Optimization Problems on Network Flows with Degree Constraints

by

Yuchong Pan

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

BECHELOR OF SCIENCE

in

The Faculty of Science

(Combined Honours Computer Science and Mathematics)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2021

Abstract

In a vertex-capacitated directed graph with sources and sinks, we would like to concurrently route demands from the sources to the sinks. This model has many applications in the real world. However, conditions in the reality usually incur new side constraints to this general model. For instance, the *next-hop routing* in Internet protocol (IP) networks requires each router to have one *single* next destination, called the *next hop*, for each incoming packet destined to an IP address. A flow with this property is said to be *confluent*. If this constraint is relaxed to allow d next destinations at each vertex, then such a flow is said to be *d-furcated*. In general, side constraints concerning bounded out-degree of each vertex give rise to *network flows with degree constraints*. Such constraints contribute to simplicity of resulting network flow models.

Given a network flow model, several optimization problems can be asked. For instance, how the demands can be routed so that the flow does not exceed the capacity at each vertex too much? Another natural question is to find a subset of the demands with the maximum amount which can be routed subject to the vertex capacities. In this thesis, we survey algorithms that find approximate solutions close to optimum values within theoretically guaranteed factors.

Preface

This thesis is a survey of existing results in the literature for network flows with degree constraints, and therefore does not contain original research. Chapter 2 contain several classic results on maximum flow algorithms, the parametric push-relabel algorithm, and their applications in optimization problems on fractional flows. Chapter 3 is based upon the work of Chen et al. [2]. Chapter 4 is based upon the work of Donovan et al. [7]. Section 4.5 is from the unpublished journal version of [7].

Table of Contents

List of Figures

List of Algorithms and Procedures

Acknowledgements

In the past twenty-three years of my life, I have owed many thanks to people who have raised me, taught me, supported me, inspired me and enlightened me. I would not be the person I am today without them. Indeed, I never thought I would write a thesis in theoretical computer science and continue to pursue a doctoral degree in this intriguing area. The following people helped me achieve this.

- Bruce Shepherd

  Bruce is the reason I am here today. I remember first meeting Bruce in his combinatorial optimization course. I was not able to register for that course because of the university policy. Bruce did whatever he could to help me, although he eventually failed—he wrote me "Not for lack of trying but we finally have lost this 'battle.' " Bruce is always supportive in situations like this.

  Bruce is a superb advisor. He encouraged me to try various open questions, and was always there to help when I was stuck, providing me with endless ideas and guidance. I came to UBC knowing almost nothing about theoretical computer science, but Bruce witnessed me to become a rookie researcher. He was at every step of the journey to cheer for my success. I can feel how deeply Bruce cares about his students. I need to worry nothing with Bruce as my advisor, and I am forever grateful for working with Bruce.

- Nick Harvey

  Nick gave us bonus questions to attempt in each assignment of his complexity course, because of which he introduced me to Bruce. Nick also taught me how to be a good teacher and how to write nice notes. I am inspired by Nick's passion and modesty about what he does.

  Before I came to UBC, I was recommended many of Nick's graduate courses by Nick's former student Keyulu. I wish that I could have taken one of these courses as an undergrad at UBC.

- Hu Fu

  Hu always explains things with maximal clarity. Hu introduced me to many topics and techniques in theoretical computer science such as linear programming, semidefinite programming and approximation algorithms, which are central to my research. It is a pity that Hu's algorithmic game theory course got cancelled in my last undergraduate semester due to COVID-19.

- Algorithms Lab in UBC Computer Science

  Talks and the summer reading group organized by the Algorithms Lab allowed me to step out of my own research area, broaden my horizon in different field of theoretical computer science, and enlarge my research tool box. I would also like to thank the Algorithms Lab for financially supporting me to attend conferences.

- Josh Zahl and Malabika Pramanik

  Josh and Malabika introduced me to mathematics and taught me to think mathematically. They are the reason I decided to major in and work on mathematics.

- The competitive programming group at Shaoxing No. 1 High School, jointly and separately: Zhizhou Ren, Wentao Wang, Yuanzheng Tao, Zhaohui Ye, Yang Li and many others

  I learned most algorithms I know so far while I was participating in competitive programming in high school. People in the competitive programming group at Shaoxing No. 1 High School are not only my peers but also my teachers. We learned things together and taught each other. You gave me the first taste of theoretical computer science, and your persistence taught me what a good scholar should be like.

- Nadie Yiluo LiTenn

  Yiluo's passion about physics has constantly encouraged me to pursue research that I truly love—that's why I chose theoretical computer science as my research area. Yiluo is a great friend to whom I can always go when I was down. She gave me immense support by helping me analyze problems with rationality and humour.

  We share a common interest in aviation. Yiluo once told me that she would probably like to be a commercial pilot during a sabbatical after she became a professor. Surely you're joking, Yiluo!

- Xingyu Zhou, Grace Yin and Adam Jozefiak

  We worked on assignments and course projects together. I will never forget the nights we spent in study rooms pondering difficult questions. Indeed, I can hardly imagine how I would study mathematics and computer science without them. It is always joyful and rewarding to talk with Xingyu, Grace and Adam.

- Jerry Dong and Snow Wang

  We did many things together in our leisure time—skiing on Cypress Mountain, making dumplings, exploring restaurants, taking a trip to the Sunshine Coast, cooking New Year's Eve Dinner, etc. You made my undergraduate years colourful. You are like my family in Canada.

- Zhen Hu

  We encouraged each other when we were applying for PhD programs last year, and that is how we became familiar. Now you have become an indispensable part of my life. Our best days are yet to come. I look forward to holding your hands and exploring the boundary of human knowledge in our respective fields in the rest of our time.

- Hong Feng

  Mom, nothing is possible without your unconditional love, support and sacrifice. Mom is always positive, believing that I am able to do anything. Mom is not a mathematician, but she taught me to do mathematics intuitively when I was very young. Mom also taught me to make my own decisions, and backed every decision I made.

Thank you. You have all made this thesis possible, directly or indirectly. I also owe deep gratitude to many more people not on the list; there are too many people who have left a mark in my life and to whom my gratitude is beyond words.

Yuchong Pan
Vancouver, British Columbia
April 2021

Dedicated to my late great-grandfather.

"I want to be a scientist when I grow up."
— Me, responding to my great-grandfather

Chapter 1

Introduction

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth.

— Robert Frost, *The Road Not Taken*

Flows in networks are ubiquitous. They naturally model phenomena in the real world, such as transportation in a highway system, routing in a telecommunication network, currents in an electrical circuit, commodity delivery in a logistical task, etc. In the most general form, each of these problems can be abstracted into a graph of vertices and edges, with several ingredients:

- A set of *sources*, which is a subset of vertices, to send demands;
- A set of *sinks*, which is a subset of vertices, to receive demands;
- *Capacities* on edges or vertices, which indicate the amount of flow allowed to pass on each edge or through each vertex;
- *Costs* on edges, which indicate the unit cost of flow that passes on each edge or through each vertex;
- *Flows* on edges which satisfy *capacity constraints* (i.e., the flow on each edge or through each vertex must not exceed the capacity) and *conservation constraints* (i.e., the flow into a vertex must equal the flow out of the vertex).

In addition to their most natural applications, flows in networks are able to model and solve many seemingly unrelated problems. Examples are clustering, image segmentation, project selection, railway scheduling and machine assignment. General problems on network flows have been well studied. See, for instance, the Ford-Fulkerson algorithm for finding a maximum flow

[11] and the successive shortest path algorithm (which can be viewed as a generalization of the Ford-Fulkerson algorithm) for finding a minimum-cost flow [8]. We will review some of classical results for general network flow problems in this chapter.

The most direct and important applications of flows in networks are perhaps due to telecommunication networks. However, new technologies often impose additional constraints to flow networks. For instance, in Internet protocol (IP) routing, a router has the *next hop table* which contains an entry of the next hop for each IP address. If a router receives a packet for an IP address, it simply forwards the packet to the next router associated with the IP address stored in the table. Therefore, all flows of packets for the same IP address that pass through a vertex must exit on a single outgoing edge. A flow in a directed network with this additional constraint is said to be *confluent.* Other examples include *bifurcated* flows where a router is allowed to have d next hops for each IP address, and *unsplittable* flows where each source must send all of its demands along a single path, which is inspired by the SONET (synchronous optical networks) standard.

Kleinberg [17, 18] introduces the following related optimization problems on single-sink unsplittable flows. These questions can also be generalized to other classes of network flows with additional constraints.

- ***Minimum Congestion.*** What is the smallest $\alpha \geq 1$ such that if all (edge or vertex) capacities are multiplied by α, then a feasible flow of a certain class that satisfies all demands exists?

  We call this minimum value of α the *congestion* of the network, or the *congestion* of such a feasible flow.

- ***Maximum Routable Demands.*** Find a subset of the sources whose demands can be routed by a feasible flow of a certain class and which maximizes the routed demands.

- ***Minimum Number of Rounds.*** How many rounds are necessary to route all demands by a feasible flow of a certain class?

Each of the above problems is considered in a setting with no edge costs. Equivalently, we consider a situation where the costs are uniform, i.e., where all costs equal the same value. In addition to these problems, the problem of ***Minimum Congestion with Arbitrary Edge Costs***, a "constrained" variant of the well-studied minimum-cost flow problem, is often considered. Since this problem has two objective functions, i.e., cost and congestion, we use a *bicriteria approximation* (α, β) to measure a feasible solution to this problem, in which the congestion of the flow is at most α, and the cost of the flow is at most β times the optimum cost of a flow with no additional constraints.

In this thesis, we survey techniques and approximation algorithms that approximately solve the aforementioned optimization problems on network flows with degree constraints.

1.1 Definitions

Before delving into techniques and approximation algorithms on network flows, we provide mathematical definitions for network flows with degree constraints in the most general form, which we use throughout this thesis. In this thesis, we consider congestion minimization and demand maximization only. Therefore, we assume that edge costs are uniform (equivalently, the cost of each edge equals 1).

Let $G = (V, E)$ be a directed graph with vertex capacities $c : V \to \mathbb{R}_+$[1] and a set $T = \{t_1, \ldots, t_k\}$ of sinks. We denote $n = |V|$ and $m = |E|$, and use $\delta^+_G(v)$ and $\delta^-_G(v)$ to denote the set of outgoing and incoming edges at v in G, respectively. Let $d : V \to \mathbb{R}_+$ be demands on vertices. For each vertex v, we would like to route $d(v)$ units of flow from v to sinks. (Therefore, the set of *sources*, denoted as S, is $\{v \in V : d(v) > 0\}$.) Given a subset E' of E, we use $f(E') = \sum_{e \in E'} f(e)$ as a shorthand. A *flow* is a function $f : E \to \mathbb{R}_+$ that satisfies the *flow conservation equations*:

$$f\left(\delta^+_G(v)\right) - f\left(\delta^-_G(v)\right) = d(v), \qquad \forall v \in V \setminus T.$$

The *vertex congestion* of a vertex v, denoted by $f(v)$ with an abuse of notation, is defined to be $(f(\delta^-_G(v)) + d(v))/c(v)$. We say that a flow f is *feasible* if $f(\delta^-_G(v)) + d(v) \le c(e)$ for each vertex v. The *vertex congestion* of a flow f is defined to be the minimum vertex congestion of the vertices.

We can apply contraints on the maximum out-degree of the support network of a flow. We say that a flow is *d-furcated* if the out-degree of each vertex is at most d in the support network. In particular, the special cases of d-furcated network flows where $d \in \{1, 2, \infty\}$ are considered in this thesis:

- $d = \infty$ ***(Fractional flows).*** Each vertex can send flow *fractionally* to any number of outgoing edges.

[1]We add the subscript + to a set of real numbers to denote the subset of non-negative numbers in the set.

- $d = 1$ ***(Confluent flows).*** Each vertex can send flow to at most one outgoing edge. In other words, the support network of the flow forms vertex-disjoint in-arborescences rooted at the sinks.
- $d = 2$ ***(Bifurcated flows).*** Each vertex can send flow to at most two outgoing edges.

To simplify the problems in question, we assume the special case of unit vertex capacities (i.e., $c(v) = 1$ for each vertex v) throughout this thesis. Given this assumption, the *vertex congestion* of a vertex v is $f(\delta_G^-(v))+d(v)$, and a flow is *feasible* if $f(\delta_G^-(v)) + d(v) \leq 1$. The more general versions of many optimization problems with non-uniform capacities, however, are siginificantly more difficult to solve than their uniform-capacity counterparts and wide open to the best of our knowledge. For confluent flows, Chen et al. [2] write "an interesting direction for future research is to extend some or all of our results to graphs with heterogeneous vertex capacities." For d-furcated flows with $d \geq 2$, Donovan et al. [7] asks "what happens in networks with non-uniform capacities or costs?" The difficulty of non-uniform capacities for congestion minimization, according to Chen et al. [2], stems from the fact that the most congested vertex in a flow is no longer necessarily one of the sinks.

Without loss of generality, we assume that a vertex with no incoming edges must have a positive demand; otherwise, flows on its outgoing edges must be zero, and these outgoing edges can be removed from the graph. Moreover, we assume that sinks are exactly vertices with no outgoing edges. If there is a non-sink vertex v with no outgoing edges, then we must have $f(\delta_G^-(v)) = 0$ because $f(\delta_G^+(v)) = 0$, $f(\delta_G^-(v)) \geq 0$ and $d(v) \geq 0$ due to the flow conservation equations. Hence, we can remove v and all of its incoming edges in this case. On the other hand, if there is a sink v with outgoing edges, then we add a vertex w to V and to T, add an edge (v, w) to E, and remove v from T.

1.2 Thesis Organization

Chapter 2 describes several classical algorithms on fractional flows; a great portion of Chapter 2 is devoted to an application of the parametric push-relabel algorithm of Gallo et al. [12] for finding a congestion-minimizing fractional flow. The approximation algorithms in Chapter 3 and Chapter 4 assume a congestion-minimizing fractional flow as their starting point.

Chapter 3 studies congestion minimization and demand maximization on confluent flows. Moreover, we give an instance which shows an H_k lower bound for congestion minimization on confluent flows, where H_k is the k^{th} harmonic number.

Chapter 4 studies congestion minimization on d-furcated flows where $d \geq 2$. It is also shown that the technique presented in Chapter 4 can be applied to solve congestion minimization on β-confluent flows.

Chapter 2, Chapter 3 and Chapter 4 are completely disjoint. A reader who would like to learn only techniques and approximation algorithms for network flows with degree constraints can skip Chapter 2.

Chapter 2

Fractional Flows

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to," said the Cat.

"I don't much care where–" said Alice.

"Then it doesn't matter which way you go," said the Cat.

"–so long as I get somewhere," Alice added as an explanation.

"Oh, you're sure to do that," said the Cat, "if you only walk long enough."

— Lewis Carroll, *Alice's Adventures in Wonderland*

Every approximation algorithm in the literature for network flows with side constraints, to the best of our knowledge, rounds a fractional network flow to a flow with certain side constraints. In this chapter, we examine several classic algorithms and reductions to them for solving optimization problems on *fractional* network flows, i.e., network flows *without* degree constraints.

2.1 The Ford-Fulkerson Algorithm

As mentioned in Chapter 1, one of the basic optimization problems on network flows is ***Maximum Routable Demand***: Find a subset of the sources whose demands can be routed by a feasible flow and which maximizes the routed demand. We say that this problem asks for an *all-or-nothing* network flow in the sense that such a flow routes either all demand or zero demand for each source. In the context of fractional network flows, we loosen this requirement by allowing a fractional portion of the demand of a source to be routed (alternatively, our reduction below to the Ford-Fulkerson algorithm also works if the demands of all sources equal to 1 or the same value). The special case where there is exactly one source and one sink is the celebrated *(single-source single-sink) maximum flow* problem:

> Given a directed graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, and edge capacities $c : E \to \mathbb{R}_+$, what is a network flow f with the maximum *value* $|f|$, i.e., the amount of flow that goes out of the source s?

Ford and Fulkerson [11] give a pseudo-polynomial-time greedy algorithm, called the *Ford-Fulkerson algorithm*, which solves the maximum flow problem. The running time of the Ford-Fulkerson algorithm with integral capacities is $O(Cm)$, where C is the sum of the edge capacities out of the source s. The basic idea of the Ford-Fulkerson algorithm is to successively augment along an s-t path in a *residual graph* along which each of the edges has available capacity. Such s-t paths are called *augmenting paths*. There also exist strongly-polynomial-time variations of the Ford-Fulkerson based upon different ways to find augmenting paths. For instance, the Edmonds-Karp algorithm uses breadth-first search to find a shortest augmenting path each time, achieving a running time of $O(m^2 n)$ [8]. Dinitz's algorithm is similar to the Edmonds-Karp algorithm but uses additional techniques, including the *level graph* and *blocking flows*, to achieve a better running time of $O(mn^2)$ [5]. This running time of Dinitz's algorithm can be further improved to $O(mn \log n)$ by utilizing data structures such as dynamic trees. In this thesis, we would like to focus on high-level algorithmic ideas instead of delving too much into details of specific implementations or data structures. Therefore, we present below the most general Ford-Fulkerson algorithm only.

Two key ingredients of the Ford-Fulkerson algorithm are *residual graphs* and *augmenting paths*. Let $G = (V, E)$ be a directed graph with edge capacities $c : E \to \mathbb{R}_+$. Let $s, t \in V$ be the source and the sink, respectively. Given a flow $f : E \to \mathbb{R}_+$, we define the *residual graph* G_f (with edge capacities) by the following:

- the vertex set of G_f is V;
- each $e \in E$ with $f(e) < c(e)$ is also in G_f with capacity $c(e) - f(e)$, called a *forward edge*;
- for each $(u, v) \in E$ with $f(e) > 0$, there exists an edge (v, u) in G_f with capacity $f(e)$, called a *backward edge*.

An s-t path[2] in the residual graph G_f is called an *augmenting path*. The minimum capacity of an edge along an augmenting path in the residual graph G_f

[2]The definition of paths is ambiguous in the literature. Throughout this thesis, a *walk* is a sequence of vertices $v_1, \ldots, v_\ell$ such that (v_i, v_{i+1}) is in the edge set for each $i \in [\ell - 1]$, and a *path* is a walk which does not repeat vertices.

is called the *bottleneck* of the augmenting path, denoted by bottleneck(P, f). Moreover, we define the operation `Augment`(P, f) of *augmenting along an augmenting path P in G_f* in Procedure 2.1. The Ford-Fulkerson algorithm, given in 2.2, repeatedly searches for an augmenting path P in the residual graph G_f, and augments along P.

```
1 $b \leftarrow$ bottleneck$(P, f)$
2 foreach forward edge $e$ along $P$ do
3     $f(e) \leftarrow f(e) + b$
4 foreach backward edge $(u, v)$ along $P$ do
5     $f(v, u) \leftarrow f(v, u) - b$
```

Procedure 2.1: `Augment`(P, f).

```
1 $f(e) \leftarrow 0$ for all $e \in E$
2 repeat
3     search for an augmenting path in $G_f$
4     if an augmenting path $P$ exists then
5         Augment$(P, f)$
6     else
7         return $f$
```

Algorithm 2.2: The Ford-Fulkerson algorithm.

