Sequential and Distributed Algorithmic Frameworks for the Maximum Concurrent Flow Problem

Christofi, Michalis

Awarding institution:
King's College London

Sequential and Distributed Algorithmic Frameworks for the Maximum Concurrent Flow Problem

Michalis Christofi

Department of Informatics

King's College London

A thesis submitted for the degree of

Doctor of Philosophy

May 2016

Acknowledgements

I want to express my gratitude to my advisor Professor Tomasz Radzik. His support, guidance and patience were very precious all along this time. I also deeply acknowledge his sharing of experience and knowledge and I will try to keep his good practices as examples. I also want to thank my examiners for the many suggestions which improved the presentation of this thesis.

Finally, I would like to thank my partner, parents and family for their support and encouragement all these years. This PhD is a testament to your faith in me, I hope I have made you proud.

Abstract

Networks are everywhere, changing the way we communicate with each other, transport goods and share information. The problems of efficient operation of such networks can often be stated as (abstract) network flow problems. In a problem of this type we want to send some commodity (goods, messages, data, electricity, vehicles) from supply points to demand points in an underlying network, which is modeled as a graph. There are various constraints on the characteristics of the routes, such as capacities and costs. There may be a number of different optimization objectives, depending on the problem setting. Network flow problems form one of the most important and most frequently encountered classes of optimization problems. They lie at the intersection of several scientific fields including computer science, mathematics and operational research. We are interested in the computer science aspect of network optimization problems, that is, in development and analysis of efficient algorithms for such problems.

In this thesis we study algorithmic frameworks for multicommodity flow problems, which can be described in the following way. The input is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is the set of edges, and specifications of k commodities. Each edge has an associated capacity $c(e)$ and each commodity has an associated source-sink pair of nodes (s_i, t_i) and a demand value d_i. The goal is to design simultaneous flow of all commodities that satisfies their demands, takes into account the capacities of the edges and optimizes a specified objective function. We focus on the problem of minimizing the overall congestion, which is often referred to as the Maximum Concurrent Flow problem. We consider both sequential and distributed models of computation. We show that the two main sequential algorithmic Maximum Concurrent Flow frameworks - the rerouting framework and the incremental framework - are more closely related than previously assumed. We prove that the running time of some distributed Maximum Concurrent Flow algorithms shown recently are asymptotically tight. We also propose a heuristic for these algorithms to improve their performance on some types of inputs.

Contents

List of Figures

Part I

Introduction and Background

Chapter 1

Introduction

Contents

1.1 Context of the Research

The multicommodity flow problem is a network flow problem with multiple commodities between different origin-destination pairs. Many real-life applications may be formalized under the form of a multicommodity flow problem. It arises in a variety of applications, from distribution of goods and transportation problems to computer and communication networks. It consists in routing a set of commodities through a given network from some origins to some destinations optimizing a specified objective function. We focus on the problem of minimizing the overall congestion, which is often referred to as the Maximum Concurrent Flow (MCF) Problem. We consider both sequential and distributed models of computation.

1.2 Algorithmic Approaches

Two approaches to solving the MCF problem under the sequential computation model were established in the beginning of 1990s, the rerouting framework and the incremental framework. The rerouting method starts the computation from some initial flow and then keeps redistributing (rerouting) the flow from more congested to less congested paths, while the incremental method builds a solution from scratch by iteratively adding small amounts of flow.

Distributed computing arose from the integration of computer and networking technologies which enabled parallelism. There are two main models of distributed computing for the multicommodity flow problem. In 1990s the "classical" model was investigated, in which computing units are associated with the nodes of the network and make local decisions on how the flows of the commodities should be forwarded. More recently a new model of distributed computing for multicommodity flow problems was proposed by Awerbuch et al. [7] and Awerbuch and Khandekar [4]. In this model each computing unit (agent) is associated with each commodity. A "billboard" maintains the current total flows on the edges of the network. The agents are allowed to read the values on the billboard at the beginning of each round and decide how to reroute the flows of their commodities. Each agent has information about the total edge flows (from the billboard) and the edge flows of its commodity, that is, they cannot distinguish between the flows of other commodities. At the end of each round the agents submit their flows to the billboard again. The agents are not allowed to co-ordinate with each other in any other way than through the billboard, in particular, they cannot read the flows of other agents or have information about their decisions in the current round. The only co-ordination that exists is that the agents synchronize their executions using the notion of a round.

