University of Michigan–Ann Arbor

Department of Electrical Engineering and Computer Science

EECS 498 004 **Advanced Graph Algorithms**, Fall 2021

Lecture 21: Part 1 - Push-Relabel Flow Algorithms Introduction

December, 2021

Instructor: Thatchaphol Saranurak

Scribe: Syed Akbari

Contents

1 Introduction and Overview **2**

2 Definitions **3**

2.1 Flow . . . 3

2.1.1 Definitions . . . 3

2.1.2 Points to note . . . 3

2.2 Demand Functions . . . 4

2.2.1 Definitions . . . 4

2.2.2 Points to note . . . 4

2.3 Source/Sink Functions . . . 5

2.3.1 Definitions . . . 5

2.3.2 Points to note . . . 5

2.4 Preflow . . . 7

2.4.1 Definitions . . . 7

2.4.2 Points to note . . . 7

3 The Push-Relabel Framework **8**

3.1 Inputs, Structure, and Invariants of the Algorithm . . . 8

3.2 Algorithm . . . 9

3.3 Analysis . . . 11

3.3.1 Input/Output . . . 11

3.3.2 Counting push and relabel calls . . . 12

3.3.3 Basic Implementation . . . 13

3.4 Total Running Time . . . 14

4 Applications **14**

4.1 Max-Flow (h = $n + 1$) . . . 14

4.1.1 Inputs/Outputs: . . . 14

4.1.2 Algorithms and Analysis: . . . 15

4.2 Sparse Cut or Matching Embedding (h = $O(\frac{log(n)}{\phi})$) . . . 15

4.2.1 Inputs/Outputs: . . . 15

4.2.2 Algorithm and Analysis . . . 16

4.3 Balanced Sparse Cut or Matching Embedding With Fake Edge . . . 18

5 Faster Implementation With Dynamic Trees **19**

5.1 Top Tree Data Structure . . . 19

5.2 Top Tree PUSH/RELABEL Implementation . . . 20

5.2.1 Rephrased Invariants . . . 20

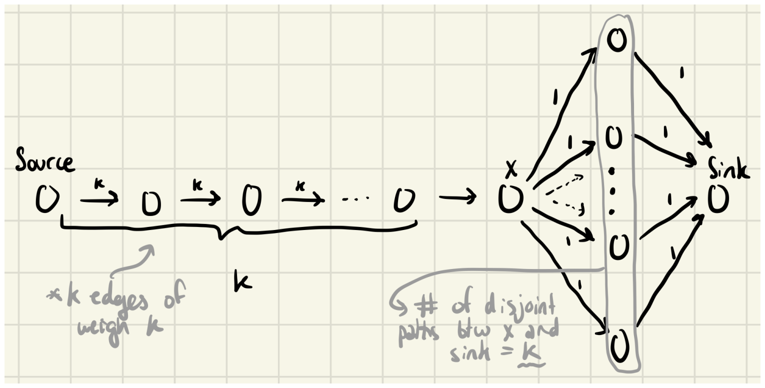
5.2.2 Updated Algorithm . . . 21

5.2.3 Analysis . . . 21

1 Introduction and Overview

In this lecture, we will be introducing a new classical framework for working with flow problems: the push-relabel framework. Like the augmenting path framework we've done most of our work in so far (or blocking flow and multiplicative weight update frameworks if you're familiar with them), push-relabel refers to a general strategy or way of thinking about and working with flow problems.

Push-relabel has lots of really neat properties, as it is fast, dynamic, and is one of the most used algorithms in practice. The key property however that makes the push-relabel paradigm so powerful is that it allows for "amortization"; roughly speaking, this means it divides up the work very well locally on each node and edge, such that it can solve complex flow problems defined on all n nodes of the graph in only one call of the push-relabel algorithm, whereas most strategies we've seen require n calls . Consider the following flow problem as a motivating example, where there are there are k edges of k weight each going between the source and x, and between x and the sink, there are k disjoint paths, each of unit length:[1]



For this example, let k be some large number, like 100,000. Consider the augmenting-path based algorithm for solving this problem, the Ford-Fulkerson algorithm, which runs in $\Omega(k^2)$ time on this graph. Notice that at each iteration, the algorithm wastes time re-exploring the long path between the source and x, even though it doesn't change. What feels more intuitive would be to simple route the flow once in $O(k)$ time from the source to x, then from x to the sink, push the the flow again in simply $O(k)$ time through each of the disjoint paths- kind of like water flowing down all the available paths of least resistance. So intuitively, we see an an approach solving this $O(k)$ time overall.

Push-relabel is a framework built to, first, capture this essential idea in a way to work with any arbitrary graph, and second, be flexible enough to deal with all sorts of flow problems. To capture these two ideas in their full generality however, there are a several definitions that must first be introduced, but fortunately, they are all fairly intuitive to understand. Using these definitions, we can define the general push-relabel algorithm. After this point, we will first show a basic example of push-relabel in use, to find max-flow, and then a very cool modern example of push-relabel, to

[1]This example is drawn from: https://www.youtube.com/watch?v=0hI89H39USg&t=184s

efficiently find sparse-cuts and an efficient matching player strategy for the cut-matching game.

2 Definitions

You are all familiar at this point with the basic ideas of flow, demand, and source/sink with (s,t)-flows. Below however, we will redefine each of these in the context of push-relabel. They will still essentially be very similar to the (s,t)-flow style definitions, but they are just generalizations that run the engine under-the-hood for push-relabel. Redefining these three will allow us to define one new idea, preflow, after which we will have everything we need to start talking about the algorithm itself.

All the definition below are defined, given the following:

Let $G = (V, E, c)$ be an undirected capacitated graph where edge capacities are c$\in \mathbb{R}^E_{\geq 0}$.
For any $S \subseteq V$ let $\delta(S) = c(E(S, V \setminus S))$ denote the size of the cut S.

2.1 Flow

2.1.1 Definitions

Definition 2.1 (flow)**.** A **flow** $f : V \times V \to \mathbb{R}$ satisfies $f(u,v) = -f(v,u)$ and $f(u,v) = 0$ for $\{u,v\} \notin E$.

- The notation $f(u,v) > 0$ means that **mass** is routed in the direction from u to v.

Definition 2.2 (Congestion)**.** The **congestion** of f is $\max_{\{u,v\}\in E} \frac{|f(u,v)|}{c(u,v)}$. If the congestion is at most 1, we say that f is **feasible** or **respects capacities**.

- So in feasible flow, we have $-c(u,v) \leq f(u,v) \leq c(u,v)$. (Negative just means sending mass from v to u).

Definition 2.3 (Flow out/in of f)**.** The **net flow going out of** u is $f_{out}(u) = \sum_{v\in V} f(u,v)$. As you would expect, **net flow coming into** u is $f_{in}(u) = \sum_{v\in V} f(v,u)$.

Definition 2.4 (Flow out/in of S)**.** Similarly, the **net flow going out of** S is $f_{out}(S) = \sum_{u\in S} f_{out}(S)$ and the **net flow coming into** S is $f_{in}(S) = \sum_{u\in S} f_{in}(S)$.