The correctness of the Ford-Fulkerson algorithm follows from the following two straightforward lemmas about the operation `Augment`(P, f).

Lemma 2.1. *Let $f : E \to \mathbb{R}_+$ be a flow in a flow network (G, c). Then the augmentation along an augmenting path P in G_f results in a flow f' with value $|f| + b$, where b is the bottleneck along P with respect to f.*

Lemma 2.2. *Let $f : E \to \mathbb{R}_+$ be a flow in the flow network. If there is no augmenting path in G_f, then f is a maximum flow.*

In addition, we give the running time of $O(Cm)$ with integral edge capacities claimed above.

Lemma 2.3. *Suppose that all edge capacities are integers. Let*

$$C = \sum_{e \in \delta^+_G(s)} c(e).$$

The Ford-Fulkerson algorithm terminates in at most C rounds. The running time of the Ford-Fulkerson algorithm is hence $O(Cm)$.

On the other hand, the case of multiple sources and multiple sinks can be reduced to the single-source single-sink case. We add a *super source* s^* and a *super sink* t^*. For each source v in the original graph, we connect an edge from s^* to v with edge capacity $d(v)$. For each sink v in the original graph, we connect an edge from v to t^* with edge capacity ∞. It can be shown that the single-source single-sink maximum flow problem in the new graph gives the maximum demand in the original network.

A *cut* is a 2-part partition $(X, \overline{X})$ of the vertex set V such that $s \in X$ and that $t \in \overline{X}$. The *capacity* of a cut $(X, \overline{X})$ is defined to be

$$c\left(X, \overline{X}\right) = \sum_{\substack{(v,w) \in E \\ v \in X, w \in \overline{X}}} c(v, w).$$

Finally, the celebrated maximum-flow minimum-cut theorem of Ford and Fulkerson [11] states the following:

Theorem 2.4 (Maximum-Flow Minimum-Cut Theorem, [11]). *The maximum flow value equals the minimum cut capacity in a flow network.*

Moreover, the proof of the maximum-flow minimum-cut theorem in [11] shows that $(X, \overline{X})$ is a minimum cut in G, where X is the set of vertices reachable from s in the residual graph G_f. In particular, we use this result to compute a minimum cut in Section 2.3.

2.2 The Push-Relabel Algorithm

The Ford-Fulkerson algorithm and its variants are called *augmenting path algorithms* for the maximum flow problem because of their `Augment`(P, f) operation. In an augmenting path algorithm, we maintain a feasible flow at any point of the algorithm, and aim to disconnect the source and the sink in the residual graph. In this section, we introduce a second paradigm called the *push-relabel algorithm*, due to Goldberg and Tarjan [14], for computing the maximum flow problem. In contrast to the Ford-Fulkerson algorithm, this new paradigm maintains that the source and the sink are disconnected in the residual graph, and aim to construct a feasible flow as the algorithm progresses. We include the push-relabel algorithm in this chapter because we would eventually like to reduce an algorithm for finding a congestion-minimizing fractional flow to the push-relabel algorithm in Section 2.6.

In addition to residual graphs introduced in Section 2.1, the push-relabel algorithm has an additional key ingredient—the notion of a *preflow.* A *preflow* is a function $f : E \to \mathbb{R}_+$ that satisfies the *capacity constraints*:

$$f(e) \leq c(e) \qquad\qquad \forall e \in E,$$

and the *relaxed conversation constraints*:

$$f\left(\delta_G^-(v)\right) \geq f\left(\delta_G^+(v)\right), \qquad\qquad \forall V \setminus \{s\}.$$

Note that the demand function $d : V \to \mathbb{R}_+$ is not present in the single-source single-sink maximum flow problem. In other words, a preflow is a feasible flow except that the flow conservation constraints are relaxed so that the amount of flow entering a vertex $v \in V \setminus \{s\}$ is at least the amount of flow leaving v. Given a preflow $f : E \to \mathbb{R}_+$, we define the *excess* of a vertex $v \in V \setminus \{s, t\}$ to be

$$\varepsilon_f(v) = f\left(\delta_G^-(v)\right) - f\left(\delta_G^+(v)\right).$$

Note that a preflow is a feasible flow if and only if each vertex $v \in V \setminus \{s, t\}$ has a zero excess. The general idea of the push-relabel algorithm is to gradually transform a preflow towards a feasible flow.

In the augmenting path paradigm, since we maintain a feasible flow throughout the algorithm, we are restricted to augmentations along an entire *s*-*t* path in the residual graph in order to preserve the conservation constraints. In the push-relabel paradigm, however, we are granted more flexibility by the relaxed conservation constraints of preflows. Therefore, we define the operation `Push`(v, f) of *augmenting along an outgoing edge from a vertex v in G_f* in Procedure 2.3. If $b = \varepsilon_f(v)$ in `Push`(v, f), then the push is *saturating*; otherwise, the push is *non-saturating.*

```
1 choose an outgoing edge $e$ from $v$ in $G_f$
2 $b \leftarrow \min\{\varepsilon_f(v), \text{capacity of } e \text{ in } G_f\}$
3 if $e$ is forward then
4     $f(e) \leftarrow f(e) + b$
5 else
6     $f(e) \leftarrow f(e) - b$
```

Procedure 2.3: `Push`(v, f).

The `Push`(v, f) operation is not sufficient, as it is possible that the algorithm pushes flow along a cycle indefinitely. To ensure that the algorithm

1. $h(s) = n$.
2. $h(t) = 0$.
3. For each edge (v, w) in the residual graph, $h(v) \leq h(w) + 1$.

Figure 2.1: The three invariants maintained in the push-relabel algorithm.

terminates, we define the *height* function $h : V \to \mathbb{Z}_+$ with the invariants outlined in Figure 2.1 to be maintained throughout the algorithm.

Recall from the opening paragraph of this section that, in contrast to the augmenting path paradigm, we would like the source and the sink to be *disconnected* in the residual graph, and gradually construct a feasible flow throughout the algorithm. Intuitively, the height decreases by at most one along an edge in the residual graph. Therefore, there is no s-t path in the residual graph. By Lemma 2.2, it follows that if a preflow $f : E \to \mathbb{R}_+$ is a feasible flow, then f is a maximum flow. Accordingly, we modify the `Push`(v, f) operation in Procedure 2.4 so that we only push flow "downhill," i.e., from a vertex v to another vertex w with $h(v) = h(w) + 1$. The push-relabel algorithm, given in Algorithm 2.5, repeatedly choose the higheset vertex v with a positive excess, and either pushes flow from v (if possible) or "relabel" $h(v)$ to make a push from v possible. The initialization from the first four lines of Algorithm 2.5 ensures that the three invariants outlined in Figure 2.1 are satisfied initially.

```
choose an outgoing edge $(v, w)$ from $v$ in $G_f$ with $h(v) = h(w) + 1$
$b \leftarrow \min\{\varepsilon_f(v, w), \text{capacity of } (v, w) \text{ in } G_f\}$
**if** $(v, w)$ *is forward* **then**
    $f(v, w) \leftarrow f(v, w) + b$
**else**
    $f(v, w) \leftarrow f(v, w) - b$
```

Procedure 2.4: `Push`(v, f), modified.

It is straightforward to verify that the three invariants outlined in Figure 2.1 are maintained throughout the push-relabel algorithm. By the discussion in the previous paragraph, if this algorithm terminates, then the flow f is a maximum flow. This gives the following two lemmas.

Lemma 2.5. *The three invariants outlined in Figure 2.1 are maintained*

```
h(s) ← n
h(v) ← 0 for all v ∈ V \ {s}
f(e) ← c(e) for all e ∈ δ^+_G(s)
f(e) ← 0 for all e ∈ E \ δ^+_G(s)
while ∃v ∈ V \ {s, t} with ε_f(v) > 0 do
    choose such v with the maximum h(v)
    if ∃(v, w) ∈ δ_{G_f}(v) with h(v) = h(w) + 1 then
        Push(v, f)
    else
        h(v) ← 1 + min{d(w) : (v, w) ∈ δ_{G_f}(v)} // Relabel(v, f)
```

Algorithm 2.5: The push-relabel algorithm.

throughout Algorithm 2.5.

Lemma 2.6. *If Algorithm 2.5 terminates, then f is a maximum flow.*

It remains to show that the algorithm does terminate, and that the running time of the algorithm is $O(n^3)$. We give this result as the following theorem.

Theorem 2.7. *Algorithm 2.5 terminates after $O(n^2)$ `Relabel`(v, f) operations and $O(n^3)$ `Push`(v, f) operations.*

2.3 A Parametric Maximum Flow Algorithm

In this section, we present an algorithm due to Gallo et al. [12] that naturally extends the push-relabel algorithm to solve the *parametric maximum flow problem.* Later, we shall apply this algorithm to solve the *perfect sharing problem* in Section 2.5 and eventually give an algorithm for finding a congestion-minimizing fractional flow in Section 2.6.

In the parametric maximum flow problem, each edge capacity $c_\lambda(e)$ is a function of a real-valued parameter λ that satisfies the following properties:

- $c_\lambda(e)$ is non-decreasing in λ for all $e = (s, v) \in \delta^+_G(s)$ with $v \neq t$;
- $c_\lambda(e)$ is non-increasing in λ for all $e = (v, t) \in \delta^-_G(t)$ with $v \neq s$;
- $c_\lambda(e)$ is constant for all $e = (v, w) \in E$ with $v \neq s, w \neq t$;
- $c_\lambda(e)$ can be computed in constant time for all $e \in E$.

When the context is clear, a *maximum flow* in the parametric maximum flow problem means a maximum flow in the network with a particular value of the parameter λ. Moreover, the problem receives an increasing sequence of parameter values $\lambda_1 < \ldots < \lambda_\ell$ and asks for a maximum flow for each parameter value in the *online* fashion, meaning that a maximum flow for λ_i is computed with no information about λ_j with $j > i$.

Note that if the value of the parameter λ is increased, then the capacities of outgoing edges from s are increased and those of incoming edges to t are decreased according to the previously outlined properties. The main idea of this parametric maximum flow algorithm rests upon the fact that f remains a preflow with the three invariants outlined in Figure 2.1 still satisfied if we set $f(e)$ by $\min\{f(e), c_\lambda(e)\}$ for all $e \in \delta_G^-(t)$ and set $f(e)$ be $c_\lambda(e)$ for all $e = (s, v) \in \delta_G^+(s)$ with $h(v) < n$. Therefore, we can again apply the push-relabel algorithm to compute a maximum flow.

Moreover, we use the remark in the last paragraph of Section 2.1 to compute a corresponding minimum cut $(X, \overline{X})$ for each parameter value λ, where X is the set of vertices reachable from s in the residual graph G_f. The minimum cuts $(X_i, \overline{X_i})$ for the parameter values λ_i and their properties are essential in Section 2.4. For all $v, w \in V$, define $d_f(v, w)$ to be the minimum number of edges from v to w in the residual graph G_f, or ∞ if there exists no v-w path in G_f. We claim that this minimum cut $(X, \overline{X})$ can be computed by `ComputeMinCut`(G, f) defined in Procedure 2.6, and that `ComputeMinCut`(G, f) does not decrease the heights of vertices. We note that $d_f(v, s)$ and $d_f(v, t)$ for all $v \in V$ can be obtained by two breadth-first searches in $O(m)$ time on the reverse graph of G_f from s and t, respectively.

```
1 $h(v) \leftarrow \min\{d_f(v,s) + n, d_f(v,t)\}$ for all $v \in V$
2 $X \leftarrow \{v \in V : h(v) \geq n\}$
3 return $(X, \overline{X})$
```

Procedure 2.6: `ComputeMinCut`(G, f).

Lemma 2.8. *If $f : E \to \mathbb{R}_+$ is a maximum flow on $G = (V, E)$, then* `ComputeMinCut`(G, f) *returns a minimum cut.*

Lemma 2.9. `ComputeMinCut`(G, f) *does not decrease $h(v)$ for each $v \in V$.*

Furthermore, Theorem 5.5 of Ford and Fulkerson [10] implies the following lemma, which we shall use below.

Lemma 2.10. *Let $(Y, \overline{Y})$ be any minimum cut. Let $(X, \overline{X})$ be the minimum cut computed in* `ComputeMinCut`(G, f)*. Then $Y \subseteq X$.*

The discussion in the previous paragraphs leads to the *parametric push-relabel algorithm*, given in Algorithm 2.7, which solves the parametric maximum flow problem and which also produces a corresponding minimum cut $(X_i, \overline{X_i})$ for each parameter value λ_i.

The correctness of Algorithm 2.7 is straightforward and follows from that of the push-relabel algorithm. The running time of the naïve implementation of Algorithm 2.7 is $O(n^2(\ell + m))$. Moreover, Gallo et al. [12] note that the running time of Algorithm 2.7 can be improved to $O(n^3 + \ell n^2)$ using a queue and further to $O((n + \ell)m \log \frac{n^2}{m})$ using the dynamic tree data structures [26, 27].

```
1 $h(s) \leftarrow n$
2 $h(v) \leftarrow 0$ for all $v \in V \setminus \{s\}$
3 $f(e) \leftarrow 0$ for all $e \in E$
4 for $i \leftarrow 1, \ldots, \ell$ do
5     $f(e) \leftarrow \min\{f(e)c_{\lambda_i}(e)\}$ for all $e \in \delta_G^-(t)$
6     $f(e) \leftarrow c_{\lambda_i}(e)$ for all $e = (s, v) \in \delta_G^+(s)$ with $d(v) < n$
7     run the push-relabel algorithm with the current $f$ and $h$
8     $(X_i, \overline{X_i}) \leftarrow$ ComputeMinCut$(G, f)$
```

Algorithm 2.7: The parametric push-relabel algorithm.

Finally, we note that the minimum cuts $(X_i, \overline{X_i})$ computed in Algorithm 2.7 satisfy the *nesting property*, which is stated in the following lemma. Indeed, this property has been discovered in several application scenarios, including record segmentation in shared databases [9] and critical load factors in two-processor distributed systems [28]. Algorithm 2.7 gives a direct proof of this property.

Lemma 2.11 (Nesting Property)**.** *Given an increasing sequence of parameter values $\lambda_1 < \ldots < \lambda_\ell$, Algorithm 2.7 produces a sequence of minimum cuts $(X_1, \overline{X_1}), \ldots (X_\ell, \overline{X_\ell})$ such that $X_1 \subseteq \ldots \subseteq X_\ell$.*

Proof. Let $v \in V$. By Lemma 2.9, `ComputeMinCut`(G, f) does not decrease $h(v)$. Note also that `Relabel`(v, f) in the push-relabel algorithm does not decrease $h(v)$. Therefore, $h(v)$ does not decrease during the execution of Algorithm 2.7 for all $v \in V$. Since `ComputeMinCut`(G, f) generates a minimum cut $(X, \overline{X})$ where $X = \{v \in V : h(v) \geq n\}$, then $X_1 \subseteq \ldots \subseteq X_\ell$. □

Goldberg and Tarjan [13], Goldberg and Tarjan [14] and Gallo et al. [12] note that a maximum flow need not be computed in order to obtain a minimum cut only. This variant of the push-relabel algorithm, called the *minimum-cut parametric algorithm*, modifies the original push-relabel algorithm by changing the condition of the main **while** loop in Algorithm 2.5 to "$\exists v \in V\backslash\{s,t\}$ with $\varepsilon_f(v) > 0$ and $h(v) < n$" and produces a minimum cut $(X, \overline{X})$ and a preflow f with $\varepsilon_f(t) = c(X, \overline{X})$ upon termination. We shall use this minimum-cut parametric algorithm to prove the following structural lemma in a parametric flow network, which is a strengthening of Lemma 2.11:

Lemma 2.12. *Let $(X, \overline{X}), (Y, \overline{Y})$ be minimum cuts for parameter values λ_1, λ_2 with $\lambda_1 \leq \lambda_2$, respectively. Then $(X \cap Y, \overline{X} \cup \overline{Y})$ and $(X \cup Y, \overline{X} \cap \overline{Y})$ are minimum cuts for parameter values λ_1, λ_2, respectively.*

Proof. We run the minimum-cut parametric algorithm for parameter values λ_1, λ_2. Lemma 2.10 implies that $h(v) \geq n$ for all $v \in X$ after computing a minimum cut for λ_1. Since $(X, \overline{X})$ is a minimum cut for λ_1, then $f(e) = c_{\lambda_1}(e)$ for each edge e across $(X, \overline{X})$. The new **while** loop condition of the minimum-cut parametric algorithm implies that $f(e) = c_{\lambda_2}(e)$ for each edge e across $(X, \overline{X})$ after computing a minimum cut for λ_2. Since $(Y, \overline{Y})$ is a minimum cut for λ_2, then $f(e) = c_{\lambda_2}(e)$ for each edge e across $(Y, \overline{Y})$. Furthermore, since $\varepsilon_f(t) = c_{\lambda_2}(X, \overline{X})$, then $\varepsilon_f(v) = 0$ for all $v \in \overline{Y} \setminus \{t\}$.

Therefore, each edge e across the cut $(X \cap Y, \overline{X} \cup \overline{Y})$ satisfies $f(e) = c_{\lambda_2}(e)$. Moreover, each $v \in \overline{X} \cup \overline{Y} \setminus \{t\} \subseteq \overline{Y} \setminus \{t\}$ satisfies $\varepsilon_f(v) = 0$. This implies that the capacity of $(X \cap Y, \overline{X} \cup \overline{Y})$ equals $\varepsilon_f(t)$ and hence the minimum cut capacity. Hence, $(X \cup Y, \overline{X} \cap \overline{Y})$ is a minimum cut for λ_2.

We note that the reverse graph G^r of G with source t and sink s, given parameter values λ_2 and then λ_1, satisfies the three invariants outlined in Figure 2.1. Applying the minimum-cut parametric algorithm on G^r with parameter values λ_2, λ_1 proves that $(X \cap Y, \overline{X} \cup \overline{Y})$ is a minimum cut for λ_1. □

Lemma 2.12 directly implies the following "intermediate-value" corollary, which plays an essential role in proving the concavity of the minimum-cut capacity function in Section 2.4.