1.3 Contributions of the Thesis

The main contributions of this thesis are summarized below.

Sequential Algorithms

The rerouting and incremental frameworks have traditionally been described and analyzed separately as two different methods, but when executed, they exhibit similar patterns of computation. First, we exhibit the two methods, explain their computation and discuss their main properties. This helps us to highlight the differences and similarities between the two methods. Then we modify algorithms proposed under each framework and show that each method can be viewed as a variation of the other.

For the incremental method we propose a new exponential length function, which is similar to the length function used in the rerouting method. We show that this length function fits in the original analysis of the incremental method. The new length function proposed, unlike the one in the original method, has a functional relation with the flow. This modification converts the incremental method into an instance of the rerouting method.

In the rerouting method, the most successful technique for updating the flows in each iteration is by using minimum cost flow computations. We show that we can use a simple approach to find suitable approximate minimum cost flow under the rerouting framework using the Successive Shortest Path algorithm (SSP). We explain how to use this algorithm and prove a running time bound. The SSP algorithm uses shortest paths to create the updated flow, in a similar way as in the incremental framework. Essentially, this modification converts the rerouting method into an instance of the incremental method. Moreover, it has been shown that the SSP algorithm is faster in practice than the other algorithms proposed for the calculation of minimum cost flows [85, 13]. Therefore our modification could make the rerouting method better in practice.

Distributed Algorithms

We study the two main distributed algorithms for the multicommodity flow problem proposed by Awerbuch et al. [7] and Awerbuch and Khandekar [4]. These two algorithms need $\widetilde{O}(L)$ rounds to terminate, where L denotes the maximum path size in the network and the $\widetilde{O}$ notation hides polylogarithmic factors. The upper bound analysis in both [7] and [4] was based on the analysis of the flow change on one edge of the shortest path. In each iteration, there is one edge with an increase of flow by at least some small amount. These increases are then totalled to get the flow change on the whole path and this result is used to derive the upper bound on the number of rounds.

The upper bound analysis in both [7] and [4] was only counting the increases of flow on the saturated edges. However, when the flow is updated on paths, then in addition to (at least one) saturated edge, the flow would increase on all edges of the path. So, it was reasonable to expect that the derived upper bounds were not tight. We construct a worst-case network and show that the running time bounds of these algorithms are actually tight.

We then propose a heuristic improvement of these algorithms and analyze its performance on the worst-case inputs. In the original algorithms the flow updates depend on the computation of a blocking flow under some flow control constraints. We impose an extra requirement on the computed blocking flow to distribute flow evenly among available approximate shortest paths. We show how to achieve such a distribution without increasing the asymptotic running time. We also show that our heuristic significantly speeds up the process at least on our worst-case input. This result indicates potential for better performance of our heuristic than the previous algorithms.

This thesis also serves as a large review of the different Maximum Concurrent Flow formulations and algorithms found in the literature. Moreover, we establish new connections between various Maximum Concurrent Flow algorithms and explicate the effect of these connections on the running time of the different algorithms proposed to solve the problem.

1.4 Outline of the Thesis

This thesis is divided into tree main parts. Part I is devoted to introductory material, definitions and applications of the Multicommodity Flow Problem. In Chapter 2 we introduce the basic background for Network Flow Problems. In Chapter 3 we introduce the Maximum Concurrent Flow Problem and give its formulations and its applications. In Chapter 4 we describe the main solution methods used to solve the MCF problem.