2.1.2 Points to note

For building intuition, it's nice to play around with the definitions to see what are simple things we can deduce immediately. For example, directly from these definitions we get:

- $f_{out}(u) + f_{in}(u) = 0$ for all u.
- $f_{out}(V) = \sum_u f_{out}(u) = 0$

Also, compare this new definition of flow with the old (s,t)-definition. For any old-school (s,t)-flow f, $f_{out}(u) = 0$ for $u \neq s,t$, and f must respect capacity, whereas for our modern generalized idea of flow, both of these constraints are relaxed.

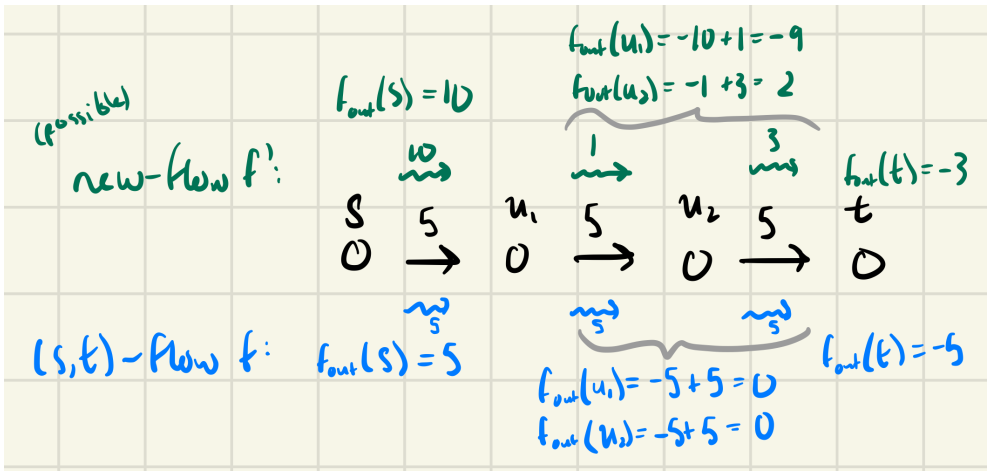


Figure 1: Relaxation of $f_{out}(u) = 0$, and edges can be greater than capacity.

2.2 Demand Functions

2.2.1 Definitions

Definition 2.5 (Demand function)**.** A **demand function** is simply some function mapping vertices in V to a number, so $\mathsf{dem} : V \to \mathbb{R}$.

Definition 2.6 (Flow satisfying demand)**.** We say that flow f **satisfies demand** dem if $\mathsf{dem}(u) = f_{out}(u)$ for all $u \in V$.

Definition 2.7 (Flow satisfying demand)**.** We say that flow f **satisfies demand** dem if $\mathsf{dem}(u) = f_{out}(u)$ for all $u \in V$.

Definition 2.8 (Total demand)**.** For any $S \subseteq V$, let $\mathsf{dem}(S) = \sum_{v \in S} \mathsf{dem}(v)$ be the **total demand** on S.

2.2.2 Points to note

Consider any of demand function we define (using this new idea of demand) such that:

$$\mathsf{dem}(u) = \begin{cases} \mathsf{val} & \text{if } u = s \\ -\mathsf{val} & \text{if } u = t \\ 0 & \text{if } u \neq s, t \end{cases}$$

A flow f satisfies this demand if and only if it is an (s,t)-flow. clearly then, this new definition is a generalization of our old-school (s,t)-flow idea of demand.

We also proved the following two ideas in HW2:

- A demand dem is satisfiable iff $\sum_u \mathsf{dem}(u) = 0$ (assuming that G is connected).
- A demand dem is satisfiable by some feasible flow iff $|\mathsf{dem}(S)| \leq \delta(S)$ for all $S \subseteq V$. *Hint*: Max-flow min-cut.

2.3 Source/Sink Functions

2.3.1 Definitions

Definition 2.9 (Source/sink function)**.** Let $\Delta : V \longrightarrow \mathbb{R}_{\geq 0}$ be a **source function** and $T : V \longrightarrow \mathbb{R}_{\geq 0}$ be a **sink function**. (Note: they map to non-negative numbers)

- $\Delta(v)$ specifies the amount of mass initially placed on v.

  Definition 2.10 (Source value for $S \subseteq V$)**.** $\Delta(S) = \sum_{u \in S} \Delta(u)$

- $T(v)$ specifies the capacity of v as a sink.

  Definition 2.11 (Sink value for $S \subseteq V$)**.** $T(S) = \sum_{u \in S} T(u)$

Definition 2.12 (Flow 'at' a node)**.** $f_{at}(v) = \Delta(v) + f_{in}(v)$, so the flow 'at' a v is the **net flow ending at** v after the routing f.

Definition 2.13 (Flow source-sink)**.** f satisfies source-sink (Δ, T) iff:

1. $f_{out}(v) \leq \Delta(v)$ for all v
   (i.e. the net flow out of v is at most its initial mass)

2. $f_{at}(v) \leq T(v)$ for all v
   (i.e. the net flow ends at v is at most its sink capacity)

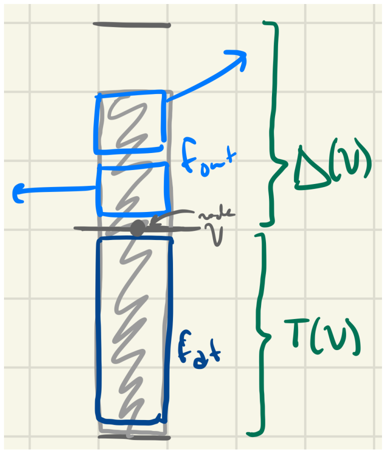


Figure 2: $f_{out}(v) \leq \Delta(v)$ and $f_{at}(v) \leq T(v)$.

2.3.2 Points to note

These ideas of source and sink functions can be thought of as connecting the previous two generalized ideas of flow and demand, into a workable framework in terms of flows that "satisfy (Δ, T)".

Consider the following problem: Given $A, B \subseteq V$ where $|A| \ll |B|$:

- route the flow such that every vertex in A should send 1 unit of flow to some node in B, and...

- each vertex in B can receive at most 1 unit.

Thinking about this in terms of $\textsf{dem}$ and (s, t)-flows, there is no clear idea of how to approach this. We could try to add a new super source node and new super sink node, and then work off of that, but if we want to solve this problem in the original graph, there is no natural way to do so. However, "(Δ, T) satisfaction" can deal with these types of problems quite naturally. Consider the following exercise.

Exercise 2.1. Given a satisfiable demand $\textsf{dem}$ (i.e. $\sum_u \textsf{dem}(u) = 0$), we can define (Δ, T) that capture the $\textsf{dem}$. Let

$$\Delta(u) = \begin{cases} \textsf{dem}(u) & \text{if } \textsf{dem}(u) \geq 0 \\ 0 & \text{else} \end{cases}$$

$$T(u) = \begin{cases} -\textsf{dem}(u) & \text{if } \textsf{dem}(u) < 0 \\ 0 & \text{else} \end{cases}$$

Observe that f satisfies $\textsf{dem}$ iff f satisfies (Δ, T).

