Confluent, bifurcated and unsplittable flows

Loïc Séguin-Charbonneau

Master of Science

Department of Mathematics and Statistics

McGill University

Montreal, Quebec

October 5, 2009

A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of M. Sc. Mathematics and Statistics

ACKNOWLEDGEMENTS

I would like to thank my parents for continuous support and encouragements. My close friends, Xavier, Julien, Philippe and Romain have been there support me in the hard moments of my studies and I thank them for that. Also, I am grateful to my office mates Atefeh, Sarah, Thomas and Darrio for the great discussions and the relief of a couple of jokes. I want to thank Bruce Shepherd for being a great supervisor, supportive and motivating. He has provided me with lots of fun and challenging problems to chew on during the last two years and working with him was a pleasure. Finally, I want to thank the National Science and Engineering Research Council, Bruce Shepherd and the Department of Mathematics and Statistics for their financial support.

ABSTRACT

This thesis studies network flow problems. More specifically, we mostly consider single-sink multicommodity flows with constraints on how nodes may "process" flow. In the unsplittable flow problem, the demand sent from a source to its destination must follow a single path. In d-furcated flows, each node is allowed to send flow to at most d out-neighbours. The special case with $d = 1$ is called confluent flow. We make a survey of many of the known algorithms to tackle these problems. Finally, we present a new result which uses confluent flows and a special type of clustering we call rooted clustering to give an approximation algorithm for the maximum edge-disjoint path problem. This algorithm routes a constant fraction of the demand with maximum edge congestion at most 3, thus improving the previous known bound of 4.

ABRÉGÉ

Cette thèse étudie les problèmes de flots sur réseau. Plus particulièrement, nous portons notre attention sur les flots à puits unique et à multiples sources avec des contraintes de degré. Dans le problème de flot indivisible, la demande envoyée par une source doit suivre un chemin unique. Dans le problème de flot d-furqué, chaque sommet peut envoyer du flot à d voisins au plus. Le cas particulier avec $d = 1$ est appelé flot confluent. Nous présentons un survol de certains des algorithmes utilisés pour attaquer ces problèmes. Finalement, nous présentons un nouveau résultat qui utilise les flots confluents et un type particulier de regroupement des sommets que nous appelons regroupement enraciné pour obtenir un algorithme d'approximation pour le problème de maximisation des chemins disjoints (en terme d'arêtes). Cet algorithme satisfait une fraction constante de la demande totale avec une congestion d'arête d'au plus 3. Il s'agit donc d'une amélioration de la meilleure borne précédente de 4.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

ABSTRACT iii

ABRÉGÉ iv

LIST OF FIGURES vii

1 Introduction 1

2 General network flow theory 4

2.1 Definitions, the maximum flow and the minimum cut problems . . 4
2.1.1 Ford-Fulkerson algorithm 8
2.2 Single-sink Multicommodity flows 11
2.3 General multicommodity flows 14
2.3.1 Column generation for general multicommodity flows . . . 15
2.3.2 Polynomial time algorithms for multicommodity flows . . . 17
2.4 The undirected case 17
2.5 Minimum cost flow problem 18

3 Unsplittable flows 22

3.1 Statement of the single-sink unsplittable flow problem 22
3.2 Known results 24
3.3 Solving the congestion minimization unsplittable flow problem . . 27
3.4 The ring loading problem 29
3.5 Minimum cost unsplittable flows 35
3.5.1 d_{min}-integral demands 35
3.5.2 Arbitrary demands 40

4 Bifurcated flows and d-furcated flows 42

4.1 Statement of the problem 42

4.2 Known results . . . 44
4.3 Sawtooth cycles and sawtooth cycle-free digraphs . . . 44
4.3.1 Detecting clean cycles . . . 47
4.3.2 Structure theorem for sawtooth cycles . . . 48
4.4 A congestion $1 + 1/(d-1)$ algorithm for d-furcated flows . . . 48
4.4.1 Phase I . . . 48
4.4.2 Phase II . . . 50
4.5 d-furcated flows with costs . . . 54
4.5.1 Hardness of finding negative-cost clean cycles . . . 56

5 Confluent flows . . . 64

5.1 Statement of the problem and known results . . . 64
5.2 Minimizing congestion . . . 66
5.3 Maximizing satisfied demand . . . 71

6 Rooted clustering and the maximum edge-disjoint path problem in planar graphs . . . 74

6.1 Statement of the MEDP problem and known results . . . 74
6.1.1 Maximum edge-disjoint paths . . . 74
6.1.2 LP relaxation . . . 75
6.1.3 Outline of the congestion 4 algorithm . . . 76
6.2 Rooted Clustering . . . 79
6.2.1 Edge-disjoint rooted clustering . . . 81
6.2.2 Partial clusterings via trees . . . 81
6.2.3 Reducing to Node-Normalized Instances. . . . 83
6.2.4 From Confluent Flows to Clusters . . . 84
6.2.5 Proof of Theorem 6.2.2 . . . 86
6.2.6 Proof of Theorem 6.2.1 . . . 87
6.3 Congestion 3 algorithm for EDP in planar graphs . . . 87
6.3.1 The congestion 3 algorithm . . . 87

7 Conclusion . . . 89

References . . . 90

LIST OF FIGURES

Figure page

3–1 An instance of the ring loading problem. Node names are indicated in regular type and edge names are in italics. The chords show demands between various pairs of terminals. 30

4–1 The acyclic digon-tree representation of a digraph with sources s_1, s_2, s_3 and sinks $x_1, x_2, \ldots, x_6$. 51

4–2 Clause gadget for clause $C_i = x_1 \vee \bar{x}_2 \vee x_3$. Note that the ending arc for path p_8^i has cost 1 since this path corresponds to the unique assignment $\bar{x}_1, x_2, \bar{x}_3$ which makes the clause false. 57

4–3 The clause gadgets attached together. 58

4–4 An example of how the literal gadgets are attached together. Here, the literal gadget x_k^1 is attached to each of the gadgets $\bar{x}_k^p$ by doing the node identifications along the dotted lines. 59

4–5 x_k-gadgets are shown with solid arcs while $\bar{x}_k$-gadgets are shown with dotted arcs from an $\bar{x}_k$ node. Here we see the eight appearances of the x_k variable in clause C_i and how they are linked together. The arcs in the digons have cost $-L$, the arcs not in the digons have cost L and the arcs leaving s_i have cost 0. 60

6–1 There is no arc-disjoint rooted total clustering (splittable or unsplittable). 82

CHAPTER 1
Introduction

The concept of "network" is now in the common knowledge. Networks occur in transportation and logistics [26], food chains and social relationships [34], scheduling [50], protein interactions [66], the Internet [24], computer chips design [6], etc. These "objects" that can be found almost everywhere where one dares to look are also very rich and interesting from a mathematical point of view. Not only does the mathematical study of networks allow us to understand the "real" networks better, but it also gives rise to numerous beautiful results (see for e.g., [17]).

Mathematically, we represent networks as graphs, i.e., sets of nodes some pairs of which are linked by edges. Graph theory formally started with the work of Euler in 1736 and the famous Königsberg bridge problem was probably the first graph theoretic question to be considered. Since then, graph theory has come a long way and the structure of graphs is now much better understood. It is only in the middle of the twentieth century, however, that network flows began to be studied. The first published results were mainly by Ford and Fulkerson [28, 26, 27, 29] even though Menger [51] published a result that was basically the Maximum-flow Minimum-cut Theorem of Ford and Fulkerson a couple of years before them. However, Ford and Fulkerson were really the first ones to introduce network flows as we know them today with costs and capacities.

In a graph, some nodes might want to send "information" to other nodes. For instance, lets consider an Internet graph. A server might want to send files to other computers. This is done by splitting the file into packets and sending these packets of information along wires towards some intermediary computers until the packets reach the source. This flow of information is constrained by how much information can transit on a given edge (e.g.: wire, wireless link, optical fibre, etc.) at the same time. Such constraints are called *capacity constraints* in the jargon of flow theory. It is also possible that a given node, i.e., intermediary computer, can only send flow along a single wire. Even if files are often divided into packets, the packets themselves are often "unsplittable", i.e., they have to travel as a whole unit throughout the network. These sorts of constraints are ones we will consider in this thesis.

First, we introduce the basics of network flows and do a quick survey of some of the known algorithms to solve network flow problems. We present various network flow problems (single-commodity, multicommodity, demand maximization, etc.). Among the algorithms that we consider is the well-known Ford-Fulkerson algorithm for the single-commodity maximum flow problem.

In Chapter 3, we consider the *unsplittable flow problem* where the flow originating at a node must travel to its destination along a single path. These flows were introduced by Cosares and Saniee [14] and were later studied by Kleinberg [43, 44] who actually coined the term "unsplittable". They have been the subject of a huge amount of literature in the last decade. The notion of

sawtooth cycles is introduced in this chapter and plays a predominant role in the other chapters.

We then present flows where nodes have out-degree constraints in Chapter 4. These flows are called d-furcated. Such flows are of particular importance for the Internet network routing since some routing protocols constrain nodes to only send flow to a very small set of other nodes. This is due to the very nature of hop by hop routing using next-hop tables. We present an algorithm from [21] that approximately solves the congestion minimization problem for d-furcated flows. There is no known results for the cost version of d-furcated flows and we present a complexity proof of a related result. This proof has been published before in [59], but some details were omitted. We present the complete proof.

The special case of d-furcated flows for $d = 1$, i.e., every node can send flow along only one arc, is called *confluent flows*. It was studied in [12, 11, 10]. We defer the study of such flows to Chapter 5 since the techniques used are somewhat different from d-furcated flows.

Finally, Chapter 6 contains a new result concerning the maximum edge-disjoint paths problem in planar graphs. We present a constant factor approximation (with respect to the number of connected terminals) for this problem where each edge is used at most 3 times, thus improving the previous bound of 4 [9]. The approximation algorithm is based on the demand maximization algorithm for confluent flows and on the theory of the *rooted clustering problem* which we define and study in this chapter.

CHAPTER 2
General network flow theory

We introduce the basic notions of network flows needed in the rest of this document. Most of the content of this chapter is covered in greater details in [1] and [13]. We assume the reader knows basic graph theory (see, for instance, [17]).

2.1 Definitions, the maximum flow and the minimum cut problems

We start by presenting the theory of *single-commodity flows* in directed graphs. Consider a digraph $D = (V, A)$ with two special nodes: a *source* s and a *sink* t. The pair (s, t) is called a *commodity*. We suppose that $|V| = n$ and $|A| = m$. The source wishes to send flow to the sink along the arcs of the graph. We define the *capacity* of the arcs to be a function $u : A \to \mathbb{R}_+ \cup \{\infty\}$ that assigns to each arc $a \in A$ a non-negative[1] capacity $u(a)$. Since the arc set is discrete, the capacities can also be seen as a vector $u \in (\mathbb{R}_+ \cup \{\infty\})^A$, where u is indexed by the arc set, i.e., $u_a = u(a)$. Both approaches are equivalent, but we adhere to the second one. For simplicity, we denote an arc (u, v) simply as uv. When we need them, we denote (undirected) edges by $\{u, v\}$. Note that we allow the capacity of an arc to be ∞ which simply means that this arc has no upper bound on the amount of flow it can receive. A *network flow* (or just a *flow*) on graph D with

[1] in this document, $\mathbb{R}_+ = \{x \in \mathbb{R} : x \geq 0\}$

capacities u, is a vector $f \in \mathbb{R}_+^A$ that assigns a flow value to each arc and that satisfy the two following constraints.

$$f_a \leq u_a \quad \forall\, a \in A \tag{2.1}$$

$$\sum_{u:uv \in A} f_{uv} = \sum_{w:vw \in A} f_{vw} \quad \forall\, v \in V \setminus \{s,t\} \tag{2.2}$$

Equations 2.1 are the *capacity constraints* which ensure that no arc receives more flow than its capacity, and Equations 2.2 are the *flow conservation* constraints which ensure that the flow arriving at a node leaves that node, i.e., only the source can "generate" flow and no flow is allowed to accumulate at any node except the sink. The *value* of a flow is the net flow out of the source

$$\sum_{v:sv \in A} f_{sv} - \sum_{w:ws \in A} f_{ws}. \tag{2.3}$$

Note that this is equal to the net flow into the sink by flow conservation.

To simplify the notation, we use $f_+(v) = \sum_{w:vw \in A} f_{vw}$ to represent the *outflow* at node v, i.e., the flow leaving v. Similarly, we define $f_-(v) = \sum_{u:uv \in A} f_{uv}$ to be the *inflow* at node v, i.e., the flow coming into v. The flow conservation constraints can then be written simply as

$$f_+(v) - f_-(v) = 0 \quad \forall\, v \in V \setminus \{s,t\}. \tag{2.4}$$

Given a set of nodes $S \subseteq V$, the *cut* induced by S, denoted by $\delta_D(S)$, is the set of arcs with tail in S and head in $V \setminus S$. When it is clear from the context that the cut is in graph D, we denote the cut by $\delta(S)$. If S consists of a single node v, we abuse the notation and write $\delta(v)$. An (s,t)-*cut* is a cut $\delta(S)$ where S contains s

and $V \setminus S$ contains t. If $x \in \mathbb{R}^A$ is a vector of arc variables, and $F \subseteq A$, we use the notation

$$x(F) = \sum_{a \in F} x_a. \tag{2.5}$$

Most of the time, the value of interest is the *capacity of a cut* $u(\delta(S))$. If x is a flow vector, we define the *net outflow* from set S to be $x(\delta(S)) - x(\delta(V \setminus S))$. Similarly, the *net inflow* into set S is $x(\delta(V \setminus S)) - x(\delta(S))$. Note that the net outflow at s is equal to the value of the flow.

A natural question is to ask what is the maximum amount of flow that can be sent from s to t. We use variables x_a to denote the flow on arc a. Since all such flow must leave the source by some arc incident to it, the quantity that is being maximized is the value of the flow $x(\delta(s)) - x(\delta(V \setminus s))$. The *maximum flow problem* can be formulated as the following linear program (LP).

$$\begin{aligned}
\max \quad & x(\delta(s)) - x(\delta(V \setminus s)) && && (2.6)\\
\text{s.t.} \quad & x_+(v) - x_-(v) && = 0 \quad \forall\, v \in V \setminus \{s,t\} && (2.7)\\
& x_a && \leq u_a \quad \forall\, a \in A && (2.8)\\
& x_a && \geq 0 \quad \forall\, a \in A && (2.9)
\end{aligned}$$

The dual of this LP is

$$\begin{aligned}
\min \quad & \textstyle\sum_{a \in A} y_a u_a & & & (2.10)\\
\text{s.t.} \quad & -z_u + z_v + y_{uv} & \geq 0 \quad & \forall\, uv \in A, u, v \in V \setminus \{s,t\} & (2.11)\\
& -z_v + y_{sv} & \geq 1 \quad & \forall\, sv \in A & (2.12)\\
& z_v + y_{vs} & \geq -1 \quad & \forall\, vs \in A & (2.13)\\
& -z_v + y_{tv} & \geq 0 \quad & \forall\, tv \in A & (2.14)\\
& z_v + y_{vt} & \geq 0 \quad & \forall\, vt \in A & (2.15)\\
& y_a & \geq 0 \quad & \forall\, a \in A & (2.16)
\end{aligned}$$

This dual is closely related to the *minimum cut problem* which consists of finding the (s,t)-cut in D that has the smallest capacity. If a dual solution is fractional, it cannot be interpreted as a cut since in a cut, an arc is either selected or not, so its corresponding variable should be integral. However, the Ford-Fulkerson algorithm that is presented in Section 2.1.1 implies that any basic solution to this dual LP is $0-1$ valued, i.e., it identifies an (s,t)-cut. Weak LP duality implies that the value of any (s,t)-cut is an upper bound on the value of the maximum flow. Strong duality implies that the optimal value for the primal and the dual are the same thus, solving the minimum cut problem is equivalent to solving the maximum flow problem [51, 28, 48].

Theorem 2.1.1 (Maximum-flow Minimum-cut)**.** *If a maximum flow exists, then the value of the maximum flow is equal to the value of the minimum* (s,t)*-cut.*

An operation that is often done on a flow to simplify its structure is to eliminate directed cycles. Define the *support* of a flow f to be the set of arcs of D with nonzero flow. Suppose a flow f has a directed cycle C in its support. Let $\epsilon = \min\{f_a : a \in C\}$. Then subtracting ϵ from the flow on each arc of C will bring the flow on at least one arc to zero, thus "breaking" the cycle. This is called *augmenting the flow* on the cycle C. In doing so, the net flow at each node in the cycle does not change since flow on one incoming arc is decreased by ϵ but so is the flow on one outgoing arc. So the new vector is still a flow and its value remains the same.

We are sometimes interested in node capacity as opposed to the arc capacities considered up to now. One way to transform node capacities $u \in \mathbb{R}_+^V$ into arc capacities is to *split* every node v into two nodes v_- and v_+. Add a *node arc* v_-v_+ of capacity u_v for every node and let the capacity of every other arcs be infinity. All uv arcs in the original graph are replaced by uv_- arcs and all vw arcs are replaced by v_+w arcs. This new instance is a flow problem with arc capacities which can be solved using techniques shown in this chapter. Once done, one needs only to contract the node arcs to get a solution for the node capacitated instance.

2.1.1 Ford-Fulkerson algorithm

We present one of the most well known efficient combinatorial algorithms to solve the maximum flow problem [28]. This algorithm also proves the Maximum-flow Minimum-cut Theorem. Note that more recent algorithms have an improved running time, e.g.: [35] (see [1] and [56] for an overview).

The Ford-Fulkerson algorithm starts with a *feasible flow*, i.e., a flow that satisfies the constraints 2.7, 2.8 and 2.9. For instance, one starts with a zero flow $x = 0$. The *auxiliary digraph* of D is a digraph D_x that has the same node set, V, as D and that has the following arcs. If $a = uv \in A$, and $x_a < u_a$, then arc uv is in $A(D_x)$; it is called a *forward arc*. If $a = uv \in A$, and $x_a > 0$, then arc vu is in $A(D_x)$; it is called a *reverse arc*. At every iteration of the algorithm, we have a feasible flow x and we build the auxiliary digraph D_x. In this digraph, we look for a directed (s, t)-path. If none exists, the algorithm stops. If such a path P exists, then it is possible to *augment* the flow along this path, i.e., to increase the flow on forward arcs and to decrease it on reverse arcs. Let $\epsilon_1 = \min\{u_a - x_a : a \text{ is a forward arc}\}$, $\epsilon_2 = \min\{x_a : a \text{ is a reverse arc}\}$ and $\epsilon = \min(\epsilon_1, \epsilon_2)$. Then, the augmentation consists of increasing the flow on forward arcs by ϵ and decreasing the flow on reverse arcs by ϵ. It is possible to do so since by definition, forward arcs can sustain an increase of ϵ in their flow without violating their capacity, and reverse arcs have at least ϵ flow on them, so their flow can be decreased by ϵ without violating the non-negativity constraint.