In Part II we examine the sequential algorithms for the MCF problem. More specifically, in Chapter 5 we describe the main sequential algorithms for solving the problem, the incremental and the rerouting method. In Chapter 6 we introduce a new length function for the incremental method which is exponential in the value the flow. This length function is similar to the one used in the rerouting framework. We show how this length function fits in the original analysis of the incremental method, prove its correctness and show the running time bounds. In Chapter 7 we introduce a different way to calculate minimum cost flows under the rerouting framework. We show how we can do this using the Successive Shortest Path algorithm (SSP) in a modified network. We prove that our modification fits in the analysis of Radzik's [65] algorithm and show the running time. Our results are slightly inferior to the current best running times but using the SSP algorithm we show that that the rerouting algorithm is closer to the incremental one than previously considered. Implementing the SSP algorithm for computing minimum cost flow paths is also much simpler than the previous methods used.

Part III is devoted to distributed algorithms for the MCF problem. In Chapter 8 we describe the distributed model of computation and give some of its applications. We also review the previous results for solving the MCF problem under the distributed computing framework. In Chapter 9 we examine the Approximate Steepest Descent Framework (DGD-MCF) proposed in [7]. We construct a worst-case input and show the running times of the DGD-MCF algorithm on this network. We also propose a heuristic improvement which speeds up significantly the running time of the DGD-MCF algorithm. Finally, in Chapter 10 we analyze the Greedy Distributed Framework (GDR-MCF) proposed in [4]. We show that its running time bottleneck is an artifact of the flow control constraints imposed. We also show that even by relaxing the flow control constraints the running time does not improve significantly when analyzed on our worst-case network. We propose a heuristic improvement to improve the running time of the GDR-MCF algorithm on our worst-case network.

Chapter 2

Network Flows: Background and Terminology

Contents

Network flow problems are one of the most important and most frequently encountered types of optimization problems [11]. They lie at the intersection of several scientific fields including computer science, mathematics and operational research. Applications of network flow problems can be found everywhere in our daily lives; in transportation, in telecommunication, in manufacturing and distribution of goods. In all of these application domains we want to send some commodity (products, messages, electricity, vehicles, data) from supply points to demand points in an underlying network, which is modeled as a graph. The network consists of several routes (paths) which connect these points and are used to transfer the requested commodity. Usually there are constraints on the characteristics of the routes, such as capacities or costs. The objective, in the most general sense, is to route these commodities in the network in the most efficient and typically least expensive way.

In this chapter we introduce the background of the research area examined in this thesis. In Sections 2.1, 2.2, 2.3 we introduce the notation and terminology associated with the problems addressed in this thesis. In Section 2.4, we provide an overview of the main approaches used to solve them efficiently. In Section 2.5 we provide the measures used to analyze the algorithms we develop in this thesis. Finally, in Section 2.6 we give a brief description of the available computing methods and discuss how these lead us in designing appropriate algorithms for solving the multicommodity flow problem.

The following textbooks were used as a basis for the concepts we introduce in this chapter and are suggested for further details on Network Flow Problems: *Network Flows: Theory, Algorithms, and Applications* by Ahuja et al. [1], *Network Optimization: Continuous and Discrete Models* by Bertsekas [11], *Linear Programming and Network Flows* by Bazaraa et al. [9] and *The Design of Approximation Algorithms* by Williamson and Shmoys [81].

2.1 Graphs

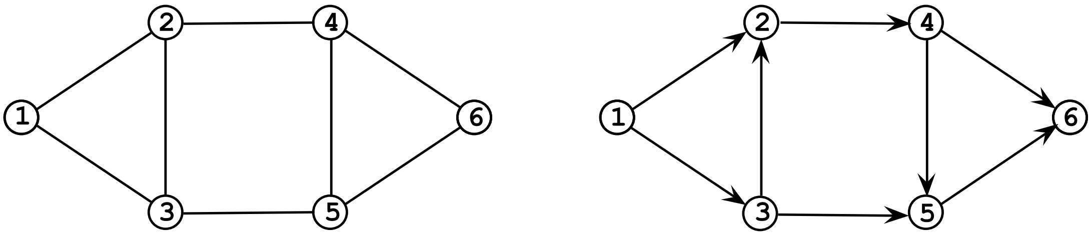
This section presents some basic graph-theoretic concepts. Examples are given to clarify the expositions.