Corollary 2.13. *Let $(X, \overline{X}), (Y, \overline{Y})$ be minimum cuts for parameter values λ_1, λ_2 with $X \subseteq Y$ and $\lambda_1 \leq \lambda_2$. For all $\lambda_3 \in [\lambda_1, \lambda_2]$, there exists a minimum cut $(Z, \overline{Z})$ for λ_3 such that $X \subseteq Z \subseteq Y$.*

Proof. Let $(Z', \overline{Z'})$ be a minimum cut for λ_3. Let $Z = (Z' \cup X) \cap Y$. Then $X \subseteq Z \subseteq Y$. Applying Lemma 2.12 on Z' and X shows that $(Z' \cup X, \overline{Z' \cup X})$ is a minimum cut for λ_3. Applying Lemma 2.12 again on $Z' \cup X$ and Y shows that $(Z, \overline{Z})$ is a minimum cut for λ_3. □

2.4 The Minimum-Cut Capacity Function

In this section, we analyze the properties of the *minimum-cut capacity function* $\kappa(\lambda)$ and discuss algorithms to compute part or all of information about the function. We shall use one of these algorithms to solve the perfect sharing problem in Section 2.5. In a parametric flow network, we define the *minimum-cut capacity function* $\kappa(\lambda)$ to be the minimum-cut capacity for the parameter value λ. In addition to the four properties outlined in Section 2.3, we assume that the edge capacity functions are linear functions throughout this section; i.e., $c_\lambda(s,v) = a_0(v) + \lambda a_1(v)$ for all $(s,v) \in \delta^+_G(s)$ and $c_\lambda(v,t) = b_0(v) - \lambda b_1(v)$ for all $(v,t) \in \delta^-_G(s)$, where $a_0(v), b_0(v), a_1(v), b_1(v)$ are constant coefficients and $a_0(v), b_0(v)$ are non-negative for each v. We shall first prove the following lemma, which describes the capacity of a cut $(X, \overline{X})$ in a parametric network.

Lemma 2.14. *If $(X, \overline{X})$ is a minimum cut for some parameter value λ_0, then the capacity of $(X, \overline{X})$ is a linear function in the parameter value λ. In particular,*

$$c_\lambda(X, \overline{X}) = \alpha + \lambda\beta,$$

where

$$\beta = \sum_{(s,v)\in E, v\in\overline{X}} a_1(v) - \sum_{(v,t)\in E, v\in X} b_1(v), \qquad \alpha = c_{\lambda_0}(X, \overline{X}) - \lambda_0\beta.$$

Proof. Let $(X, \overline{X})$ be a minimum cut for some parameter value λ_0. Since $c_\lambda(v,w)$ is constant for all $(v,w) \in E$ with $v \neq s$ and $w \neq t$, then for an arbitrary parameter value λ, we have that

$$\begin{aligned}
c_\lambda\left(X,\overline{X}\right) - c_{\lambda_0}\left(X,\overline{X}\right) &= \sum_{(v,w)\in E, v\in X, w\in\overline{X}} \left(c_\lambda(v,w) - c_{\lambda_0}(v,w)\right) \\
&= \sum_{(s,v)\in E, v\in\overline{X}} \left(c_\lambda(s,v) - c_{\lambda_0}(s,v)\right) - \sum_{(v,t)\in E, v\in X} \left(c_\lambda(v,t) - c_{\lambda_0}(v,t)\right) \\
&= \sum_{(s,v)\in E, v\in\overline{X}} \left(\lambda-\lambda_0\right) a_1(v) - \sum_{(v,t)\in E, v\in X} \left(\lambda-\lambda_0\right) b_1(v) \\
&= \left(\lambda-\lambda_0\right) \left(\sum_{(s,v)\in E, v\in\overline{X}} a_1(v) - \sum_{(v,t)\in E, v\in X} b_1(v) \right) \\
&= \left(\lambda-\lambda_0\right)\beta
\end{aligned}$$

Therefore, we have that

$$c_\lambda\left(X,\overline{X}\right) = c_{\lambda_0}\left(X,\overline{X}\right) + \left(\lambda-\lambda_0\right)\beta = c_{\lambda_0}\left(X,\overline{X}\right) + \lambda\beta - \lambda_0\beta = \alpha + \lambda\beta.$$

This completes the proof. □

A direct consequence of Lemma 2.11, Corollary 2.13 and Lemma 2.14 is the following key lemma, which gives a characterization of the minimum-cut capacity function $\kappa(\lambda)$. The proof of this key lemma is an adaptation of the proof of Lemma 1 in [15].

Lemma 2.15. *The minimum-cut capacity function $\kappa(\lambda)$ is a piecewise-linear concave function with at most $n-2$ breakpoints, where a* breakpoint *is a value of λ at which the slope of $\kappa(\lambda)$ changes.*

Proof. Let λ_1, λ_2 be parameter values with $\lambda_1 < \lambda_2$. Let $(X_1,\overline{X_1}), (X_2,\overline{X_2})$ be minimum cuts for λ_1 and λ_2, respectively. By Lemma 2.11, we can assume without loss of generality that $X_1 \subseteq X_2$. Suppose that $X_2 \setminus X_1$ contains no vertices adjacent to s or t that are neither s nor t (note that $c_\lambda(s,t)$ is constant if $(s,t) \in E$). Since $c_\lambda(e)$ is constant for all $e = (v,w) \in E$ with $v \neq s, w \neq t$, then Lemma 2.14 implies that $c_\lambda(X_1,\overline{X_1})$ and $c_\lambda(X_2,\overline{X_2})$ are linear functions in λ with the same slope. Therefore, $c_\lambda(X_1,\overline{X_1})$ and $c_\lambda(X_2,\overline{X_2})$ are either coincident or parallel.

Case 1. $c_\lambda(X_1,\overline{X_1})$ and $c_\lambda(X_2,\overline{X_2})$ are coincident. Let $\lambda_3 \in [\lambda_1,\lambda_2]$. By Corollary 2.13, there exists a minimum cut $(X_3,\overline{X_3})$ for λ_3 such that $X_1 \subseteq X_3 \subseteq X_2$. Suppose for the sake of contradiction that $c_{\lambda_3}(X_3,\overline{X_3}) < c_{\lambda_3}(X_1,\overline{X_1})$. By Lemma 2.14, the slope of $c_\lambda(X_3,\overline{X_3})$ is at most the slope of $c_\lambda(X_1, \overline{X_1})$ and at least the slope of $c_\lambda(X_2, \overline{X_2})$. Since $c_\lambda(X_1, \overline{X_1})$ and $c_\lambda(X_2, \overline{X_2})$ have the same slope, then $c_\lambda(X_3, \overline{X_3})$ has the same slope. However, since $c_{\lambda_3}(X_3, \overline{X_3}) < c_{\lambda_3}(X_1, \overline{X_1})$, then $c_\lambda(X_3, \overline{X_3}) < c_\lambda(X_1, \overline{X_1})$ for any parameter value λ. Hence, $(X_3, \overline{X_3})$ is a cut whose capacity is less than the capacity of $(X_1, \overline{X_1})$ for λ_1. This contradicts the minimality of $(X_1, \overline{X_1})$ for λ_1. Since $(X_3, \overline{X_3})$ is a minimum cut for λ_3, then $c_{\lambda_3}(X_3, \overline{X_3}) = c_{\lambda_3}(X_1, \overline{X_1})$.

Case 2. $c_\lambda(X_1, \overline{X_1})$ and $c_\lambda(X_2, \overline{X_2})$ are parallel. Without loss of generality, we assume that $c_\lambda(X_1, \overline{X_1}) > c_\lambda(X_2, \overline{X_2})$ for any parameter value λ. Then $(X_2, \overline{X_2})$ is a cut whose capacity is less than the capacity of $(X_1, \overline{X_1})$ for λ_1. This contradicts the minimality of $(X_1, \overline{X_1})$ for λ_1.

We have shown that if $X_2 \setminus X_1$ contains no vertices adjacent to s or t that are neither s nor t, then $\kappa(\lambda)$, $c_\lambda(X_1, \overline{X_1})$ and $c_\lambda(X_2, \overline{X_2})$ coincide on $[\lambda_1, \lambda_2]$. In other words, if $c_\lambda(X_1, \overline{X_1})$ and $c_\lambda(X_2, \overline{X_2})$ have different slopes, then $X_2 \setminus X_1$ contains a vertex adjacent to s or t that is neither s nor t. Since there exist at most $n-2$ vertices adjacent to s or t that are neither s nor t, then there exist at most $n-2$ parameter values $\lambda_1 < \ldots < \lambda_\ell$ and minimum cuts $(X_1, \overline{X_1}), \ldots, (X_\ell, \overline{X_\ell})$ for $\lambda_1, \ldots, \lambda_\ell$, respectively, such that the linear functions $c_\lambda(X_i, \overline{X_i})$ for $i \in [\ell]$ have distinct slopes and that $\kappa(\lambda)$ coincides with $c_\lambda(X_i, \overline{X_i})$ on $[\lambda_i, \lambda_{i+1}]$ for each $i \in [\ell - 1]$.

Finally, we assume without loss of generality that $X_1 \subseteq \ldots \subseteq X_\ell$ by Lemma 2.11. By Lemma 2.14, the slope of $c_\lambda(X_i, \overline{X_i})$ decreases as i increases for $i \in [\ell]$. Therefore, $\kappa(\lambda)$ is concave. This proves that $\kappa(\lambda)$ is a piecewise-linear concave function with at most $n-2$ breakpoints. $\square$

The piecewise-linear concavity of the minimum-cut capacity function $\kappa(\lambda)$ suggests the possible existence of efficient algorithms for computing part or all of the information about $\kappa(\lambda)$. Gallo et al. [12] present three algorithms for computing the smallest (or equivalently, the largest) breakpoint of $\kappa(\lambda)$, a maximum of $\kappa(\lambda)$, and all breakpoints of $\kappa(\lambda)$, respectively, each of which uses the parametric push-relabel algorithm given in Section 2.3 as a subroutine. In the rest of this section, we shall discuss in detail the algorithm for computing the smallest breakpoint of $\kappa(\lambda)$ only, which we shall use in Section 2.5 for solving the perfect sharing problem.

The algorithm for computing the smallest breakpoint of $\kappa(\lambda)$ was initially given by Gusfield [15] for solving the *transmission scheduling problem* stated in [16]. Since $\kappa(\lambda)$ is a piecewise-linear concave function with at most $n-2$ breakpoints, we use $L_1(\lambda) = \alpha_1 + \lambda\beta_1, \ldots, L_\ell(\lambda) = \alpha_\ell + \lambda\beta_\ell$ for some $\ell \in [n-1]$ to denote the linear functions that correspond to the line segments in $\kappa(\lambda)$, i.e., $\kappa(\lambda) = \min_{i \in [\ell]} L_i(\lambda)$. Let λ_0 be the smallest breakpoint of $\kappa(\lambda)$. On one hand, this algorithm can be regarded as an adaptation of Newton's method that computes the unique root of $L_1(\lambda) - \min_{i \in \{2,\ldots,\ell\}} L_i(\lambda)$. Therefore, this algorithm starts with an initial guess $\lambda^* \geq \lambda_0$ and sets

$$\lambda^* \leftarrow \lambda^* - \frac{L_1(\lambda^*) - L_i(\lambda^*)}{L_1'(\lambda^*) - L_i'(\lambda^*)} = \lambda^* - \frac{(\alpha_1 + \lambda^*\beta_1) - (\alpha_i + \lambda^*\beta_i)}{\beta_1 - \beta_i} = \frac{\alpha_i - \alpha_1}{\beta_1 - \beta_i}$$

at each step, where $L_i(\lambda)$ is the linear function corresponding to the line segment of $\kappa(\lambda)$ above λ^*, i.e., $\kappa(\lambda^*) = L_i(\lambda^*)$. Alternatively, we can regard this algorithm as initially fixing an interval $[\lambda_1, \lambda_2]$ that includes λ_0, and shrinking the interval by strictly decreasing λ_2 to the next breakpoint on the left at each step. We give the formal description of this algorithm in Algorithm 2.8, where we use $L_X(\lambda) = \alpha_X + \lambda\beta_X$ to denote the linear function corresponding to the cut $(X, \overline{X})$.

```
1 choose $\lambda_1, \lambda_2$ such that $\lambda_0 \in [\lambda_1, \lambda_2]$
2 compute a minimum cut $(X_1, \overline{X_1})$ for $\lambda_1$
3 repeat
4     compute a minimum cut $(X_2, \overline{X_2})$ for $\lambda_2$
5     if $L_{X_1}(\lambda_2) = L_{X_2}(\lambda_2)$ then
6         return $\lambda_2$
7     else
8         $\lambda_2 \leftarrow (\alpha_{X_2} - \alpha_{X_1})/(\beta_{X_1} - \beta_{X_2})$
```

Algorithm 2.8: An algorithm for computing the smallest breakpoint of the minimum-cut capacity function $\kappa(\lambda)$.

We note that the value of λ_2 is strictly decreasing during the execution of the algorithm; indeed, the algorithm sets λ_2 to be the value of λ at the intersection of $L_{X_1}(\lambda)$ and $L_{X_2}(\lambda)$, i.e., the value of λ such that $L_{X_1}(\lambda) = L_{X_2}(\lambda)$. Therefore, the **repeat** loop of Algorithm 2.8 terminates after at most $n-2$ iterations. Since λ_2 is strictly decreasing, we can use the parametric push-relabel algorithm on the reverse graph of G with source t and sink s to maintain a minimum cut for the current value of λ_2. Therefore, Algorithm 2.8 runs in $O(mn \log \frac{n^2}{m})$ time, which improves the running time of $O(mn^2 \log n)$ in [15] by a factor of n. Moreover, we note that a variant of Algorithm 2.8 can be used to compute the largest breakpoint of $\kappa(\lambda)$ by repeatedly increasing λ_1 and running the parametric push-relabel algorithm on G instead of repeatedly decreasing λ_2 and running the parametric push-relabel algorithm on the reverse graph of G.

Finally, there exist various valid choices of the initial values of λ_1 and λ_2 in the first line of Algorithm 2.8. Gallo et al. [12] provide the following initial values of λ_1 and λ_2, where $a(v) = 0$ for all $v \in V$ with $(s, v) \notin \delta_G^+(s)$ and $b(v) = 0$ for all $v \in V$ with $(v, t) \notin \delta_G^-(t)$.

$$\lambda_1 = \min_{\substack{v \in V \setminus \{s,t\} \\ a_1(v)+b_1(v)>0}} \frac{b_0(v) - a_0(v) - \sum_{u \in V \setminus \{s,t\}} c(u, v)}{a_1(v) + b_1(v)} - 1,$$

$$\lambda_2 = \max_{\substack{v \in V \setminus \{s,t\} \\ a_1(v)+b_1(v)>0}} \frac{b_0(v) - a_0(v) + \sum_{u \in V \setminus \{s,t\}} c(u, v)}{a_1(v) + b_1(v)} + 1.$$

2.5 The Perfect Sharing Problem

A class of important applications of the parametric push-relabel algorithm and the algorithms for computing information about $\kappa(\lambda)$ is the class of *flow sharing problems*. We shall briefly describe the motivation of flow sharing problems. The food bank would like to supply food to a set of affected areas during a famine so that each person would receive an equal share, while maximizing the total amount of food supplied. In other words, the goal of distributing food is to equalize the ratio of the food supplied for each area to the population of that area. This specific example leads to the *perfect sharing problem*. Other criteria of distribution may also be of interest. For instance, one may want to distribute food so that the minimum ratio defined above amongst all areas is maximized, or the maximum ratio amongst all areas is minimized.

The formal formulation of general flow sharing problems is given in the following. Let $G = (V, E)$ be a capacitated directed graph with a single sink s and a single sink t. Let $S = \{s_1, \ldots, s_k\}$ be the set of vertices adjacent to s, called the *(source) frontier vertices*. Let $w : S \to \mathbb{R}_+$ denote the positive weight associated with each frontier vertice. Given a flow $f : E \to \mathbb{R}_+$, we define $u_f : S \to \mathbb{R}_+$ to denote the flow through the edge (s, s') for each frontier vertex s', i.e., $u_f(s') = f(s, s')$ for each frontier vertex s'. The following flow sharing problems are considered by Gallo et al. [12]:

- ***Perfect Sharing.*** Find a flow f whose value is maximized amongst flows with $u_f(s')/w(s')$ equal for all $s' \in S$.
- ***Maximin Sharing.*** Find a flow f that maximizes the smallest value of $u_f(s')/w(s')$ for $s' \in S$ amongst maximum flows.
- ***Minimax Sharing.*** Find a flow f that minimizes the largest value of $u_f(s')/w(s')$ for $s' \in S$ amongst maximum flows.

- ***Optimal Sharing.*** Find a flow f that simultaneously maximizes the smallest value of $u_f(s')/w(s')$ for $s' \in S$ and minimizes the largest value of $u_f(s')/w(s')$ for $s' \in S$ amongst maximum flows.
- ***Lexicographic Sharing.*** Find a flow f that lexicographically maximizes the k-vector whose j^{th} component is the j^{th} smallest value of $u_f(s')/w(s')$ for $s' \in S$ amongst maximum flows.

We briefly note that the above formulations of these five flow sharing problems are said to be *multiple-source one-sink* in [12] (because the source s is called the *super-source* and the frontier vertices $s_1, \dots, s_k$ are called the *sources* in [12]). It is equivalent to consider these five flow sharing problems in the *one-source multiple-sink* fashion in which we replace G with its reverse graph, interchange the source and the sink, and re-define the frontier vertices to be those adjacent to the sink. Gallo et al. [12] and Megiddo [21] also note that the last four problems outlined above can be formulated in the *multiple-source multiple-sink* fashion defined analogously.

In this section, we chiefly discuss an algorithm based upon the parametric push-relabel algorithm for solving the perfect sharing problem, to which we shall reduce the problem of finding a congestion-minimizing fractional flow in Section 2.6. Indeed, all of these five flow sharing problems can be solved with the following parametric formulation: For each frontier vertex s', we set the capacity of the edge (s, s') to be $w(s')\lambda$, where $\lambda \in [0, \infty)$.

For the perfect sharing problem, Gusfield [15] proposes a very simple algorithm, given in Algorithm 2.9, which is based upon the piecewise-linear concavity of $\kappa(\lambda)$. This algorithm was initially used to solve the transmission scheduling problem, which we shall also discuss briefly below as an application of the perfect sharing problem. It is straightforward to see that the running time of Algorithm 2.9 is $O(mn \log \frac{n^2}{m})$. We prove the correctness of Algorithm 2.9 in the following lemma.

```
1 $c_\lambda(s, s') \leftarrow w(s')\lambda$ for all $s' \in S$
2 $\lambda_s \leftarrow$ smallest breakpoint of $\kappa(\lambda)$
3 **return** *any maximum flow for* $\lambda_s$
```

Algorithm 2.9: An algorithm for the perfect sharing problem.