The stopping condition is crucial in proving the Maximum-flow Minimum-cut Theorem. One can find an (s, t)-path by greedily growing an arborescence rooted at s using, for instance, depth-first search. If no such path is found, then we have a maximum arborescence T (with respect to the number of nodes) rooted at s such that all arcs of D_x having a node of T as its tail also has a node of T as its head. Otherwise, there would be an arc uv with $u \in T$ and $v \notin T$ and adding uv to T we would obtain an arborescence with one more node, a contradiction. Let $S = V(T)$

be the set of vertices in T. Then, $\delta_{D_x}(S) = \emptyset$. The following lemma shows that this is a minimum capacity (s,t)-cut in D.

Lemma 2.1.2. *If S is obtained as described above and $\delta_{D_x}(S) = \emptyset$, then $\delta_D(S)$ is a minimum capacity (s,t)-cut.*

Proof. Let a be an arc in $\delta_D(S)$. Since $a \notin \delta_{D_x}(S)$, we have that $x_a = u_a$. Let $r = uv \in \delta_D(V \setminus S)$. Since $vu \notin \delta_{D_x}(S)$, we have that $x_r = 0$. Hence the net outflow from S is $x(\delta_D(S)) = u(\delta_D(S))$. Since the capacity of a cut is an upper bound on the value of a flow by weak LP duality, x is a maximum flow and $\delta_D(S)$ is a minimum cut. □

Note that we obtain the Maximum-flow Minimum-cut Theorem as an immediate corollary. Also, the algorithm gives a nice existence condition for maximum flows. The *capacity of a directed path* is the minimum of the capacities of every arc on this path. To determine whether a maximum flow exists, one can use the following simple corollary.

Corollary 2.1.3. *A maximum flow exists if and only if there is no directed (s,t)-path of infinite capacity.*

The Ford-Fulkerson algorithm runs in weakly polynomial time when all capacities are rational, more precisely, it has a running time of $O(mM)$, where M is the maximum flow value (which is not necessarily polynomially bounded in the size of the input). If capacities are irrational, the algorithm may fail to converge to a maximum flow [29]. An improvement to this algorithm was proposed at the same time by Edmonds and Karp [23] and by Dinitz [19]. The idea is to always choose the shortest (s,t)-path, with respect to the number of arcs. The running time of

this improved version is $O(nm^2)$ which is strongly polynomial (does not depend on the capacities). The first strongly polynomial algorithm for minimum cost flow was suggested by Tardos [63].

Since the Ford-Fulkerson algorithm guarantees that solutions to the primal and the dual LP are integral, the maximum flow problem can also be solved using any algorithm that solves linear programs such as the simplex algorithm [16]. This algorithm is, however, not polynomial. It is also possible to apply the ellipsoid algorithm [42] to get an optimal fractional solution that is sufficiently close to an integral solution to be able to find this integral solution in polynomial time.

2.2 Single-sink Multicommodity flows

A natural generalization of the single-commodity flow problem is to consider the case where there is more than one sink and/or more than one source. Consider the case where there is only one sink t but there are k sources $s_1, s_2, \ldots, s_k$. This is referred to as the *single-sink multicommodity flow problem*. This case can be reduced to the single-commodity case by adding a new node s to D and adding an arc ss_i with infinite capacity for each $i \in \{1, 2, \ldots, k\}$. The maximum (s, t)-flow in this new modified instance is the same as the maximum single-sink multicommodity flow. In this thesis, we will mainly consider single-sink multicommodity problems with additional constraints such as the ones in [60]. These constraints make it impossible to use the above reduction and new tools have to be developed.

A variant of the single-sink multicommodity flow problem is that each source s_i is assigned some *demand* d_i. The demand is the amount of flow that the source

wants to send to the sink. For such problems, it is convenient to use the *path formulation* as opposed to the *arc formulation* used for the single-commodity flows. The path formulation is defined as follow. Let $\mathcal{P}_i$ be the set of directed paths from s_i to t and $\mathcal{P} = \cup_i \mathcal{P}_i$. For each source s_i, we want to find what amount of flow x_P to send along each $P \in \mathcal{P}_i$ so that the following constraints are satisfied.

$$\sum_{P:a\in P} x_P \leq u_a \quad \forall\, a \in A \tag{2.17}$$

$$x_P \geq 0 \quad \forall\, P \in \mathcal{P} \tag{2.18}$$

We call a flow satisfying these constraints *fractional* or *standard* (as opposed to unsplittable, confluent or d-furcated). Note that since we have a variable for each path, this results in an exponential number of variables. A flow x *satisfies* the demand of commodity (s_i, t) if

$$\sum_{P\in\mathcal{P}_i} x_P = d_i \tag{2.19}$$

We then say that the demand has been *routed*. With demands, the multicommodity flow problem becomes a decision problem where we ask whether or not a flow that satisfies all demands exists.

Clearly, if there exists a cut $\delta(S)$ with $t \notin S$ such that the sum of the demands d_i for $s_i \in S$ is greater than the capacity of the cut, there can be no flow satisfying all demands. Interestingly, the converse is also true. Indeed, consider adding a source s with arcs ss_i of capacity d_i. It is easy to show that the minimum capacity (s,t)-cut in this new (single-commodity) instance has capacity $\sum_i d_i$ and thus, by the Maximum-flow Minimum-cut Theorem, the maximum flow has the same

value. In the original instance, this gives a flow satisfying all demands. This result is known as the *cut condition* (see, for instance, [1]).

Theorem 2.2.1 (Cut condition)**.** *Let D be a digraph with sources $\{s_1, s_2, \ldots, s_k\}$, demands $d = (d_1, d_2, \ldots, d_k)$ and arc capacities u. A fractional flow f satisfying all demands exists if and only if*

$$\sum_{i:s_i \in S} d_i \leq u(\delta(S)) \tag{2.20}$$

for all $S \subseteq V$ such that $t \in V \setminus S$.

We usually make a couple of simplifying assumptions. Denote the maximum demand by $d_{max} = \max\{d_1, d_2, \ldots, d_k\}$ and the minimum capacity by $u_{min} = \min\{u_a : a \in A\}$. Throughout this thesis, we suppose, unless otherwise mentioned, that $d_{max} \leq u_{min}$, i.e., the maximum demand can be routed on the smallest capacity arc. This condition is referred to as the *no bottleneck assumption*. By scaling the demands and the capacities by a constant factor, we can make the assumption that all demands are at most 1 and all capacities are at least 1. Finally, most of the time we consider *uniform capacities*, i.e., all arcs have the same capacity[2] .

Also, if we are considering an instance with node congestion and we do the node splitting operation described in Section 2.1, the demand of node v has to be assigned to v_- in order to reduce the problem to the edge capacitated problem.

[2] We also refer to instances with *uniform costs* and *uniform demands*, meaning that all arcs have the same cost and all nodes have the same demand, respectively.

If the cut condition does not hold, there are three main questions of interest. In the *demand maximization problem*, one asks what is the subset of demands with the maximum value that can be routed while still obeying the capacity constraints. Another question of interest is if we allow to increase the capacities, what is the minimum increase that will result in a flow satisfying all demands. In this context, the flow on an arc is called its *congestion* and the maximum flow on an arc of D is the *congestion of the flow*. The question thus becomes to minimize the congestion of the flow. This is the *congestion minimization problem*. Finally, in the *rounds minimization problem*, we try to route the demands in a minimum number of *rounds*, i.e., to find a set of feasible flows $f_1, f_2, \ldots, f_\nu$ such that the union of these flows satisfies all demands and ν is minimized. These questions will come back later when we talk about bifurcated, unsplittable and confluent flows.

2.3 General multicommodity flows

When sources and sinks come in pairs, i.e., a source has to route its flow to a given sink, we have a *general multicommodity flow problem*. Let the set of all sources and all sinks be called the set of *terminals* T. To be as general as possible, we allow to have more than one terminal at a given node. Let X be a set of terminal pairs (s_i, t_i), also called *commodities*, composed of one source and one sink. The question is to determine the maximum flow that can be sent from sources to their destination without violating the edge capacities. We refer to this problem as the *throughput maximization problem*.

2.3.1 Column generation for general multicommodity flows

General multicommodity flows are substantially more complicated both theoretically and computationally. For instance, on the theory side, the cut condition is no longer sufficient for the existence of a general multicommodity flow (instances on expander graphs for uniform demand are one class of examples). Empirically they have proved more difficult as well. In this section, we present an algorithm to solve the multicommodity flow throughput maximization problem using the path formulation. Let $\mathcal{P}_i$ be the set of directed paths from s_i to t_i and $\mathcal{P} = \cup_i \mathcal{P}_i$. For each terminal pair, we want to find what amount of flow x_P to send along each $P \in \mathcal{P}_i$ so that the following constraints are satisfied.

$$\sum_{P:a\in P} x_P \leq u_a \quad \forall\, a \in A \tag{2.21}$$

$$x_P \geq 0 \quad \forall\, P \in \mathcal{P} \tag{2.22}$$

These constraints, except the non-negativity ones, can be written as $Cx \leq u$ where $C \in \mathbb{R}^{A\times\mathcal{P}}$, $x \in \mathbb{R}^{\mathcal{P}}$, $u \in \mathbb{R}^A$. The columns of C are just the characteristic vectors of paths in $\mathcal{P}$, i.e., for a given path P, $(c_P)_a$ is 1 if $a \in P$ and 0 if $a \notin P$. The objective is to maximize $\sum_{P\in\mathcal{P}} x_P$. This formulation can have an exponential number of variables, thus any algorithm that solves linear programs will possibly take an exponential time to solve this problem. However, in a basic solution a huge number of variables will have value 0 and we can use the strategy of *column generation* to set them to positive values as needed. This leads to a non polynomial, but sometimes useful, approach for solving the multicommodity flow

problem. The following algorithm is a variation of the simplex algorithm due to Fork and Fulkerson [27].

First, add a slack variable σ_a to each capacity constraint to make it an equality constraint and let $x' = \begin{pmatrix} x \\ \sigma \end{pmatrix}$ where σ is the vector of slacks. The capacity constraints become $[C\ I]x' = u$. Let $w \in \mathbb{R}^{\mathcal{P}+A}$ have an entry of 1 for all paths and an entry of 0 for all arcs. Then the objective function is maximize $w^T x'$. Consider the basic solution obtained by setting all $x_P = 0$ and all $\sigma_a = u_a$. The columns of $[C\ I]$ corresponding to non-zero variables form a *basis* B. The algorithm moves from basis to basis while ensuring that an improving move is made at all steps.

The dual of this problem is the following.

$$\min \quad u^T y \tag{2.23}$$

$$\text{s.t.} \quad y^T c_P \geq 1 \quad \forall\, P \in \mathcal{P} \tag{2.24}$$

$$y_a \geq 0 \quad \forall\, a \in A \tag{2.25}$$

where $y \in \mathbb{R}^A$. For any basis of the primal problem, there is a corresponding solution to the dual problem obtained by solving $y^T B = w_B$ where w_B is the part of w corresponding to the variables in the basis. Once y is found, one can check whether it satisfies the constraints of the dual. Checking for the non-negativity of the arc variables is easy. The key to tackling the path constraints 2.24 is to consider the y_a as costs on the arcs of the graph and to run a shortest path algorithm such as Dijkstra's [18], Floyd-Warshall's [25] or Johnson's [41]. With

this interpretation, the path constraint says that every path should have cost at least 1. Hence, if the shortest (s_i, t_i)-path has length less than 1, we can move this path into the basis. The variable that leaves the basis is chosen by solving for z in $Bz = e_a$ (where e_a is the vector in $\mathbb{R}^A$ consisting of all zeroes except for the a^{th} entry which is a 1) if the entering variable is σ_a or $Bz = c_P$ if the entering variable is path x_P.

2.3.2 Polynomial time algorithms for multicommodity flows

There exists (theoretically) more efficient algorithms to solve the multicommodity flow problem in polynomial time such as the one by Grötschel et al. [37]. The dual separation problem for multicommodity flows is similar to the shortest path approach described above. Thus, the ellipsoid algorithm can be used to solve the problem. There are also a wealth of polynomial time approximation algorithms such as those presented in [65, 49, 53, 32].

2.4 The undirected case

In the previous sections, we saw how to reformulate the maximum flow problem using the path formulation instead of the arc formulation. This formulation is useful if we want to define maximum flow problems on undirected graphs $G = (V, E)$. The flow conservation constraints can not be implemented in the undirected case since there is no notion of inflow or outflow for undirected graphs. However, we can try to find the maximum flow sent on paths from a source to a sink such that the capacity constraints are satisfied.

In particular, the multicommodity flow problem on undirected graphs has the same path formulation as the directed version where we replace directed paths by

undirected paths and arcs by edges. The problem can be solved using the column generation method presented in the preceding section.

For single-commodity flow, the path formulation also gives a way to solve the undirected version of the problem. Another approach is based on bidirecting edges. More precisely, given and undirected graph $G = (V, E)$ one can *bidirect* it, i.e., replace each edge $e = \{u, v\} \in E$ by two arcs $a_1 = uv$ and $a_2 = vu$. Then the graph is transformed into a directed graph and the usual algorithms (such as the Ford-Fulkerson algorithm) can be used. Since we can always augment along directed cycles to remove them, flow is never going to be sent on both arcs corresponding to the same edge. Thus, the flow can be reinterpreted as an undirected flow in the original graph.

2.5 Minimum cost flow problem

The *minimum cost flow problem* generalizes maximum flow problems, including shortest path problems. For instance, a maximum flow problem is basically a minimum cost problem with uniform costs and a shortest path problem is a minimum cost problem where every arc has unit cost, infinite capacity and every node wants to send 1 unit of flow to every other node. As such, minimum cost flow problems are usually harder to solve and require more sophisticated algorithms. Lots of algorithms have been developed to solve these problems, some of which run in polynomial time. These algorithms are quite involved and we do not describe them in detail here. We just present the important notion of negative cost cycles which will be useful in understanding why the cost version of the d-furcated flow problem is hard to solve.

Consider a directed graph $D = (V, A)$ with cost $c \in \mathbb{R}^A_+$ on the arcs, arc capacities $u \in \mathbb{R}^A_+$, and demands $d \in \mathbb{R}^V$. The cost of an arc is to be interpreted as the cost per unit of flow. The *cost of a flow* x is just $\text{cost}(x) = \sum_{a \in A} c_a x_a$. The demand of a node is the amount of flow it wants to send (if $d_v > 0$) or receive (if $d_v < 0$). We assume that as much flow needs to be sent as to be received, i.e.,

$$\sum_{v \in V} d_v = 0 \tag{2.26}$$

The minimum flow problem is then the following linear program.

$$\begin{aligned}
\min \quad & \textstyle\sum_{a \in A} c_a x_a && && (2.27)\\
\text{s.t.} \quad & x_+(v) - x_-(v) && = d_v \quad \forall\, v \in V && (2.28)\\
& x_a && \leq u_a \quad \forall\, a \in A && (2.29)\\
& x_a && \geq 0 \quad \forall\, a \in A && (2.30)
\end{aligned}$$

The first set of constraints are the equivalent of flow conservation where we take into account the demand of the node.

As for the maximum flow problem, we construct an auxiliary digraph D_x with $V(D_x) = V$. If $uv \in A$, add a forward arc uv to $A(D_x)$ if $x_{uv} < u_{uv}$ and give it cost c_{uv} and capacity $u_{uv} - x_{uv}$. If $uv \in A$, add reverse arc vu to $A(D_x)$ if $x_{uv} > 0$ and give it cost $-c_{uv}$ and capacity x_{uv}. Using this auxiliary digraph, it is relatively easy to establish an optimality condition for a feasible flow x. Define the *cost of a directed path* to be the sum of the cost of its arcs. We need the following cycle decomposition theorem stated in [1].

Theorem 2.5.1. *Let x_1 and x_2 be two feasible flows. Then x_2 can be obtained by augmenting x_1 along at most m cycles in the auxiliary digraph D_{x_1} and the cost of x_2 is equal to the cost of x_1 plus the cost of the augmenting cycles.*

Theorem 2.5.2. *If x is a feasible solution to the minimum cost flow problem, it is an optimal solution if and only if the auxiliary digraph does not contain a negative cost cycle.*

Proof. ($\Rightarrow$) Suppose there exists a cycle C of cost $-\delta$ in D_x, $\delta > 0$. Let $\epsilon_1 = \min\{u_a - x_a : a \in C, a \text{ is a forward arc}\}$, $\epsilon_2 = \min\{x_a : a \in C, a \text{ is a reverse arc}\}$ and $\epsilon = \min(\epsilon_1, \epsilon_2)$. Then one can augment the flow along C by increasing x by ϵ on forward arcs and decreasing it by ϵ on reverse arcs. We thus obtain a solution x' which is still feasible but which has cost $\text{cost}(x) - \delta\epsilon < \text{cost}(x)$.

($\Leftarrow$) Suppose x is a feasible solution and $x^* \neq x$ is an optimal solution. By Theorem 2.5.1, x^* can be obtained by augmenting x along at most m cycles in D_x. Each of these cycles has non-negative cost so $\text{cost}(x^*) \geq \text{cost}(x)$ but x^* being optimal means that $\text{cost}(x^*) \leq \text{cost}(x)$. Hence, $\text{cost}(x^*) = \text{cost}(x)$ and x is optimal. □

Theorem 2.5.2 provides a nice characterization of when a flow is optimal. It also suggests a simple algorithm to find a minimum cost flow. The pseudo code for this algorithms is shown below.

NegCycleAugmentation

`Start with a feasible flow` x

While D_x `contains a negative cost cycle` C

$\epsilon_1 = \min\{u_a - x_a : a \in C, a$ `is a forward arc`$\}$

$\epsilon_2 = \min\{x_a : a \in C, a$ `is a reverse arc`$\}$

$\epsilon = \min(\epsilon_1, \epsilon_2)$

`Augment` x `by` ϵ `along` C

There exists a lot of more efficient algorithms to solve the minimum cost flow problem such as successive shortest path [40, 39], minimum cost mean cycle augmentation [36] and the network simplex [15].