Proof. Suppose f satisfies $\textsf{dem}$. Then, we have $f_{out}(u) = \textsf{dem}(u) \leq \Delta(u)$ and also $f_{at}(u) = \Delta(u) + f_{in}(u) = \Delta(u) - \textsf{dem}(u)$, which means

$$f_{at}(u) = \begin{cases} \textsf{dem}(u) - \textsf{dem}(u) = 0 \leq T(u) & \text{if } \textsf{dem}(u) \geq 0 \\ 0 - \textsf{dem}(u) \leq T(u) & \text{if } \textsf{dem}(u) < 0. \end{cases}$$

Next, suppose f satisfies (Δ, T). Then, we have

$$f_{out}(u) \leq \Delta(u)$$

and

$$\Delta(u) - f_{out}(u) \leq T(u) \iff f_{out}(u) \geq \Delta(u) - T(u).$$

So for u where $\textsf{dem}(u) \geq 0$, we have

$$f_{out}(u) = \Delta(u) = \textsf{dem}(u).$$

Now, for u where $\textsf{dem}(u) < 0$, we have

$$\textsf{dem}(u) \leq f_{out}(u) \leq 0.$$

We claim that $\textsf{dem}(u) = f_{out}(u)$ too in this case. Otherwise, suppose for contradiction that there is u where $\textsf{dem}(u) < f_{out}(u) \leq 0$. Then, we get a contradiction below

$$0 = \sum_{u:\textsf{dem}(u)\geq 0} \textsf{dem}(u) + \sum_{u:\textsf{dem}(u)<0} \textsf{dem}(u) < \sum_{u:\textsf{dem}(u)\geq 0} f_{out}(u) + \sum_{u:\textsf{dem}(u)<0} f_{out}(u) = 0.$$

□

Exercise 2.2. We have

- (Δ, T) is satisfiable by some flow iff $\Delta(V) \leq T(V)$. (Assuming that G is connected.)
- (Δ, T) is satisfiable by some feasible flow iff $\Delta(S) - T(S) \leq \delta(S)$ for all $S \subseteq V$. *Hint*: Max-flow min-cut.

Proof. The work here carries over straightforwardly from 2.2.2. □

2.4 Preflow

2.4.1 Definitions

Given source/sink functions (Δ, T)...

Definition 2.14 (Preflow)**.** We say that f is a **preflow** w.r.t. (Δ, T) iff:

- $f_{out}(v) \leq \Delta(v)$ for all v
- **but we allow** $f_{at}(v) > T(v)$
  (this is the 'excess' unit(s) of flow left that push-relabel is trying to 'push' away to some sink, so we aren't 'finished' building the flow, hence the name 'preflow')

Definition 2.15 (Absorbed mass)**.** Let $\text{ab}_f(v) = \min\{f_{at}(v), T(v)\}$ be the **absorbed mass** on v (note: nonnegative).

Definition 2.16 (Saturated sink)**.** When $\text{ab}_f(v) = T(v)$, we say that v**'s sink is saturated**.

Definition 2.17 (Excess)**.** Let $\text{ex}_f(v) = f_{at}(v) - \text{ab}_f(v)$ be the **excess on** v (note: nonnegative).

2.4.2 Points to note

The first key point to note is that f satisfies source sink (Δ, T) if and only if there is no excess.

Proposition 2.3. *f satisfies (Δ, T) iff $\text{ex}_f(v) = 0$ for all v.*

Proof. $\text{ex}_f(v) = 0 \iff f_{at}(v) = \text{ab}_f(v) \iff f_{at}(v) \leq T(v)$. □

Proposition 2.4. *Let f be a preflow w.r.t (Δ, T). Then, $f_{at}(v) \geq 0$ for all v.*

Proof. We have $f_{at}(v) = \Delta(v) + f_{in}(v) = \Delta(v) - f_{out}(v)$, but $f_{out}(v) \leq \Delta(v)$. □

So all three of the terms we care about, $\text{ex}_f, \text{ab}_f, f_{at}$, are nonnegative, and can be understood naturally in terms of each other.

Exercise 2.5. For any vertex set $S \subseteq V$, we have

$$\underbrace{\Delta(S)}_{\text{initial mass}} = \underbrace{f_{out}(S)}_{\text{net flow out}} + \underbrace{\text{ab}_f(S)}_{\text{absorbed mass}} + \underbrace{\text{ex}_f(S)}_{\text{excess}}$$

Proof. For any u, we have two inequalities

$$f_{at}(u) = \Delta(u) + f_{in}(u)$$
$$f_{at}(u) = \text{ex}_f(u) + \text{ab}_f(u).$$

So

$$\Delta(u) = f_{out}(u) + \text{ex}_f(u) + \text{ab}_f(u)$$

and we are done by summing over $u \in S$. □

3 The Push-Relabel Framework

Given the above terminology, we have everything we need to describe the structure and actual algorithm of the PUSH/RELABEL framework.

3.1 Inputs, Structure, and Invariants of the Algorithm

Inputs:

- graph $G = (V, E, c)$
- source function Δ
- sink function T
- parameter h (height)

Maintains (keeps track of these structures at all steps) :

- a preflow f w.r.t. (Δ, T)
- a label for each vertex $\ell : V \to \{0, \ldots, h\}$

Definition 3.1 (Valid state)**.** We say that (f, ℓ) (aka: flow f w.r.t the status of the vertex labels ℓ at the current iteration in the algorithm) is a (G, Δ, T)**-valid state** iff:

1. For all nodes $u, v \in V$, whenever $\ell(u) > \ell(v) + 1$, then f fully saturates (u, v) from u to v (i.e. $f(u, v) = c(u, v)$)
2. For all nodes u s.t. $\ell(u) \geq 1$, f has completely filled the sink for u, or in other words, u is fully absorbed (i.e. $\text{ab}(u) = T(u)$).

Definition 3.2 (Valid solution)**.** We say that (f, ℓ) is a (G, Δ, T)**-valid solution** if (f, ℓ) is a (G, Δ, T)**-valid state**, and additionally, we have that

1. If $\ell(u) < h$, then u has no excess (i.e. $\text{ex}(u) = 0$)

Given these inputs and structures, we can state the key property for the algorithm. Assuming we are given a (G, Δ, T)-valid state as input, the goal is to keep (f, ℓ) a (G, Δ, T)-valid state at all times. So the two **invariants** come from the two parts of definition 3.1:

1. *"For all nodes $u, v \in V$, whenever $\ell(u) > \ell(v) + 1$, then f fully saturates (u, v) from u to v (i.e. $f(u, v) = c(u, v)$)"*

   This can be understood as saying that "only edges between nodes in adjacent levels in ℓ have some left over capacity over which you can still route some flow".

   It can be helpful to think of the directed **residual graph** $G_f = (V, E_f)$ where $E_f = \{(u, v) \mid r_f(u, v) > 0\}$, where $r_f(u, v) = c(u, v) - f(u, v)$ is the **residual capacity** of (u, v). In these terms, this variant can be understood as saying that all the "downward edges" in G_f cannot skip levels.

2. *"For all nodes u s.t. $\ell(u) \geq 1$, f has completely filled the sink for u, or in other words, u is fully absorbed (i.e.* $\text{ab}(u) = T(u)$*)."*

   This is basically saying that all the nodes at non zero levels have no more sink space left inside them, so provided they have any excess, it must be routed elsewhere.