Definition 1. *An undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, consists of a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$ whose elements are unordered pairs of distinct nodes.*

Definition 2. *A directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, consists of a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$ whose elements are ordered pairs of distinct nodes.*

Figure 2.1(a) shows an example of an undirected network. Figure 2.1(b) shows an example of a directed network.

The number of nodes is denoted by n and the number of edges is denoted by m. We note that some authors use a different terminology by referring to the



(a) Undirected Graph

(b) Directed Graph

Figure 2.1: Directed and Undirected Graphs

nodes as vertices and to edges as arcs. An edge is *directed* if it is assumed ordered and it is denoted by (u, v), $u, v \in \mathcal{N}$. If unordered, the edge is referred to as *undirected* and is denoted by $\{u, v\}$, $u, v \in \mathcal{N}$. Note that we use both e and (u, v) or $\{u, v\}$ to denote an edge depending on the context to make our notation simpler.

Definition 3. *The capacity of an edge $c(u, v)$ is a function $c : \mathcal{E} \rightarrow \mathbb{R}_+$, where $\mathbb{R}_+$ is the set of non-negative real numbers. We extend this definition to all node pairs by letting $c(u, v) = 0$ if $(u, v) \notin \mathcal{E}$.*

A *network* is a graph (directed or undirected) with associated values on its nodes or edges, e.g. capacities or demands. The edges and nodes in this thesis always have edge or node values associated with them. For convenience, we will often use the term graph to refer to networks. We will mostly deal with directed graphs since they are convenient for our problem domain. Therefore we often omit the term "*directed*" and thus whenever referring to a graph we implicitly assume that the graph is directed unless stated otherwise.

We do not exclude the possibility of the presence of edges connecting two nodes in both directions in a directed graph. However for simplicity of notation we do not allow two or more edges in the same direction between the same pair of nodes (so called multiple edges). In the network flow problems that we consider, we can replace parallel edges with one "super-edge" which has the same direction and whose capacity is equal to the sum of the capacities of the edges that have been replaced.

Definition 4. *A graph $\mathcal{G}' = (\mathcal{N}', \mathcal{E}')$, is called a subgraph of $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, if $\mathcal{N}' \subseteq \mathcal{N}$ and $\mathcal{E}' \subseteq \mathcal{E}$.*

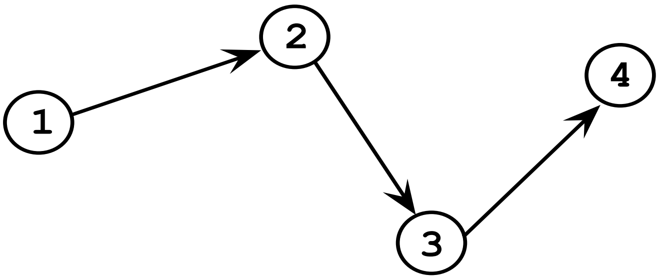


Figure 2.2: Path (1,2,3,4)

Definition 5. *A cut in a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, often denoted as $[S, \overline{S}]$, is a partition of the node set $\mathcal{N}$ into two disjoint sets, $S \subseteq \mathcal{N}$ and $\overline{S} = \mathcal{N} - S$. A cut-set is the set of edges which have their one end in the set S and their other end in the set $\overline{S}$.*

In Figure 2.1 the sets $S = \{1, 2, 3\}$ and $\overline{S} = \{4, 5, 6\}$ define a cut with the edges $\{2, 4\}, \{3, 5\} \in [S, \overline{S}]$ in the cut.

Definition 6. *A path P is a graph whose nodes can be listed in the order $(\nu_1, \nu_2, \dots, \nu_k)$, such that the edges are $(\nu_i, \nu_{i+1}) \in \mathcal{E}$, where $i = 1, 2, \dots, n-1$.*

Figure 2.2 shows an example of a path.