CHAPTER 3
Unsplittable flows

Unsplittable flows have been introduced Cosares and Saniee in their paper on the ring loading problem [14]. Later, Kleinberg studied them and coined the name "unsplittable" [43, 44]. Unsplittable flows have since been the subject of a huge amount of literature (see, for e.g., [3, 4, 20, 46, 47, 61, 62]). Like network flows presented in Chapter 2 there exists multicommodity and single commodity unsplittable flow problems (UFP). We will only consider single-sink multicommodity UFP. There are several types of questions that one can ask about unsplittable flows but we will focus on only two of them. First, it is not true, in general, that the cut condition is sufficient for the existence of an unsplittable flow. Thus, it is natural to ask how much extra capacity we must add to the edges to guarantee that such a flow exists. We call this the *congestion minimization problem*. In particular we will present the algorithm of Dinitz et al. [20] and the ring loading algorithm of Schrijver et al. [57]. The second question that we will consider is to determine how much capacity one needs to add to get an unsplittable flow whose cost is at most the cost of an optimal splittable flow. This is the *minimum cost UFP*.

3.1 Statement of the single-sink unsplittable flow problem

Consider a graph $G = (V, E)$ directed or undirected with arc capacities $u \in \mathbb{R}^E$, terminals $\{s_1, s_2, \ldots, s_k\}$ and a single sink $t \in V$. Terminals have demand

$d_1, d_2, \ldots, d_k$ respectively. An *unsplittable* network flow is a flow f such that for every terminal s_i there exists $P \in \mathcal{P}_i$ with $f(P) = d_i$ and hence flow is routed along a single path. In other words, the demand coming from a source cannot be divided into smaller demands that route to the sink using different paths. Note that there can be more than one terminal at any given node and each of these terminal can use a distinct path for their demand.

This problem is a specialization of the more general UFP where we are given terminal pairs (s_i, t_i) and we are asked to route the demand between any terminal pair along a single path. This general problem has a lot in common with the edge-disjoint path problem (see Chapter 6). Indeed, a solution to the edge-disjoint path problem gives an unsplittable flow since every terminal pair gets routed along a single path. Of course, MEDP is more restrictive since is requires the paths to be edge-disjoint. Raghavan and Thompson [54] gave a logarithmic approximation for the congestion minimization problem for unsplittable flows using the randomized rounding technique. They actually applied this technique to integral multicommodity flow problems but the same ideas extend to unsplittable flows. Unsplittable flows had actually not been introduced yet when they were published. Hence, it is known that a logarithmic approximation exists for the congestion minimization problem on general instances. It is interesting to note that for the single- sink problem, there is an $O(1)$ approximation algorithm despite it being NP-hard. We later go into considerable details on the current best known algorithm due to Dinitz et al. [20].

The four main questions of interest are the same as the ones presented in Section 2.2. Namely, we can consider the *feasibility problem* (i.e., to determine whether an unsplittable flow exists for a given instance), the *congestion minimization problem* (i.e., to determine what is the minimum increase in congestion for which we can get an unsplittable flow), the *demand maximization problem* (i.e., to determine the subset of maximum demand that can be routed unsplittably without increasing congestion) and the *round minimization problem* (i.e., to find the minimum number of rounds of feasible unsplittable flow necessary to satisfy all demands). All these questions were considered in the thesis of Kleinberg [44]. Note that the exact same questions are of interest for d-furcated and confluent flows which we consider later.

Let OPT_c be the optimal value for the congestion minimization problem, OPT_d be the optimal value for the demand maximization problem and OPT_r be the optimal value for the number of rounds minimization problem.

3.2 Known results

The first published results regarding single-sink unsplittable flows were by Kleinberg [43, 44]. Note that Kleinberg considered flow from t to the s_i's and it was thus a single-source flow problem. In order to be consistent with Chapter 5, we consider the equivalent single-sink version. He presented constant factor polytime approximation algorithms to solve the congestion minimization, the demand maximization and the number of rounds minimization problems. He made the no bottleneck assumption and scaled the demands and capacities such that all capacities are at least 1 and all demands are at most 1. The key result that he

uses is due to Ford and Fulkerson [29] (note that a similar result is also used by Skutella in his cost minimization algorithm for unsplittable flows [61]). For $d \in \mathbb{R}$, we say that $x \in \mathbb{R}$ is *d-integral* if it is a multiple of d. A real vector is d-integral if all its components are d-integral.

Theorem 3.2.1. *Let $G = (V, E)$ be an undirected graph, $t \in V$ be the sink, $\{s_1, s_2, \ldots, s_k\} \subseteq V$ be terminals with uniform demand d. Suppose that the capacity vector is d-integral. Then, there exists an unsplittable flow whose value is equal to the value of a maximum (fractional) flow and that flow can be found in polynomial time.*

We now give a high level sketch of the idea used by Kleinberg. The basic idea is to find a particular "tree cover" for G, i.e., a set of trees in G such that every node of G is in at least one such tree. This tree cover is such that every tree contains approximately the same amount of demand and that edges of G are in a "small" number of trees. It can be found by performing a depth-first search on a spanning tree T of G and grouping the terminals as they are encountered while making sure that the total demand in a group is within predefined bounds. Then, he chooses a node to be the *leader* in each tree and he finds a path from the leader to the sink. Using the tree covering, it is then relatively easy to route the nodes in each tree to their leader using the edges of the tree and concatenating this path with the path from the leader to the sink we get the desired unsplittable flow. This idea is very similar to the one we use in Chapter 6 to solve the MEDP problem. Our tree cover is built differently and in particular we wish to choose edge-disjoint trees. These techniques allow Kleinberg to devise an algorithm that routes all

demands unsplittably with congestion at most $(1 + O(\sqrt{d_{max}/\mathrm{OPT}_c}))\mathrm{OPT}_c$. We also briefly mention his other results. His demand maximization algorithm routes $(1 - O(\sqrt{d_{max}}))\mathrm{OPT}_d$. Finally, if $d_{max} < 7/4 - \sqrt{3}$ and a maximum fractional flow exists, then he can route unsplittably in two rounds. On a more negative note, Kleinberg pointed out that all four problems stated above are NP-hard. He presented a reduction from the PARTITION problem.

Kolliopoulos and Stein [46] gave improved approximation algorithms for all problems except the feasibility problem. They start with a maximum flow satisfying all the demands and modify it to make it unsplittable. For the congestion minimization problem, their algorithm is a $(3.23 + o(1))$-approximation algorithm, for demand maximization they obtain a 0.075-approximation algorithm, and for the minimum number of rounds they are able to route all demand in at most 13 rounds, whatever the demand values. The algorithms they use for all these problems are very similar to each other. They are based on a sort of divide-and-conquer approach where they consider the terminals with similar demands as a single subproblem that can be solved independently.

The next major advance in unsplittable flow theory appears in the paper by Dinitz et al. [20] and it largely forms the state of the art for most versions of the problem today. For the congestion minimization problem, they obtain an unsplittable flow where the increase in congestion is at most d_{max}.

Theorem 3.2.2. *Given a fractional flow f satisfying all demands in an instance where the no bottleneck condition holds, there exists a polynomial time algorithm*

that finds an unsplittable flow such that the congestion on any arc a is at most $f_a + d_{max}$.

If all capacities are at most 1, this gives an unsplittable flow with congestion at most 2. They also show that this is best possible by providing an example where any unsplittable flow has congestion arbitrarily close to 2. For the rounds minimization problem, they give an algorithm that routes all demands in 5 rounds and for demand maximization, they can route 0.226 of the total demand provided that the cut condition holds, i.e., a fractional flow for the demands exists. Furthermore, they extend their results to the case where the cut condition is not satisfied, i.e., there does not exists a fractional flow satisfying all demands. The general ideas of their algorithm are given in Section 3.3.

More recently, there is interest in a generalization of unsplittable flows called *k-splittable flows* [5, 22, 45] where each demand can be routed along k paths instead of just one. We do not discuss this problem.

3.3 Solving the congestion minimization unsplittable flow problem

Dinitz et al. [20], like Kolliopoulos and Stein, start from a fractional flow satisfying all demands in their single-sink (actually single-source for them) multicommodity flow problem, and then modify it to get an unsplittable flow. To be consistent with the rest of this thesis, we adapt their result to the case of single-sink multicommodity flows. This can be done by simply reversing the direction of every arc. The capacity of an arc can be taken to be the (fractional) flow on that arc in the originally computed flow, call it f. Their algorithm proceeds to "move" the terminals towards the sink while keeping track of the path that

each terminal follows. When all terminals have reached the sink, the paths they followed gives the unsplittable flow. At any point, a terminal is said to be *irregular* if its demand is less than the capacity on an arc out of the node containing the terminal. Such a terminal can be moved along that arc and the arc capacity reduced accordingly. When all terminals have been moved so that they are *regular*, i.e., all arcs leaving the node at which a terminal sits have capacity less than the demand of that terminal, they start to look for so-called *sawtooth cycles* (which they call *alternating cycles* in their paper).

A sawtooth cycle (whose name was coined in [10]) is a simple directed cycle of the form $(P_1, \bar{P}_2^{-1}, P_3, \bar{P}_4^{-1}, \ldots, P_{\nu-1}, \bar{P}_\nu^{-1})$ where P_i is a directed path called a *forward path* and $\bar{P}_j^{-1}$ is a *reverse path*. Reverse paths are the reverse order of the nodes in a directed path $\bar{P}_j$. Moreover, these paths have the following properties:

- the last node of P_i is the last node of $\bar{P}_{i+1}$ for all $i \in \{1, 3, \ldots, \nu - 1\}$;
- the first node of $\bar{P}_i$ is the first node of P_{i+1} for all $i \in \{2, \ldots, \nu\}$ where we define $P_{\nu+1} = P_1$;
- all nodes on a forward path except the first one have outdegree one.

Sawtooth cycles are a recurrent theme in this thesis. They are used to find d-furcated flows and confluent flows for instance. In Section 4.3 we see that in the applications we are interested in, forward paths can be replaced by a single forward arc as a result of so-called "contraction" operations to be defined later.

We now sketch the algorithm of Dinitz et al. for congestion minimization. Dinitz et al. look for sawtooth cycles of a particular kind; namely, ones in which the first node of each forward path is a terminal. They augment the flow on

these sawtooth cycles by increasing the flow on forward paths and decreasing it on reverse paths. By doing the augmentations carefully and then moving the terminals according to specific rules they guarantee that, for each arc, by removing the flow from at most one terminal, the remaining flow on that arc is at most its original capacity. Thus, in the final flow, the total demand on each arc is at most its original capacity plus d_{max}. This implies Theorem 3.2.2.

3.4 The ring loading problem

The ring loading problem is a subclass of the general unsplittable flow problem. The interest of this class of problems arose from the design of optical networks. For instance, the Synchronous Optical Network (SONET) standard uses rings of optical fibres. In the ring loading problem, we consider an undirected graph $G = (V, E)$ which is a cycle of length n. The nodes are named sequentially on the cycle from 1 to n. The edge between nodes i and $i + 1$ is named edge i. See Figure 3–1 for naming conventions. We use the interval notation to denote only the integer values in this interval, e.g., $[i, j)$ denotes the set $\{i, i + 1, \ldots, j - 1\}$. Arithmetic is done modulo n. The demands can be represented as a symmetric $n \times n$ matrix whose entry $d_{i,j}$ is the demand for a terminal pair (s, t) with s at node i and t at node j. We suppose that $d_{i,i} = 0$ for all i. We abuse the terminology and refer to (i, j) with i and j in V as a terminal pair. Let $X = \{(i, j) : d_{i,j} > 0 \text{ and } i < j\}$ be the set of terminal pairs with nonzero demand. For each terminal pair $(i, j) \in X$, we simply want to determine whether the flow is going to be sent on edges $[i, j)$ (we then say that flow is sent *clockwise*) or on edges $[j, i)$ (we then say that flow is sent *counter-clockwise*). Let $\phi \in \{0, 1\}^X$ be an *assignment*

or *routing* vector that determines whether the demand for a given pair is sent clockwise ($\phi_{i,j} = 1$) or counter-clockwise ($\phi_{i,j} = 0$).

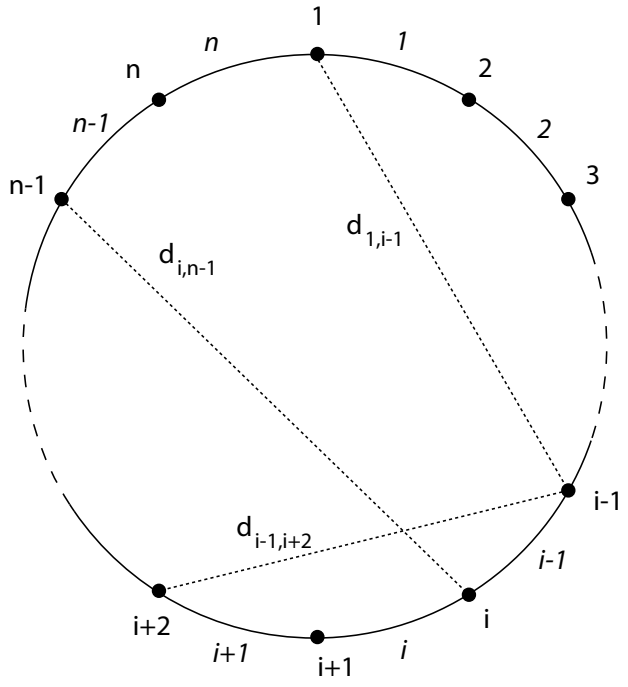

Figure 3–1: An instance of the ring loading problem. Node names are indicated in regular type and edge names are in italics. The chords show demands between various pairs of terminals.

Once ϕ has been determined, the flow on each edge k is given by

$$f_k = \sum\{d_{i,j} : \phi_{i,j} = 1 \text{ and } k \in [i,j)\} + \sum\{d_{i,j} : \phi_{i,j} = 0 \text{ and } k \notin [i,j)\} \qquad (3.1)$$

and the congestion minimization problem, also called the *ring loading problem*, is to find ϕ such that $\max_{1 \le k \le n} f_k$ is minimized. This in an integer programming

problem since we have the restriction that ϕ is an integral vector and the decision version of this optimization problem is NP-complete as can be shown by a reduction from PARTITION [31]. A natural relaxation of this problem is to allow $\phi \in [0,1]^X$ (where $[0,1]$ represents the real interval). Let ϕ^* be such a fractional assignment. Then, $\phi^*_{i,j}$ is the fraction of the demand that is routed clockwise and $1 - \phi^*_{i,j}$ is the fraction that is routed counter-clockwise. The flow on edge k in this fractional routing is given by

$$f^*_k = \sum \{d_{i,j}\phi^*_{i,j} : k \in [i,j)\} + \sum \{d_{i,j}(1 - \phi^*_{i,j}) : k \notin [i,j)\} \tag{3.2}$$

This relaxation can be solved in polynomial time using general LP but there are also simple polynomial time combinatorial methods (see the paper by Schrijver et al., for instance, [57]).

We now present an overview of the algorithm proposed by Shrijver et al. [57] to solve the ring loading problem. Their main result is the following.

Theorem 3.4.1. *Let ϕ^* be a solution to the relaxed ring loading problem that gives a fractional flow f^* of minimum congestion. Then, there is an integral assignment ϕ such that* $\max_k f_k \leq (\max_k f^*_k) + \frac{3}{2}d_{max}$.

First, they propose a polynomial time algorithm that finds an assignment ϕ^* such that at most $n/2$ of the terminal pairs have their demand *split*, i.e., $0 < \phi^*_{i,j} < 1$. Once they have such a solution, they go through an "unsplitting" phase where split demands are sent either completely clockwise or completely counter-clockwise while trying to keep the congestion as low as possible. We now describe some of the details.

If $(i,j) \in X$ and $(g,h) \in X$, we say that their demands *cross* if exactly one of i or j lies in the interval (g,h). If demands do not cross, they are *parallel*. As in Figure 3–1, demands can be represented as chords and we say that edges lying between the chords of parallel demands are *between* the demands. An assignment ϕ^* for the relaxed ring loading problem is said to be *minimal* if for all other assignments ϕ' the flow on every edge is $f'_k \geq f^*_k$. For a minimal assignment ϕ^*, we have the following lemma.

Lemma 3.4.2. *Let ϕ^* be a minimal assignment for the relaxed ring loading problem. Then, if an edge lies between two parallel demands it does not receive flow from both demands.*

Proof. Suppose edge k receives flow a from demand $d_{i,j}$ and flow b from demand $d_{g,h}$, and these demands are parallel. Suppose, without loss of generality, that a is less than b. Consider sending an amount of flow a in the opposite direction for both demands. Then, the flow on all edges between the demands decreases by $2a$ and the flow on the remaining part of the ring stays the same. Doing so results in a new feasible flow satisfying all demands but where at least one edge, namely k, has a strictly lower flow. This contradicts the minimality of ϕ^*. Thus, any edge between parallel demands does not receive flow from both demands. □

This lemma says that in a minimal routing ϕ^*, all split demands cross. Hence, we can assume from now on that we have such a flow. It is now possible to remove all the nodes that are not an endpoint for a split demand. Suppose (i,j) is not split and is routed on the path $[i,j)$. Then, remove $d_{i,j}$ from the flow on each edge of this path and delete the demand. If a node i is not part of any terminal pair,

delete it and add an edge $\{i-1, i+1\}$ with $f^*_{\{i-1,i+1\}} = f^*_{i-1}$. Repeat until only split demands remain.

After performing these operations, we have a ring with an even number of nodes where all demands are split. We now limit ourselves to crossing demand instances on a ring. Since a demand must cross all others, each demand must be of the form $d_{i,i+n/2}$ where n is the number of nodes in the ring.

Define $u_i = \phi^*_{i,i+n/2} d_{i,i+n/2}$ and $v_i = (1-\phi^*_{i,i+n/2}) d_{i,i+n/2}$. Since the demands are split, both of these quantities are greater than zero. There are two possible choices to unsplit the demand $d_{i,i+n/2}$: either we send the clockwise flow u_i counter-clockwise, or we send the counter-clockwise flow v_i clockwise. Either way, the flow on one of the intervals $[i, i+n/2)$ or $[i+n/2, i)$ increases and the flow decreases by the same amount on the other interval. In particular, the change in flow on edge $j \in [i, i+n/2)$ is the negative of the change in flow on edge $j+n/2$. Let z_i be a variable which is equal to v_i if we set the variable $\phi_{i,i+n/2}$ to 1 and to $-u_i$ otherwise. We need to define these quantities only for $i \in [1, 2, \ldots, n/2]$ since this covers all the demands. Then, the change in load on edge $j \in [1, n/2)$ in the new unsplittable flow is given by

$$M_j = \sum_{i=1}^{j} z_i - \sum_{i=j+1}^{n/2} z_i \tag{3.3}$$

and by symmetry the change on edge $j+n/2$ is $-M_j$. Schrijver et al. show that there is a way to set the values of the z_i's such that the sum of the first j values is always in the interval $[-d_{max}/2, d_{max}/2]$.