Lemma 2.16. *Algorithm 2.9 produces a flow f whose value is maximized amongst flows with $u_f(s')/w(s')$ equal for all $s' \in S$.*

Proof. If $\lambda = 0$, then $c_0(s, s') = 0$ for each $s' \in S$, and therefore $(\{s\}, V \setminus \{s\})$ is a minimum cut with capacity $c_0(\{s\}, V \setminus \{s\}) = 0$. Since λ_s is the smallest breakpoint of $\kappa(\lambda)$, then $\kappa(\lambda) = c_\lambda(\{s\}, V \setminus \{s\})$ on $[0, \lambda_s]$, and $\kappa(\lambda) < c_\lambda(\{s\}, V \setminus \{s\})$ on (λ_s, ∞). Let λ be a parameter value such that $\lambda > \lambda_s$ and that there exists a flow f such that $u_f(s')/w(s') = \lambda$ and therefore $u_f(s') = w(s')\lambda = c_\lambda(s')$ for all $s' \in S$. By the maximum-flow minimum-cut theorem, the maximum flow value and the minimum cut capacity both equal

$$\kappa(\lambda) = \sum_{s' \in S} c_\lambda(s') = \sum_{s' \in S} w(s')\,\lambda = \lambda \sum_{s' \in S} w(s') = c_\lambda(\{s\}, V \setminus \{s\}).$$

Since λ_s is the maximum parameter value λ such that $\kappa(\lambda) = c_\lambda(\{s\}, V \setminus \{s\})$, then any maximum flow for λ_s solves the perfect sharing problem. □

Moreover, since the capacities of the incoming edges into t are constant in our parametric formulation for the aforementioned flow sharing problems, then we can prove the following lemma which gives a tighter bound on the number of breakpoints in $\kappa(\lambda)$ than Lemma 2.15 using the same argument.

Lemma 2.17. *In the parametric network defined above with k frontier vertices, the minimum-cut capacity function $\kappa(\lambda)$ is a piecewise-linear concave function with at most k breakpoints, each per frontier vertex s'.*

As a digression, we briefly discuss two problems that can be reduced to the perfect sharing problem. The first problem is the aforementioned *transmission scheduling problem*, which was first introduced by Itai and Rodeh [16] and for which Algorithm 2.9 was initially designed by Gusfield [15]. In the simplified formulation given in [15], the transmission scheduling problem is defined by the following. Let $G = (V, E)$ be a directed garph with edge capacities $c : E \to \mathbb{R}_+$, which denotes the transmission rate of each edge (in bits per second). Let $t \in V$ be the sink and $S \subseteq V \setminus \{t\}$ the set of sources. Let $w : S \to \mathbb{R}_+$ denote the number of bits that each source would like to send to the sink. The goal of the transmission scheduling problem is to find a minimum number T such that the demands of all sources can be completed within T seconds.

We shall show a reduction from the transmission reduction problem to the perfect sharing problem. We add a super-source s and an edge (s, s') from s to each source s'. Let f be a flow with $u_f(s')/w(s')$ equal for all $s' \in S$. Let λ be such that $\lambda = u_f(s')/w(s')$ for all $s' \in S$. Let $T = \frac{1}{\lambda}$. Then the value of f is given by

$$\sum_{s' \in V} u_f(s') = \sum_{s' \in V} w(s')\,\lambda = \lambda \sum_{s' \in V} w(s') = \frac{1}{T} \sum_{s' \in V} w(s').$$

This implies that the minimum value of T equals the maximum value of a flow f with $u_f(s')/w(s')$ equal for all $s' \in S$. Therefore, the perfect sharing problem solves the transmission scheduling problem. We note that solving the transmission scheduling problem using the perfect sharing problem takes $O(mn \log \frac{n^2}{m})$ time, which improves the two algorithms of Itai and Rodeh [16] with running times of $O(kmn^2)$ and $O(kmn \log n)$, respectively, where k is the number of sources.

The other problem reducible to the perfect sharing problem that we shall discuss is the problem of the *strength of a directed network* introduced by Cunningham [4]. This problem is formulated as follows. Let $G = (V, E)$ be a directed graph with a source $s \in V$. Let $c : E \to \mathbb{R}_+$ be the edge weights, which denote the cost to delete each edge e. Let $d : V \setminus \{s\} \to \mathbb{R}_+$ be the vertex weights, which denote the value of each vertex if it is reachable from s. Note that deleting a subset A of edges with total cost $f(A) = \sum_{e \in A} c(e)$ may disconnect a subset $V_A \subseteq V \setminus \{s\}$ of vertex from s, and hence induce a total loss of $g(A) = \sum_{v \in V_A} d(v)$. The *strength* of G is then defined to be

$$\lambda^* = \min \left\{ \frac{f(A)}{g(A)} : A \subseteq E, g(A) > 0 \right\}.$$

Gallo et al. [12] discuss several approaches for computing the strength of a network, one of which is to reduce this problem to the perfect sharing problem (in the one-source multiple-sink version). To see this reduction, we note that $\lambda^* \leq \lambda$ if and only if $\min\{f(A) - \lambda g(A) : A \subseteq E\} \geq 0$. The latter optimization problem is called an *attack problem* in [4]. It is straightforward that the strength problem can be converted to a sequence of attack problems by binary search in the case of integral values. Indeed, Cunningham [4] gives an algorithm that computes the strength problem by solving at most n attack problems. The key observation of Cunningham [4] that is essential for this reduction is the following lemma:

Lemma 2.18. *$f(A) - \lambda g(A)$ over $A \subseteq E$ attains the minimum value at $\delta^-_G(B)$ for some minimum cut $(\overline{B}, B)$ with $B \subseteq V \setminus \{s\}$.*

For a cut $(\overline{B}, B)$ with $B \subseteq V \setminus \{s\}$, we note that B is the set of vertices not reachable from s. Therefore, given $\lambda > 0$, the attack problem $\min\{f(A) - \lambda g(A) : A \subseteq E\}$ can be converted to the following optimization problem:

$$\min \left\{ f\left(\delta^-_G(B)\right) - \lambda d(B) : B \subseteq V \setminus \{s\} \right\}.$$

This optimization problem can be computed by solving the perfect sharing problem formulated in the following. We add a sink t and an edge (v, t) for each $v \in V \setminus \{s\}$. Let f be a flow whose value is maximized amongst flows with $u_f(v)/d(v)$ equal for all $v \in V \setminus \{s\}$. Let λ be such that $\lambda = u_f(v)/d(v)$ for all $v \in V \setminus \{s\}$. We set the capacity of (v,t) to be $d(v)\lambda$ for each $v \in V \setminus \{s\}$. For each cut $(V \setminus (B \cup \{t\}), B \cup \{t\})$ with $B \subseteq V \setminus \{s\}$,

$$\begin{aligned} c(V \setminus (B \cup \{t\}), B \cup \{t\}) &= f\left(\delta_G^-(B)\right) + \lambda d(V \setminus (B \cup \{s\})) \\ &= \left(f\left(\delta_G^-(B)\right) - \lambda d(B)\right) + \lambda d(V \setminus \{s\}). \end{aligned}$$

Therefore, any minimum cut $(V \setminus (B \cup \{t\}), B \cup \{t\})$ with $B \subseteq V \setminus \{s\}$ implies that B minimizes $f(\delta_G^-(B)) - \lambda d(B)$. This completes the reduction of the strength problem to the perfect sharing problem.

2.6 Finding a Minimum Congestion Flow

Equipped with necessary ingredients, namely the parametric push-relabel algorithm from Section 2.3 and the perfect sharing problem from Section 2.5, we are now at a position to compute a fractional flow with minimum congestion by a reduction to the perfect sharing problem.

Let $G = (V, E)$ be a directed graph with vertex demands $d : V \to \mathbb{R}_+$ and the set S of sinks. We construct an auxiliary directed graph $G' = (V^+ \cup V^-, E')$ by the following procedure. For each vertex $v \in V$, we add two copies v^+, v^- to V^+ and V^-, respectively, and an edge (v^-, v^+) to E'. For each edge $(v, w) \in E$, we add an edge (v^+, w^-) to E'. In addition, we add a super-source s, a super-sink t, an edge (s, v^-) for each vertex $v \in V$ with $d(v) > 0$, and an edge (v^+, t) for each sink $v \in S$. We set the weight $w(v^-) = d(v)$ for each vertex $v \in V$ with $d(v) > 0$. Finally, we define edge capacities $c : E \to \mathbb{R}_+$. We set $c(s, v^-) = \infty$ for each $v \in V$ with $d(v) > 0$, and $c(e) = 1$ for each $e \in E' \setminus \{(s, v^-) : v \in V, d(v) > 0\}$.

We show that the minimum vertex congestion problem on G is equivalent to the perfect sharing problem on G'. Let $f' : E' \to \mathbb{R}_+$ be a flow on G' whose value is maximized amongst flows with $u_{f'}(v^-)/w(v^-)$ equal for all $v \in V$ with $d(v) > 0$. Let λ be such that $\lambda = u_{f'}(v^-)/w(v^-)$ for all $v \in V$ with $d(v) > 0$. Then the value of f' is given by

$$\sum_{\substack{v \in V \\ d(v) > 0}} u_{f'}\left(v^-\right) = \sum_{\substack{v \in V \\ d(v) > 0}} w\left(v^-\right)\lambda = \sum_{\substack{v \in V \\ d(v) > 0}} d(v)\lambda = \lambda \sum_{\substack{v \in V \\ d(v) > 0}} d(v).$$

Let $f : E \to \mathbb{R}_+$ be defined by $f(v, w) = \frac{1}{\lambda} f'(v^+, w^-)$ for each edge $(v, w) \in E$. Note that f satisfies the conservation constraints on G by the construction of the auxiliary directed graph G'. For each vertex $v \in V$ with $d(v) = 0$, the congestion of v is given by

$$\begin{aligned} f\left(\delta_G^-(v)\right) &= \sum_{(u,v)\in\delta_G^-(v)} f(u,v) = \sum_{(u,v)\in\delta_G^-(v)} \frac{f'\left(u^+, v^-\right)}{\lambda} = \frac{1}{\lambda} \sum_{e\in\delta_{G'}^-(v^-)} f'(e) \\ &= \frac{f'\left(v^-, v^+\right)}{\lambda} \leq \frac{c\left(v^-, v^+\right)}{\lambda} = \frac{1}{\lambda}. \end{aligned}$$

Moreover, for each vertex $v \in V$ with $d(v) > 0$, $d(v) = w\left(v^-\right) = \frac{1}{\lambda} u_{f'}(v^-) = \frac{1}{\lambda} f'(s, v^-)$, and hence the congestion of v is given by

$$\begin{aligned} f\left(\delta_G^-(v)\right) + d(v) &= \sum_{(u,v)\in\delta_G^-(v)} f(u,v) + \frac{f'\left(s, v^-\right)}{\lambda} \\ &= \sum_{(u,v)\in\delta_G^-(v)} \frac{f'\left(u^+, v^-\right)}{\lambda} + \frac{f'\left(s, v^-\right)}{\lambda} = \frac{1}{\lambda} \sum_{e\in\delta_{G'}^-(v^-)} f'(e) \\ &= \frac{f'\left(v^-, v^+\right)}{\lambda} \leq \frac{c\left(v^-, v^+\right)}{\lambda} = \frac{1}{\lambda}. \end{aligned}$$

The maximality of f implies that $\frac{1}{\lambda}$ is minimized. Hence, $\frac{1}{\lambda}$ is the minimum vertex congestion of G. This proves the reduction from the minimum vertex congestion problem to the perfect sharing problem.

Chapter 3

Confluent Flows

I pick up my life
And take it away
On a one-way ticket–
Gone up North,
Gone out West,
Gone!

— Langston Hughes, *One-Way Ticket*

The simplicity of the confluent network flow model and its arborescence-shaped support graph gained its universality in real-world applications. One of the major applications of confluent flows is IP routing on the Internet because IP routing often selects a shortest path tree for each destination, as discussed in Chapter 1. Other applications are mentioned in [3]. For instance, another scenario of the confluent network flow model is emergency evacuation in a hotel, where fleeing people "meet and merge" by following emergency exit signs, forming a tree of their evacuation routes. Moreover, servers in content delivery networks (CDNs) are usually connected in the shape of a rooted tree (rooted at the network operations centre) so that data are transmitted between parents and their children, and vice versa. Lastly, the increasing popularity of wireless networks over the past several decades gives rise to new applications of the confluent flow model. An ad hoc network is a decentralized wireless network in which each vertex in the network participates in routing, which therefore does not depend upon pre-existing network infrastructures such as routers. Ad hoc networks are also usually organized as a rooted tree (rooted at the wired access point connected to the Internet) so that each access point in the ad hoc network transmits data with its parent to connect to the Internet.

Several works study the impact of confluent networks on the congestion of a netowk heuristically and practically in the setting of IP routing; see, for instance, [20] and [29]. However, these two works does not go beyond the trivial approximation ratio of $O(n)$. In this paper, we summarize the approximation algorithms in [2] to give theoretical upper bounds for several optimization problems on confluent flows, including the *congestion minimization* problem and the *demand maximization* problem. Moreover, we give an instance which admits a fractional flow with congestion at most 1, yet no confluent flow with congestion smaller than H_k, the k^{th} harmonic number, where k is the number of sinks. This example shows that the minimum congestion is inevitably unbounded for a confluent flow.

Recall that a flow is *confluent* if each vertex can send flow to at most one outgoing edge. Throughout this chapter, we consider uniform vertex capacities (i.e., $c(e) = 1$ for each $e \in E$) and assume that there exist k sinks $t_1, \ldots, t_k$. The algorithm in Section 2.6 produces a fractional flow satisfying all demands and capacity constraints at all vertices. Hence, by scaling all demands and flows by the congestion value, we can assume without loss of generality that there exists an optimal fractional flow with congestion 1. Moreover, we assume that the support network of the optimal fractional flow is acyclic by repeatedly applying the operation of directed cycle cancellation, defined in Procedure 3.1, which subtracts ε flow on each edge along C with ε chosen to be the minimum flow on an edge along C whenever there exists a directed cycle C in the support network.

```
1 $\varepsilon \leftarrow \min\{f(e) : e \in E(C)\}$
2 $f(e) \leftarrow f(e) - \varepsilon$ for all $e \in E(C)$
```

Procedure 3.1: `EliminateCycle`(C, f).

3.1 Congestion Minimization

In this section, we present a $(1 + \lg k)$-approximation algorithm, due to Chen et al. [2], for the congestion minimization problem on confluent flows, where the logarithm has base 2. We shall see, after the expositions in this chapter, that a repeatedly emerging pattern in approximation algorithms for confluent flows is a combination of three operations on a flow and its support graph, which we introduce below.

Recall that a vertex is a *frontier vertex* if it is adjacent to a sink. The first operation is then to *aggregate* a frontier vertex v with all of its outgoing edges to the same sink t_i. Such a frontier vertex is called a *decided vertex*. Specifically, the operation marks one of the outgoing edges, contract v into t_i, and add $d(v)$ to $d(t_i)$. We define this operation in Procedure 3.2.

The operation of *vertex aggregation* can be interpreted from two points of view. On one hand, we can view this operation in the shrunken graph as described above, and the graph gets smaller after performing a vertex aggregation operation. On the other hand, it is sometimes more convenient to visualize this operation on the original graph. A straightforward inductive proof shows that a sink in the shrunken graph always corresponds to an in-arborescence rooted at the corresponding sink in the original graph, and that each vertex aggregation operation grows the in-arborescence from one of its leaves. Since each vertex can be aggregated into at most one sink, then the sinks in the shrunken graph correspond to vertex-disjoint sink trees in the original graph. We call such an in-arborescence rooted at a sink in the original graph a *sink tree*. The discussion in this paragraph is summarized in the following lemma.

Lemma 3.1. *The sinks in the shrunken graph correspond to vertex-disjoint sink trees in the original graph formed by the marked edges.*

```
1 mark one of the outgoing edges of $v$
2 $d(t_i) \leftarrow d(t_i) + d(v)$
3 foreach edge $e$ to $v$ do
4     set the head of $e$ to $t_i$
```

Procedure 3.2: `Aggregate`(v, t_i), where v is a decided vertex with all outgoing edges to the same sink t_i.

A *sawtooth cycle* is a collection

$$C = \{(u_0, v_0), P_0, (u_1, v_1), P_1, \ldots, (u_{r-1}, v_{r-1}), P_{r-1}\}, \tag{3.1}$$

where (u_i, v_i) is an edge, u_i is a sink, and P_i is a u_{i+1}-v_i directed path in the support graph of f, with subscripts modulo r. In other words, a *sawtooth cycle* is a cycle of length at least 3 in an auxiliary graph $\hat{G}$ obtained from G by adding a reverse edge for each edge from a frontier vertex to a sink. An example of a sawtooth cycle is given in Figure 3.1. In Chapter 4, we shall see that removing the restriction that only edges from a frontier vertex to a sink can be reversed gives the definition of a general sawtooth cycle, which admits a nice characterization in terms of the *acyclic digon-tree representation*. In this chapter, however, the restricted definition of sawtooth cycles suffices.

The second operation, defined in Procedure 3.3, *eliminates* a sawtooth cycle in the support network of a flow by adding ε flow to each edge (u_i, v_i) and subtracting ε flow along directed paths P_i, where ε is chosen appropriately so that at least one edge along the sawtooth cycle vanishes.

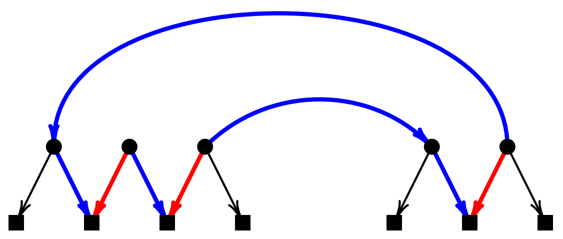


Figure 3.1: An example of a sawtooth cycle, where the red edges denote the reverse edges of edges from frontier vertices to sinks, and square vertices denote sinks.

```
1 $\varepsilon \leftarrow \min\{f(e) : e \in E(P_i), i = 0, \ldots, r-1\}$
2 $f(u_i, v_i) \leftarrow f(u_i, v_i) + \varepsilon$ for all $i = 0, \ldots, r-1$
3 $f(e) \leftarrow f(e) - \varepsilon$ for all $e \in E(P_i)$ and $i = 0, \ldots, r-1$
```

Procedure 3.3: `EliminateSawtooth`(C, f), where C is a sawtooth cycle as defined in (3.1).

The third operation, *sink deactivation*, is concerned with flow redirection between two outgoing edges at a frontier vertex. We say that a sink is a *remote sink* if it has only one in-neighbour, which has at least one other sink out-neighbour. We give an example of this structure in Figure 3.2. We denote by $f(v)$ the congestion at a vertex v for some flow f on G. In [2], sink deactivation is defined as in Procedure 3.4.

The three operations introduced above can be applied repeatedly until only the sinks remain in the shrunken graph. This is summarized in the following lemma.