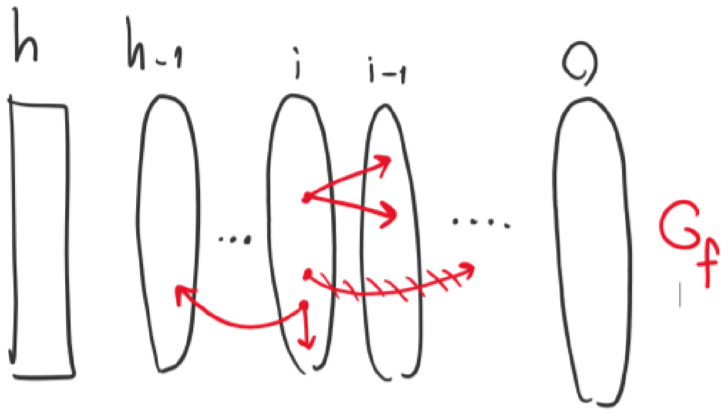


Figure 3: Residual graph interpretation of invariant 1.

The goal of the algorithm is then to, while keeping in line with these invariants, end with (f, ℓ) as a (G, Δ, T)**-valid solution**.

Initialization:
You can think of G, Δ, and T, as being the "raw inputs", so in practice, these would be given to you. h, as will become clearer below, is something you set, depending on the type of problem you are working with.

Now, while the algorithm will be maintaining and updating f and ℓ at each iteration, you can choose what state you want to start them off in. As we want the algorithm to keep (f, ℓ) in a (G, Δ, T)-valid state at all times (recall why we have the two invariants we have), it makes sense that we would like to feed it in some (G, Δ, T)-valid state. $f \equiv 0$ and $\ell \equiv 0$ is trivially a (G, Δ, T)-valid state and works as an input, but a really neat fact that you will see in section 4 below is that actually every (G, Δ, T)-valid state works as input! We will use this fact to get some interesting local and dynamic properties of push-relabel next lecture.

3.2 Algorithm

These two simple definitions will allow us to describe the algorithm. The role each definition plays will become clear by looking at the algorithm.

Definition 3.3 (Active vertex)**.** A vertex u is **active** if $\ell(u) < h$ and $\text{ex}(u) > 0$.

- In other words, a vertex is active if the algorithm hasn't given up on it ($\ell(u) < h$), and if there is still some excess that needs to be routed to a sink.

Definition 3.4 (Admissible arc)**.** An arc (u, v) in G_f is **admissible** if $r_f(u, v) > 0$ and $\ell(v) = \ell(u) - 1$.

- In other words, an arc is admissible if it has some capacity left over that's not being used, and is also a valid arc to send flow through, keeping in line with invariant 1.

Now, we will actually describe the algorithm, first through words and intuition, then we will provide the pseudo code.

Algorithm in words:

1. Find an **active** vertex u.
   - If there isn't that means that, either, we've given up on all of them ($\forall u \in V \ \ell(u) \geq h$), or all the excess has been successfully routed to the sinks ($\forall u \in V \ ex(u) = 0$) and we have in a (G, Δ, T)-valid solution. In either case, it makes sense to terminate.
   - Alternately, if there is some active node, that means we've got a node that has some excess to route, preventing us from having a (G, Δ, T)-valid solution, and we still haven't give up on it. So, we can continue onward in the algorithm to try to push the excess off.
2. If u has some **admissible** edge (u, v) attached to it, push as much excess as possible down that edge.
   - An edge (u, v) in G is admissible if the flow going through it hasn't yet maxed out it's capacity ($r_f(u, v) > 0$, and if it's attached to some node that's in the immediately smaller ℓ level below it.
   - Intuitively, you should think of ℓ as the height. So if a node is in ℓ level 3, it clearly can't send flow upwards to 4, and also, before it can send flow to ℓ level 2, the flow has to flow over ℓ level 3. Moreover, for the nodes v in level 2, because they are more than one level below u, by invariant 1, the edges (u, v) are have no capacity left anyways ($\ell(v) < \ell(u) - 1 \implies r_f(u, v) = 0$).
3. Else, **relabel** u by putting it in a higher level.
   - If there all the edges going from u to a lower level are full, then it makes sense to try the edge between u and the nodes in a higher level. To do this, we increase u's height.
4. Repeat!
   - This process can only terminate once there are no active edges (so in step 1).

Algorithm pseudocode:

```
BasicPushRelabel($G, \Delta, T, h, (f, \ell)$)
.   Assertion: $(f, \ell)$ is $(G, \Delta, T)$-valid.
.   While $\exists$ active vertex $u$ (i.e. $\ell(u) < h$ and $\text{ex}(u) > 0$)
.   .   Push/Relabel($u$).
```

```
Push/Relabel($u$)
.   If $\exists$ admissible arc $(u, v)$ (i.e. $r_f(u, v) > 0$, $\ell(u) = \ell(v) + 1$)
.   .   Push($u, v$).
.   Else Relabel($u$).
```

```
PUSH$(u, v)$
.   **Assertion**: $u$ is active and $(u, v)$ is admissible.
.   $\psi = \min\left(\text{ex}(u), r_f(u, v)\right)$
.   Send $\psi$ units of supply from $u$ to $v$: $f(u, v) \leftarrow f(u, v) + \psi$
```

```
RELABEL$(u)$
.   **Assertion**: $u$ is active and there is no admissible arc $(u, v)$
.   $\ell(u) \leftarrow \ell(u) + 1$.
```

3.3 Analysis

3.3.1 Input/Output

For the following lemmas, assume (f, ℓ) is initialized to a (G, Δ, T)-valid state at the beginning of the algorithm.

Lemma 3.1. *(f, ℓ) remains (G, Δ, T)-valid state throughout.*

Proof. To show that (f, ℓ) remains a (G, Δ, T)-valid state, it suffices to show that the two invariants are respected by every operation.

- Invariant 1: "$\forall u, v \in V \quad \ell(u) > \ell(v) + 1 \implies f(u, v) = c(u, v)$".

  We only change labels in RELABEL, so given a valid state, this invariant could only be broken in RELABEL. But if $f(u, v) < c(u, v)$ after calling RELABEL(u), then (u, v) was admissible before calling RELABEL(u), so PUSH would have been called instead of RELABEL, hence a contradiction. Therefore invariant 1 is maintained.

- Invariant 2: "if $\ell(u) \geq 1$, then $\text{ab}(u) = T(u)$":

  Because $\ell(u)$ increased from 0 to 1, that means that u must have been active at some point. This means that it must have had some non-zero excess ($\text{ex}(u) > 0$) at some point, but $\text{ex}(u) > 0$ only if u's sink was already full/fully absorbed ($\text{ab}(u) = T(u)$). Now because $\text{ab}(u)$ is never decreased, as we only PUSH excess out, $\text{ab}(u) = T(u)$ will be maintained throughout.

□

Lemma 3.2. *After termination, (f, ℓ) is a (G, Δ, T)-valid solution.*

Proof. Given a valid state, by the previous lemma we know that (f, ℓ) will remain a valid state at termination. We only terminate if there is no active vertex, so only when $\forall u \in V \ \ell(u) = h$ or $\text{ex}(u) = 0$, so if it terminates, we have the additional condition that makes (f, ℓ) a (G, Δ, T)-valid solution. □

Corollary 3.3. *Suppose after termination, $\forall u \in V \ \ell(u) < h$. Then f is a feasible flow satisfying (Δ, T).*

Proof. Note, from the above proof, that we only terminate when $\forall u \in V \ \ell(u) = h$ or $\text{ex}(u) = 0$, so if no node is at level h, then every node has zero excess. By proposition 2.2, we get then that f satisfies (Δ, T). Because we never PUSH flow that's greater than the capacity of an edge, $\forall (u, v) \in E$ $|f(u, v)| \leq c(u, v)$, so $\frac{|f(u,v)|}{c(u,v)} \leq 1$, giving us that f is also feasible.