Lemma 3.4.3. *It is possible to set the values of z_i such that*

$$\sum_{i=1}^{j} z_i \in \left[-\frac{d_{max}}{2}, \frac{d_{max}}{2}\right]$$

for all $j \in \{1, 2, \ldots, n/2\}$.

Proof. The proof is by induction. It is true for z_1 since $u_1 + v_1 = d_1 \leq d_{max}$ thus at least one of u_1 or v_1 is less than or equal to d_{max}. Setting z_1 accordingly ensures that the inequality holds for $i = 1$. Suppose that $\sum_{i=1}^{j} z_i \in [-d_{max}/2, d_{max}/2]$. If $\sum_{i=1}^{j} z_i - u_{j+1} < -d_{max}/2$, then

$$\begin{aligned}
\sum_{i=1}^{j} z_i + v_{j+1} &= \sum_{i=1}^{j} z_i - u_{j+1} + u_{j+1} + v_{j+1} \\
&\leq \sum_{i=1}^{j} z_i - u_{j+1} + d_{max} \\
&< \frac{d_{max}}{2}
\end{aligned}$$

and thus setting $z_{j+1} = v_{j+1}$ gives the desired inequality. If $\sum_{i=1}^{j} z_i - u_{j+1} \geq -d_{max}/2$, then we may set $z_{j+1} = -u_{j+1}$. □

Theorem 3.4.1 easily follows. Note that

$$\begin{aligned}
M_j &= \sum_{i=1}^{j} z_i - \left(\sum_{i=1}^{n/2} z_i - \sum_{i=1}^{j} z_i\right) \\
&= 2\sum_{i=1}^{j} z_i - \sum_{i=1}^{n/2} z_i
\end{aligned}$$

and both of these sums are in $[-d_{max}/2, d_{max}/2]$ so $M_j \in [-\frac{3}{2}d_{max}, \frac{3}{2}d_{max}]$. The maximum increase in congestion is thus $\max_{1 \leq j \leq n/2} |M_j| \leq \frac{3}{2}d_{max}$.

3.5 Minimum cost unsplittable flows

In this section we present the algorithm of Skutella [61] for approximating the minimum cost single-sink UFP in directed graphs. Throughout, we make the no bottleneck assumption. We consider an instance of the UFP where arcs have costs $c \in \mathbb{R}^A_+$. Note that the congestion algorithms discussed so far do not consider the existence of costs. Skutella gives a 3-approximation algorithm in terms of congestion that gives an unsplittable flow $\bar{f}$ of cost no more than a fractional flow f of minimum cost.

3.5.1 d_{min}-integral demands

We first consider the case where all demands are multiples of one another, i.e., for any pair d_i, d_j we have that $d_i|d_j$ or $d_j|d_i$. By renumbering the demands, we can suppose, without loss of generality, that $d_1|d_2|\ldots|d_k$. We present an algorithm for these particular instances and then we generalize it to arbitrary demands by using a rounding procedure. The following well-known result (see, for e.g., [1]) is crucial in proving that the algorithm works.

Theorem 3.5.1. *Let $D = (V, A)$ be a directed graph with arc capacities $u \in \mathbb{R}^A$, terminals $s_1, s_2, \ldots, s_k \in V$ and a single sink $t \in V$. Terminals have demand $d_1, d_2, \ldots, d_k$ respectively. Suppose that every capacity and demand is a-integral for some nonnegative real number a and that the cut condition holds. Then, there exists a minimum cost a-integral flow satisfying all demands and computable in polynomial time.*

The pseudo code for the algorithm follows. It roughly proceeds as follows. First, start with a fractional flow f_0^1 of minimum cost satisfying all demands. If the

algorithm ever finds an arc with 0 flow, it is removed. Otherwise, it considers the demands in non-decreasing order. For simplicity, we suppose that all demands are different, i.e., $d_1 < d_2 < \ldots < d_k$. If this is not the case, demands that are equal are treated at the same time. At each iteration i, we process the terminal with the next smallest demand d_i. We use the flow from the previous iteration f^1_{i-1} to set new arc capacities u^i_a. Then, we find a d_i-integral feasible flow f^0_i whose cost is at most the cost of the previous flow (such a flow exists by Theorem 3.5.1). Since this flow is d_i-integral and satisfies all demand, there is at least one path P_i from s_i to t and all arcs on this path have at least d_i flow. Hence, the demand of source s_i can be routed along P_i and the flow on this path is then decreased by d_i. The flow at the end of this iteration is denoted by f^1_i. When the algorithm ends, we have a set of paths $P_1, P_2, \ldots, P_k$ along which demand for each terminal can be routed to the sink. The flow on any arc a[1] is at most $f^1_0(a) + d_{max}$ and the cost of the flow is less than the cost of f^1_0. Here is the pseudo code for the algorithm called MinCostDivisible.

[1] In this section, and only in this section, $f^j_i(a)$ is the flow on arc a. We use this notation to avoid confusion in the indices.

MinCostDivisible(G, d, f_0^1)

For `all` i `in` $\{1, 2, \ldots, k\}$

For `all arcs` $a \in A$

$u_a^i \leftarrow \left\lceil \frac{f_{i-1}^1(a)}{d_i} \right\rceil d_i$

$f_i^0 =$ `FindIntegralFlow(`G`,` d_i`)`

`RemoveZeroArcs(`G`,` f_i^0`)`

`Find an` (s_i, t)`-path` P_i

For `all arcs` $a \in P_i$

$f_i^1(a) \leftarrow f_i^0(a) - d_i$

`RemoveZeroArcs(`G`,` f_i^1`)`

`Remove terminal` s_i

RemoveZeroArcs(G, f)

For `all arcs` $a \in A$

If $f(a) == 0$

`Remove` a `from` G

In order to show that the algorithm works, we need the following simple lemma.

Lemma 3.5.2. *Let a and c be distinct positive real numbers, and b be a nonnegative real number. Suppose that b and c are multiples of a. Let x be the smallest multiple of c greater than or equal to b. Then $x \leq b + c - a$.*

Proof. We can write $x = mc$ for some integer $m \geq 0$. Then, $b \in ((m-1)c, mc]$. Since a is also a divisor of c, $(m-1)c = (m-1)na$ for some integer $n > 0$ and

$b \in ((m-1)na, mc]$. But b is a multiple of a so $b \in [((m-1)n+1)a, mc]$, i.e.,

$$\begin{aligned} b + c - a &= b + c - \frac{c}{n} \\ &\geq ((m-1)n+1)a + c - \frac{c}{n} \\ &= (mn - n + 1)\frac{c}{n} + c - \frac{c}{n} \\ &= mc = x \end{aligned}$$

□

Theorem 3.5.3. MinCostDivisible *finds an unsplittable flow $\hat{f}$ whose cost is bounded above by the cost of a minimum cost fractional flow f and for which the congestion increased by at most d_{max}. Moreover, it runs in polynomial time.*

Proof. Since FindIntegralFlow finds a flow of minimum cost given the current arc capacities and these capacities are increasing at every iteration, the cost of the flow never increases. Thus, the final flow has cost bounded by the cost of the original fractional flow f. Moreover, at every iteration we obtain a flow satisfying all remaining demands. If we added back the flow paths found from previous iterations, we would have a flow satisfying all demands, and the previous demands would be routed unsplittably.

In every iteration, finding a d_i-integral flow can be done in polynomial time by Theorem 3.5.1 and finding an (s_i, t)-path can be done in linear time. Hence, the algorithm runs in polynomial time since it executes in exactly k of these iterations.

We only have to show that the congestion increase is at most d_{max}. Consider an arc a. The total flow through a at the end of iteration j, denoted by $\phi_j(a)$, is

$f_j^1(a)$ plus the flow path on P_j if it uses the arc a plus the sum of any flow path from previous iterations using arc a.

$$\phi_j(a) = f_j^1(a) + \sum_{a \in P_j} d_j + \sum_{i<j, a \in P_i} d_i \tag{3.4}$$

The sum of the first two terms is clearly bounded above by the capacity of a at iteration j since $f_j^1(a) + \sum_{a \in P_j} d_j = f_j^0(a) \leq u_a^j$. Now apply Lemma 3.5.2 with $a = d_{j-1}, c = d_j$ and $b = f_{j-1}^1(a)$. Since u_a^j is the smallest multiple of d_j greater than $f_{j-1}^1(a)$, we get that $u_a^j \leq f_{j-1}^1(a) + d_j - d_{j-1}$. Thus

$$\phi_j(a) \leq f_{j-1}^1(a) + d_j - d_{j-1} + \sum_{i<j, a \in P_i} d_i \tag{3.5}$$

Observe that this only holds for $j > 1$. To make this hold for $j = 1$ we need only to define $d_0 > 0$ small enough. Any $d_0 \leq \min(\{d_1\} \cup \{f_0^1(a) \mod d_1 : f_0^1(a) \text{ not } d_1\text{-integral}\})$ will make the inequality hold.

In Equation 3.5, the sum of the first and the last terms is just $\phi_{j-1}(a)$. So we obtain the following recursive formula.

$$\phi_j(a) \leq \phi_{j-1}(a) + d_j - d_{j-1} \tag{3.6}$$

Going through the recursion shows that the sum of d_i is telescoping and considering $j = k$ gives

$$\phi_k(a) \leq \phi_0(a) + d_k - d_0 \leq f_0^1(a) + d_k \tag{3.7}$$

i.e., the increase in congestion is at most $d_k = d_{max}$. □

3.5.2 Arbitrary demands

It is striking perhaps that in the case where demands are multiples of each other, we can achieve a minimum cost flow whose congestion is at most that of Dinitz et al. (which is known to be the best possible). It is conjectured by Goemans that this is actually possible for all demands. However, the best result so far is due to Skutella, and we now outline his extension to general demands. Here he must give up a little more congestion to guarantee a minimum cost flow.

For arbitrary demands $d_1 < d_2 < \ldots < d_k$ (again, if demands are equal, they are treated at the same time), start with a minimum cost fractional flow f_0^1. The idea is to first round down demand d_i to $\bar{d}_i = 2^{\lfloor \log(d_i/d_1) \rfloor} d_1$ for all i so that they satisfy the conditions of Theorem 3.5.1. Then, decrease the flow f so that it is a valid flow for the rounded down demands $\bar{d}_i$. To do this, for each source s_i, as long as the outflow at s_i is greater than $\bar{d}_i$, we find a maximum cost (s_i, t)-path and we decrease the flow on this path as much as possible (we can decrease the flow either by the minimum amount of flow on an arc of the path or by the difference between the outflow and $\bar{d}_i$, whichever is smaller), removing any arc with zero flow. Once this is done, we have a new instance with a fractional flow satisfying all demands. We can apply MinCostDivisible to obtain the set of paths $P_1, P_2, \ldots, P_k$ along which to route the demands of each terminal to the sink. Now, instead of just routing the rounded down demand $\bar{d}_i$ along the (s_i, t)-path P_i, route all of the original demand d_i. Clearly, this gives an unsplittable flow satisfying all demands. The pseudo code for this algorithm is shown below.

MinCostUnsplittable(G, d)

$f_0^1 =$ `MinCostFlow(`G`,` d`)`

For `all` i `in` $\{1, 2, \ldots, k\}$

$\bar{d}_i = 2^{\lfloor \log(d_i/d_{min}) \rfloor} d_{min}$

While $f_{0-}^1(s_i) > \bar{d}_i$

`Find a maximum cost` (s_i, t)`-path` P

$\epsilon = \min(\{f_{0-}^1(s_i) - \bar{d}_i\} \cup \{f_0^1(a) : a \in P\})$

For `all arcs` a `in` P

$f_0^1(a) \leftarrow f_0^1(a) - \epsilon$

`MinCostDivisible(`G`,` $\bar{d}$`,` f_0^1`)`

Using this algorithm, Skutella can show the following result.

Theorem 3.5.4. MinCostUnsplittable *finds an unsplittable flow of cost at most than the cost of a minimum cost fractional flow and the increase in congestion on arc a is at most $f_0^1(a) + d_{max}$.*

The author also prove that the problem of finding an unsplittable flow whose cost is less than the cost of an optimal fractional flow while keeping the congestion as low as possible is NP-hard to approximate with a performance guarantee strictly better than $(1 + \sqrt{5})/2$, i.e., it is NP-hard to find an unsplittable flow where the maximum congestion is strictly less than $(1 + \sqrt{5})/2$ times the optimal solution. Thus, there is still a gap between their algorithm with performance guarantee 3 and the theoretical limit. Our personal attempts to improve this algorithm have failed.

CHAPTER 4
Bifurcated flows and d-furcated flows

We examine a single-sink multicommodity flow problem with constraints on the number of outgoing arcs allowed at a node. This category of problems, includes confluent flows (introduced in [11]) which are discussed in Chapter 5 and d-furcated flows (introduced in [21]) with are discussed in this chapter.

4.1 Statement of the problem

Let $D = (V, A)$ be a directed graph with sink t and terminals $s_1, s_2, \ldots, s_k$ with demand $d_1, d_2, \ldots, d_k$ respectively. Define the *load* of a node v under some single-sink multicommodity flow as the sum of its inflow and its own demand. Alternatively, this is just the outflow at a node. Every node has a *capacity* which is defined to be the maximum load it can receive. The *congestion* of a node is its load divided by its capacity. Since we generally only consider digraphs with uniform node capacities U, the congestion of every node will be the same as its load divided by U. Hence, the load and congestion are the same up to a multiplicative factor. The congestion of a flow is the maximum congestion of a node in the graph. Since we consider uniform capacities, we may also assume that U is simply 1 by scaling. We suppose that the no bottleneck assumption holds, i.e., if each capacity is 1, then $d_{max} \leq 1$.

Without loss of generality, since we only consider node congestion we can also suppose that D has no parallel arcs and no loops. Indeed, parallel arcs can

be replaced by a single arc and loops can be removed by the usual directed cycle augmentation. A flow f is *d-furcated* if every node has at most d outgoing arcs with positive flow. If we allow d to be infinity (which is equivalent to $d \geq n-1$), then there is no constraint on the number of outgoing arcs and the problem is just the standard single-sink multicommodity flow. If $d = 2$, we call the flow *bifurcated*. Finally, for $d = 1$, the flows are called *confluent*. Confluent flows were introduced in [11] and later the theory was more fully developed in [10]; for $d \geq 2$ the notions appear to be first studied in [21].

In this chapter, we will focus on finding constant factor congestion bounds for d-furcated flows for $d \geq 2$. This is normally attacked as follows: first find a fractional flow on the digraph D with the smallest possible congestion. This can be done by doing the node splitting operation described in Section 2.1 and then solving the following linear program.

$$\begin{aligned}
\min \quad & L & & & (4.1)\\
\text{s.t.} \quad & L & \geq x_a \quad & \forall \text{ node arcs } a & (4.2)\\
& x_+(v_-) - x_-(v_-) & = d_v \quad & \forall\, v \in V & (4.3)\\
& x & \geq 0 & & (4.4)
\end{aligned}$$

One thus finds a flow f satisfying all demands with minimum node capacity U. In a second phase, one tries to modify this flow to get a d-furcated flow with a constant (multiplicative) factor increase in congestion. Let the congestion of an optimal fractional flow be OPT and the congestion of an optimal d-furcated flow be OPT^d. The worst case ratio OPT^d/OPT over all instances is called the

congestion gap for d-furcated flows. As mentioned earlier, by scaling the demands, we can suppose that the maximum fractional flow has congestion 1.

4.2 Known results

The congestion minimization problem for d-furcated flows has dramatically different answers in the case where $d = 1$ and $d \geq 2$. In particular, in Chapter 5 we will see that when $d = 1$ we cannot even guarantee a bounded node congestion. We will see that a congestion of $\log(n)$ is possible and that it is best possible. Interestingly, if one allows one extra arc out of each node (i.e., $d = 2$) the situation is a lot better. Donovan et al. [21] showed that the congestion gap for d-furcated flows is bounded above by $1 + 1/(d - 1)$. In particular, for bifurcated flows we have a congestion gap of at most 2. We give a comprehensive sketch of the methods used to obtain these results.

There are no known results for the cost version either of the d-furcated or the confluent flow problem. We have tried, without success, to generalize the algorithm in [21] to the case where arcs have costs. We discuss this further and give a complexity proof for a related problem in the last section of this chapter.

4.3 Sawtooth cycles and sawtooth cycle-free digraphs

We defined sawtooth cycles in Chapter 3. They were first used in [20, 10], but both used particular types of sawtooth cycles. In [21], a structure theorem is given for the general sawtooth cycle detection problem. In particular, they show that either there is a sawtooth cycle somewhere in the digraph or the digraph has a specific structure. This specific structure is that the graph is acyclic and "layered"

in a sense that we make clear later. They also present a polynomial time algorithm that either finds a sawtooth cycles or concludes that there is none.

The definition of a sawtooth cycle that we gave in Chapter 3 is different from the definition that is given in [21] but they are equivalent as we explain now. In [21], they call a sawtooth cycle any sequence $(u_1v_1, P_1^{-1}, u_2v_2, P_2^{-1}, \ldots, u_rv_r, P_r^{-1})$ where u_iv_i are called *forward arcs* and P_i is a path from u_{i+1} to v_i called a *reverse path* (index arithmetic modulo r). However, they also perform *contraction* operations defined as follows. Call a node *decided* if it has outdegree 1. A contraction operation consists of contracting a decided node and its out-neighbour into a single node. In our definition, every node on a forward path except the first one have outdegree one and is thus a decided node. Performing contractions on these nodes gives a sawtooth cycle according to the definition in [21]. For the remainder of the thesis, we adhere to the definition of [21]. In other words, we assume contraction operations are being performed, and hence the forward segments all have length 1.

We now describe the polynomial time algorithm that either finds a sawtooth cycle or shows that none exists. Consider a digraph $D = (V, A)$ without parallel arcs. Check if D has a directed cycle using, for instance, Tarjan's algorithm [64]. If it does, then D has a sawtooth cycle since any directed cycle is a sawtooth cycle. Thus, we suppose from now on that D is acyclic. First, we build an auxiliary digraph $\hat{D}$ as follows. For each node $v \in V$, $\hat{D}$ has two nodes v_- and v_+ and a *node arc* v_-v_+. For every arc $uv \in A$, D has two arcs: a *real arc* u_-v_+ and a *complementary arc* v_+u_-. Define a *clean cycle* to be a simple directed cycle of length at least 3. They then show the following result.

