torben Hagerup, and Kurt Mehlhorn. "Can a Maximum Flow Be Computed in o(Nm) Time?" In: *Automata, Languages and Programming*. Ed. by Michael S. Paterson. Vol. 443. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1990, pp. 235–248. ISBN: 978-3-540-52826-5. DOI: 10.1007/BFb0032035.

[4] Joseph Cheriyan and Kurt Mehlhorn. "An Analysis of the Highest-Level Selection Rule in the Preflow-Push Max-Flow Algorithm". In: *Information Processing Letters* 69.5 (Mar. 1999), pp. 239–242. ISSN: 00200190. DOI: 10.1016/S0020-0190(99)00019-8.

[5] Thomas H. Cormen et al. "Maximum Flow". In: *Introduction to Algorithms*. 3rd ed. Cambridge, Mass: MIT Press, 2009, pp. 708–766. ISBN: 978-0-262-03384-8 978-0-262-53305-8.

[6] E.A. Dinic. "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation". In: *Doklady Akademii Nauk SSSR* 11 (1970), pp. 1277–1280. URL: https://www.cs.bgu.ac.il/~dinitz/D70.pdf (visited on 05/03/2021).

[7] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *J. ACM* 19.2 (Apr. 1972), pp. 248–264. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321694.321699.

[8] L. R. Ford and D. R. Fulkerson. “Maximal Flow Through a Network”. In: *Can. j. math.* 8 (1956), pp. 399–404. ISSN: 0008-414X, 1496-4279. DOI: 10.4153/CJM-1956-045-5.

[9] A V Goldberg and R E Tarjan. “A New Approach to the Maximum Flow Problem”. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing - STOC ’86*. The Eighteenth Annual ACM Symposium. Berkeley, California, United States: ACM Press, 1986, pp. 136–146. ISBN: 978-0-89791-193-1. DOI: 10.1145/12130.12144.

[10] Dorit S. Hochbaum. “The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem”. In: *Operations Research* 56.4 (Aug. 2008), pp. 992–1009. ISSN: 0030-364X, 1526-5463. DOI: 10.1287/opre.1080.0524.

[11] Alexander V. Karzanov. “Determining the Maximal Flow in a Network by the Method of Preflows”. In: *Soviet Mathematics - Doklady* 15 (1974), pp. 434–437.

[12] V. King, S. Rao, and R. Tarjan. “A Faster Deterministic Maximum Flow Algorithm”. In: *Journal of Algorithms* 17.3 (Nov. 1994), pp. 447–474. ISSN: 01966774. DOI: 10.1006/jagm.1994.1044.

[13] Jon Kleinberg and Éva Tardos. *Algorithm design*. New international edition. First edition. Harlow: Pearson Education Limited, 2014. ISBN: 978-1-292-02394-6 1-292-02394-5.

[14] Tim Roughgarden. “The Push-Relabel Algorithm for Maximum Flow”. A Second Course in Algorithms (Stanford University). Jan. 13, 2016. URL: https://www.youtube.com/watch?v=0hI89H39USg (visited on 05/03/2021).

[15] Robert Endre Tarjan. “A Simple Version of Karzanov’s Blocking Flow Algorithm”. In: *Operations Research Letters* 2.6 (Mar. 1984), pp. 265–268. ISSN: 01676377. DOI: 10.1016/0167-6377(84)90076-2.

[16] Wikipedia. *Push-Relabel Maximum Flow Algorithm*. Wikipedia. URL: https://en.wikipedia.org/wiki/Push-relabel_maximum_flow_algorithm (visited on 05/03/2021).

[17] David P. Williamson. *Network Flow Algorithms*. Cambridge, United Kingdom ; New York, NY, USA: Cambridge University Press, 2019. ISBN: 978-1-107-18589-0 978-1-316-63683-1.

Index

Definitions

Theorems

Appendix A

Implementation

A.1 maximum-index push-relabel

```
#include <vector>
#include <queue>
#include <tuple>
#include <algorithm>
using namespace std;

struct todoEntry {
    int height, index, node;
    bool operator<(const todoEntry& other) const {
        return make_tuple(      height,       index,       node)
            < make_tuple(other.height, other.index, other.node);
    }
};

struct flowGraph {
    int numNode, source, sink;
    vector<int> height, index, numAtLayer, excess, currentEdge;
    vector<vector<int>> flow, capacity, neighbors;
    priority_queue<todoEntry> todo;

    flowGraph(int _numNode, int _source, int _sink) {
        numNode = _numNode; source = _source; sink = _sink;
        capacity.assign(numNode,vector<int>(numNode,0));
        flow.assign(numNode,vector<int>(numNode,0));
        neighbors.resize(numNode);
        currentEdge.assign(numNode, 0);
        excess.assign(numNode, 0); excess[source] = 1e9;
        height.assign(numNode, 0); height[source] = numNode;
        index.assign(numNode, 0);
        numAtLayer.assign(numNode+1, 1); numAtLayer[0] = numNode;
    }

    void add_capacity(int from, int to, int diff) {
        if (capacity[from][to] == 0 and capacity[to][from] == 0) {
            neighbors[from].push_back(to); neighbors[to].push_back(from);
        }
        capacity[from][to] += diff;
    }
```

```
    int residual(int from, int to) {
        int res = 0;
        if (height[to] < height[from])  res += capacity[from][to];
        if (height[to] <= height[from]) res -= flow[from][to];
        return max(0, res);
    }

    void push(int from, int to) {
        if (excess[to] == 0 and to != source and to != sink)
            todo.push({height[to],index[to],to});
        int diff = min(residual(from, to), excess[from]);
        excess[from] -= diff; excess[to] += diff;
        flow[from][to] += diff; flow[to][from] -= diff;
    }

    void discharge(int from) {
        while (0 < excess[from]) {
            int to = neighbors[from][currentEdge[from]];
            push(from, to);

            if (excess[from] == 0) continue;
            currentEdge[from]++;
            if (currentEdge[from] < (int) neighbors[from].size()) continue;

            height[from]++;
            currentEdge[from] = 0;
            index[from] = numAtLayer[height[from]];
            numAtLayer[height[from]]++;

            sort(neighbors[from].begin(), neighbors[from].end(), [&](int a, int b) {
               return make_pair(height[a], index[a]) < make_pair(height[b], index[b]);
            });
        }
    }

    void calculate() {
        for (int to : neighbors[source])
            push(source, to);
        while (not todo.empty()) {
            int from = todo.top().node; todo.pop();
            discharge(from);
        }
    }
};

int main() {
    // Counterexample
    // 0 = source, 1 .. n = storage, n+1 .. 2n = conveyor, 2n+1 = sink
    int n = 200;
    flowGraph graph4(2*n+2, 0, 2*n+1);
    for (int i = 1; i <= n; i++) {
        graph4.add_capacity(0, i, n);
        graph4.add_capacity(i, n+1, n-1);
        graph4.add_capacity(i, 2*n, 1);
    }
    for (int i = n+1; i < 2*n; i++)
        graph4.add_capacity(i, i+1, n*(n-1));
    graph4.add_capacity(2*n, 2*n+1, n*(n-1));
    graph4.calculate();
    return 0;
}
```