Lemma 3.2. *If there is neither a decided vertex nor a sawtooth cycle, then there is a remote sink, which can be found by a polynomial time algorithm.*

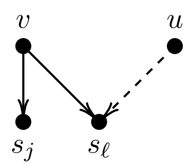


Figure 3.2: An example of a remote sink s_j with its only in-neighbour v, which has at least one other sink out-neighbour s_ℓ.

```
1 if $f(t_j) + f(v, t_\ell) < f(t_\ell) - f(v, t_\ell)$ then
2     redirect flow on $(v, t_\ell)$ to $(v, t_j)$
3     remove edge $(v, t_\ell)$
4 else
5     redirect flow no $(v, t_j)$ to $(v, t_\ell)$
6     remove edge $(v, t_\ell)$
7     deactivate $t_j$
```

Procedure 3.4: `DeactivateSink`(v, t_j, t_ℓ), where t_j is a remote sink whose only in-neighbour is v that has another sink out-neighbour t_ℓ.

Proof. Let $\hat{G}$ be an auxiliary graph obtained from G by adding a reverse edge for each edge from a frontier vertex to a sink. We give such a polynomial time algorithm in Algorithm 3.5.

```
1 walk along the edges of $\hat{G}$, obeying the following three rules:
2 (a) when possible, an edge $(u, v)$ is not followed by its reverse edge
3 (b) when possible subject to (a), void visiting sinks
4 (c) stop after visiting a vertex twice
5 $v \leftarrow$ the vertex visited twice
6 $(w, v) \leftarrow$ the last visited edge
7 return $w$
```

Algorithm 3.5: A polynomial time algorithm to find a remote sink, provided that there exist neither decided vertices nor sawtooth cycles in the support network of f.

It is straightforward to see that the running time of Algorithm 3.5 is $O(n)$ since each step in the walk visits at least one vertex and since the walk stops immediately after it visits a vertex twice.

Let v, w be the vertices as in Algorithm 3.5. We show that w is a remote sink whose only in-neighbour is v. By construction, each vertex of $\hat{G}$ has at least one outgoing edge. Hence, the walk would continue indefinitely without (c). Since there does not exist a sawtooth cycle in G, then there does not exist a cycle of length at least 3 in $\hat{G}$. This implies that the last two visited edges in the walk are (v, w) and (w, v). Note that at least one of v and w is a sink because otherwise (v, w) and (w, v) would form a cycle, which contradicts the acyclicity of G. Moreover, note that v is not a sink because otherwise w would be a frontier vertex all of whose outgoing edges go to v due to (a), which would therefore satisfy the condition of a vertex aggregation operation. Therefore, w is a sink, and (a) implies that v is a frontier vertex all of whose outgoing edges go to w. Lastly, note that v must have one other sink out-neighbour because otherwise one of the following two cases would happen:

- all of the outgoing edges of v go to w, which satisfies the condition of a vertex aggregation operation;
- it has an edge to another non-sink vertex, which would have been visited instead of (v, w) due to (b).

This proves that w is a remote sink whose only in-neighbour is v. □

The $(1+\lg k)$-approximation algorithm of Chen et al. [2] simply consists of a main loop which executes the three aforementioned operations whenever possible, until only the sinks remain in the shrunken graph. We give this algorithm in Algorithm 3.6, where f is a fractional flow on G with the congestion at any vertex at most 1. It is straightforward to see that each vertex aggregation removes a vertex, and that each sawtooth cycle elimination or sink deactivation removes an edge. Therefore, repeatedly applying the three aforementioned operations takes finite steps. Moreover, we note that vertex aggregation does not change the congestion at any vertex, that sawtooth cycle elimination does not increase the congestion at any vertex, and that sawtooth cycle elimination does not change the congestion at any sink.

```
1 while $V(G) \neq S$ do
2     do one of the following three operations whenever possible:
3     (a) Aggregate($v, t_i$) for some frontier vertex $v$
4     (b) EliminateSawtooth($C, f$) for some sawtooth cycle $C$
5     (c) DeactivateSink($v, t_j, t_\ell$) for some remote sink $t_j$
```

Algorithm 3.6: A $(1+\lg k)$-approximation algorithm for congestion minimization on confluent flows.

Lemma 3.3. *Each vertex aggregation operat does not change the congestion of any vertex.*

Lemma 3.4. *Each elimination of a sawtooth cycle does not increase the congestion of any vertex, and does not change the congestion of any sink in the shrunken graph.*

It then remains to show that sink deactivation increases the congestion of the sink to which flow is redirected by at most $\lg k$. This analysis uses a potential function argument. Given a flow f and a sink t_i, we define the potential at t_i to be $\phi(t_i) = 2^{f(t_i)}$, and the potential of the flow f to be the sum of the potentials at the *active* sinks. We note that this potential function $\phi(\cdot)$ is convex. The following lemma shows that sink deactivation does not increase the congestion of the current flow.

Lemma 3.5. *Sink deactivation does not increase the potential of the flow. In particular, the potential of the deactivated sink, if any, is no more than the potential of the flow before the sink deactivation.*

Proof. We denote by f' the new flow after the redirection, and by ϕ' the new potential after the redirection. If $f(t_j)+f(v,t_\ell) < f(t_\ell)-f(v,t_\ell)$, then we redirect flow from (v,t_ℓ) to (v,t_j) and remove (v,t_ℓ). Since we increase and decrease the congestions at t_j and t_ℓ by the same amount, then there exists $\lambda \in (0,1)$ such that

$$f'(t_j) = \lambda f(t_j) + (1-\lambda) f(t_\ell)\,,$$
$$f'(t_\ell) = (1-\lambda) f(t_j) + \lambda f(t_\ell)\,.$$

By the convexity of $2^{(\cdot)}$, we have that

$$\phi'(t_j) = 2^{f'(t_j)} \le \lambda 2^{f(t_j)} + (1-\lambda)2^{f(t_\ell)} = \lambda\phi(t_j) + (1-\lambda)\phi(t_\ell)\,, \tag{3.2}$$
$$\phi'(t_\ell) = 2^{f'(t_j)} \le (1-\lambda) 2^{f(t_j)} + \lambda 2^{f(t_\ell)} = (1-\lambda)\phi(t_j) + \lambda\phi(t_\ell)\,. \tag{3.3}$$

Adding (3.2) and (3.3) gives that

$$\phi'(t_j) + \phi'(t_\ell) \le \phi(t_j) + \phi(t_\ell)\,.$$

In other words, sink deactivation does not increase the potential of the flow if $f(t_j) + f(v,t_\ell) < f(t_\ell) - f(v,t_\ell)$.

On the other hand, if $f(t_j) + f(v,t_\ell) \ge f(t_\ell) - f(v,t_\ell)$, i.e., $f(t_j) \ge f(t_\ell) - 2f(v,t_\ell)$, then we redirect flow from (v,t_j) to (v,t_ℓ), remove (v,t_j), and deactivate t_j. The increase in the potential of the flow is given by

$$\begin{aligned}\phi'(t_\ell) - \phi(t_j) - \phi(t_\ell) &= 2^{f'(t_\ell)} - 2^{f(t_j)} - 2^{f(t_\ell)} \\ &= 2^{f(t_\ell)+f(v,t_j)} - 2^{f(t_j)} - 2^{f(t_\ell)} \\ &\le 2^{f(t_\ell)+f(v,t_j)} - 2^{f(t_\ell)-2f(v,t_\ell)} - 2^{f(t_\ell)}.\end{aligned}$$

Since $f(t_\ell) - f(v, t_\ell) = \frac{1}{2}(f(t_\ell) - 2f(v, t_\ell)) + \frac{1}{2}f(t_\ell)$, then the convexity of $2^{(\cdot)}$ implies that

$$2^{f(t_\ell)-2f(v,t_\ell)} \leq \frac{1}{2} \cdot 2^{f(t_\ell)-2f(v,t_\ell)} + \frac{1}{2} \cdot 2^{f(t_\ell)}.$$

Therefore, we have that

$$2^{f(t_\ell)-2f(v,t_\ell)} + 2^{f(t_\ell)} \geq 2 \cdot 2^{f(t_\ell)-2f(v,t_\ell)} = 2^{f(t_\ell)-2f(v,t_\ell)+1}.$$

Since the congestion of f is 1, then $f(v) \leq 1$. Hence, we have that

$$\begin{aligned}(f(t_\ell) + f(v, t_j)) - (f(t_\ell) - f(v, t_\ell) + 1) &= f(v, t_j) + f(v, t_\ell) - 1 \\ &\leq f(v) - 1 \leq 1 - 1 = 0.\end{aligned}$$

This implies that $f(t_\ell) + f(v, t_j) \leq f(t_\ell) - f(v, t_\ell) + 1$ and therefore that $2^{f(t_\ell)+f(v,t_j)} \leq 2^{f(t_\ell)-f(v,t_\ell)+1}$. This eventually gives that the increase in the potential of the flow is at most

$$2^{f(t_\ell)+f(v,t_j)} - 2^{f(t_\ell)-2f(v,t_\ell)} - 2^{f(t_\ell)} \leq 2^{f(t_\ell)+f(v,t_j)} - 2^{f(t_\ell)-2f(v,t_\ell)+1} \leq 0.$$

This proves that sink deactivation does not increase the potential of the flow. Moreover, the second assertion in the lemma is straightforward. □

The lemmas stated and proved in this section lead to the next theorem concerning the correctness of Algorithm 3.6.

Theorem 3.6. *Given a fractional flow satisfying all demands with vertex congestion* 1*, Algorithm 3.6 finds a confluent flow that satisfies all demands and has congestion at most* $1 + \lg k$*.*

Proof. By Lemma 3.1, the sinks in the shrunken graph correspond to vertex-disjoint sink trees in the original graph formed by the marked edges. Hence, if the vertex set of the shrunken graph contains the sinks only, then the marked edges form a confluent flow.

Since the congestion at each sink is at most 1 in the initial fractional flow, then the potential of the initial flow is at most $\sum_{i=1}^{k} \phi(t_i) = k \cdot 2^1 = 2k$. Lemma 3.3, Lemma 3.4 and Lemma 3.5 show that the potential of the flow does not increase throughout the algorithm, and that the potential of any deactivated sink is at most $2k$. Therefore, the potential of any sink is at most $2k$ upon the termination of the algorithm. It follows that the congestion of any sink with respect to the final confluent flow is at most $\lg 2k = 1 + \lg k$. □

To conclude this section, we note that Chen et al. [2] slightly improves the approximation ratio of Algorithm 3.6 to $1 + \ln k$, where the logarithm uses the natural base. Specifically, this improved $(1 + \ln k)$-approximation algorithm generalizes sink deactivation to *parsimonious sink deactivation*, which performs flow redirection in an induced subgraph G_1 of G with the following properties:

- Every edge in G_1 connects a frontier to a sink.
- G_1 contains all of the outgoing edges of its frontier edges and all of the incoming edges of its sinks.
- Every frontier vertex of G_1 is adjacent to at least two sinks.

Moreover, the improves algorithm uses convex minimization to perform the *balancing* operation, which redistributes flow from the frontier vertices in G_1 to the sinks in G_1. The details of this improved algorithm is more involved than those of Algorithm 3.6, so we refer the reader to their original paper for detailed expositions.

3.2 Demand Maximization

The three operations in the last section—*vertex aggregation*, *sawtooth cycle elimination* and *sink deactivation*—are of great importance to optimization problems on confluent flows. In this section, we present a slight variant of Algorithm 3.6, which gives a constant-ratio approximation algorithm for the demand maximization problem on confluent flows. Recall that the demand maximization problem asks for a subset of the sources whose demands can be routed by a feasible flow and which maximizes the routed demands.

In Algorithm 3.6, we assume that there exists a fractional flow which satisfies *all* demands with congestion 1, which can be produced by the congestion-minimizing algorithm in Section 2.6. This assumption gives rise to a 3-approximation algorithm for the demand maximization problem for confluent flows. However, if we only want a constant-ratio approximation for demand maximization, then we can relax this assumption by allowing any fractional flow with congestion 1 which satisfies a constant fraction of the maximum satisfiable demands. Thus, any constant-ratio approximation algorithm for the demand maximization problem on single-sink unsplittable flows can be used to give such an initial flow. For instance, Kolliopoulos and Stein [19] gives a 1.33-approximation algorithm for demand maximization on single-sink unsplittable flows, and this approximation ratio is improved to 4.43 by Dinitz et al. [6], which is the best known approximation ratio. Given a 4.43-approximation single-sink unsplittable flow as the initial flow, we can obtain a 13.29-approximation algorithm for demand maximization on confluent flows.

We assume that there exists a fractional flow that satisfies D demands with congestion 1. The only differences of the 3-approximation algorithm for demand maximization from Algorithm 3.6 are a modified version of sink deactivation and a post-processing procedure. More generally, as in [22], we introduce an extra parameter κ to sink deactivation, giving Procedure 3.7. Note that sink deactivation is called *pivoting* in [22]. For instance, the 3-approximation algorithm for demand maximization uses $\kappa = \frac{1}{2}$, and the clustering algorithm of Seguin-Charbonneau and Shepherd [22] uses $\kappa \geq 1$. Moreover, it is conjectured by Seguin-Charbonneau and Shepherd [22] that $\kappa = -1$ may be promising for the open question of routing all demands in a constant number of confluent rounds, in which case $f(t_j) - f(v, t_j) > -1$ and hence the deactivation branch is always executed.

```
1 if $f(t_j) - f(v, t_j) \leq \kappa$ then
2     redirect flow on $(v, t_\ell)$ to $(v, t_j)$
3     remove edge $(v, t_\ell)$
4 else
5     redirect flow no $(v, t_j)$ to $(v, t_\ell)$
6     remove edge $(v, t_\ell)$
7     deactivate $t_j$
```

Procedure 3.7: `DeactivateSink`$_\kappa(v, t_j, t_\ell)$, where t_j is a remote sink whose only in-neighbour is v which has another sink out-neighbour t_ℓ.

As discussed in [22], this parametrized version of sink deactivation is based upon the intuition that if the congestion at a sink is "small", then it is not deactivated; if the congestion exceeds $\kappa + 1$, then it is deactivated, and its flow is redirected to some other sink. Therefore, a sink is either deactivated or has small congestion at most $\kappa + 1$.

The second modification of the 3-approximation algorithm for demand maximization is a post-processing procedure that selects which demands to satisfy for each of the sink trees after obtaining the confluent flow. Let $T_1, \ldots, T_k$ be the sink trees corresponding to $t_1, \ldots, t_k$, respectively. Let $\hat{b}_i$ be the total demand in T_i for each $i \in [k]$. For each sink tree T_i, we proceed as follows based upon the total demand $\hat{b}_i$ in T_i:

Case 1. If $\hat{b}_i \leq 1$, then routing all demands in T_i does not make the congestion exceed 1. Therefore, the total demand routed in T_i is $\hat{b}_i$.

Case 2. If $1 < \hat{b}_i \leq \frac{3}{2}$, then we *greedily* partition the demands into groups such that each group has total demand at most $\frac{2}{3}\hat{b}_i$, and then route the demands in the group with the maximum total demands. This greedy partition procedure is given in Procedure 3.8. We denote $d(U) = \sum_{u \in U} d(u)$.

```
1 $\mathcal{U} \leftarrow \varnothing$
2 **while** *not all vertices $v$ in $T_i$ with $d(v) > 0$ have been marked* **do**
3     $U \leftarrow \varnothing$
4     **while** $\exists$ *unmarked $v \in V(T_i)$ with $d(v) > 0$, $d(U \cup \{v\}) \leq \frac{2}{3}\hat{b}_i$* **do**
5         $U \leftarrow U \cup \{v\}$
6         mark $v$
7     $\mathcal{U} \leftarrow \mathcal{U} \cup \{U\}$
8 **return** $U \in \mathcal{U}$ *such that $\sum_{u \in U} d(u)$ is maximized*
```

Procedure 3.8: `GreedyPartition`(T_i).

Procedure 3.8 selects a subset of the demands in T_i whose total amount is at least $\frac{1}{2}\hat{b}_i$, while the congestion of T_i does not exceed 1.

Lemma 3.7. *Procedure 3.8 applied to a sink tree T_i with $1 < \hat{b}_i \leq \frac{3}{2}$ returns a subset U of the demands in T_i with $\frac{1}{2}\hat{b}_i \leq d(U) \leq 1$.*

Proof. Suppose that the groups in the paritition $\mathcal{U}$ have demands $d_1, \ldots, d_r$ with $d_1 \leq \ldots \leq d_r$, respectively. First, we observe that $d_j \geq \frac{1}{3}\hat{b}_i$ for all $j \geq 2$; otherwise, we would have that $d_1 \leq d_2 < \frac{1}{3}\hat{b}_i$ and hence that $d_1 + d_2 < \frac{2}{3}\hat{b}_i$, so the groups corresponding to d_1 and d_2 would have been combined. Similarly, $d_1 + d_r > \frac{2}{3}\hat{b}_i$; otherwise, we would have that $d_1 + d_r \leq \frac{2}{3}\hat{b}_i$, so the groups corresponding to d_1 and d_2 would have been combined. This implies that $r = 2$ because otherwise we would have that $d_1 + d_2 + d_r > \frac{2}{3}\hat{b}_i + \frac{1}{3}\hat{b}_i = \hat{b}_i$, a contradiction. Thus, $d_1 + d_r = \hat{b}_i$ and $d_1 \leq d_r$. This shows that $d_r \geq \frac{1}{2}\hat{b}_i$. Moreover, $d_r \leq \frac{2}{3}\hat{b}_i \leq \frac{2}{3} \cdot \frac{3}{2} \leq 1$. □

Case 3. If $\hat{b}_i > \frac{3}{2}$, then we use another greedy procedure to select a subset of the demands in T_i whose total amount is at least $\frac{1}{2}$. We give this procedure in Procedure 3.9.

It is straightforward to see that Procedure 3.9 selects a subset of the demands in T_i whose total amount is at least $\frac{1}{2}$, and routing which does not make the congestion exceed 1.

```
$U \leftarrow \varnothing$
foreach vertex $v \in V(T_i)$ with $d(v) > 0$ do
    if $d(v) < \frac{1}{2}$ then
        $U \leftarrow U \cup \{v\}$
        if $d(U) \geq \frac{1}{2}$ then
            return $U$
    else
        return $\{v\}$
```

Procedure 3.9: `GreedySelect`(T_i).

Lemma 3.8. *Procedure 3.9 applied to a sink tree T_i with $\hat{b}_i > \frac{3}{2}$ returns a subset U of the demands in T_i with $\frac{1}{2} \leq d(U) \leq 1$.*

Proof. If there exists $v \in V(T_i)$ with $d(v) \geq \frac{1}{2}$, then $\{v\}$ is returned when the loop visits v, and $d(\{v\}) \geq \frac{1}{2}$. Otherwise, $d(v) < \frac{1}{2}$ for all $v \in V(T_i)$. Since $\hat{b}_i > \frac{3}{2}$, then there exists a subset of the demands in T_i with total demand at least $\frac{1}{2}$. Moreover, if $d(U) > 1$, then removing any vertex v from U would give that $d(U \setminus \{v\}) > 1 - \frac{1}{2} = \frac{1}{2}$, so the last vertex added to U would not have been added, a contradiction. Therefore, $\frac{1}{2} \leq d(U) \leq 1$. □

```
for $i \leftarrow 1, \ldots, k$ do
    if $\hat{b}_i \leq 1$ then
        route all of the demands in $T_i$
    else if $\hat{b}_i \leq \frac{3}{2}$ then
        $U \leftarrow$ GreedyPartition$(T_i)$
        route all of the demands in $U$
    else
        $U \leftarrow$ GreedySelect$(T_i)$
        route all of the demands in $U$
```

Procedure 3.10: `PostProcess`$(T_1, \ldots, T_k)$.