Theorem 4.3.1. *An acyclic directed graph D without parallel arcs contains a sawtooth cycle if and only if $\hat{D}$ contains a clean cycle.*

Proof. ($\Leftarrow$) Let C be a clean cycle in $\hat{D}$. First, observe that D is bipartite with all nodes indexed by "+" in one stable set, we call these the *plus* nodes, and all nodes indexed by "−" in the other stable set, we call these the *minus* nodes. Suppose C has no real arc. Since every arc starting at a plus node is a complementary arc and every arc starting at a minus node that is not real is a node arc, C consists of alternating complementary and node arcs. However, such a cycle corresponds to the reverse of a directed cycle in D and we supposed that D is acyclic. Thus, C must contain at least one real arc u_-v_+. A real arc can only be followed by an odd number of alternating complementary and node arcs since otherwise the ending node a plus node and no real arc starts at a plus node. Thus C consists of sections composed of a real arc followed by an odd number of alternating complementary and node arcs. In D, this corresponds to a forward arc followed by a reverse path, i.e., a sawtooth cycle.

($\Rightarrow$) If S is a sawtooth cycle in D, then it corresponds to a cycle in $\hat{D}$. Since a sawtooth cycle has length at least three (because we do not allow parallel arcs), the resulting cycle in D is clean. □

Thus, looking for a sawtooth cycle in D is the same as looking for a clean cycle in $\hat{D}$. The next step is to determine whether $\hat{D}$ contains a clean cycle. This can be done in polynomial time as is shown in the next subsection.

4.3.1 Detecting clean cycles

The objective is to find a clean cycle in an arbitrary digraph $\hat{D}$. To do this, we introduce the notion of a *digon-tree representation*. Consider the subgraph of $\hat{D}$ obtained as follows. First, eliminate all arcs not in a digon[1] . Now replace every digon by a single undirected edge and call the resulting graph H. Search for cycles in this undirected graph using, say, breadth-first search. If there is a cycle, it must have length at least three since $\hat{D}$ contains no loop and we just removed all digons. Hence, this corresponds to a clean directed cycle in the original digraph. Thus, if $\hat{D}$ does not have a clean cycle, we must have that H is a forest. Next, consider contracting every component of H into a single node to obtain a minor $\mathcal{D}$ of $\hat{D}$. $\mathcal{D}$ is called the digon-tree representation of $\hat{D}$. Note that if $\mathcal{D}$ has a directed cycle, then this would give rise to a clean directed cycle in $\hat{D}$. So we assume that this is not the case, and so $\mathcal{D}$ has an acyclic ordering of its nodes, i.e., of the shrunken components of H. The following theorem is shown in [59].

Theorem 4.3.2. *If $\hat{D}$ is a directed graph without loops, then $\hat{D}$ has no clean cycle if and only if it has an acyclic digon-tree representation.*

Once we have the digon-tree representation, it is easy to check for directed cycles using, for instance, Tarjan's algorithm [64]. Hence, we have outlined a polynomial time algorithm to find clean cycles.

[1] Recall that a digon is a directed cycle of length 2.

4.3.2 Structure theorem for sawtooth cycles

Theorem 4.3.2 can now be used to give a good characterization of the structure of a digraph D without sawtooth cycle. Such a graph has an auxiliary digraph $\hat{D}$ whose digon-tree representation is acyclic. Thus, it can be shown that D can be partitioned into edge-disjoint trees such that every node is in at most two trees, all incoming arcs to a node are in the same tree and all outgoing arcs from a node are in the same tree. Moreover, these trees have an acyclic ordering. The details are provided in [21].We give a pictorial representation of the structure result in Figure 4–1 to help the reader visualize.

4.4 A congestion $1 + 1/(d-1)$ algorithm for d-furcated flows

We now present the algorithm of Donovan et al. [21]. Start by finding a fractional flow f satisfying all demands with minimum congestion. Scale this flow so that the maximum node congestion is 1. Perform the usual simplification of removing any directed cycle by reducing the flow along any such cycle until some arc disappears. Let $X = \{x \in V : xt \in A\}$ be the set of in-neighbours of the sink. We may remove the sink and consider a flow where the terminals have to route to any combination of nodes in X. If at any point of the algorithm an arc has zero flow, we remove it from the graph, i.e., we only work with the support of the flow. The algorithm has two phases.

4.4.1 Phase I

The first operation that is used is Contraction of decided nodes as described in Section 4.3. When such a contraction is performed, we keep track of the

demands by assigning to the new node the sum of the demands of the contracted nodes.

CONTRACTION(D)

While `there exists a decided node` u `with out-neighbour` v

`Contract arc` uv `into a single node` u'

`Assign demand` $d_{u'} = d_u + d_v$ `to the new node`

The second operation is to find a sawtooth cycle and "break" it with a routine called BREAKSAWTOOTH. Suppose we found a sawtooth cycle $S = (u_0v_0, P_0^{-1}, u_1v_1, P_1^{-1}, \ldots, u_rv_r, P_r^{-1}$ where P_i is a reverse path. Let $\epsilon = \min\{f_a : a \in P_i$ for some i$\}$. Then, it is possible to decrease the flow along reverse paths by ϵ and to increase it along forward arcs by ϵ. Doing so does not violate the flow conservation constraints, and the congestion of every node either decreases or stays the same. Indeed, the congestion of internal nodes in a reverse path decreases by ϵ and the congestion of nodes that are the endpoints of a forward arc stays the same. This augmentation drives the flow on at least one arc on a reverse path to zero, which "breaks" the sawtooth cycle.

BreakSawtooth(G, f)

While `there exists a sawtooth cycle` S

$\epsilon \leftarrow \min\{f_a : a \in P_i$ `for some i`$\}$

For `all arcs` $a \in S$

If a `is a reverse arc`

$f_a \leftarrow f_a - \epsilon$

Else

$f_a \leftarrow f_a + \epsilon$

Phase I repeatedly executes Contraction and BreakSawtooth until it is no longer possible. Each of these operations runs in polynomial time. Since the former removes a node at each iteration and the later removes an arc at each iteration, Phase I runs in polynomial time.

4.4.2 Phase II

Phase II starts with an acyclic sawtooth cycle-free graph D (see Figure 4–1). For a source node $s \in D$, the node s_+ in $\hat{D}$ will be in a digon-tree consisting of only itself since no arc of D points into sources. Whenever we have a source node s, we can thus contract s_+ into s_- in $\hat{D}$. Hence, if D still contains some arcs, there is at least one digon-tree in $\hat{D}$ with least two nodes whose minus nodes are sources. A *round* of Phase II consists of processing a digon-tree T^* that has no arcs leaving it, i.e., we process the digon-trees in the reverse of the acyclic ordering given by Theorem 4.3.2. A minus node $v_- \in T^*$ has only complementary incoming arcs, thus the corresponding node v of G only has outgoing arcs in the subgraph of G

corresponding to T^*, i.e., it acts as a source in this subgraph. Also, observe that every minus node in T^* has at least two distinct neighbours. Otherwise, in G, this would correspond to a node v with only one out-neighbour, but then v would be a decided node and a contraction would have been performed. The *steps* in a round consists of processing the minus nodes of T^*.

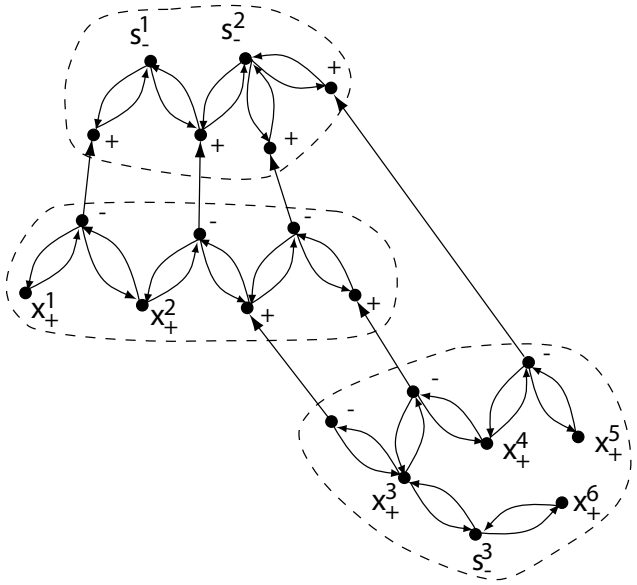

Figure 4–1: The acyclic digon-tree representation of a digraph with sources s_1, s_2, s_3 and sinks $x_1, x_2, \ldots, x_6$.

To choose the minus node to be processed, Donovan et al. use the following lemma.

Lemma 4.4.1. *In a tree T with bipartition classes A and B, if every node in A has degree at least 2, then there exists a node $v \in A$ whose neighbours are all leaves except at most one.*

Applying this lemma to the undirected tree T^* insures that we can find a minus node s_- whose neighbours $u_1, u_2, \ldots, u_r$ are all leaves except at most one, say u_r. This is the node that is going to be processed next using the d-FURCATE routine. At the time where s is processed, it may already have some extra congestion ω coming from previous steps or rounds. Once the flow at node s has been d-furcated, s is removed from G.

d-FURCATE$(G,\ s,\ u_1, \ldots, u_r,\ \omega)$

If $r \leq d$

$f_{su_1} \leftarrow f_{su_1} + \omega$

Else

$\Omega = 0$

For `arcs` su_i `such that` $i > d$

$\Omega \leftarrow \Omega + f_{su_i}$

$f_{su_i} \leftarrow 0$

For `arcs` su_j `such that` $j \leq d$

$f_{su_j} \leftarrow f_{su_j} + (\Omega + \omega)/d$

We no ensure that d-FURCATE operations do not increase the congestion of any node above $1 + 1/(d-1)$.

Theorem 4.4.2. *There exists a d-furcated flow with node congestion at most* $1 + 1/(d-1)$.

Proof. The proof is by induction. Clearly, for the first processed node, there is no extra congestion. Suppose we are processing node s and it has extra congestion

at most $1/(d-1)$ by induction. If $r \leq d$, then its neighbour u_1 receives this extra congestion but it is a leaf, thus after removal of s, it becomes a source whose congestion is still bounded by the desired constant.

If $r \geq d+1$, then the leaf neighbours $u_1, \ldots, u_d$ of s receive the extra congestion of s as well as the flow from the other neighbours $u_{d+1}, \ldots, u_r$. The increase in congestion for the first d leaf neighbours is bounded by

$$\begin{aligned} \frac{1}{d}\left(\sum_{i \geq d+1} f_{su_i} + \frac{1}{d-1}\right) &\leq \frac{1}{d}\left(\sum_{i=1}^{r} f_{su_i} + \frac{1}{d-1}\right) && (4.5) \\ &\leq \frac{1}{d}\left(1 + \frac{1}{d-1}\right) && (4.6) \\ &= \frac{1}{d-1} && (4.7) \end{aligned}$$

The first inequality holds since the outflow at s is greater then the flow sent to any combination of the out-neighbours of s. The second inequality holds since the outflow at s is precisely the congestion at s which is at most 1. Thus, all neighbours that receive extra flow have their congestion increase by at most $1/(d-1)$ and they become sources after the removal of s. Thus the induction carries through and we get the desired result. □

In [21], they also point out that this bound is tight. More precisely, they show the following theorem.

Theorem 4.4.3. *For any $\epsilon > 0$, there exists a digraph admitting a fractional flow with maximum node congestion 1 but for which all d-furcated flows have congestion at least $1 + 1/(d-1) - \epsilon$.*

4.5 d-furcated flows with costs

It is an open question to find the congestion gap for the d-furcated flow problem on digraphs with costs on the arcs. More precisely, we ask what is the congestion required in order to find a d-furcated flow whose cost is no more than the optimal fractional flow. We also do not know of a nontrivial bicriteria approximation. That is, we do not know if we can get within a constant factor of the optimal cost, while maintaining a constant factor congestion. There has been work done on cost version of other bounded degree problems, e.g., minimizing the cost of a spanning tree with bounded degree [55]. In trying to modify the bifurcated algorithm to incorporate costs, the first problem arises with the sawtooth cycle operation. This is because we must reduce flow on the reverse paths in order to maintain the congestion bound. However, it may be the case that this is the expensive direction in terms of cost augmentation.

Suppose we start the algorithm with a fractional flow of minimum cost satisfying all demands. One idea is to modify this flow in order to get an unsplittable flow whose cost is at most the cost of the fractional flow. This could result in an increased congestion, but as long as this increase is within a constant factor of the optimal congestion, we will be satisfied. Build an auxiliary digraph $\hat{D}$ from D in the same way as in Section 4.3. Node arcs all have cost 0, real arc u_-v_+ has cost equal to the cost of uv, and complementary arc v_+u_- has cost equal to the negative of the cost of uv. We add *reverse node arcs* u_+u_- with cost 0. In this auxiliary digraph, every clean cycle corresponds either to a sawtooth cycle or to the reverse of a sawtooth cycle in D, and the cost of the clean cycle is the same

as the cost of the corresponding sawtooth cycle (or reverse sawtooth cycle). If we find a negative cost clean cycle in $\hat{D}$ and it corresponds to a sawtooth cycle in D, then the usual flow augmentation on this sawtooth cycle breaks it and the cost of the new flow is at most the cost of the original flow. If the negative cost clean cycle corresponds to the reverse of a sawtooth cycle, then we could try to augment along the opposite direction, i.e., decrease flow along forward arcs and increase it on reverse paths. Doing so may increase the load of internal nodes on the reverse paths, but we can try to bound this increase by some other fashion.

Suppose there is a good bound on the increase in congestion. Then, we could eliminate all sawtooth cycles while never increasing the cost of the flow. The flow obtained would have a sawtooth free support and thus it would have the nice layered structure described above. We could then hope to find a way to generalize the d-furcation phase of the algorithm. However, even before trying to find a bound on the increase in congestion, there is a major problem with this approach. It lies in the fact that the procedure to find sawtooth cycles using the digon-tree representation outlined in Section 4.3 does not extend to find negative cost sawtooth cycles. To do so we would need to find negative cost clean cycles in $\hat{D}$, but it turns out that this problem is NP-hard. The proof of this is given in [59]. We present this proof but we give all the details in the next section.

Note that this does not imply that the cost version of the d- furcated flow problem is intractable. It only means that new ideas will be needed to tackle the problem.

4.5.1 Hardness of finding negative-cost clean cycles

We want to show that given a digraph $D = (V, A)$ and cost vector $c \in \mathbb{Q}^A$ it is NP-hard to find negative cost clean cycles. This result was shown in [59], however we describe the proof more completely below. Recall that a clean cycle is a directed cycle of length at least 3. The heart of the proof is to show that finding a shortest clean path is NP-hard.

Theorem 4.5.1. *Given a digraph $D = (V, A)$ and cost cost vector $c \in \mathbb{Q}^A$ it is NP-hard to find shortest clean paths.*

Proof. To prove this, we show a reduction from 3-SAT. We have an instance of 3-SAT consisting of m clauses $C_1, C_2, \ldots, C_m$ and n variables $x_1, x_2, \ldots, x_n$. Variable x_k appears in n_k clauses. We now build a directed graph D using two kinds of gadgets.

For each clause there is a clause gadget (see Figure 4–2). Consider a clause C_i with variables x_1, x_2, x_3. There are 8 different assignments for those three variables. The clause gadget has 8 node-disjoint directed paths $p_1^i, p_2^i, \ldots, p_8^i$ and each of these paths has three segments, one for each variable. These segments will consist of a literal gadget which we describe later. Add a vertex s_i and arcs of cost 0 from s_i to the first vertex of each p_j^i. Add another vertex t_i and arcs from the last vertex of each p_j^i to t_i. These arcs all have cost 0 except for the arc that links to the path whose corresponding variable assignment makes the clause false. This arc has cost 1. The clause gadgets are then attached together by identifying t_i and s_{i+1} for $i = 1, 2, \ldots, n-1$ as shown in Figure 4–3.

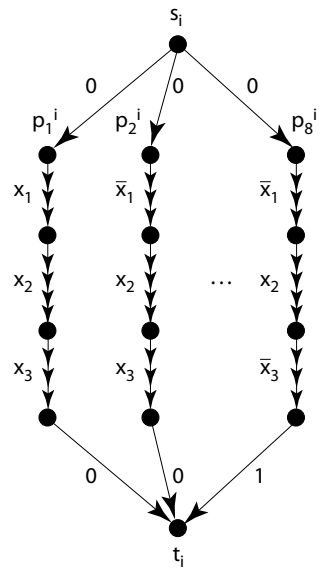

Figure 4–2: Clause gadget for clause $C_i = x_1 \vee \bar{x}_2 \vee x_3$. Note that the ending arc for path p_8^i has cost 1 since this path corresponds to the unique assignment $\bar{x}_1, x_2, \bar{x}_3$ which makes the clause false.

Notice that in a clause gadget for a clause having variable x_k, there are four x_k-gadgets and four $\bar{x}_k$-gadgets. And since x_k appears in n_k clauses, the whole graph has $4n_k$ x_k-gadgets and $4n_k$ $\bar{x}_k$-gadgets.

Now, we describe the literal gadgets. For variable x_k, a literal gadget is a directed path of length $8n_k$ whose arcs have cost L and $-L$ alternatively, starting with an arc of cost L. Thus there are $4n_k$ arcs of negative cost in this path.

We attach the literal gadgets together. Consider variable x_k and the $8n_k$ corresponding literal gadgets labelled $x_k^1, x_k^2, \ldots, x_k^{4n_k}, \bar{x}_k^1, \bar{x}_k^2, \ldots, \bar{x}_k^{4n_k}$. Let x_k^j be the path $u_0^j u_1^j \ldots u_{8n_k}^j$ where the u_l^j are vertices. Similarly, let $\bar{x}_k^j$ be the path $v_0^j v_1^j \ldots v_{8n_k}^j$. Identify u_{2p-1}^j with v_{2j}^p, and u_{2p}^j with v_{2j-1}^p for all $j = 1, 2, \ldots, 4n_k$ and

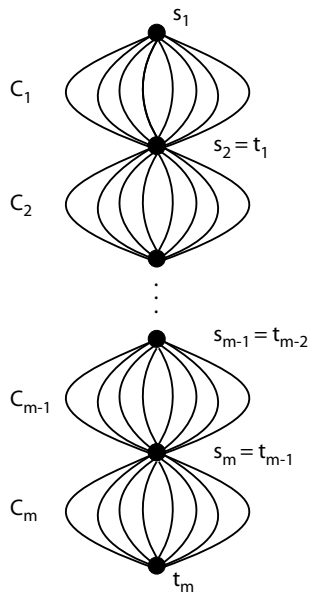

Figure 4–3: The clause gadgets attached together.