2.2 Networks and Flows: The single-commodity case

In the applications of network flows we send a commodity (e.g. electric current, messages, cars) through a network. We introduce variables which measure the amount of commodity sent over different edges. The variable associated with edge (u, v) is defined as the *flow of this edge* and it is denoted by $f(u, v)$. The flow of an edge can only be positive or zero. We extend this definition to all node pairs by letting $f(u, v) = 0$ if $(u, v) \notin \mathcal{E}$.

Definition 7. *A flow in a network with a set of nodes $\mathcal{N}$, a set of edges $\mathcal{E}$, a source s and a sink t, is a real-valued function $f : \mathcal{N} \times \mathcal{N} \to \mathbb{R}$ with the following properties:*

Capacity Constraints: $0 \leq f(u,v) \leq c(u,v), \;\; \forall \;\; (u,v) \in \mathcal{N} \times \mathcal{N},$

Flow Conservation: $\quad \sum_{v \in \mathcal{N}} f(u,v) - \sum_{v \in \mathcal{N}} f(v,u) = 0, \; \forall \; u \in \mathcal{N} \setminus \{s,t\}.$

Note that if $(u,v) \notin \mathcal{E}$ then $f(u,v) = 0$. The capacity constraints state that the total flow on an edge cannot be greater than the capacity of the edge. The flow conservation property states that the total flow arriving at a node is equal to the total flow departing from the node. This is true for all nodes except the source and the sink. At the source we put flow into the network so the net flow $\sum_{v \in \mathcal{N}} f(s,v)$ will be positive and at the sink we take out flow from the network and thus the net flow is negative.

Definition 8. *The output of the maximum flow problem is a flow in $\mathcal{G}$ which maximizes $\sum_{v \in \mathcal{N}} f(s,v)$.*

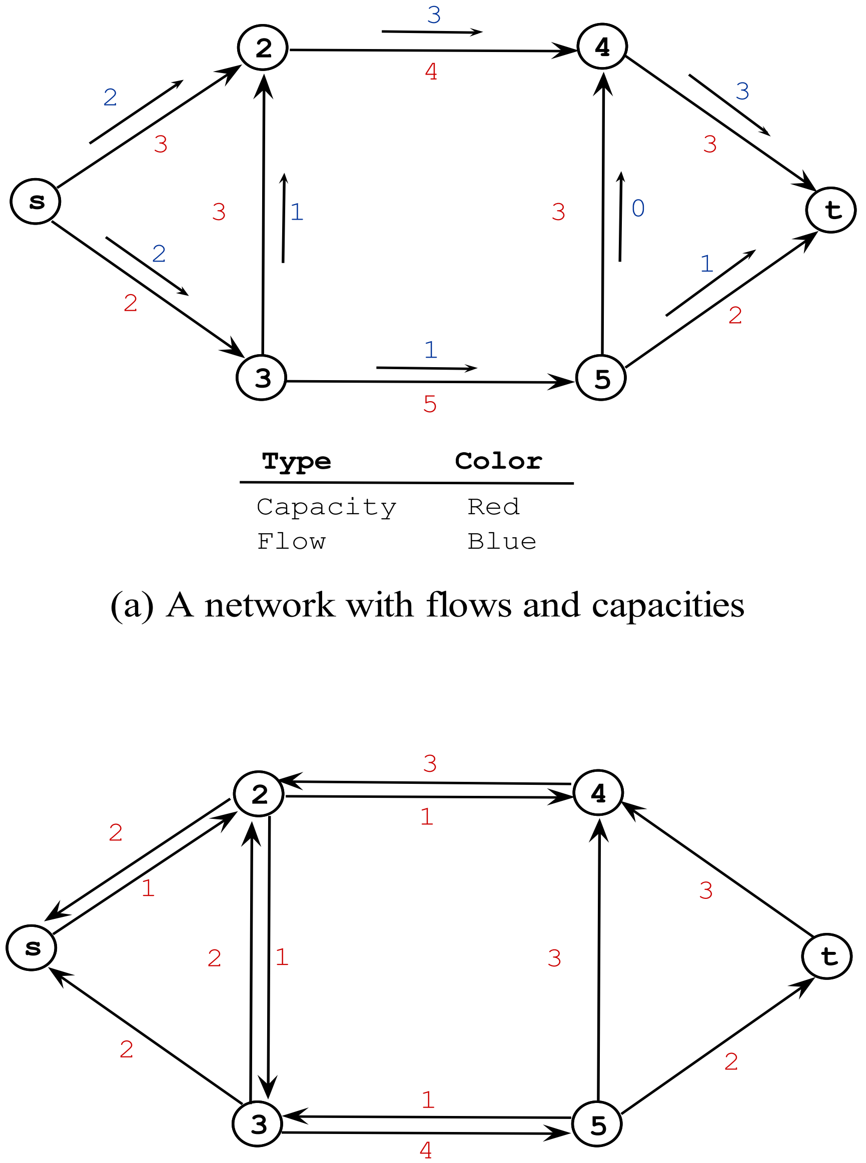
2.3 Residual Networks and Blocking Flows