The entire post-processing procedure is given in Procedure 3.10, and the 3-approximation algorithm for demand maximization on confluent flows is given in Algorithm 3.11. By Lemma 3.7 and Lemma 3.8 with the trivial first case, selecting demands to route in each sink tree according to the above

```
1 while $V(G) \neq S$ do
2     do one of the following three operations whenever possible:
3     (a) Aggregate$(v, t_i)$ for some frontier vertex $v$
4     (b) EliminateSawtooth$(C, f)$ for some sawtooth cycle $C$
5     (c) DeactivateSink$_{1/2}(v, t_j, t_\ell)$ for some remote sink $t_j$
6 $T_1, \ldots, T_k \leftarrow$ sink trees corresponding to $t_1, \ldots, t_k$, respectively
7 PostProcess$(T_1, \ldots, T_k)$
```

Algorithm 3.11: A 3-approximation algorithm for demand maximization on confluent flows.

three cases gives the congestion at most 1 and a total demand of at least

$$\sum_{\substack{i\in[k]\\ \hat{b}_i\le 1}} \hat{b}_i + \frac{1}{2}\sum_{\substack{i\in[k]\\ 1\le\hat{b}_i\le\frac{3}{2}}} \hat{b}_i + \sum_{\substack{i\in[k]\\ \hat{b}_i>\frac{3}{2}}} \frac{1}{2} \tag{3.4}$$

It then remains to show that (3.4) achieves an approximation ratio of 3 for demand maximization, i.e., that the value of (3.4) is at least $\frac{D}{3}$. Chen et al. [2] prove this using a "pictorial" approach by plotting the congestions of the sinks as a histogram.

Lemma 3.9. *Let $\hat{b}_i$ be the total demand (i.e., the congestion) of T_i for each $i \in [k]$. Then we have that*

$$\sum_{\substack{i\in[k]\\ \hat{b}_i\le 1}} \hat{b}_i + \frac{1}{2}\sum_{\substack{i\in[k]\\ 1<\hat{b}_i\le\frac{3}{2}}} \hat{b}_i + \sum_{\substack{i\in[k]\\ \hat{b}_i>\frac{3}{2}}} \frac{1}{2} \ge \frac{D}{3}. \tag{3.5}$$

Proof. First, we show that

$$2\sum_{\substack{i\in[k]\\ \frac{1}{2}<\hat{b}_i\le 1}} \hat{b}_i + \frac{1}{2}\sum_{\substack{i\in[k]\\ 1<\hat{b}_i\le\frac{3}{2}}} \hat{b}_i \ge \sum_{\substack{i\in[k]\\ \hat{b}_i>\frac{3}{2}}} \left(\hat{b}_i - \frac{3}{2}\right). \tag{3.6}$$

We plot as a histogram the congestion $f(t_i)$ at each sink t_i in the shrunken graph. The conservation of flow implies that the total area of the histogram is always D. Lemma 3.3 and Lemma 3.4 imply that vertex aggregation and sawtooth cycle elimination do not change the congestion at any sink. Hence, the histogram only changes during sink deactivation.

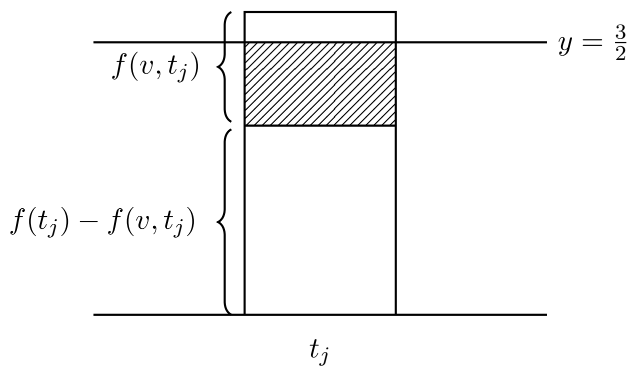


Figure 3.3: An illustration of the histogram of the congestion at a sink. The area of the shaded region gives an upper bound of the increase of A.

We divide the histogram into two parts by the horizontal line $y=\frac{3}{2}$. Let A be the total area of the upper part. Note that flow redirection in sink deactivation corresponds to moving of area in the histogram. We have the following two cases:

Case 1. If $f(t_j)-f(v,t_j)\le\frac{1}{2}$, then we have that

$$f\left(t_j\right)+f\left(v,t_\ell\right)\le\frac{1}{2}+f\left(v,t_j\right)+f\left(v,t_\ell\right)\le\frac{1}{2}+f(v)\le\frac{1}{2}+1=\frac{3}{2}.$$

Hence, A does not increase.

Case 2. If $f(t_j)-f(v,t_j)>\frac{1}{2}$, then t_j is deactivated according to $\texttt{DeactivateSink}_{1/2}(v,t_j,t_\ell)$, implying that $\hat{b}_j=f(t_j)-f(v,t_j)$. Pictorially, the increase of A is at most the area of $f(v,t_j)$ below the horizontal line $y=\frac{3}{2}$ in Figure 3.3. Therefore, we have the following sub-cases: *Case 2(a).* If $\frac{1}{2}<f(t_j)-f(v,t_j)\le 1$, then the increase of A is at most

$$\frac{3}{2}-\left(f\left(t_j\right)-f\left(v,t_j\right)\right)<\frac{3}{2}-\frac{1}{2}=2\cdot\frac{1}{2}<2\left(f\left(t_j\right)-f\left(v,t_j\right)\right)=2\hat{b}_j.$$

Case 2(b). If $1<f(t_j)-f(v,t_j)\le\frac{3}{2}$, then the increase of A is at most

$$\frac{3}{2}-\left(f\left(t_j\right)-f\left(v,t_j\right)\right)<\frac{3}{2}-1=\frac{1}{2}<\frac{1}{2}\left(f\left(t_j\right)-f\left(v,t_j\right)\right)=\frac{1}{2}\hat{b}_j.$$

Case 2(c). If $f(t_j)-f(v,t_j)>\frac{3}{2}$, then A does not increase.

On the other hand, the total increase of A after the algorithm is given by the area of the upper part of the histogram, i.e., $\sum_{i:\hat{b}_i>3/2}(\hat{b}_i-\frac{3}{2})$. Therefore, combining the two cases above yields (3.6). Therefore, we have that

$$2\sum_{\substack{i\in[k]\\ \hat{b}_i\le 1}}\hat{b}_i+\frac{1}{2}\sum_{\substack{i\in[k]\\ 1<\hat{b}_i\le\frac{3}{2}}}\hat{b}_i\ge 2\sum_{\substack{i\in[k]\\ \frac{1}{2}<\hat{b}_i\le 1}}\hat{b}_i+\frac{1}{2}\sum_{\substack{i\in[k]\\ 1<\hat{b}_i\le\frac{3}{2}}}\hat{b}_i\ge\sum_{\substack{i\in[k]\\ \hat{b}_i>\frac{3}{2}}}\left(\hat{b}_i-\frac{3}{2}\right). \tag{3.7}$$

Since the total demand is D by definition, then we have that

$$\sum_{\substack{i\in[k]\\ \hat{b}_i\le 1}}\hat{b}_i+\sum_{\substack{i\in[k]\\ 1<\hat{b}_i\le\frac{3}{2}}}\hat{b}_i+\sum_{\substack{i\in[k]\\ \hat{b}_i>\frac{3}{2}}}\hat{b}_i=D. \tag{3.8}$$

Therefore, adding (3.7) and (3.8) and dividing both sides of the resulting inequality by 3 yields (3.5). □

Combining the lemmas stated and proved in this section gives the next theorem which asserts the correctness of Algorithm 3.11.

Theorem 3.10. *Given a fractional flow with vertex congestion* 1 *and total demand D, Algorithm 3.11 finds a confluent flow with vertex congestion at most* 1 *which satisfies a subset of demands with total amount at least* $\frac{D}{3}$.

3.3 An H_k Lower Bound

The simplicity of confluent flows is a mixed blessing. While confluent flows have simple and omnipresent tree-shaped support network, they suffer from an unbounded lower bound; in other words, Chen et al. [2] show an instance of a graph G which admits a fractional flow with congestion 1, but which does not admit a confluent flow with congestion less than H_k, where H_k is the k^{th} harmonic number. Since $H_k = \ln k + \gamma - o(1)$, where $\gamma \approx 0.577$ is Euler's constant, then the upper bounds of $1 + \lg k$ and of $1 + \ln k$ presented in Section 3.1 are almost tight within a multiplicative constant less than 2 and an additive constant less than 1, respectively.

We give such an instance in Figure 3.4. Let $k \in \mathbb{N}$. Let G be a directed graph with $\frac{k(k+1)}{2}$ sources of the form (i, j) with $i, j \in [k]$ and $i \le j$, and k sinks of the form t_i for $i \in [k]$. Each (i, j) with $i, j \in [k]$ and $i \le j < k$ has two outgoing edges to $(i, j+1)$ and to $(i+1, j+1)$, respectively. Moreover, each (i, k) with $i \in [k]$ has an outgoing edge to t_i. The demand at each source (i, j) is $\frac{1}{j}$ for $i, j \in [k]$ and $i \le j$. Intuitively, this instance consists of a triangle of vertices where each vertex not in the last row sends flow to the two vertices immediately below it. We have the following two lemmas

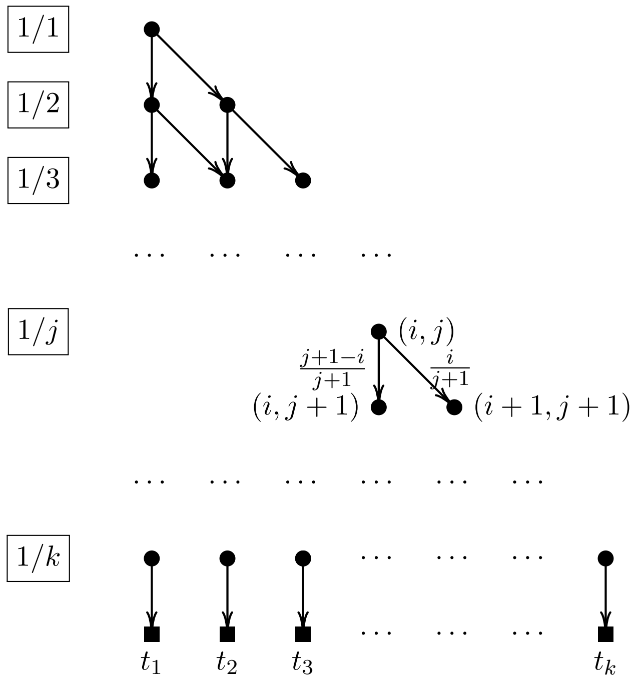


Figure 3.4: An instance which admits a fractional flow that satisfies all demands with vertex congestion 1, yet which does not admit a confluent flow that satisfies all demands with vertex congestion less than H_k, the k^{th} harmonic number.

regarding fractional flows and confluent flows, which give an H_k gap between fractional flows and confluent flows.

Lemma 3.11. *There exists a fractional flow on G that satisfies all demands with vertex congestion* 1.

Proof. For each source (i, j) with $i, j \in [k]$ and $i \leq j$, we send $\frac{j+1-i}{j+1}$ units of flow from (i, j) to $(i, j+1)$, and $\frac{i}{j+1}$ units of flow from (i, j) to $(i+1, j+1)$. Thus, the total outgoing flow from each source (i, j) with $i, j \in [k]$ and $i \leq j$ equals $\frac{j+1-i}{j+1} + \frac{i}{j+1} = 1$.

It suffices to prove by induction on $j \in [k]$ that the congestion at each source (i, j) is 1 for each $i \in [j]$. Since the demand at $(1, 1)$ is 1 and $(1, 1)$ has no incoming edges, then the congestion at $(1, 1)$ is 1, proving the base case. Let $j \in \{2, \ldots, k\}$. Suppose by induction that the congestion at (i, j) is 1 for each $i \in [j-1]$. Let $i \in [j]$. Note that (i, j) has incoming edges from $(i, j-1)$ and $(i-1, j-1)$ with $\frac{j-i}{j}$ units of flow and $\frac{i-1}{j}$ units of flow, respectively. Note also that the demand at (i, j) is $1/j$. Therefore, the congestion at (i, j) is $\frac{1}{j} + \frac{j-i}{j} + \frac{i-1}{j} = 1$, which equals its total outgoing flow. This completes the induction step. $\square$

Lemma 3.12. *Any confluent flow on G that satisfies all demands has vertex congestion at least H_k.*

Proof. Note that any confluent that satisfies all demands must satisfy the demand at $(1, 1)$. The confluence implies that the flow must be routed along a path from $(1, 1)$ to (i, k) for some $i \in [k]$, passing through one vertex from each "layer" $j = 1, \ldots, k$ with demands $1, \frac{1}{2}, \ldots, \frac{1}{k}$, respectively. Since all of the demands along this path must be sent to the sink t_i, then the congestion at t_i equals $1 + \frac{1}{2} + \ldots + \frac{1}{k} = H_k$. $\square$

Chapter 4

Bi-/d-furcated Flows

The earth expanding right hand and left hand,
The picture alive, every part in its best light,
The music falling in where it is wanted, and stopping where it is not wanted,
The cheerful voice of the public road, the gay fresh sentiment of the road.

— Walt Whitman, *Song of the Open Road*

As discussed in Chapter 1, one of the chief motivations for the confluent network flow model is IP routing, whose strength derives from its low cost and operational simplicity. The simplicity of IP routing is reflected in its routing protocol described as follows. In particular, with an IP router is associated a *next hop table* which includes an entry *next*(t) denoting the single *next hop* for each possible IP address t. In other words, each possible IP address has a single next hop from an IP router.

On the other hand, nevertheless, the simplicity of IP routing also entails several burdens of the confluent network flow model. For instance, we have shown in Chapter 3 that there exist instances in which confluent flows suffer from an $\Omega(\log k)$ congestion grow, and Chen et al. [2] also prove that it is NP-hard to approximate the congestion of an optimal confluent flow to within a factor of $\frac{\lg k}{2}$. This is an almost tight lower bound which matches its $(1 + \ln k)$-upper bound. Such an unbounded lower bound puts the confluent network flow model in a less satisfactory position for practical applications.

A natural idea to possibly break this $\Omega(\log k)$ lower bound of the confluent flow model while preserving its simplicity to a large extent is to allow an IP router to send outgoing requests to two next hops, or d next hops, for each possible IP address. In their 2004 paper, Chen et al. [2] ask the following question about such a variation:

> Another variation is to allow the flow leaving a vertex to use a constant number of outgoing edges, instead of one outgoing edge in the confluent setting. For instance, one interesting question is

the following: if the graph admits a splittable flow of congestion 1, can we always achieve congestion $O(1)$ using a flow in which every vertex uses only $O(1)$ outgoing edges.

It is quite surprising that very little has been studied for bifurcated and d-furcated flows. Therefore, we primarily study the congestion minimization problem of such degree constraint relaxations of the confluent flow model in this chapter. In particular, we focus on the paper of Donovan et al. [7] for its $(1 + \frac{1}{d-1})$-congestion algorithm, and perhaps more interestingly, for a structural theorem which gives a characterization of sawtooth-cycle-free directed graphs. We say that a flow is *bifurcated* if each vertex can send flow to at most two outgoing edges, and more generally, that a flow is *d-furcated* if each vertex can send flow to at most d outgoing edges. Moreover, in the congestion minimization problem of d-furcated flows, we consider uniform vertex capacities (i.e., $c(e) = 1$ for each $e \in E$) and assume that there is a single sink t. By the algorithm given in Section 2.6, we assume that there exists a fractional flow satisfying all demands and capacity constraints at all vertices. It turns out, surprisingly, that even allowing one additional next hop (i.e., a bifurcated flow) drastically reduces the $\Theta(\log k)$ congestion to a constant congestion. In addition, we assume that the support network of the optimal fractional flow is acyclic by repeatedly applying `EliminateCycle`(C, f) defined in Procedure 3.1 whenever we find a directed cycle C.

4.1 Edge Contractions and Sawtooth Cycles

We introduce two operations used in the algorithm of Donovan et al. [7]. First, we *contract* edges so that each edge has out-degree either 0 or at least 2. This operation is defined in Procedure 4.1. One can view the edge contraction operation as shrinking the edge (v, w) and aggregating the two vertices v and w. Therefore, in a shrunken graph, an edge may corresponding to multiple edges in the original graph. It is straightforward to verify that performing `Contract`(v) gives an equivalent instance. We note that edge contractions can be regarded as a generalized version of vertex aggregation operations in the approximation algorithms for confluent flows, in the sense that vertex aggregations only aggregate a frontier vertex and a sink, whereas edge contractions may happen between any vertex of out-degree 1 and its only out-neighbour.

Lemma 4.1. *The congestion at each vertex is not changed after performing* `Contract`(v).

```
1 if $v$ has out-degree 1 then
2     $w \leftarrow$ only out-neighbour of $v$
3     $d(v) \leftarrow d(v) + d(w)$
4     foreach edge $e$ from $w$ do
5         set the tail of $e$ to be $v$
```

Procedure 4.1: `Contract`(v).

We have already seen that the notion of sawtooth cycles and the operation of breaking sawtooth cycles play an essential role in the $(1 + \ln k)$-congestion algorithm of Chen et al. [2]. Recall that a *sawtooth cycle* in [2] is defined as a cycle in an auxiliary graph $\hat{G}$ obtained from G by adding a reverse edge for each edge from a frontier vertex to a sink. In other words, a sawtooth cycle in [2] may only have "reverse" edges from frontier vertices to sinks. Donovan et al. [7] generalize this strict notion of sawtooth cycles by defining a *(general) sawtooth cycle* to be a collection

$$C = \{(u_0, v_0), P_0, (u_1, v_1), P_1, \ldots, (u_{r-1}, v_{r-1}), P_{r-1}\},$$

where (u_i, v_i) is an edge and P_i is a u_{i+1}-v_i directed path in the support graph of f for $i \in \{0, \ldots, r-1\}$, with subscripts modulo r. We use "sawtooth cycles" and "general sawtooth cycles" interchangeably in this chapter.

Sawtooth cycles are not wanted in the $(1 + \frac{1}{d-1})$-congestion d-furcated flow algorithm of Donovan et al. [7]; indeed, in subsequent paragraphs we present a graph-theoretic characterization of directed graphs that do not have sawtooth cycles. As in the approximation algorithms for confluent flows presented in Chapter 3, a common operation in the approximation algorithm of Donovan et al. [7] is the operation of eliminating sawtooth cycles, given in Procedure 3.3. It is straightforward that the following two lemmas hold for the operation of eliminating sawtooth cycles.