$p = 1, 2, \ldots, 4n_k$. An example of this identification with $j = 1$ is provided in Figure 4–4.

The results of this operation is shown in part in Figure 4–5.

This completes the reduction from 3-SAT. We refer to the graph just described as G. In this graph, each x_k-gadget crosses each $\bar{x}_k$-gadget at exactly one directed digon. Each digon is composed of two arcs of cost $-L$ and all arcs of negative cost are in a digon.

We claim that any directed clean path P has cost $\text{cost}(P)$ greater than or equal to the cost of the first arc of the path. This is proved in Lemma 4.5.2. Since a directed path starting at s_1 first encounters an arc of cost 0, we know that any directed clean path starting from s_1 has nonnegative cost. In particular, we have that there is no $s_1 - t_m$ clean path of negative cost.

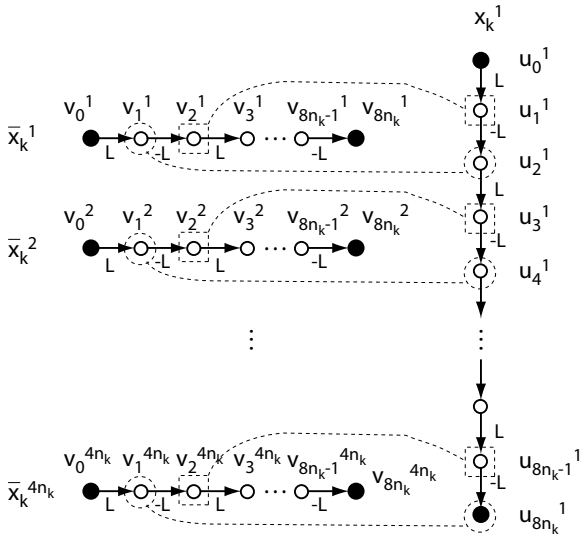

Figure 4–4: An example of how the literal gadgets are attached together. Here, the literal gadget x_k^1 is attached to each of the gadgets $\bar{x}_k^p$ by doing the node identifications along the dotted lines.

Now, we show that finding a shortest clean $s_1 - t_m$ path in G is equivalent to finding a satisfying assignment in the 3-SAT instance.

First, suppose we have a satisfying assignment. Then the shortest clean $s_1 - t_m$ path is obtained as follow. In each clause gadget, follow the path p_j^i from s_i to t_i that corresponds to the assignment. For instance, if clause C_j has variables x_1, x_2 and x_3 and the satisfying assignment is $\bar{x}_1, x_2, x_3$ then follow p_j^i whose literal gadgets are $\bar{x}_1, x_2, x_3$. Since this is a satisfying assignment, in each clause gadget we get a path of cost 0. Indeed, recall that only the false assignment has an ending arc with cost 1, and every literal gadget has as many positive cost arcs as negative cost arcs. So we end up with a unique path of cost 0, i.e., a shortest path since there are no negative cost clean paths. This path is clean since we use the same

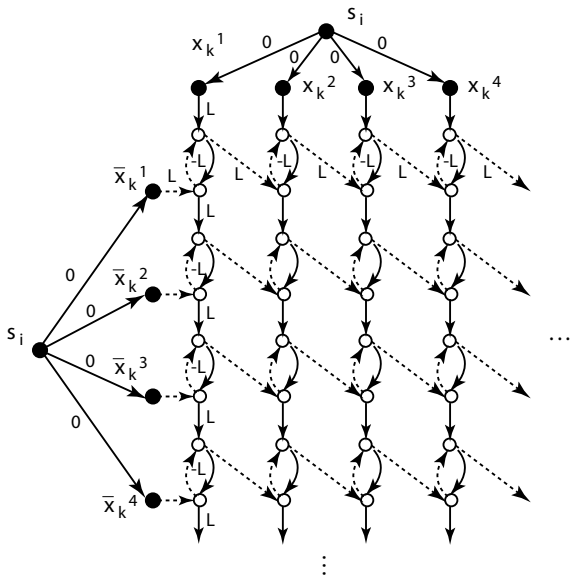

Figure 4–5: x_k-gadgets are shown with solid arcs while $\bar{x}_k$-gadgets are shown with dotted arcs from an $\bar{x}_k$ node. Here we see the eight appearances of the x_k variable in clause C_i and how they are linked together. The arcs in the digons have cost $-L$, the arcs not in the digons have cost L and the arcs leaving s_i have cost 0.

literal for each variable in each clause gadget where this variable appears and the only way to get a digon would be to use x_k in some clause gadget and $\bar{x}_k$ in some other clause gadget. Thus, a satisfying assignment gives a unique shortest clean $s_1 - t_m$ path.

Then, suppose we have found a shortest clean $s_1 - t_m$ path P. If every time this path uses one arc in a literal gadget then it uses all the arcs in that gadget (i.e., in each clause gadget a path p^i_j is completely traversed), we say that the literal gadget is completely traversed. In a shortest path, a literal gadget is either completely traversed or not traversed at all. Indeed, if a literal gadget is not completely traversed, then at some point an arc a_1 of cost L is immediately

followed by another arc a_2 of cost L since this is the only way to leave a literal gadget. Let P_1 be the part of P from s_1 to the endpoint of a_1 and P_2 be the part of P from the endpoint of a_2 to t_m. By Lemma 4.5.2, we have $\text{cost}(P_1) \geq L$ and $\text{cost}(P_2) \geq -L$. Thus

$$\begin{aligned} \text{cost}(P) &= \text{cost}(P_1) + L + \text{cost}(P_2) \\ &\geq L + L + (-L) = L \end{aligned}$$

Thus, the cost of P is at least L. Note that here exists a clean $s_1 - t_m$ path Q of cost at most m which is just a path corresponding to an assignment for which some of the clauses are false. This path completely traverses literal gadgets that it touches. If we select $L > m$, then P is a shortest path only if it traverses variable gadgets completely or not at all since otherwise $\text{cost}(P) \geq L > m = \text{cost}(Q)$ and P is not a shortest path. As a consequence, a shortest clean path always corresponds to a consistent assignment. If this path has cost 0, then the assignment satisfies all the clauses. If not, then the 3-SAT instance is not satisfiable.

Since 3-SAT is NP-hard, so is shortest clean path.

□

Now we prove the following lemma which we used in the preceding proof.

Lemma 4.5.2. *In the graph G described in the proof of Theorem 4.5.1, let P be a directed clean path such that the first arc has cost c_f and the last arc has cost c_l. Then the cost of P is bounded by* $\text{cost}(P) \geq c_f$. *Moreover, if P has more than one arc then* $\text{cost}(P) \geq c_f + \max\{0, c_l\}$.

Proof. We prove this by induction on the number of arcs of the path. If the path has only one arc, it has cost c_f. Suppose we have a path P with $l-1$ arcs and cost $\text{cost}(P) \geq c_f + \max\{0, c_l\} \geq c_f$. If the last arc of P has cost L, we know by induction that removing this arc from P gives a clean path P' with cost $\text{cost}(P') \geq c_f$. Thus P has cost at least $c_f + L$ and adding any arc with cost c'_l at the end of P gives a path of l arcs with cost at least $c_f + \max\{0, c'_l\}$. If the last arc of P has cost $-L$, then we just followed an arc in a digon. The only arc of negative cost we can append to P is in the same digon but this is not permitted in a clean path. So we can only add an arc of nonnegative cost. If the last arc of P has cost 1, we are at a vertex s_i and the only arcs leaving s_i have cost 0. Finally, if the last arc has cost 0, then we are either at a vertex s_i or at the first vertex of a literal gadget. In both cases, there are no negative cost arc leaving that vertex. Thus, the only arcs we can append to P lead to a longer clean path of cost at least $\text{cost}(P) + c'_l \geq c_f + c'_l$ where c'_l is the cost of the appended arc. □

Lemma 4.5.3. *There is no negative cost clean cycles in the graph G described in the proof of Theorem 4.5.1.*

Proof. For any directed clean cycle C we can choose two vertices on the cycle, v_1 and v_2, and consider the paths P_1 from v_1 to v_2 and P_2 from v_2 to v_1 such that $C = P_1 \cup P_2$. By Lemma 4.5.2

$$\text{cost}(C) = \text{cost}(P_1) + \text{cost}(P_2) \geq c_f^1 + c_f^2 + \max\{0, c_l^1\} + \max\{0, c_l^2\}$$

where c_f^i and c_l^i are the costs of the first arc and the last arc of P_i respectively. If c_f^1 and c_f^2 are nonnegative, we are done. Thus we only need to worry about

the case where at least one of the first arc has cost $-L$. If $c_f^1 = -L$, then $c_l^2 = L$ otherwise C is not clean. Thus $\text{cost}(C) \geq -L + c_f^2 + \max\{0, c_l^1\} + L = c_f^2 + \max\{0, c_l^1\}$. If c_f^2 is nonnegative, we are done. Otherwise, $c_l^1 = L$ and we are also done. □

We can use Theorem 4.5.1 to show that finding a negative cost clean cycle is NP-hard.

Corollary 4.5.4. *Finding a negative cost clean cycle is NP-hard.*

Proof. We use the same reduction as in the preceding proof. Add an arc of cost -1 from t_n to s_1. Then, if we find a shortest clean $s_1 - t_m$ path of cost 0 we obtain a clean cycle of cost -1 by appending the arc $t_n s_1$ to this path and by Lemma 4.5.3 this is the only negative cost clean cycle. So a satisfying assignment gives a negative cost clean cycle and a negative cost clean cycle gives a satisfying assignment. So finding a negative cost clean cycle is NP-hard. □

CHAPTER 5
Confluent flows

We study the *single-sink multicommodity confluent flow* problem. This problem was introduced in [11] and further studied in [12, 10]. First, we state the problem and give some known results. We then present two algorithms: one for the congestion minimization problem and the other for the demand maximization problem.

5.1 Statement of the problem and known results

In the *single-sink multicommodity confluent flow* problem, one is given a simple graph (directed or undirected) $G = (V, E)$ and a sink t. In addition, we have terminals $s_1, s_2, \ldots, s_k$ where each s_i wishes to route d_i units of flow to t. A *confluent flow* is one where the flow out of any node must travel on a single edge. Confluent flows are a special case of d-furcated flows where $d = 1$ and also of unsplittable flows. Alternatively, one can view confluent flows as partitioning V into node-disjoint trees (or arborescences, for directed graphs) T_i rooted at distinct neighbours t_i of t, and the flow is routed from the terminals along the edges of T_i's accordingly. The *load* (or *congestion*, assuming that all node capacities are 1) of a confluent flow is then simply the largest flow on one of the edges into t. Since we are only concerned with the maximum node load and uniform capacities, we can scale our demands and hence assume without loss of generality that the d_i's lie in $[0, 1]$. We then refer to a confluent flow as *feasible* if its load is at most 1.

One natural objective is to find a confluent flow such that the maximum load is minimized, i.e., the congestion minimization problem. In [10] it is proved that there is a confluent flow whose maximum load is $1 + \log k$. Moreover, they show that, in directed graphs, it is NP-hard to determine the minimum congestion of a confluent flow to within a factor of $\frac{1}{2} \log k$. Given this hardness result, it is perhaps surprising that they can also devise an $O(1)$-approximation for the demand maximization confluent flow problem. Namely, if the instance admits a fractional flow satisfying all demands, there is a feasible confluent flow which routes $\Delta/3$ of the demand where $\Delta = \sum_i d_i$. A few comments are in order. First, in the factor 3-approximation, they guarantee that in the confluent flow of $\Delta/3$, each demand either entirely routes its d_i units, or routes nothing (the *unsplittable* case). Second as they point out, the initial instance may not have a standard flow for all the demands (with maximum load of 1). In this case, one may run an algorithm, [20] or [43], for the maximum unsplittable flow problem, to unsplittably route some amount of demand within a constant factor (where the constant is 0.226) of the optimal. Applying the confluent algorithm to the resulting flow, then yields a factor 13.29-approximation for the maximum confluent flow problem.

In [10], they also raise the question of whether there is an $O(1)$ approximation for the "routing in rounds" version of confluent flow. More specifically, they point out that their techniques show that any instance can be routed in $O(\log n)$ rounds of feasible confluent routing but that it may be that a constant, or even two, rounds suffice. This remains an intriguing open question.

5.2 Minimizing congestion

We present the congestion minimization algorithm of Chen et al. [10]. Throughout, we let $D = (V, A)$ be the simple input digraph (the undirected case follows from the directed version) and we suppose that we have an initial standard network flow f that routes all of the demands such that the maximum node load is 1. As they do, we actually ignore the sink t, and consider only the neighbours of t denoted by $T = \{t_1, t_2, \ldots, t_k\}$; they call these *sinks* since, without loss of generality, we may assume our starting flow has an acylic support and also that there are no arcs between these nodes (since we only require node loads to be at most 1). As the algorithm runs, it will modify the flow, and we let $b \in \mathbb{R}^k_+$ denote the current vector of node loads of the sinks.

A *frontier node* is a node u that has an *out-neighbour* which is a sink, i.e., there is an arc ut_j for some j. (By our assumptions, no sink is a frontier node.) A *decided node* is a frontier node that has exactly one out-neighbour (note that in Chapter 4 we defined a decided node as any node with exactly one out-neighbour; for simplicity, we use the new definition in this chapter). *Node aggregation* is an operation that refers to *marking* an arc uv for some decided node u and then aggregating u into v, removing any loops thus created.

A *remote node* is some sink t_j that has only one in-neighbour u, called a *pivot node*, amongst the frontier nodes, but u itself also has at least one other sink out-neighbour t_i. Finally, let $\hat{D}$ be the graph obtained from $D \setminus \{t\}$, by adding a reverse arc $t_j u$ for every arc of the form ut_j. In [10], they refer to a simple directed cycle of length greater than two in $\hat{D}$ as a *sawtooth cycle*. Note that this is just a

particular case of the more general definition of a sawtooth cycle given in Chapter 3.

The algorithm in [10] repeatedly performs three operations: *node aggregation*, *sawtooth cycle breaking*, and *pivoting*. (We defer the definitions of the latter two for a moment.) These operations gradually contract arcs from decided nodes to sinks until the only nodes remaining are the sinks t_i. Reversing the contractions then reveals a collection of node-disjoint arborescences rooted at the t_i's. Note that these steps are similar to the ones in the congestion minimization algorithm for unsplittable flows presented in Chapter 3. Indeed, aggregating a decided node with its sink is basically the same as moving the "source" located at that decided node towards the sink. Chen et al. show that if there are still non-sink nodes and there is no possible node aggregation or sawtooth cycle operation, then there is some sink node that is remote. Hence we can perform the PIVOT operation. We now specify the last two operations in more detail.

First, consider some sawtooth cycle S. In D, the sawtooth cycle is a cycle with *forward arcs*, i.e., arcs of D, and *reverse arcs*, i.e., arcs of D that are used in the reverse direction. Note that the reverse arcs are always used from a sink to a frontier node and so any reverse subpath is of length 1 exactly. It follows that the operation does not increase the load of any node since flow is only increased on reverse arcs. Moreover, at least one of the arcs will have its flow decreased to 0; that arc may then be eliminated from $\hat{D}$.

BreakSawtooth(D, S, f)

$f_{min} = \min\{f_a : a \in S\}$

For `all forward arcs` a `of` S

$f_a \leftarrow f_a - f_{min}$

For `all reverse arcs` a `of` S

$f_a \leftarrow f_a + f_{min}$

We now show that if there are still non-sink nodes in the graph and it is no longer possible to do the node aggregation or sawtooth cycle breaking operations, then there exists a remote node on which to do the Pivot operation.

Lemma 5.2.1. *Suppose that D has at least one non-sink node and it is no longer possible to do the node aggregation or sawtooth cycle breaking operations, then there exists a remote node.*

Proof. Start a walk at any node of $\hat{D}$ such that, whenever possible, the walk does not go to a sink and it does not use arc uv right after vu. Every node in $\hat{D}$ has outdegree at least 1 since every sink as at least one reverse arc into a frontier node and every other node is connected with the rest of the graph. Thus, it is possible to walk indefinitely. Since the graph has a finite number of nodes, there is a node v that will be the first to be visited twice. Since there is no sawtooth cycle in D, there is no directed cycle of length greater than two in $\hat{D}$. Hence, the last two arcs in the walk were vw and wv for some node w. Moreover, D has no directed cycles so one of these arcs must be a reverse arc, i.e., v or w is a sink. If possible, the walk would have avoided arc wv, so w has all its outgoing arcs into v. If v was a sink, w would be a decided node which contradicts the impossibility of doing

a node aggregation. Thus, w is a sink and it has only one in-neighbour v. v has at least another out-neighbour otherwise it would be a decided node and all its out-neighbours are sinks since otherwise the walk would have avoided going to w. Thus, w is a remote node. □

We refer to the following operation where we move flow from one arc to another as *pivoting*. Pivoting starts with a frontier node u with two out-neighbours t_i, t_j where in addition t_j is remote.

PIVOT(D, u, b, f)

If $b_j + f_{ut_i} \leq b_i - f_{ut_i}$

`Remove` ut_i

$f_{ut_j} \leftarrow f_{ut_j} + f_{ut_i}$

Else

`Remove` ut_j

$f_{ut_i} \leftarrow f_{ut_i} + f_{ut_j}$

`Deactivate sink` t_j

Pivoting is the only operation that increases the load at some sink. To show the $O(\log(n))$ bound on congestion, we use a *potential function* $\phi : T \to \mathbb{R}$ defined for sink t_i by $\phi(t_i) = 2^{b_i}$. The sum of the potential of all active sinks is referred to as the *potential of the flow*. Note that for the initial flow, the potential is at most $2k$.