□

These lemmas basically give us the confirmation that our algorithm 'makes sense'. Now, we need to work to see how long it takes.

3.3.2 Counting push and relabel calls

Note that $\ell(u)$ only increases, and only stops increasing one $\ell(u) = h$. This is an easy to see fact that will be relevant in bounding the time.

Lemma 3.4 (Relabel)**.** *The total number of* RELABEL *calls is at most nh.*

Proof. n vertices, and each is labeled at most h times. □

Bounding the number of PUSH calls is a bit more challenging. Note that whenever we call PUSH, we push either $\text{ex}(u)$ flow, or $r_f(u,v)$ flow- whichever is smaller. So, it makes sense to break bounding PUSH calls into two cases, those which "fill up edges" ($r_f(u,v) \leq \text{ex}(u)$), and those which don't ($\text{ex}(u) < r_f$):

Definition 3.5 (saturating PUSH)**.** This is a push where the edge (u,v) is "filled up". It happens only when $r_f(u,v) \leq \text{ex}(u)$ before the PUSH, and after the PUSH, $f(u,v) = c(u,v)$ (so $r_f(u,v) = 0$).

Definition 3.6 (unsaturating PUSH)**.** This is a push where the edge still has space; if a push isn't saturating, it is unsaturating. Any unsaturating PUSH(u,v) will push $\text{ex}(u)$ units of flow.

Now we can bound each of these types of PUSH calls.

Lemma 3.5 (saturating PUSH)**.** *The total number of saturating* PUSH *is at most mh.*

Proof. Given some edge (u,v), let's count the number of saturating pushes we can make on it. Suppose we make our first saturating PUSH(u,v). Note that $\ell(u) = \ell(v) + 1$.

Even though the edge is full right now, suppose at some future iteration, there is some reverse direction PUSH(v,u) that empties out the edge somewhat. Note that v can only push to u if $\ell(v) = \ell(u) + 1$, so $\ell(v)$ must have increased 2 levels. Similarly, for there to be another saturating PUSH(u,v) for us to count, $\ell(u)$ must also increase 2 levels.

By induction, there can be at most $\frac{h}{2}$ saturating PUSH(u,v). There are m edges, and for each edge, you can have a saturating push in both directions, there at most $2m \times (\frac{h}{2}) = mh$ saturating pushes. □

Lemma 3.6 (unsaturating PUSH)**.** *The total number of unsaturating* PUSH *is at most $O(mh^2)$.*

Proof. Consider the following potential function, which is summing up the levels of each of the nodes with excess:

$$\Lambda = \sum_{v:\text{ex}(v)>0} \ell(v).$$

Because every unsaturating PUSH(u,v) pushes exactly $\text{ex}(u)$ flow off from u, thereby making $\text{ex}(u) = 0$, PUSH(u,v) could remove $\ell(u)$ from the sum. However, PUSH(u,v) could also give

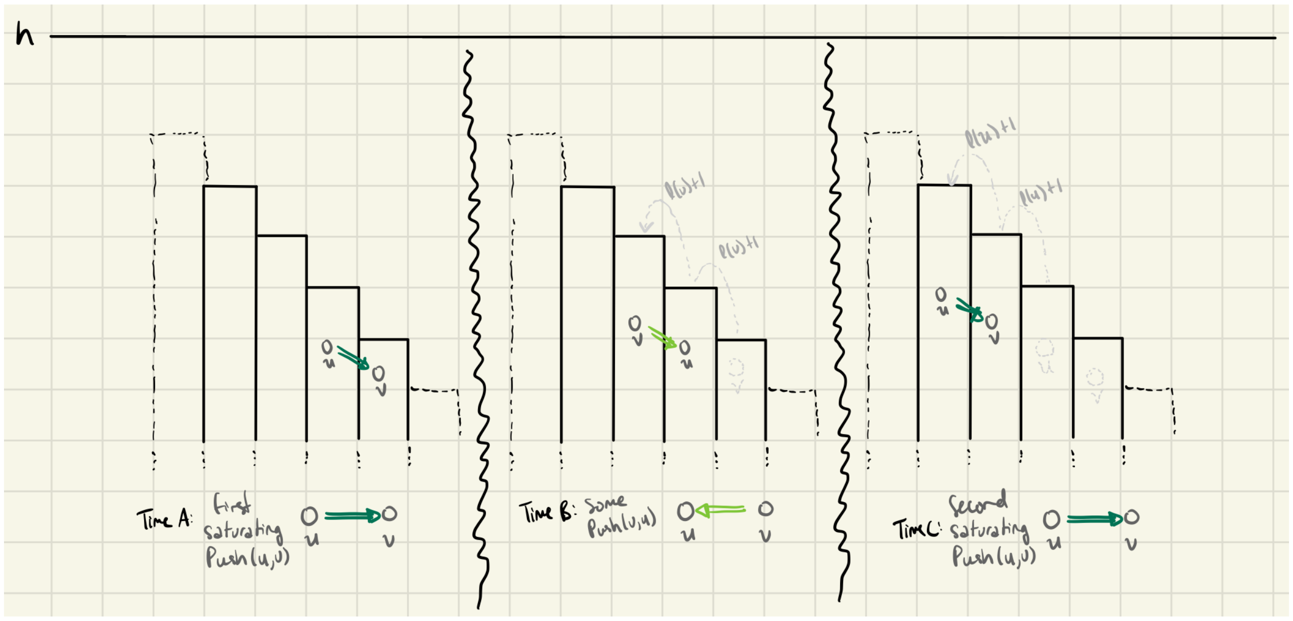

h
Time A: First saturating Push(u,v)
Time B: Some Push(v,u)
Time C: Second saturating Push(u,v)
u
v
ℓ(v)+1
ℓ(u)+1

Figure 4: Proof of saturating-pushes bound visualization.

v excess, and thereby add $\ell(v)$ to the sum. However, $\ell(u) = \ell(v) - 1$, so at the very least, every unsaturating PUSH(u, v) decreases Λ by 1.

Consider a saturating PUSH(u, v). This won't take away the excess at u, but could potentially give v excess; therefore, it can't decrease Λ, but could potentially add $\ell(v)$ to Λ. $\ell(v)$ could be at most $h - 1$, so each saturating PUSH(u, v) adds at most $h - 1$ to the sum. There can only be mh saturating pushes (by lemma 3.5), so in one full run of the algorithm, saturating PUSH calls can increase Λ by at most $mh(h - 1)$.

Each RELABEL call increases Λ by at most 1, and the total number of RELABEL calls is at most nh (by lemma 3.3), so in one full run of the algorithm, RELABEL calls can increase Λ by at most nh. Finally, note that Λ can start off as at most nh, because each node can be of level h or less.

Combining all of these, we get that the total number of unsaturating PUSH will be at most $O(mh(h-1) + 2nh) = O(mh^2)$ □

3.3.3 Basic Implementation

We've bounded the number of PUSH and RELABEL calls. To finish understanding the algorithm and its performance, we need to specify how to implement the remaining two pieces quickly: getting an active node, and getting an admissible edge. Recall that we look for active nodes at the start of each iteration of the while loop, and given an active node, we decide to PUSH or RELABEL depending on whether that node has an admissible edge or not.