Lemma 4.2. *If the congestion at each vertex is bounded by* 1*, performing* `EliminateSawtooth`(C, f) *guarantees that the congestion at each vertex in the original graph is still bounded by* 1*.*

Lemma 4.3. *Performing* `EliminateSawtooth`(C, f) *for some sawtooth cycle C eliminates C, i.e., C is no longer a sawtooth cycle.*

We note that, due to edge contraction operations, unshrinking a vertex may lead to a directed path in the original graph whose internal vertices have out-degrees 1. Hence, it is not entirely trivial to see that the congestion at each of these internal vertices is still bounded by 1 after performing `EliminateSawtooth`(C, f). However, we observe that the congestions at the endpoints of such a directed path must be at least the congestion of any internal vertex. Therefore, Lemma 4.2 holds.

Finally, we note that performing `Contract`(v) may lead to new sawtooth cycles, and that performing `EliminateSawtooth`(C, f) may lead to new vertices with out-degrees 1. However, each edge contraction reduces the number of vertices by 1 in the graph, and each sawtooth cycle elimination reduces the number of edges by 1 in the support graph. The following lemma therefore follows.

Lemma 4.4. *If we perform* `Contract`(v) *and* `EliminateSawtooth`(C, f) *whenever possible, then within a polynomial number of steps, we obtain a directed graph and a flow on it whose support graph is acyclic and contains neither sawtooth cycles nor vertices of out-degrees* 1.

4.2 Acyclic Digon-Tree Representations

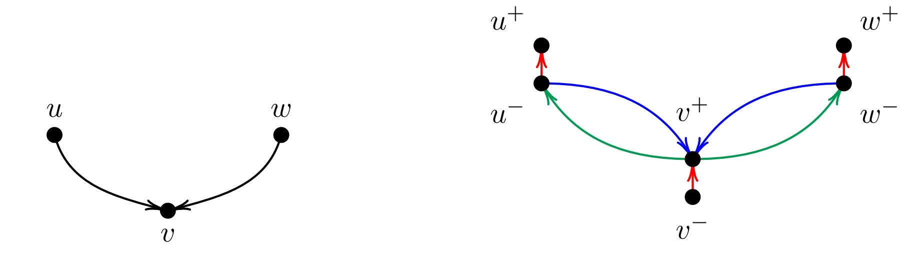
Having defined the operation of eliminating sawtooth cycles in a directed graph, two question that naturally arises is the following: What does a directed graph look like if it does not contain sawtooth cycles? How does it help with algorithm design for d-furcated flows?

It is indeed very intriguing that a directed graph with no sawtooth cycles admits a very nice "layered" structure, called an *acyclic digon-tree representation*, that naturally inspires an algorithm that processes the vertices in a layered order (i.e., from sources to sinks, or from sinks to sources).

To describe this characterization, we first associate with each directed graph G an *auxiliary graph* $\hat{G} = (V^+ \cup V^-, \hat{E})$ defined as follows. For each vertex $v \in V$, we add two copies v^+ and v^- to V^+ and V^-, respectively. The edge set $\hat{E}$ of $\hat{G}$ contains three types of deges:

- For $v \in V$, we add (v^-, v^+) to $\hat{E}$, called a *vertex edge.*
- For $(v, w) \in E$, we add (v^-, w^+) to $\hat{E}$, called a *real edge.*
- For $(v, w) \in E$, we add (w^+, v^-) to $\hat{E}$, called a *complementary edge.*

According to this construction, $\hat{G}$ is bipartite. In graph theory, a *digon* is a pair of two edges that form a simple directed cycle of length 2. Therefore, according to the construction above, an edge in the original graph G corresponds to a digon in its auxiliary graph $\hat{G}$. A comparison between the original graph G and its auxiliary graph $\hat{G}$ is given in Figure 4.1.



(a) The original graph G. (b) The auxiliary graph $\hat{G}$.

Figure 4.1: A comparison between the original directed graph G and its auxiliary graph $\hat{G}$, where red, blue and green edges denote vertex, real and complementary edges, respectively.

The first graph-theoretic characterization of Donovan et al. [7] asserts that whether a directed graph contains sawtooth cycles or not can be determined by the maximum length of a directed cycle in the auxiliary graph. We note, however, that this equivalence is not very practical in algorithm design in the sense that the problem of finding a longest cycle in a directed graph is NP-hard, and this hardness result can be shown by a reduction from the *Hamiltonian cycle problem.* An even stronger hardness result, due to Björklund et al. [1], states that the problem of finding a longest cycle cannot be approximated within a factor of $n^{1-\varepsilon}$ in polynomial time for any $\varepsilon > 0$ unless P = NP.

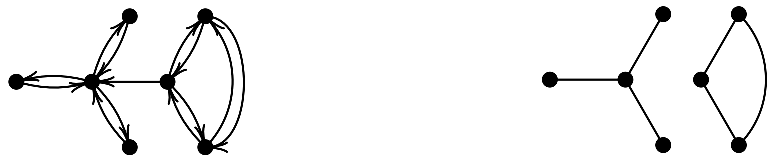
Lemma 4.5. *A directed acyclic graph G contains a sawtooth cycle if and only if the auxiliary graph $\hat{G}$ contains a simple circle of length at least 3 (and hence at least 4 since $\hat{G}$ is bipartite).*

Proof. ($\Longleftarrow$) Let $C = (v_1^-, v_2^+, v_3^-, v_4^+, \ldots, v_{2r}^+)$ be a directed cycle in $\hat{G}$ of length at least 4 for some $v_1, \ldots, v_{2s} \in V$. We observe the following patterns of edges in the auxiliary graph $\hat{G}$:

- a vertex edge is followed by a complementary edge;
- a real edge is followed by a complementary edge;
- a complementary edge is followed by a real edge or a vertex edge.

Therefore, we have the following two cases:

- C has alternating complementary and vertex edges. This, however, corresponds to a cycle in G, contradicting the acyclicity of G.



(a) The directed graph D. (b) The undirected graph $\tilde{D}$.

Figure 4.2: An example of an undirected graph $\tilde{D}$ edge-induced by the digons in a directed graph D. In this example, D does not admit an acyclic digon-tree representation because $\tilde{D}$ is not a forest.

- $E(C)$ can be partitioned into blocks of a real edge followed by an odd, alternating path of complementary and vertex edges. Each real edge corresponds to an edge in G, and each odd, alternating path of complementary and vertex edges corresponds to the reverse of a path in G. Therefore, C corresponds to a sawtooth cycle in G.

($\Longrightarrow$) Let C be a sawtooth cycle in G of the form (3.1). Then each edge (u_i, v_i) corresponds to a real edge in $\hat{G}$, and the reverse each directed path P_i corresponds to an odd, alternating path of complementary and vertex edges, which has length at least 2. Therefore, C corresponds to a directed cycle of length at least 3 in $\hat{G}$. $\square$

While the above characterization of sawtooth cycles in terms of directed cycles in the auxiliary graph is not directly practical in algorithm design, it bridges the gap between sawtooth cycles and *digon-tree representations*, which we define below. Let D be a directed graph such that the underlying undirected graph $\tilde{D}$ *edge-induced* by the digons of D is a forest. We say that the subgraph of D corresponding to each tree in $\tilde{D}$ is a *digon tree*. That is, there exists a bijection between the digons of D and the edges of $\tilde{D}$. We define the *digon-tree representation* $\mathcal{D}$ of D to be obtained by contracting each digon of D. In other words, each forest in $\tilde{D}$ corresponds to a vertex in $\mathcal{D}$, each inter-tree edge corresponds to an edge between the associated vertices, and each intra-tree edge corresponds to a self-loop at the associated vertex. A vertex in $\mathcal{D}$ is called a *digon-tree vertex*. A vertex that is not the endpoint of a digon in D corresponds to a *singleton digon-tree veretx* in $\mathcal{D}$.

The second half of the acyclic digon-tree representation characterization connects the maximum length of a cycle in a directed acyclic graph and the acyclicity of its digon-tree representation. This result first appears in [25].

Lemma 4.6. *Let D be a directed graph with no self-loops. Then D has no directed cycles of length at least 3 if and only if it has an acyclic digon-tree representation.*

Proof. ($\Longrightarrow$) Suppose that D has no directed cycles of length at least 3. If $\tilde{D}$ contained a cycle (of length at least 2), then it would correspond to a cycle of length at least 4 in D, a contradiction. Hence, $\tilde{D}$ is a forest, so D has a digon-tree representation $\mathcal{D}$.

Suppose for the sake of contradiction that $\mathcal{D}$ contains a self-loop e at a digon-tree vertex T. Since e cannot correspond to an edge in a digon (otherwise it would be contracted), then e corresponds to an edge (u, v) in D for some u, v non-adjacent in T. Then (u, v) and the digons corresponding to the edges in the unique path between u and v in T contain a directed cycle of length at least 3 (which consists of (u, v) and one of the two edges in each digon), a contradiction.

Suppose for the sake of contradiction that $\mathcal{D}$ contains a directed cycle $\mathcal{C} = (T_0, \ldots, T_{r-1})$ for some $r \geq 2$. Then for each $i \in \{0, \ldots, r-1\}$, there exists $(u_i, v_i) \in E(D)$ such that $u_i \in V(T_i), v_i \in V(T_{i+1})$, with subscripts modulo r. Moreover, for each $i \in \{0, \ldots, r-1\}$, either $v_{i-1} = u_i$, or there exists a directed v_{i-1}-u_i path, with subscripts modulo r. This implies that there exists a directed cycle C in D. Note that C cannot be a digon, because otherwise C would correspond to an edge in $\tilde{D}$, which will be contracted in $\mathcal{D}$, and hence fail to correspond to an edge in $\mathcal{D}$. Therefore, C is a directed cycle of length at least 3 in D, a contradiction.

($\Longleftarrow$) Let $\mathcal{D}$ be an acyclic digon-tree representation. Suppose for the sake of contradiction that D contains a directed cycle $C = (v_1, \ldots, v_r)$ of length at least 3. If $v_1, \ldots, v_r \in V(T)$ for some tree T in $\tilde{D}$, then there would exist an edge e in C that is not an edge in T (since T is acyclic), but then e would be an intra-tree edge in D and hence correspond to a self-loop in $\mathcal{D}$, a contradiction. Therefore, there exist $T_1, \ldots, T_s \in V(\mathcal{D})$ visited by C in that order for some $s \geq 2$. This implies that $(T_1, \ldots, T_s)$ is a directed cycle in $\mathcal{D}$, a contradiction. $\square$

Since the auxiliary graph $\hat{G}$ of a direct graph G does not contain self-loops by our construction, then combining Lemma 4.5 and Lemma 4.6 gives the following nice characterization of sawtooth-cycle-free direct acyclic graphs in terms of digon-tree representations.

Theorem 4.7. *A directed acyclic graph G contains sawtooth cycles if and only if its auxiliary graph $\hat{G}$ has an acyclic digon-tree representation.*

4.3 A Layered Structure of Digon Trees

According to Theorem 4.7, we obtain an acyclic digon-tree representation after performing sawtooth cycle eliminations. How would such a structural characterization guide us to devise an algorithm? A benefit of an acyclic digon-tree representation is a layered structure of digon trees, which allows us to design an algorithm that processes vertices in an acyclic ordering (which can be regarded as a generalized topological ordering), redirecting flow to transform a fractional flow to a d-furcated flow.

Theorem 4.8. *Let G be a directed acyclic graph with no sawtooth cycles. Then the underlying undirected graph $\tilde{G}$ is the union of vertex-induced trees of $\tilde{G}$ such that*

- *any vertex v is contained in exactly two of the trees (which possibly correspond to a singleton digon-tree vertex);*
- *all outgoing edges from any vertex v are contained in the same tree;*
- *all incoming edges into any vertex v are contained in the same tree.*

Proof. By Theorem 4.7, the auxiliary graph $D = \hat{G}$ has an acyclic digon-tree representation $\mathcal{D}$. Since each digon in D corresponds to an edge in G, then each digon tree in $\tilde{D}$ corresponds to a tree in the underlying undirected graph $\tilde{G}$ of G. Therefore, each vertex v is contained in at most two trees corresponding the two digon trees containing v^+ and v^- in D.

We note that v^+, v^- cannot be contained in the same digon tree for each vertex $v \in V(G)$, because otherwise (v^-, v^+) would correspond to a self-loop in the digon-tree representation $\mathcal{D}$ of $\hat{G}$, contradicting its acyclicity.

We show that each digon tree in D corresponds to an induced tree in G. Let $\hat{T}$ be a digon tree in D, with T the corresponding tree in $\tilde{G}$. Suppose for the sake of contradiction that there exists $uv \in E(\tilde{G}) \setminus E(T)$ such that $u, v \in V(T)$. This implies that the digon between u^- and v^+ is not contained in T, so this digon is contained in some other digon tree $T' \neq T$. Note that the vertex edges (v^-, v^+) and (u^-, u^+) form a directed cycle between the digon-tree vertices T, T_2' in $\mathcal{D}$, contradicting the acyclicity of $\mathcal{D}$.

Next, let $v \in V(G)$ of out-degree at least 1, and $(v, w) \in \delta_G^+(v)$. By the construction of D, there exists a digon between v^- and w^+, so this digon is contained in a digon tree in D. Therefore, v and w are contained in the same induced tree.

Similarly, let $v \in V(G)$ of in-degree at least 1, and $(u, v) \in \delta_G^-(v)$. By the construction of D, there exists a digon between u^- and v^+, so this digon is contained in a digon-tree in D. Therefore, v and w are contained in the same induced tree. □

Moreover, the construction of the auxiliary graph $\hat{G}$ of G implies that s^+ is not incident to a digon for each source s, and that t^- is not incident to a digon for each sink t. Given Theorem 4.8, the following corollary follows.

Corollary 4.9. *For each sink t in G, t^- is in a singleton digon-tree vertex in $\mathcal{D}$. For each source s in G, s^+ is in a singleton digon-tree vertex in $\mathcal{D}$.*

Finally, the construction of $\hat{G}$ ensures that the partition $\{V^+, V^-\}$ of $V(\hat{G})$ gives a two-colouring of each digon tree in $\hat{G}$. That is, the vertex set of each digon tree $\hat{T}$ in $\hat{G}$ can be partitioned into $X \cup Y$ such that $X \subseteq V^+$ and $Y \subseteq V^-$. By Theorem 4.8, we have the following corollary.

Corollary 4.10. *The vertex set of each digon tree $\hat{T}$ in $\hat{G}$ can be partitioned into $X \cup Y$ such that $X \subseteq V^+, Y \subseteq V^-$, and that X, Y correspond to disjoint subsets $X', Y' \subseteq V(G)$, respectively. Moreover, $\delta^+_G(X') = Y', \delta^-_G(Y') = X'$.*

To conclude this section, we discuss some intuition beyond the structural results proved in this section. Given a directed graph G, the acyclicity of the corresponding digon-tree representation $\mathcal{D}$ ensured by Theorem 4.7 naturally gives rise to a "source-to-sink" topological ordering of the digon-tree vertices in $\mathcal{D}$. If we process the digon trees in the auxiliary graph in this order, then each of the vertices from the V^- part of the partition in the current digon tree $\hat{T}$ correspond to a vertex from the V^+ part of the partition in some other digon tree $\hat{T}' \neq \hat{T}$. However, the corresponding vertex edge together with the topological ordering ensures that $\hat{T}'$ must have been processed before $\hat{T}$. Therefore, if we have a subroutine to process each digon tree to d-fucate flow, then we can redirect in the entire graph to obtain a d-furcated flow.

Inspired by this acyclic digon-tree representation characterization, the remaining task, therefore, is to devise a procedure to "d-furcate" a fractional flow, assuming that the vertices from the V^- part of the partition have already been processed with a congestion at most $\frac{1}{d-1}$. This gives rise to a proof by induction on the acyclic ordering of digon trees described above. In other words, we are able to redirect flow "locally," and the acyclic digon-tree representation theorem automatically gives us a "global" d-furcated flow. This demonstrates the power of the acyclic digon-tree representation characterization! We formalize this idea in the next section.

4.4 Local Flow Redirection

As discussed in Section 4.3, the final phase of the approximation algorithm of Donovan et al. [7] is to design a procedure to "*d*-furcate" flow in each digon tree locally. The next elementary lemma is at the centre of our procedure.

Lemma 4.11. *Let T be a tree with bipartition classes X and Y such that each vertex of Y has degree at least 2. Then there exists $s \in Y$ with at most one non-leaf neighbour in T.*

Proof. Let $X' \subseteq X$ be the set of non-leaf vertices in X. Then each vertex in X' has at least two neighbours in Y. Suppose for the sake of contradiction that each vertex in Y has at least two neighbours in X'. Hence, $X' \cup Y$ induces a bipartite graph with minimum degree 2, which therefore contains a cycle, contradicting the acyclicity of T. □

Now, we assume that G contains neither directed cycles nor sawtooth cycles, and that each vertex of G has out-degree at least 2. This assumption is guaranteed by directed cycle eliminations, sawtooth cycle eliminations and edge contractions introduced in Section 4.1. Theorem 4.7 implies that the auxiliary graph $\hat{G}$ of G has an acyclic digon-tree representation $\mathcal{D}$. Then there exists a "source-to-sink" topological ordering $(T_1, \ldots, T_k)$ of the digon-tree vertices in $\mathcal{D}$. That is, an edge (T, T') in $\mathcal{D}$ implies that T' must be before T in the topological ordering. We process the digon trees in $\hat{G}$ in this order, omitting singleton digon-tree vertices.

Let T be the current digon tree in $\hat{G}$ with bipartition classes $X \subseteq V^+$ and $Y \subseteq V^-$ given in Corollary 4.10. Let $v^- \in Y$. By Theorem 4.8, v^+ is contained in some other digon tree $T' \neq T$ (which may correspond to a singleton digon-tree vertex). Since (v^-, v^+) is a vertex edge in $\hat{G}$, then (T, T') is an edge in $\mathcal{D}$. The topological ordering of the digon-tree vertices guarantees that T' must have been processed before we process T. We summarize this discussion in the following lemma.

Lemma 4.12. *There exists an order of digon trees in $\hat{G}$ such that processing the digon trees in this order ensures that the currently processed digon tree T has the following property: For each v^- in the V^- part of the bipartition of $V(T)$, we have that v^+ is contained in an already processed digon tree.*

In other words, if we remove a digon tree and its outgoing edges in $\mathcal{D}$ immediately after it is processed, then the V^- part of the bipartition of the currently processed digon tree correspond to a subset of sources in G [7].