Lemma 5.2.2. *During the execution of the algorithm, the potential of the flow never increases. Moreover, when a sink is deactivated, its potential is no more than the potential of the flow before the deactivation.*

Proof. The node aggregation and sawtooth cycle operations do not increase the load of any node thus they don't increase the potential of any sink. In a pivot operation, first suppose that no sink is deactivated, i.e., $b_j + f_{ut_i} < b_i - f_{ut_i}$. This implies $b_j < b_i$. Then, only the potential of t_i and t_j change. We can write $b_j + f_{ut_i} = \lambda b_j + (1-\lambda) f_{ut_i}$ for some $\lambda \in (0,1)$. By convexity of the potential function

$$\lambda 2_j^b + (1-\lambda) 2_i^b \geq 2^{b_j + f_{ut_i}} \tag{5.1}$$

Similarly, we can obtain

$$(1-\lambda) 2_j^b + \lambda 2_i^b \geq 2^{b_i - f_{ut_i}} \tag{5.2}$$

and adding these two inequalities gives $2_j^b + 2_i^b \geq 2^{b_j + f_{ut_i}} + 2^{b_i - f_{ut_i}}$, i.e., the potential of the flow does not increase.

Now, suppose that a sink is deactivated. Then $b_j + f_{ut_i} \geq b_i - f_{ut_i}$ which implies that $b_j > b_i - 2f_{ut_i}$. So the potential before the deactivation is

$$2^{b_j} + 2^{b_i} > 2^{b_i - 2f_{ut_i}} + 2^{b_i}. \tag{5.3}$$

The same convexity argument as above gives

$$2^{b_i - 2f_{ut_i}} + 2^{b_i} \geq 2^{b_i - f_{ut_i}} + 2^{b_i - f_{ut_i}} = 2^{b_i - f_{ut_i} + 1}. \tag{5.4}$$

Finally, since the load at u is at most 1 we have that $f_{ut_i} + f_{ut_j} \leq 1$ so $2^{b_i - f_{ut_i} + 1} \geq 2^{b_i + f_{ut_j}}$ and this last expression is just the potential of sink t_i after the deactivation. Thus $2^{b_j} + 2^{b_i} > 2^{b_i + f_{ut_j}}$ and the potential of the flow does not increase.

The potential of the deactivated sink is clearly less than or equal to the potential of the flow since the potential of the flow is the sum of the potentials of the sinks. □

It is now easy to prove the main result.

Theorem 5.2.3. *Given an instance of the unsplittable flow problem with k sinks, there is a polynomial time algorithm to find a confluent flow satisfying all demands with congestion at most $1 + \log(k)$.*

Proof. Expanding the marked arcs gives a set of node-disjoint arborescences rooted at the sinks. Thus, we obtain a confluent flow. Initially, the potential of the flow is at most $2k$ since every one of the k sinks has load at most 1. This potential is an upper bound on the potential of any deactivated sink by Lemma 5.2.2. Thus, any deactivated sink has load at most $\log(2k) = 1 + \log(k)$. Any sink that has not been deactivated has a potential that is less than $2k$. Consequently, the load of any sink is at most $1 + \log(k)$.

Finding a sawtooth cycle can be done in polynomial time. At every iteration, an arc or a node is removed. Thus, the algorithm runs in polynomial time. □

The authors of [10] also present an improved version of the algorithm with which they achieve a congestion of at most $1 + \ln(k)$.

5.3 Maximizing satisfied demand

Chen et al. [10] also give an algorithm for approximately solving the demand maximization problem. Similar to the congestion minimization algorithm, the demand maximization algorithm repeatedly performs nodes aggregations, sawtooth

cycle breaking and pivoting. However, the pivoting step is slightly different and there is a post-processing phase. We show the pseudo code for the modified PIVOT operation below. Again, s_j is a remote node with in-neighbour u who is adjacent to at least one other sink s_i.

PIVOT(D, u, b, f)

If $b_j - f_{ut_j} \leq 1/2$

`Remove` ut_i

$f_{ut_j} \leftarrow f_{ut_j} + f_{ut_i}$

Else

`Remove` ut_j

$f_{ut_i} \leftarrow f_{ut_i} + f_{ut_j}$

`Deactivate sink` t_j

Note that pivoting is the only operation that increases the load at some sink. Note also that as long as a sink's load is "small", then the pivot will shunt flow onto it. However, when its load goes above 3/2, then the flow is shunted elsewhere and t_j is shut down or *deactivated*. The reason that 3/2 is an upper bound on the cutoff for deactivation is because $f_{ut_j} \leq 1$, and hence if $b_j > 3/2$, then $b_j - f_{ut_j} > 1/2$. It follows that any sink which occurs as a remote node, will either be deactivated, or still has load at most 3/2. Once this part of the algorithm is over (i.e., it is no longer possible to do any of the three operations), we have a set of disjoint arborescences $\{T_1, T_2, \ldots, T_k\}$ where T_i is rooted at t_i. Some of these arborescences, however, may have total demand greater than 1.

The post-processing step finds a subset of the terminals with total demand at most 1 in each tree. If a tree T_i has total demand $\hat{b}_i$ in $(1, 3/2]$, partition the demands into sets of total demand at most $\frac{2}{3}\hat{b}_i$ select the largest of these sets and route the corresponding demands. For trees with total demand larger than $3/2$, it is possible to select a subset of the demands that sum up to at least $1/2$ and these demands can be routed without violating the node capacities.

Using this algorithm, Chen et al. show the following result.

Theorem 5.3.1. *Given a fractional flow f for the demands s_i, d_i with maximum node congestion* 1 *on a digraph D, there is a polynomial time algorithm that finds a confluent flow with maximum node congestion* 1 *that satisfies demands summing up to at least $\Delta/3$, where $\Delta = \sum_i d_i$.*

CHAPTER 6
Rooted clustering and the maximum edge-disjoint path problem in planar graphs

We start by presenting the maximum edge-disjoint paths (MEDP) problem. We then introduce the *rooted clustering problem* which consists of grouping demands in a graph into clusters each of which has a constant amount of demand, with the additional restriction that each cluster must be connected and contain a given root node. Then, using ideas from rooted clustering and confluent flows, we present an algorithm to give a constant factor approximation to the maximum edge-disjoint paths problem on planar graphs with edge-congestion 3, thus improving the previously best known bound of 4 given in [9]. Namely, we prove the following theorem.

Theorem 6.0.2. *There is a polynomial time constant approximation for MEDP in planar graphs, using edge congestion* 3.

The material in this chapter is joint work with Bruce Shepherd [58].

6.1 Statement of the MEDP problem and known results

6.1.1 Maximum edge-disjoint paths

MEDP is formulated as follows. We are given a graph $G = (V, E)$ and a set of terminal pairs $T = \{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ where we assume, without loss of generality, that all nodes are distinct and thus form a matching. Indeed, if a node v appears in more than one pair, say (v, t_1) and (v, t_2), we can

split that node into v_1 and v_2 and the pairs become (v_1, t_1) and (v_2, t_2). Define $X = \{s_1, s_2, \ldots, s_k, t_1, t_2, \ldots, t_k\}$. The nodes in X are called *terminals* and the two terminals in a pair are called *siblings*. For $v \in X$ we denote its sibling by $\sigma(v)$. The objective is to find a collection of paths joining as many terminal pairs as possible with the constraint that these paths need to be edge-disjoint. If a pair has a path joining its two terminals, we say that the demand for the pair has been *routed*. The MEDP problem can be formulated using flows: we search for a maximum (with respect to the number of routed demands) multicommodity unsplittable flow where arc capacities are all 1 and demands between terminal pairs are all 1.

It is well-known that the natural linear programming formulation for MEDP may have a $\Omega(n^\epsilon)$ gap [33] in the worst case (even in undirected planar graphs). In [7], however, it was shown that in planar graphs, the gap is at most $O(\log n)$ if one allows congestion 2 on each edge (i.e., our paths may use each edge up to twice). This is a bicriteria result where the first criteria is the integrality gap and the second is the congestion. More recently, it was shown [9] that there is a constant factor approximation in planar graphs if congestion 4 is allowed. It may yet be, however, that a constant factor is possible with at most congestion 2. We attempt to improve the congestion bound of 4 by focusing on a clustering phase used in the algorithm; in [9] this step incurs a congestion of 2. We show an alternative approach, based on confluent flows, that gives a congestion 1 clustering.

6.1.2 LP relaxation

The natural LP relaxation of MEDP is the following:

$$
\begin{aligned}
\max \sum_{i=1}^{k} x_i \quad & \text{s.t.} \\
x_i - \sum_{P \in \mathcal{P}_i} f(P) &= 0 \quad 1 \leq i \leq k \\
\sum_{P: e \in P} f(P) &\leq 1 \quad \forall e \in E \\
x_i, f(P) &\in [0,1] \quad 1 \leq i \leq k, P \in \mathcal{P}
\end{aligned}
$$

where $\mathcal{P}_i$ is the set of paths between s_i and t_i and $\mathcal{P} = \cup_{i=1}^{k} \mathcal{P}_i$ is the set of all paths in G. The variable x_i is the total amount of flow that is going to be sent from s_i to t_i. The flow sent on a path P is denoted by $f(P)$. It is well known that this problem can be solved in polynomial time.

This is a relaxation in two ways. First, we allow fractional x_i whereas in the MEDP problem a pair has $x_i \in \{0, 1\}$. Second, we allow the demand for a terminal pair to be routed along more than one path instead of just one for the MEDP problem, i.e., in MEDP, f is a 0-1 vector. If we let OPT denote an optimal solution for the LP relaxation, then the optimal solution for the instance of MEDP is clearly less than or equal to OPT. In [9], they show the following theorem.

Theorem 6.1.1. *For the MEDP problem in a planar graph, there is a polynomial time algorithm to route* $\Omega(\text{OPT})$ *demand pairs with congestion* 4*, where* OPT *is the value of the above multicommodity flow LP.*

6.1.3 Outline of the congestion 4 algorithm

In [9], they first do a preprocessing phase to reduce the original graph to one where the degree is bounded by 4. Given a planar graph $G = (V, E)$ it is possible

to transform it to a graph in which every vertex has degree at most 4. This can be done in polynomial time and in such a way as to preserve planarity. The procedure is detailed in [30] and [7]. Thus, they suppose, without loss of generality, that every node has degree at most 4. Also, they show that the problem can be reduced to the two-node connected case. We do not present the details of this reduction.

Solving the LP from the previous section gives a flow f of value OPT. For $v \in \{s_i, t_i\}$, let f_v be the total flow that belongs to this pair, i.e., $f_v = x_i$. f_v is called the *demand* of node v.

In [9], they then show that there exists a subgraph G_C of G, which can be found in polynomial time, with one face being C and with the following properties:

- C has exactly $\left\lceil \frac{1}{10} \sum_{v \in G_C} f_v \right\rceil$ nodes
- Each node v of G_C can send simultaneously a flow of value $f_v/10$ to nodes of C in such a way that no node of C receives more than one unit of flow.

By the first property and the bound on the degree of every vertex, it is possible to show that for such a subgraph G_C, the demand that is routed completely inside G_C is at least 1/10 of the total flow that intersects any node of G_C. Let $b_v \leq f_v$ be the amount of flow sent from node $v \in V(G_C)$ that remains completely inside G_C and

$$p := \frac{1}{2} \sum_{v \in v(G_C)} b_v \tag{6.1}$$

be the total demand that remains inside G_C. They ignore all demand not routed completely in G_C and simply try to route a constant fraction $\Omega(p)$ of the pairs inside G_C. Then they delete G_C and repeat the process. We now focus on the key step of finding $\Omega(p)$ pairs in G_C which can be routed in G_C with low congestion.

If $p \leq 10$, it suffices to route one demand pair to get a constant fraction of the optimal solution, so we can suppose that $p > 10$. The routing in G_C is done in two phases.

Phase I. First, they run a *clustering phase* where they cluster the terminals (with weights b_v) of G_C into edge-disjoint connected subgraphs (called *clusters*) $H_1, H_2, \ldots, H_h$ such that each cluster has demand $\Theta(1)$. They also find paths $P_1, P_2, \ldots, P_h$ that are edge-disjoint in G and such that each P_i has one end on C and one end in H_i. Moreover, these paths have the property that no node of C is the endpoint of more than one path. However, since the P_i's might go through edges of H_j's for $i \neq j$, this procedure leads to a congestion two clustering where the clusters are $H_i \cup P_i$. It is this phase that can be formulated as a rooted clustering problem (where the root is a node with unit capacity edges to each node of C). We defer the definition of rooted clustering to section 6.2. We would like a clustering which is edge-disjoint instead of their congestion 2 scheme.

Phase II. The second phase consists of setting up an Okamura-Seymour (OS), i.e., an instance where all demand pairs lie on the contour C. The OS theorem (see [52]) states that for any such instance, if the cut condition holds and the demands are integral, it is possible to find a routing of the pairs with a congestion of 2. Setting up the OS instance requires some work; we do not give all the details.

The final solution is obtained as follows. There are two cases to consider. First, if there are lots of clusters which contain both siblings from a demand, then we can route that pair in H_i itself; this already produces a (congestion 1 in fact)

routing of a constant fraction of demands. The hard case is where most siblings live in separate clusters H_i. For each i, let u_i denote the endpoint of P_i that lies in C. In this case, they insure that the OS instance created includes demands for pairs (u_i, u_j) with the property that there is some original pair $(v, \sigma(v))$ such that $v \in H_i$ and $\sigma(v) \in H_j$. This pair is then routed as follows. Let $Q_{v\sigma(v)}$ be the path used to route (u_i, u_j) in the OS instance. Let P_v (respectively $P_{\sigma(v)}$) be the path from v (resp. $\sigma(v)$) to an endpoint of P_i (resp. P_j) in H_i (resp. H_j). We then route on the path $P_v, P_i, Q_{v\sigma(v)}, P_j, P_{\sigma(v)}$. We call the non $Q_{v\sigma(v)}$ part of the path the *tails*. Clearly the collection of all $Q_{v\sigma(v)}$ paths induces a congestion of 2 by the OS theorem. Since the H_i's and P_i's induce an edge congestion of at most 2, the collection of all tails also has congestion 2. Thus, the overall routing has congestion 4.

6.2 Rooted Clustering

In the *rooted clustering problem* we are given a graph $G = (V, E)$ (not necessarily simple, and either directed or undirected) with a specified *root* node $t \in V$ also called a *sink*. We are also given *terminals* $s_1, \ldots, s_k$ each with a weight $d_i \in [0, 1]$. A *(unsplittable) rooted clustering* is simply a collection of connected subgraphs (called *clusters*) $H_1, \ldots, H_r$ each containing t, and an assignment $f : \{s_1, s_2, \ldots, s_k\} \to \{H_1, H_2, \ldots, H_r\}$. In other words, if $f(s_i) = H_j$, then terminal s_i is said to be *assigned* to cluster H_j; we also say that s_i is *covered* by H_j. We assume each cluster to be *valid* in the sense that

$$\sum_{s_i \text{ assigned to } H_j} d_i = \Theta(1) \tag{6.2}$$

The *congestion* of such a clustering is the maximum number of times that any edge appears in the list of clusters. If the congestion is 1, then we also refer to it as an *edge-disjoint* rooted clustering. Note that a node (including any terminal) may appear in several clusters. However, we require that each terminal is assigned to a single cluster. This condition can be relaxed to obtain a *splittable* version of the problem, where a demand of d_i may be split across multiple clusters. We also note that the existence of an edge-disjoint rooted clustering implies the existence of an unsplittable flow for the demands with congestion equal to the largest cluster size.

Clustering, i.e., grouping of some weighted terminals s_i, d_i in a graph into so-called *clusters*, is a key step in many approximation algorithms for flow problems. For instance, in [38, 2, 8] a collection of connected subgraphs are sought such that each subgraph contains $\Theta(1)$ of demand. Such clusters are usually easy to construct greedily from a spanning tree. In [9], however, they are actually looking for a rooted clustering with sink t[1] . They are able to find a rooted clustering with congestion 2. More precisely, they show that if G has bounded degree, and G, t and the s_i, d_i's admit a fractional flow with maximum edge-congestion 1, then there is a rooted clustering with congestion 2. Roughly speaking this is achieved by first taking a spanning tree and breaking the tree into edge-disjoint, valid (but non-rooted) clusters covering all of the terminals. Secondly, they use the existence

[1] In [9], the sink is in fact a face of the planar graph, but one can think of adding a new sink node t connected to all nodes of the face.

of the standard network flow for the s_i's to show that a certain cut condition is satisfied, and this implies the existence of edge-disjoint paths from each cluster to t. Combining the clusters with the paths yields the congestion 2 result.

6.2.1 Edge-disjoint rooted clustering

Our main application of rooted clustering is to MEDP where we seek to improve on the congestion 2 clustering above; thus we focus in this section on finding edge-disjoint clusterings. Unfortunately, it is not the case that any instance has an edge-disjoint rooted clustering. This is shown by the following example (see Figure 6–1).[2] In the directed setting, consider a cluster containing the top node. Since the cluster must contain a directed path P to the root, and since all arcs are directed downwards, every terminal on this path must be contained in the same cluster. Hence this cluster includes at least one terminal of demand $\frac{1}{i}$ at each level i, and hence it is invalid since it has a total demand of $\sum_{i=1}^{k} \frac{1}{i} = \Omega(\log n)$. We also believe the undirected version should not admit a valid clustering, but this is an open question.

6.2.2 Partial clusterings via trees

Instead of finding a rooted clustering, where each terminal is assigned to some cluster, we show that one can cluster a large fraction of demands if we are allowed to sacrifice a small (constant) fraction of them. We call this a *partial* clustering.

[2] This example is an extension of an example given in [10] to show a $\Omega(\log n)$ gap for the congestion minimization LP for confluent flows.

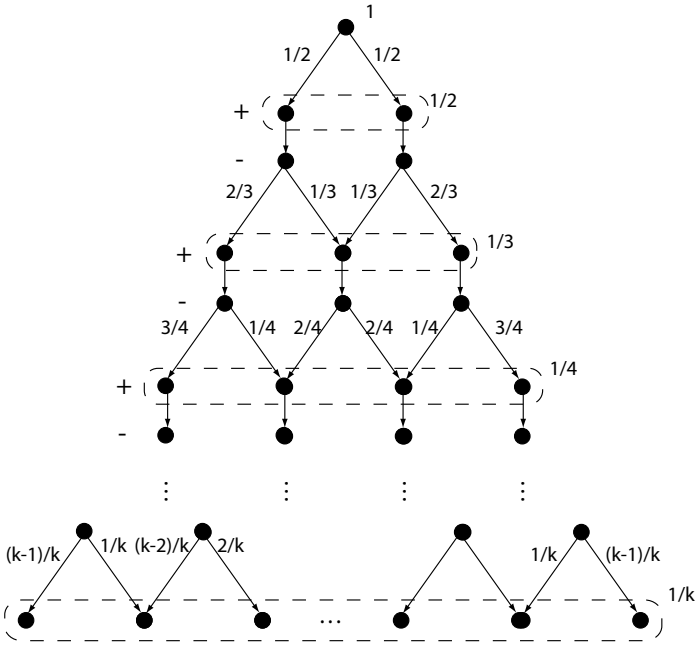

Figure 6–1: There is no arc-disjoint rooted total clustering (splittable or unsplittable).