Active node: This is the easy one. Maintaining a simple queue will give us an active node in

$O(1)$ time.

Admissible edge: Given an active node u, the trivial solution would be to look through each of the edges connected to it to find an admissible edge; so every time you wanted to find an admissible edge for u, it would take $O(deg(u))$ time to find an admissible edge and PUSH, or conclude there aren't any and RELABEL.

We can make this faster however. Note that if an edge (u, v) is not-admissible, it is guaranteed to remain not-admissible until $\ell(u)$ changes.* So, we can maintain an ordered list for each node u with each of it's edges, separated into two types: (1) the edges that have been confirmed as not-admissible since $\ell(u)$ was last updated, and (2) the edges that haven't yet been checked since $\ell(u)$ was last updated. So, if you have u as your active node and are looking for an admissible edge, just look through the nodes in (2), transferring them to (1) if you see they aren't admissible, until you find an admissible edge. In a later iteration, if you end up with u as your active node again and $\ell(u)$ hasn't been updated yet, you just need to look through the, now shorter, list of edges in (2), to find an admissible edge.

So before RELABEL(u) is called again, we only spend $O(\deg(u))$ time scanning through u's neighbors to check for admissible edges, so we can charge $O(\deg(u))$ time to each RELABEL(u). Therefore, the total time on RELABEL is $\sum_u O(\deg(u))h = O(mh)$.

*Note: In this context, u and v see the edge (u, v) differently. It's possible u sees (u, v) as not admissible and v sees it as admissible, because they can both only push flow away from themselves (u sends flow (u, v) and v sends flow (v, u)). Think of the residual graph understanding on invariant 1 on page 8.

3.4 Total Running Time

Each PUSH takes only $O(1)$ time, and there are $O(mh)$ saturating PUSH calls, and $O(mh^2)$ unsaturating PUSH calls. We know from the previous section that we can charge RELABEL with $O(mh)$ total time, so combining everything, we get that the total running time is $O(mh^2)$.

4 Applications

4.1 Max-Flow (h = $n + 1$)

4.1.1 Inputs/Outputs:

We are presenting the "decision version" of the (s,t)-max flow problem. Note that if we can solve this problem, we can solve the standard (s,t)-max flow by a simple binary search.

Inputs: G, Δ, T

Output: Return one of the following:

- YES: with a feasible flow f satisfying (Δ, T)

- NO: with the cut S where $\Delta(S) - T(S) > \delta(S)$
  (certifying that no feasible flow satisfying (Δ, T) exists, by exercise 2.2)

4.1.2 Algorithms and Analysis:

Algorithm: Simply call BASICPUSHRELABEL on (G, Δ, T) with parameter $h = n + 1$.

Analysis: BASICPUSHRELABEL can end in one of two states: (1) no vertex u with $\ell(u) = h$, so $\mathrm{ex}(V) = 0$, or (2) some vertex u with $\ell(u) = h$, so $\mathrm{ex}(V) > 0$.

Case (1): By corollary 3.3, we have f as a feasible flow satisfying (Δ, T).

Case (2): We need to find a cut S that witnesses the failure of f to be a feasible flow, so we will construct a cut S where $\Delta(S) - T(S) > \delta(S)$.

First, notice that, as $h = n + 1$, there will be some level k where there are no vertices, i.e. $L_k = \{u \mid \ell(u) = k\} = \emptyset$ (each node can only be in one level). We will argue below that the cut S as the level cut $L_{>k} = \{u \mid \ell(u) > k\}$ works. Recall:

$$\underbrace{\Delta(S)}_{\text{initial mass}} = \underbrace{f_{out}(S)}_{\text{net flow out}} + \underbrace{\mathrm{ab}_f(S)}_{\text{absorbed mass}} + \underbrace{\mathrm{ex}_f(S)}_{\text{excess}}$$

Now, note the following:

- $f_{out}(S) = \delta(S)$: This is because all the nodes in S are more than one level above all the nodes in $V \setminus S$, so the flow out of S to $V \setminus S$ will be exactly the weight of the edges between S and $V \setminus S$.
- $\mathrm{ab}_f(S) = T(S)$: All the nodes in S are above level 0, and must respect invariant 2, so they much all be fully absorbed.
- $\mathrm{ex}_f(S) > 0$: The excess for the whole graph is always pooled up in level h (at termination, there are no active nodes, so any node not at level h has no excess), but we $L_h \subseteq S$. Since the excess of the graph is nonzero (no feasible flow), $L_h > 0$, so $\mathrm{ex}_f(S) > 0$.

Plugging these in above, we get:

$$\Delta(S) > \delta(S) + T(S) \implies \Delta(S) - T(S) > \delta(S)$$

as desired.

4.2 Sparse Cut or Matching Embedding (h = $O(\frac{log(n)}{\phi})$)

4.2.1 Inputs/Outputs:

Inputs:

- $G = (V, E)$

- a cut (A, B) s.t. $vol(A) \leq vol(B)$
- a parameter ϕ

Outputs: Return one of the following:

- a cut S where $\Phi_G(S) < 1.1\phi$ (a sparse cut)
- a bipartite ($\deg_G$)-matching M of size vol(A) where $M \preccurlyeq^{\text{flow}} \frac{1}{\phi} \cdot G$
  - For all $a \in A$, $\deg_M(a) = \deg_G(a)$
  - For all $b \in B$, $\deg_M(b) \leq \deg_G(b)$

4.2.2 Algorithm and Analysis

Setting Δ and T: First, how to set Δ and T comes quite naturally from the matching we're trying to find. As we're trying to send deg(a) flow out for each node $a \in A$, and accept deg(b) flow in for each node $b \in B$, we set:

$$\Delta(u) = \begin{cases} \deg(u) & u \in A \\ 0 & u \notin A \end{cases} \text{ and } T(u) = \begin{cases} \deg(u) & u \in B \\ 0 & u \notin B \end{cases}$$

Scaling G: PUSH/RELABEL find flows, but only feasible flows. We're trying to embed M with congestion ϕ, so we can just scale down G to G_c, where $c(u, v) = 1/\phi$, and then tun the algorithm on G_c.

So, we can know that we will run PUSH/RELABEL on (G_c, Δ, T). What is left now is to set h. We will first go through an easy option, which gives us what we want but can be slow, and then will work through a more efficient, but somewhat more complicated example.

Option 1 (h = n + 1): Consider once more the two cases at the end of the algorithm, (1) where ex$(V) = 0$, and (2) where ex$(V) > 0$:

Case (1): Using the same reasoning from the analysis for Max-Flow (5.1.2), there exists a feasible flow f satisfying (Δ, T). f embeds M, and by flow-path decomposition, f corresponds to the (deg_G)-matching M that we want.

Case (2): Once more, using the reasoning for Max-Flow (5.1.2), we can easily find a cut S where $f_{out}(S) = \delta(S)$, so we get:

$$\begin{aligned} \frac{1}{\phi}\delta_G(S) = \delta_{G_c}(S) = f_{out}(S) &= \Delta(S) - \text{ab}(S) - \text{ex}(S) \\ &< \Delta(S) - T(S) \end{aligned}$$

Now, we just need to show that $\Delta(S) - T(S)$ is less that the volume of the cut (i.e. $\Delta(S) - T(S) \leq \min\{\text{vol}(S), \text{vol}(\bar{S})\}$), which will give us that $\frac{1}{\phi}\delta_G(S) < \Delta(S) - T(S) \leq \min\{\text{vol}(S), \text{vol}(\bar{S})\} \implies$

$\delta_G(S) \le \phi \min\{\text{vol}(S), \text{vol}(\bar{S})\}$ (i.e. that S is a ϕ-sparse cut).