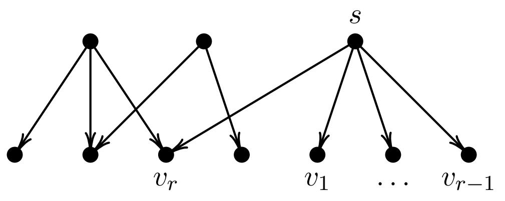


Figure 4.3: An illustration of flow redirection in the original graph, where s has at most one non-leaf neighbour v_r and $v_1, \dots, v_{r-1}$ are leaves in the underlying undirected graph.

We now describe flow redirection (i.e., d-furcation) in each digon tree by an inductive proof for the following theorem. We denote by $f(v)$ the congestion at a vertex v for some flow f on G.

Theorem 4.13. *Let f be a fractional flow with the congestion at each vertex at most 1. There exists a d-furcated flow such that the congestion at each vertex v is at most $f(v) + \frac{1}{d-1}$. In particular, there exists a d-furcated flow such that the congestion at each vertex is at most $1 + \frac{1}{d-1}$.*

Proof. Let T be the currently processed digon tree, with bipartition classes $X \subseteq V^+$ and $Y \subseteq V^-$. Let T' be the corresponding tree in the underlying undirected graph of G, with corresponding bipartition classes X', Y'. Since each vertex of G has out-degree at least 2, then each vertex in Y' has degree at least 2. Lemma 4.11 implies that there exists $s \in Y'$ with at most one non-leaf neighbour in T'. Let $v_1, \dots, v_r$ be the neighbours of s in T' such that $v_1, \dots, v_{r-1}$ are leaves. An illustration is given in Figure 4.3. Suppose by induction that the additional congestion at s is at most $\frac{1}{d-1}$. Then we have the following two cases:

Case 1. $r \le d$. We redirect the additional flow at s, which is at most $\frac{1}{d-1}$, from s to v_1. Since v_1 is a leaf in T', then Corollary 4.10 implies that the additional flow at v_1 is at most $\frac{1}{d-1}$.

Case 2. $r \ge d+1$. Then $v_1, \dots, v_d$ are leaves. Hence,

$$\sum_{j=d+1}^{r} f(s, v_j) \le \sum_{j=1}^{r} f(s, v_j) \le f(s) \le 1.$$

In this case, we redirect the flow on the edges (s, v_j) for $j \in \{d+1, \dots, r\}$, together with the additional flow which is at most $\frac{1}{d-1}$, equally on the edges $(s, v_1), \dots, (s, v_d)$. Hence, for each $j \in [d]$, the additional flow at v_j after the redirection is at most

$$\frac{1}{d}\left(\sum_{j=d+1}^{r} f\left(s, v_j\right) + \frac{1}{d-1}\right) \leq \frac{1}{d}\left(1 + \frac{1}{d-1}\right) = \frac{1}{d} \cdot \frac{d}{d-1} = \frac{1}{d-1}.$$

Since $v_1, \ldots, v_d$ are leaves in T, then the additional flows at $v_1, \ldots, v_d$ are at most $\frac{1}{d-1}$ by Corollary 4.10. $\square$

This completes the description of the $(1+\frac{1}{d-1})$-approximation algorithm of Donovan et al. [7] for the congestion minimization problem on d-furcated flows with $d \geq 2$. We summarize this algorithm in Algorithm 4.2, where f is a fractional flow on G with the congestion at any vertex at most 1.

4.5 β-Confluent Flows

As a digression, we demonstrate in this section how the approach of Donovan et al. [7] can be applied to design a constant-factor approximation algorithm for the congestion minimization problem on *β-confluent flows*, which can be regarded as a more restricted version of its counterpart on bifurcated flows. For $\beta \in [0, 1]$, we say that a flow is *β-confluent* if each vertex sends at least a β-fraction of its total incoming flow to a single outgoing edge, and all remaining flow to at most one other outgoing edge.

We have already seen in Chapter 3 and in this chapter that the vertex congestion of a bifurcated flow is at most 2, whereas the vertex congestion of a confluent flow can be as large as $\Omega(\log k)$ and hence unbounded. This gap is intriguing, and therefore it would be interesting to study the minimum congestion of a network flow "between" confluent flows and bifurcated flows. To that end, β-confluent flows can be regarded as more restricted than bifurcated flows, but looser than confluent flows. In particular, the case where $\beta = 0$ gives a confluent flow, and the case where $\beta = 1$ gives a bifurcated flow.

For $\beta \in (0, \frac{1}{2}]$, we note that a bifurcated flow is also a bifurcated flow. Consider a bifurcated flow. Let $v \in V$. Suppose that v sends a β'-fraction of its total incoming flow to a single outgoing edge, and the remaining $(1-\beta')$-fraction of its total incoming flow to one other outgoing edge, for some $\beta' \in [0, 1]$. Without loss of generality, we assume that $\beta' \geq 1 - \beta'$, which implies that $\beta' \geq \frac{1}{2} \geq \beta$. Therefore, this bifurcated flow is also a β-confluent flow. It follows that Algorithm 4.2 gives a 2-approximation for the congestion minimization problem on β-confluent flows with $\beta \in (0, \frac{1}{2}]$.

```
while $\exists$ a directed cycle $C$ in $G$ do
    $b \leftarrow \min\{f(e) : e \in E(C)\}$
    $f(e) \leftarrow f(e) - b$ for all $e \in E(C)$
while $\exists$ a sawtooth cycle or a vertex of out-degree $1$ in $G$ do
    if $\exists$ a sawtooth cycle $C$ in $G$ then
        EliminateSawtooth$(C, f)$
    else if $\exists$ a vertex of out-degree $1$ in $G$ then
        Contract$(v)$
$D \leftarrow$ auxiliary graph of $G$
$\mathcal{D} \leftarrow$ acyclic digon-tree representation of $D$
$(T_1, \dots, T_k) \leftarrow$ a "source-to-sink" topological ordering of $\mathcal{D}$
$\varepsilon(v) \leftarrow 0$ for all $v \in V$
for $i \leftarrow 1, \dots, k$ do
    if $T_i$ is a singleton digon-tree vertex in $\mathcal{D}$ then
        continue
    $\{X, Y\} \leftarrow$ bipartition classes of $T_i$ with $X \subseteq V^+, Y \subseteq V^-$
    choose $s^- \in V^-$ s.t. $s^-$ has $\leq 1$ non-leaf out-neighbour in $T_i$
    $(v_1^+, \dots, v_r^+) \leftarrow$ neighbours of $s^-$ in $T_i$ with $v_1^+, \dots, v_{r-1}^+$ leaves
    if $r \leq d$ then
        $f(s, v_1) \leftarrow f(s, v_1) + \varepsilon(s)$
        $\varepsilon(v_1) \leftarrow f(v_1) - 1$
    else
        $\Delta \leftarrow \sum_{j=d+1}^{r} f(s, v_j) + \varepsilon(s)$
        $f(s, v_j) \leftarrow f(s, v_j) + \Delta/d$ for all $j \in [d]$
        $\varepsilon(v_j) \leftarrow f(v_j) - 1$
```

Algorithm 4.2: A $(1 + \frac{1}{d-1})$-approximation algorithm for the congestion minimization problem on d-furcated flows with $d \geq 2$.

The following theorem, from the unpublished journal version of [7], gives a $(1+\frac{\beta}{1-\beta})$-approximation algorithm for the problem when $\beta \in (\frac{1}{2}, 1)$. The proof of this theorem resembles that of Theorem 4.13.

Theorem 4.14. *Let f be a fractional flow with the congestion at each vertex at most* 1. *For $\beta \in (\frac{1}{2}, 1)$, there exists a β-confluent flow such that the congestion at each vertex is at most $f(v) + \frac{\beta}{1-\beta}$. In particular, there exists a β-confluent flow such that the congestion at each vertex is at most $1 + \frac{\beta}{1-\beta}$.*

Proof. The proof only differs from that of Theorem 4.13 in the two cases. Suppose by induction that the additional congestion at s is at most $\frac{\beta}{1-\beta}$. Let T be a currently processed digon tree, with bipartition classes $X \subseteq V^+$ and $Y \subseteq V^-$. Let $s^- \in Y$ and $v_1^+, \ldots, v_r^+ \in X$ such that $v_1, \ldots, v_{r-1}$ are leaves in the corresponding tree T' in the underlying undirected graph of G. Suppose by induction that the additional congestion at s is at most $\frac{\beta}{1-\beta}$. Let ε be the amount of the additional flow at s.

Case 1. $r \leq 2$. We send the additional flow at s and redirect Δ flow from (s, v_2) to (s, v_1). In order to ensure that the flow on (s, v_1) is at least a β-fraction of $f(s)$, then Δ needs to satisfy

$$f(s, v_1) + \varepsilon + \Delta \geq \beta(f(s) + \varepsilon) = \beta\left(f\left(s, v_1\right) + f\left(s, v_2\right) + \varepsilon\right).$$

This implies that $\Delta = \max\{0, \beta f(s, v_2) - (1-\beta)(f(s, v_1) + \varepsilon)\} \leq f(s, v_2)$ satisfies the above inequality. If $\Delta = 0$, then we only send the additional flow at s to (s, v_1). Since v_1 is a leaf in T', then Corollary 4.10 implies that the additional flow at v_1 is at most $\varepsilon \leq \frac{\beta}{1-\beta}$. Otherwise, since v_1 is a leaf in T', then Corollary 4.10 implies that the additional flow at v_1 is at most

$$\begin{aligned}
\varepsilon + \Delta &= \varepsilon + \beta f\left(s, v_2\right) - (1-\beta)\left(f\left(s, v_1\right) + \varepsilon\right) \\
&= \beta\left(f\left(s, v_2\right) + \varepsilon\right) - (1-\beta) f\left(s, v_1\right) \\
&\leq \beta\left(f\left(s, v_2\right) + \varepsilon\right) \leq \beta(f(s) + \varepsilon) \\
&\leq \beta\left(1 + \frac{\beta}{1-\beta}\right) = \frac{\beta}{1-\beta}.
\end{aligned}$$

Case 2. $r \geq 3$. We send the additional flow at s, Δ flow from $(s, v_2), \ldots, (s, v_r)$ to (s, v_1), and all remaining flow to (s, v_2). Let $F = f(s) - f(s, v_1)$. In order to ensure that the flow on (s, v_1) is at least a β-fraction of $f(s)$, then Δ needs to satisfy

$$f\left(s, v_1\right) + \varepsilon + \Delta \geq \beta(f(s) + \varepsilon) = \beta\left(f\left(s, v_1\right) + F + \varepsilon\right).$$

This implies that $\Delta = \max\{0, \beta F - (1-\beta)(f(s,v_1)+\varepsilon)\}$ satisfies the above inequality. Let f' be the new flow after redirection. Since f' is β-confluent, then we have that

$$f'(s,v_1) \geq \beta\left(f'(s,v_1) + f'(s,v_2)\right) > \frac{1}{2}\left(f'(s,v_1) + f'(s,v_2)\right).$$

Therefore, $f'(s,v_2) < f'(s,v_1)$. If $\Delta = 0$, then we only send the additional flow at s to (s,v_1), and redirect all remaining flow to (s,v_2). Since v_1 is a leaf in T', then Corollary 4.10 implies that the additional flow at v_1 is at most $\varepsilon \leq \frac{\beta}{1-\beta}$. Therefore, the additional flow at v_2 is also at most $\frac{\beta}{1-\beta}$. Otherwise, since v_1 is a leaf in T', then Corollary 4.10 implies that the additional flow at v_1 is at most

$$\begin{aligned}\varepsilon + \Delta = \varepsilon + \beta F - (1-\beta)\left(f(s,v_1) + \varepsilon\right) &= \beta(F+\varepsilon) - (1-\beta) f(s,v_1) \\ &\leq \beta(F+\varepsilon) \leq \beta\left(f(s) - f(s,v_1) + \varepsilon\right) \leq \beta(f(s)+\varepsilon) \\ &\leq \beta\left(1 + \frac{\beta}{1-\beta}\right) = \frac{\beta}{1-\beta}.\end{aligned}$$

□

This gives rise to a $(1+\frac{\beta}{1-\beta})$-approximation algorithm for the congestion minimization problem on β-confluent flows with $\beta \in (\frac{1}{2}, 1)$. This upper bound, however, is not tight. We have shown in Chapter 3 that there exists a $(1+\ln k)$-approximation algorithm for the congestion minimization problems on confluent flows. Since each confluent is β-confluent for any $\beta \in [0,1]$, then we may substitute the algorithm above with the algorithm of Chen et al. [2] to obtain a better approximation ratio whenever $\frac{\beta}{1-\beta} > \ln k$.

Chapter 5

Conclusion

Does the road wind up-hill all the way?
  Yes, to the very end.
Will the day's journey take the whole long day?
  From morn to night, my friend.

— Christina Rossetti, *Up-Hill*

In this thesis, we have presented several approximation techniques in the theoretical understanding of network flow models with degree constraints. In Chapter 3, we have seen a repeatedly emerging pattern in approximation algorithms for confluent flows—a main loop consisting of three operations, *vertex aggregation*, *sawtooth cycle elimination* and *sink deactivation*. We have also briefly discussed how variants of sink deactivation can be used to solve different questions on network flows. In Chapter 4, we have generalized the notion of sawtooth cycles by relaxing the restriction of only having reverse edges from frontier vertices to sinks in [2]. We have also seen how this generalized notion of sawtooth cycles leads to a promising technique that allows us to process vertices "locally" in each digon tree. In particular, a directed acyclic graph with no (general) sawtooth cycles admits an acyclic digon-tree representation, which gives a topological ordering of the digon trees in the graph.

5.1 Open Questions

Many intriguing questions remain wide open. We list several directions for future research in network flows with degree constraints:

1. *Non-uniform vertex capacities.* As discussed in Chapter 1, it would be interesting to extend existing results for the special case of unit vertex capacities to graphs with non-uniform vertex capacities. For confluent flows with non-uniform vertex capacities, Shepherd et al. [24] give an $O(\log^6 n)$-approximation algorithm for demand maximization either if the *no bottleneck assumption* is satisfied, or if congestion 2 is

allowed. On the negative side, Shepherd and Vetta [23] prove that, for confluent flows with non-uniform vertex capacities, both congestion minimization and demand maximization cannot be approximated to within $O(m^{0.5-\varepsilon})$ for any $\varepsilon > 0$, unless P = NP.

2. *Non-uniform edge costs.* The problem of bounded congestion with arbitrary edge costs is not considered in this thesis. This problem is intrinsically more difficult to solve because a routing path involves edge costs along the entire path. Therefore, we may not be able to use a "local" approach for non-uniform edge costs.

3. *Minimum number of confluent rounds.* Another natural objective function is the minimum number of rounds that are necessary to route all demands by a feasible confluent flow. In other words, we would like a partition $\{V_1, \ldots, V_r\}$ of vertices such that r is minimized and that the demands in each V_i can be satisfied by a feasible confluent flow. Note that congestion minimization gives a trivial approximation ratio of $O(\log k)$ for this optimization problem by using the same support network. It would be interesting to ask whether it is possible to satisfy all demands in a constant number of rounds.

4. *Halfluent flows.* A flow is *halfluent* if each vertex must either send its total incoming flow to one outgoing edge, or split its total incoming flow *equally* to two outgoing edges. Donovan et al. [7] conjecture that the acyclic digon-tree representation approach may be extended to achieve a constant gap between bifurcated flows and halfuent flows.

Bibliography

[1] Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In *International Colloquium on Automata, Languages, and Programming*, pages 222–233. Springer, 2004.

[2] Jiangzhuo Chen, Robert D Kleinberg, Lászlo Lovász, Rajmohan Rajaraman, Ravi Sundaram, and Adrian Vetta. (Almost) tight bounds and existence theorems for confluent flows. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 529–538, 2004.

[3] Jiangzhuo Chen, Rajmohan Rajaraman, and Ravi Sundaram. Meet and merge: Approximation algorithms for confluent flows. *Journal of Computer and System Sciences*, 72(3):468–489, 2006.

[4] William H Cunningham. Optimal attack and reinforcement of a network. *Journal of the ACM (JACM)*, 32(3):549–561, 1985.

[5] Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[6] Yefim Dinitz, Naveen Garg, and Michel X Goemans. On the single-source unsplittable flow problem. *Combinatorica*, 19(1):17–41, 1999.

[7] Patrick Donovan, B Shepherd, Adrian Vetta, and Gordon Wilfong. Degree-constrained network flows. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 681–688, 2007.

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

[9] Mark J Eisner and Dennis G Severance. Mathematical techniques for efficient record segmentation in large shared databases. *Journal of the ACM (JACM)*, 23(4):619–635, 1976.

[10] D. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, USA, 2010. ISBN 0691146675.

[11] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[12] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[13] 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, page 136–146, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897911938. doi: 10.1145/12130.12144. URL `https://doi.org/10.1145/12130.12144`.

[14] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[15] Dan Gusfield. *On scheduling transmissions in a network.* Yale University, Department of Computer Science, 1986.

[16] Alon Itai and Michael Rodeh. Scheduling transmissions in a network. *Journal of Algorithms*, 6(3):409–429, 1985.

[17] Jon M Kleinberg. *Approximation algorithms for disjoint paths problems.* PhD thesis, Massachusetts Institute of Technology, 1996.

[18] Jon M Kleinberg. Single-source unsplittable flow. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 68–77. IEEE, 1996.

[19] Stavros G Kolliopoulos and Clifford Stein. Approximation algorithms for single-source unsplittable flow. *SIAM Journal on Computing*, 31(3): 919–946, 2001.

[20] Dean H Lorenz, Ariel Orda, Danny Raz, and Yuval Shavitt. How good can ip routing be. *DIMACS Rep*, 17, 2001.

[21] Nimrod Megiddo. Optimal flows in networks with multiple sources and sinks. *Mathematical Programming*, 7(1):97–107, 1974.

[22] Loïc Seguin-Charbonneau and F Bruce Shepherd. Maximum edge-disjoint paths in planar graphs with congestion 2. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 200–209. IEEE, 2011.

[23] F. Bruce Shepherd and Adrian R. Vetta. The inapproximability of maximum single-sink unsplittable, priority and confluent flow problems. *Theory of Computing*, 13(20):1–25, 2017. doi: 10.4086/toc.2017.v013a020. URL `http://www.theoryofcomputing.org/articles/v013a020`.

[24] F Bruce Shepherd, Adrian Vetta, and Gordon T Wilfong. Polylogarithmic approximations for the capacitated single-sink confluent flow problem. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 748–758. IEEE, 2015.

[25] FB Shepherd and A Vetta. Visualizing, finding and packing dijoins. In *Graph Theory and Combinatorial Optimization*, pages 219–254. Springer, 2005.

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

[27] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[28] Harold S. Stone. Critical load factors in two-processor distributed systems. *IEEE transactions on Software Engineering*, (3):254–258, 1978.

[29] Jun Wang and Klara Nahrstedt. Hop-by-hop routing algorithms for premium-class traffic in diffserv networks. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 705–714. IEEE, 2002.