Theorem 6.2.1 (Cluster Covering)**.** *For any $\kappa \geq 1$ and any instance which admits a fractional flow of maximum edge-congestion* 1*, there is an edge-disjoint rooted partial clustering that covers at least $\frac{\kappa-1}{2\kappa(\kappa+2)}\Delta$ of the total demand $\Delta = \sum_i d_i$.*

To maximize the covered portion of the demand, the optimal value for κ is $1+\sqrt{3}$ which gives a routing for approximately 0.134Δ.

We consider a further relaxation on the clustering that is useful to our application to MEDP. In particular, we allow *incomplete* clustering where not the full demand d_i from a terminal has to be assigned in order to accrue "profit". The techniques used extend to the case where terminals come in pairs (s_i, t_i), each with a demand d_i. (Recall that, without loss of generality, the terminals are all

distinct.) We say that a clustering *captures* $x \leq d_i$ of pair i's demand, if at least x demand from each of s_i and t_i is assigned to some cluster (not necessarily the same one). Specifically we prove:

Theorem 6.2.2 (Cluster Capture)**.** *For any $\kappa \geq 2$, and any instance which admits a fractional flow with maximum edge-congestion 1, there is an edge-disjoint rooted partial incomplete clustering satisfying at least $\frac{\kappa-1}{\kappa(\kappa+2)}\Delta$ of the total demand. If terminals come in pairs, then we can capture at least $\frac{\kappa-2}{\kappa(\kappa+2)}\Delta$ of the pairwise demand.*

In this paired case, the κ value that maximizes the routed demand is $2\sqrt{2}+2$ which gives a routing for approximately 0.0858Δ.

We now turn our attention to proving these results. Our first step is to show that we may reduce to instances where there is a fractional flow such that the load at every node is at most 1. We call these instances *node-normalized*.

6.2.3 Reducing to Node-Normalized Instances.

First, we modify our instance to be node-normalized. We may first assume that our standard flow for the instance is acyclic (note that this makes sense also in the undirected version via the standard bidirection of edges). Next, we may assume that any node v has been split into v^-, v^+ so that any flow destined to t through v traverse an edge/arc v^-v^+. If the total load on any such edge is at most 1, then we are node-normalized already. Otherwise, we make multiple copies of the node v such that each of the new nodes has load at most 1.

Once we have performed the above reduction, we can apply the ideas from [10] presented in Chapter 5 for computing confluent flows on node-normalized

instances. In particular, we use their demand maximization algorithm to route a large fraction of our demands. Note that such a confluent flow actually gives a node-disjoint rooted clustering (not just edge-disjoint). This is in the reduced graph however; it corresponds to an edge-disjoint clustering in our original graph.

6.2.4 From Confluent Flows to Clusters

We first present a modified version of the demand maximization algorithm from [10]. We also modify the analysis slightly so as to obtain the Cluster Covering Theorem.

Throughout, we let $D = (V, A)$ be the simple input digraph (the undirected case follows from the directed version) and we suppose that we have an initial standard network flow that routes all of the demands such that the maximum node load is 1. As in Chapter 5, we actually ignore the sink t, and consider only the neighbours of t denoted by $\{t_1, t_2, \ldots, t_k\}$; we call these *sinks* since, without loss of generality, we may assume our starting flow has an acylic support and also that there are no arcs between these nodes (since we only require node loads to be at most 1). As the algorithm runs, we let b denote the vector of node loads of the sinks; to start, each $b_i \leq 1$ but in time, some of these values may become quite large.

As before, the algorithm repeatedly performs three operations: *node aggregation*, *sawtooth cycle breaking*, and *pivoting*. The first two operations are unchanged, but the pivoting operation is modified as follows.

Pivoting starts with a frontier node with at least two out-neighbours t_i, t_j where in addition t_j is remote. We introduce the use of a *threshold* parameter κ which is the only distinction from the demand maximization algorithm in [10].

PIVOT(D, u, b, f)

If $b_j - f_{ut_j} \leq \kappa$

`Remove` ut_i

$f_{ut_j} \leftarrow f_{ut_j} + f_{ut_i}$

Else

`Remove` ut_j

$f_{ut_i} \leftarrow f_{ut_i} + f_{ut_j}$

`Deactivate sink` t_j

Pivoting is the only operation that increases the load at some sink. Note also that as long as a sink's load is "small", then the pivot will shunt flow onto it. However, when its load goes above $\kappa + 1$, then the flow is shunted elsewhere and t_j is shut down or *deactivated*. The reason that $\kappa + 1$ is an upper bound on the cutoff for deactivation is because $f(u, t_j) \leq 1$, and hence if $b_j > \kappa + 1$, then $b_j - f_{ut_j} > \kappa$. It follows that any sink which occurs as a remote node, will either be deactivated, or still has load at most $\kappa + 1$.

In [10] for sink deactivation they use $\kappa = 1/2$; it is more convenient for us to consider an arbitrary threshold for when to deactivate. In particular, to obtain a good clustering, we need to consider κ larger than 1 with the optimal being $1+\sqrt{3}$.

After running the algorithm, we say that a tree is *big* if it has total demand greater than $\kappa + 2$. The key idea is to show that by removing a small amount of

the demands, we no longer have any big trees. We use the resulting trees with demands scaled down to act as our clusters.

6.2.5 Proof of Theorem 6.2.2

Proof. Call a partial flow from some terminal *bad* if at any time during the execution of the algorithm it was pivoted into a sink whose current congestion was greater than $\kappa + 2$. Note that the sink's congestion could later decrease, but once some flow was labelled bad, it remains so. Call a demand *bad* if it is the source of some bad flow. Let A be the total amount of bad flow. When we perform the sink deactivation step, if a sink t_j is not deactivated, then it had congestion at most $\kappa + 1$ (since node congestion at each non-sink node remains at most 1 throughout the algorithm). After pivoting the flow from t_i into t_j, t_j thus has congestion at most $\kappa + 2$, i.e., the pivoted flow is not bad so A does not increase.

Thus A only increases only when a sink is deactivated and then it increases by at most 1 since f_{ut_j} is a most 1. If ν is the number of deactivated sinks in the algorithm execution, then we clearly have $A \leq \nu$. A deactivated tree has total demand greater than κ, and so $\kappa\nu < \Delta$ and thus $A < \Delta/\kappa$. Consider throwing away all bad demand, and hence we lose only Δ/κ of the total demand.

Routing the remaining demand would result in node congestion at most $\kappa + 2$ in each arborescence. Hence scaling down the demands by a factor $1/(\kappa + 2)$ we route $\frac{\kappa-1}{\kappa(\kappa+2)}\Delta$ demand with node congestion 1.

If demands come in pairs, then eliminating a bad demand may implicitly eliminate demand from its sibling. (We are assuming Δ accounts for the d_i from each sibling.) Hence, the overall loss of pairwise demand may be up to $2\Delta/\kappa$.

Scaling down the remaining demands we have at least

$$\frac{\kappa - 2}{\kappa(\kappa + 2)}\Delta$$

pairwise demands which are in rooted clusters with congestion 1. □

6.2.6 Proof of Theorem 6.2.1

Proof. Think again of throwing away the bad demand and look at the remaining trees. It is sufficient to show that in any such tree, we may route with congestion 1 at least $\frac{1}{2(\kappa+2)}$ fraction of its remaining demand, and we must route any demand in an all-or-nothing fashion. Note that, the act of throwing away a bad demand, may reduce some demands d_i to values $d'_i < d_i$, but we may ignore that in terms of selecting our final demands to route. Consider the full demand to be available for selection. In particular, it is sufficient to show that in each tree we may pick a subset of demands of total weight at most 1, and at least 1/2. If this were not possible, then the total demand in that tree was at most 1 to begin with, and hence we can route everything. This completes the argument. □

6.3 Congestion 3 algorithm for EDP in planar graphs

We present a constant factor approximation algorithm for the maximum edge-disjoint paths (MEDP) problem on planar graphs with edge-congestion 3. The algorithm is based on the one found in [9].

6.3.1 The congestion 3 algorithm

We now modify the clustering phase of their algorithm to achieve a congestion 3 routing algorithm. To do this, we need to ensure the OS instance can be constructed. It is actually built in two steps. The first finds a fractional OS

instance satisfying the cut condition. The second (completely independent) step, takes any such fractional instance and turns it into an integral instance with $\Omega(1)$ of the original demand.

The key ingredients to setting up the fractional OS instance is that if there is positive demand $d_{u_i u_j}$ between nodes $u_i, u_j \in C$, then this was the result of summing values $\Omega(b_v)$ for each terminal pair $v\sigma(v)$ with $v \in H_i$ and $\sigma(v) \in G_j$. Scaling down b_v appropriately results in a weighted demand graph H that one can argue satisfies the cut condition in G_C.

Thus the key is to find a similar "rooted" clustering H_i, P_i as before, except we need not have every terminal clustered. Instead it is sufficient to find some subset of the demands $X \subseteq \{1, 2 \ldots, k\}$ such that s_i, t_i are both clustered if $i \in X$, and in addition $\sum_{i \in X} b_{s_i} = \Omega(p)$. Note that it is not necessary to route the whole b_{s_i}: it is sufficient to route a constant fraction of it. This is precisely what is achieved by the Cluster Capture Theorem 6.2.2. Phase II remains unchanged and thus a congestion of 2 is still incurred there. Hence, the overall routing has congestion 3, proving Theorem 6.0.2.

CHAPTER 7
Conclusion

This thesis contained a quick survey of known results in network flows. In particular, we focused our attention on problems with various degree constraints. After considering unsplittable flows, d-furcated flows and confluent flows, we introduced the new notion of rooted clustering and used it to show an improved bound for the maximum edge-disjoint paths problem. Namely, we showed that there exists a polynomial time algorithm to route a constant fraction of the demand while incurring an edge congestion of at most 3.

There remain many interesting open question in the areas covered by this thesis. For instance, nothing is known regarding the cost version of the d-furcated flow problem. Even though we tried to design an approximation algorithm for the rounds minimization problem for confluent flows, some crucial step could not be made to work. Computer experiments suggest that there is still a chance to succeed, but further work is needed. Finally, we still do not know if it is possible to find an edge-disjoint rooted clustering in undirected graphs. We conjecture that it is not the case, but a proof is still needed.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, New Jersey, 1993.

[2] S. Antonakopoulos, C. Chekuri, B. Shepherd, and L. Zhang. Buy-at-bulk network design with protection. *Foundations of Computer Science, 2007. FOCS '07. 48th Annual IEEE Symposium on*, pages 634–644, 2007.

[3] Alper Atamtürk and Deepak Rajan. On splittable and unsplittable flow capacitated network design arc–set polyhedra. *Mathematical Programming*, 92(2):315–333, 04 2002/04/25/.

[4] Baruch Awerbuch, Yossi Azar, and Amir Epstein. Large the price of routing unsplittable flow. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 57–66, New York, NY, USA, 2005. ACM.

[5] Georg Baier, Ekkehard Köhler, and Martin Skutella. The k-splittable flow problem. *Algorithmica*, 42(3):231–248, 07 2005/07/01/.

[6] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002.

[7] C. Chekuri, S. Khanna, and F. B. Shepherd. Edge-disjoint paths in planar graphs. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 71–80, 2004.

[8] Chandra Chekuri, Sanjeev Khanna, and F. Bruce Shepherd. The all-or-nothing multicommodity flow problem. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 156–165, New York, NY, USA, 2004. ACM.

[9] Chandra Chekuri, Sanjeev Khanna, and F. Bruce Shepherd. Edge-disjoint paths in planar graphs with constant congestion. In *STOC '06: Proceedings*

of the thirty-eighth annual ACM symposium on Theory of computing, pages 757–766, New York, NY, USA, 2006. ACM.

[10] Jiangzhuo Chen, Robert D. Kleinberg, László Lovász, Rajmohan Rajaraman, Ravi Sundaram, and Adrian Vetta. (Almost) tight bounds and existence theorems for single-commodity confluent flows. *J. ACM*, 54(4):16, 2007.

[11] 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, 2006/5.

[12] Jiangzhuo Chen, Ravi Sundaram, Madhav Marathe, and Rajmohan Rajaraman. The confluent capacity of the internet: Congestion vs. dilation. *Distributed Computing Systems, International Conference on*, 0:5, 2006.

[13] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. Wiley-Interscience series in discrete mathematics and optimization. Wiley-Interscience, 1998.

[14] Steve Cosares and Iraj Saniee. An optimization problem related to balancing loads on sonet rings. *Telecommunication Systems*, 3(2):165–181, 06 1994/06/29/.

[15] G. B. Dantzig. Application of the simplex method to a transportation problem. In T. C. Koopmans, editor, *Activity analysis of production and allocation*. Wiley, 1951.

[16] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, N.J., 1963.

[17] Reinhard Diestel. *Graph Theory*. Springer, 2006.

[18] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 12 1959/12/01/.

[19] E.A. Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Doklady*, 1970.

[20] Yefim Dinitz, Naveen Garg, and Michel X. Goemans. On the single-source unsplittable flow problem. *Combinatorica*, 19(1):17–41, 01 1999/01/04/.

[21] P. Donovan, B. Shepherd, A. Vetta, and G. Wilfong. Degree-constrained network flows. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 681–688, New York, NY, USA, 2007. ACM.

[22] Christophe Duhamel and Philippe Mahey. Multicommodity flow problems with a bounded number of paths: A flow deviation approach. *Networks*, 49(1):80–89, 2007.

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

[24] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 251–262, New York, NY, USA, 1999. ACM.

[25] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[26] Jr. Ford, L. R. and D. R. Fulkerson. Solving the transportation problem. *Management Science*, 3(1):24–32, 1956.

[27] Jr. Ford, L. R. and D. R. Fulkerson. A suggested computation for maximal multi-commodity network flows. *Management Science*, 5(1):97–101, 10 1958/10/1.

[28] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[29] L.R. Ford and D.R. Fulkerson. *Flows in networks*. Princeton University Press, Princeton, N.J., 1962.

[30] A. Frank. Packing paths, cuts, and circuits - a survey. In B. Korte, László Lovász, H. J. Prömel, and Alexander Schrijver, editors, *Paths, Flows and VLSI-Layout*, pages 49–100. Springer Verlag, 1990.

[31] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, San Francisco, CA, 1979.

[32] N. Garg and J. Konemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *Foundations of Computer Science, 1998. Proceedings.39th Annual Symposium on*, pages 300–309, 1998.

[33] Naveen Garg, Vijay V. Vazirani, and Mihalis Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.

[34] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Science*, 99:7821–7826, June 2002.

[35] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[36] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36(4):873–886, 1989.

[37] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Algorithms and Combinatorics. Springer-Verlag, 1988.

[38] Refael Hassin, R. Ravi, and F. Sibel Salman. Approximation algorithms for a capacitated network design problem. *Algorithmica*, 38(3):417–431, 03 2004/03/01/.

[39] M. Iri. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan*, 3:27–87, 1960.

[40] W. S. Jewell. *Optimal flow through networks*. Operations Research Center, MIT, Cambridge, MA, 1958.

[41] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[42] L. Khachiyan. A polynomial algorithm in linear programming. *Soviet Math. Doklady*, 20(1):191–194, 1979.

[43] J. M. Kleinberg. Single-source unsplittable flow. *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 68–77, 1996.

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

[45] Ronald Koch, Martin Skutella, and Ines Spenke. Maximum k -splittable s , t -flows. *Theory of Computing Systems*, 43(1):56–66, 07 2008/07/01/.

[46] S. G. Kolliopoulos and C. Stein. Improved approximation algorithms for unsplittable flow problems. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 426–436, 1997.

[47] Petr Kolman and Christian Scheideler. Improved bounds for the unsplittable flow problem. *Journal of Algorithms*, 61(1):20–44, 2006/9.

[48] A. Kotzig. Súvislost' a pravideliná súvislost' konečných grafor. *Vysoká Škola Ekonomická*, 1956.

[49] Tom Leighton, Clifford Stein, Fillia Makedon, Éva Tardos, Serge Plotkin, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 101–111, New York, NY, USA, 1991. ACM.

[50] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.

[51] K. Menger. Zur allgemeinen kurventheorie. *Fund. Math*, 1927.

[52] Haruko Okamura and P. D. Seymour. Multicommodity flows in planar graphs. *Journal of Combinatorial Theory, Series B*, 31(1):75–81, 1981/8.

[53] Tomasz Radzik. Fast deterministic approximation for the multicommodity flow problem. *Mathematical Programming*, 78(1):43–58, 07 1996/07/01/.

[54] Prabhakar Raghavan and Clark Tompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 12 1987/12/23/.

[55] R. Ravi, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt III. Approximation algorithms for degree-constrained minimum-cost network-design problems. *Algorithmica*, 31(1):58–78, 12 2001/12/21/.

[56] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

[57] Alexander Schrijver, Paul Seymour, and Peter Winkler. The ring loading problem. *SIAM Review*, 41(4):777–791, 1999.

[58] Loïc Séguin-Charbonneau and F. Bruce Shepherd. Routing and clustering via confluent flows. Unpublished manuscript, 2009.

[59] F. B. Shepherd and A. Vetta. Visualizing, finding and packing dijoins. *Graph Theory and Combinatorial Optimization*, pages 219–254, 2005.

[60] F. Bruce Shepherd. Single-sink multicommodity flow with side constraints. *Research Trends in Combinatorial Optimization*, pages 429–450, 2009.

[61] Martin Skutella. Approximating the single source unsplittable min-cost flow problem. *Mathematical Programming*, 91(3):493–514, 02 2002/02/25/.

[62] A. Srinivasan. Improved approximations for edge-disjoint paths, unsplittable flow, and related routing problems. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 416–425, 1997.

[63] Éva Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 09 1985/09/01/.

[64] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121, 1971.

[65] P. M. Vaidya. Speeding-up linear programming using fast matrix multiplication. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 332–337, 1989.

[66] Christian von Mering, Roland Krause, Berend Snel, Michael Cornell, Stephen G. Oliver, Stanley Fields, and Peer Bork. Comparative assessment of large-scale data sets of protein-protein interactions. *Nature*, 417(6887):399–403, 05 2002/05/23/print.