The total source $\Delta(S)$ in S is at most $\text{vol}(S \cap A)$, which is clearly less than or equal to $\text{vol}(S)$, so $\Delta(S) \le \text{vol}(S)$. So, $\Delta(S) - T(S) \le \text{vol}(S)$. Simply working through definitions again shows that $\Delta(S) - T(S) \le \text{vol}(\bar{S})$ as well. This gives us that $\Delta(S) - T(S) \le \min\{\text{vol}(S), \text{vol}(\bar{S})\}$, and by the above, that S is a ϕ-sparse cut.

Option 2 (h = $O(\frac{\log n}{\phi})$): So, we know that option 1 works, but it is slow. We will show here that we can set h to be considerably smaller, and get almost the same result. The only caveat is that in the case of finding a sparse cut, it might not be quite a ϕ-sparse cut, but it will be close enough 1.1ϕ-sparse. Consider once more the two end cases, (1) where $\text{ex}(V) = 0$, and (2) where $\text{ex}(V) > 0$:

Case (1): Exactly as above, we have a feasible flow f satisfying (Δ, T).

Case (2): In the case of h = n + 1, once we found an S where the flow out was saturated in G_c, so where $f_{out}(S) = \delta_{G_c}(S) = \frac{1}{\phi}\delta_G(S)$ in G, we were done. But notice that we are also done if we can find an S where the net flow is big enough:

$$f_{out}(S) \ge \frac{1}{\phi}\delta_G(S) - O(\min\{\text{vol}(S), \text{vol}(\bar{S})\})$$

so almost saturated, with some slack. So, as we've done the last few times, we just need to prove there exists a level cut where this property holds. The following lemma gives this to us.

Lemma 4.1. *There is a level k where $L_{\ge k} = \{u \mid \ell(u) \ge k\}$ and*

$$\begin{aligned} f_{out}(L_{\ge k}) &\ge \delta_{G_c}(L_{\ge k}) - \frac{\phi}{10} \cdot \min\{\text{vol}_{G_c}(L_{\ge k}), \text{vol}_{G_c}(L_{<k})\} \\ &= \frac{1}{\phi}\delta(L_{\ge k}) - \frac{1}{10} \cdot \min\{\text{vol}(L_{\ge k}), \text{vol}(L_{<k})\} \end{aligned}$$

Proof. Consider the graph G_{con} (G 'consecutive'), where $G_{con} \subset G_c$ contains only the edges between consecutive levels $\bigcup_i E_{G_c}(L_i, L_{i-1})$. The key point to note is that there is a level $k \in [h - \frac{50 \log n}{\phi}, h]$ where $\delta_{G_{con}}(L_{\ge k}) < \frac{\phi}{20}\text{vol}_{G_{con}}(L_{\ge k})$.

To see this, note that $\text{vol}_{G_{con}}(L_{\ge k-1}) \ge \delta_{G_{con}}(L_{\ge k}) + \text{vol}_{G_{con}}(L_{\ge k})$, so if the claim is not true, then $\text{vol}_{G_{con}}(L_{\ge h-i}) \ge (1 + \frac{\phi}{20})^i \ge n^4$ when $i \ge \frac{50 \log n}{\phi}$, which would give us a contradiction. This is why we need $h \ge \frac{100 \log n}{\phi}$.

Using this argument as a ball growing argument from both sides, and using the k with the level cut with smaller cut size, we get that there is a k where:

$$\delta_{G_{con}}(L_{\ge k}) < \frac{\phi}{20} \cdot \min\{\text{vol}_{G_{con}}(L_{\ge k}), \text{vol}_{G_{con}}(L_{<k})\} \le \frac{\phi}{20} \cdot \min\{\text{vol}_{G_c}(L_{\ge k}), \text{vol}_{G_c}(L_{<k})\}.$$

Now because all the edges that skip levels are saturated, we get that:

$$f_{out}(L_{\ge k}) \ge \delta_{G_c}(L_{\ge k}) - 2\delta_{G_{con}}(L_{\ge k})$$

which in turn implies that:

$$f_{out}(L_{\geq k}) = \frac{1}{\phi}\delta(L_{\geq k}) - \frac{1}{10}\min\{\mathrm{vol}(L_{\geq k}), \mathrm{vol}(L_{<k})\}$$

as was desired. $\square$

4.3 Balanced Sparse Cut or Matching Embedding With Fake Edge

So we've now got a powerful way to use PUSH/RELABEL for finding a sparse cut, or a (deg_G)-matching embedding. Note that this is really just the step of a matching player in a cut-matching game. We will now show how to use PUSH/RELABEL for finding a balanced sparse cut, which can be used for finding balanced sparse cuts via the cut matching framework, and in turn, for computing expander decomposition.

Inputs/Outputs: **Inputs**:

- $G = (V, E)$
- a cut (A, B) s.t. $vol(A) \leq vol(B)$
- a parameter ϕ
- a parameter β

Outputs: Return one of the following:

- a cut S where $\Phi_G(S) < 2\phi$ and $\mathrm{vol}(S), \mathrm{vol}(\bar{S}) > \beta$ (a balanced sparse cut)
- a matching M' and a set of fake edge F where
  - $M = M' \cup F$ is a bipartite $(\deg_G)$-matching between A and$B$ of size $\mathrm{vol}(A)$
  - for all $a \in A$, $\deg_M(a) = \deg_G(a)$
    * for all $b \in B$, $\deg_M(b) \leq \deg_G(b)$
    * $|F| \leq \beta$
  - $M \preccurlyeq^{\text{flow}} \frac{1}{\phi} \cdot G$

Algorithm: We can use the same algorithm as last time (5.2.2), so calling BASICPUSHRELABEL on (G_c, Δ, T) with parameter $h = O(\frac{\log n}{\phi})$. The difference now, is that we will change what we do based on the threshold of $\mathrm{ex}(V)$ at the end, from 0 to β.

Case (1): Suppose $\mathrm{ex}(V) \leq \beta$. This means that you were close to embedding the matching, but there were β units of flow that doesn't go from source to sink. So, using the same argument as the last few times with corollary 3.3, via flow-path decomposition, f corresponds to the $(\deg_G)$-matching M' of size $\geq \mathrm{vol}(A) - \beta$.

Case (2): Suppose otherwise, that $\mathrm{ex}(V) > \beta$. The excess can only be on L_h, where the excess for each node in L_h can be at most the degree, so we have that $\mathrm{vol}(L_h) \geq \mathrm{ex}(V) > \beta$. Moreover, $S \supseteq L_h$, so $\mathrm{vol}(S) > \beta$. Similarly, we can show that $\mathrm{vol}(L_0) > \beta$, so $\mathrm{vol}(\bar{S}) > \beta$ as well.

5 Faster Implementation With Dynamic Trees

Recall from section 3.4 that the overall running time, using the basic implementation we discussed, is: $O(mh^2)$. The bottleneck here are the unsaturating PUSH calls, as everything else costs $O(mh)$.

Our goal here is to improve $O(mh^2)$ to $O(mh \log n)$, by making (almost) all pushes saturating, so only $O(mh)$ operations. With this improvement, our running time for max-flow would improve from $O(mn^2)$ to $O(mn \log n)$, and for finding a ϕ-sparse cut would improve from $O(m \log^2(n)/\phi^2)$ to $O(m \log^2(n)/\phi)$.

To implement this strategy, we will use **top tree** data structure.[2] We will discuss their use from a high level perspective, enough so that we can use it to improve PUSH/RELABEL.

5.1 Top Tree Data Structure

Suppose you have a collection of rooted trees $\mathcal{T} = \{T_1, \dots, T_k\}$, where each edge e is associated with value $\mathsf{val}(e)$. Morally speaking, the **top tree data structure** allows you to maintain 'anything' you'd like to dynamically on these trees. Most likely, it can handle anything you'd like to throw at it. Here are some example of the types of things it can handle, quite surprisingly, in $O(\log n)$ time:

Updates:

- link(u, v): add an edge (u, v) to combine two trees (as long as $\mathcal{T}$ remains sets of rooted tree)
- cut(u, v): delete edge (u, v) to cut a tree into two
- change-path(x, y, v): for each $e \in P_{xy}$ in the unique path from x to y, set $\mathsf{val}(e) \leftarrow \mathsf{val}(e) + v$.
- change-root(u): make u a root
- many many more..

Queries:

- root(u): return the root of $T \ni u$
- min-path(x, y): return $\min_{e \in P_{xy}} \mathsf{val}(e)$. (Can return max or sum too)
- min-subtree(x): return $\min_{e \in T_x} \mathsf{val}(e)$ where T_x is the subtree rooted at x. (Can return max or sum too)
- size-subtree(u): return size of T_u
- meet(x, y, z): find the unique "meeting vertex" of $P_{xy} \cap P_{yz} \cap P_{zx}$.
- many many more..

[2]Top trees go by many names in the literature, such as dynamic trees, link-cut trees, topological trees, etc. Read more about the detail of how they work here: https://arxiv.org/pdf/cs/0310065.pdf

A warning to keep in mind though is that, while great in theory, top trees seem to be a lot slower in practice.

Question 5.1 (Research Question)**.** *Is there some problem on trees that is easy in the static setting, but hard on the dynamic setting?*

5.2 Top Tree PUSH/RELABEL Implementation

5.2.1 Rephrased Invariants

Start off with imagining each vertex as the root of it's own tree in L_0, so $T_u = \{u\}$ for each node u. We want to maintain the following invariants:

1. Each vertex u is incident to at most one tree edge (u, v) where $\ell(v) = \ell(u) - 1$. In other words, every vertex can have only one parent in $\mathcal{T}$ in another level.
2. Only active vertices can be the root of a tree.

- The $\mathsf{val}(u, v)$ for each edge in $\mathcal{T}$ is equal to the residual capacity of the edge $r_f(u, v)$. This isn't quite an invariant, as it is can be implicitly maintained very quickly:
  - When we call link(u, v), we always set change-value$(u, v, r_f(u, v))$.
  - When we call cut(u, v), we always set $f(u, v) \leftarrow c(u, v) - \mathsf{val}(u, v)$.

  However, it's an important thing to keep in mind when understanding how this works.

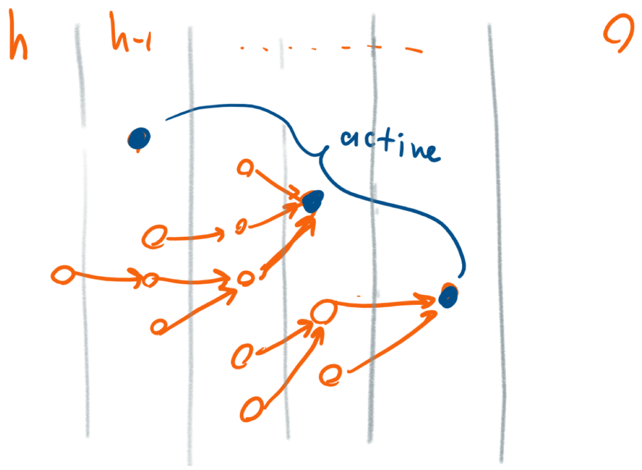


Figure 5: Top tree invariants visualization.

5.2.2 Updated Algorithm

Can start off with initializing $f \equiv 0$ and $\ell \equiv 0$, so every node is a root of its own tree.

```
TreePushRelabel($G, \Delta, T, h, (f, \ell)$)
.   Assertion: $(f, \ell)$ is $(G, \Delta, T)$-valid.
.   While $\exists$ active tree root vertex $u$
.   .   Push/Relabel($u$).
```

```
Push/Relabel($u$)
.   If $\exists$ admissible arc $(u, v)$
.   .   TreePush($u, v$).
.   Else Relabel($u$).
```

```
TreePush($u, v$)
.   Assertion: $u$ is active and $(u, v)$ is admissible.
.   link($u, v$)
.   While $u \neq$ root($u$) and ex($u$) $> 0$
.   .   $\psi = \min(\text{ex}(u), \text{min-path}(u, \text{root}(u)))$
.   .   Send $\psi$ units of supply from $u$ to root($u$): change-path($u$, root($u$), $-\psi$)
.   .   for all $e \in P_{u,\text{root}(u)}$ where val($e$) $= r_f(e) = 0$, perform cut($e$).
            //Exercise: implement this using $O(1)$ top-tree-operations per cut.
```

```
Relabel($u$)
.   Assertion: $u$ is active and there is no admissible arc $(u, v)$
.   for all tree-children $c$ of $u$, cut($c, u$).
.   $\ell(u) \leftarrow \ell(u) + 1$.
```

5.2.3 Analysis

We can use the same implementation details for getting an active vertex of an admissible edge quickly: a queue for active edges, and an ordered list for edges. The changes using top trees are with how we Push.

In TreePush, we can have lots of intermediate pushes until we get to the root. Lets call these intermediate push operations Push. We can define **saturating/unsaturating** Push as we did before: if $\psi = \text{min-path}(u, \text{root}(u))$, then this is a **saturating** Push, and otherwise, it is **unsaturating**.

The updated running time for Relabel and this saturating Push follow very directly from the analysis in 3.4, except we update for the $\log n$ operations with top trees. So:

- Relabel is called $O(nh)$ times, so total time is $O(mh \log n)$
  - We charge $O(\deg(u))$ to Relabel(u) for the cost of scanning through list of neighbors only once.
  - Also, for each Relabel, we call "cut" $O(\deg(u))$ tree operations, each of cost $O(\log n)$.
- There are $O(mh)$ saturating Push calls, so the total time is $O(mh \log n)$

It just remains now to bound the previous bottleneck, unsaturating PUSH calls. First, note that #unsaturating PUSH $\leq$ #links, but #links $\leq$ #cuts + $n - 1$. Also, note that #cuts $\leq$ #edges that become saturated + #edges scanned during RELABEL. But, because #saturating pushes $\leq mh$ and #edges scanned during RELABEL $\leq \sum_u \deg(u)h \leq 2mh$, we get that the total cost of unsaturating PUSH is $O(\#\text{link} \cdot \log n) = O(mh \log n)$.

Therefore, the overall running time is $O(mh \log n)$.