Crucial notions for network flow algorithms are the residual capacity and the residual networks. These notions are defined below, with an example in Figure 2.3.

Definition 9. *A residual capacity $c_R(u,v) = c(u,v) - f(u,v)$ is the amount of available capacity of an edge (u,v). From these capacities we can construct the residual network $\mathcal{G}_{\mathcal{R}} = (\mathcal{N}, \mathcal{E}_{\mathcal{R}})$, where $\mathcal{E}_{\mathcal{R}} = \{(u,v) \in \mathcal{E} : c_R(u,v) > 0\}$, which gives us the available capacity of the graph. The capacities of the reverse edges in the residual network are equal to $f(u,v)$.*

A *saturated edge* is an edge with residual capacity equal to zero, that is no more flow can be sent through this edge without violating the capacity constraints. The *capacity of a path* is the minimum over all capacities of the edges in the path. An *augmenting path* is a source-to-sink path in the residual network.

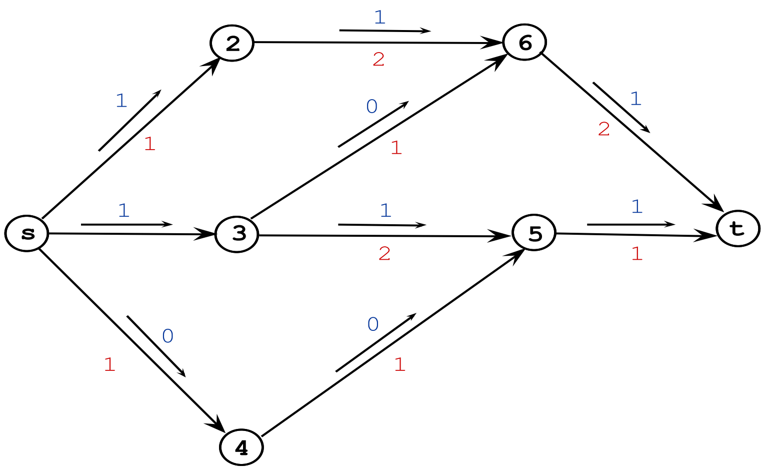
Definition 10. *A blocking flow is a flow function in which every source-to-sink path in the directed graph $\mathcal{G}$ has a saturated edge.*



(a) A network with flows and capacities

(b) Residual Network

Figure 2.3: Graph and Residual Graph



Type	Color
Capacity	Red
Flow	Blue

Figure 2.4: A Blocking Flow

We should note that a blocking flow does not guarantee a *maximum flow*. A maximum flow is achieved if we cannot find any augmenting paths in the residual network to send flow. We can observe this in Figures 2.4 and 2.6. In Figure 2.4 we have a blocking flow, that is we cannot find any path in the directed network to augment flow. However, in the residual network we can still send one unit of flow along path $s - 4 - 5 - 3 - 6 - t$. We can easily verify in Figure 2.6 that we cannot send any more flow since all the outward edges from the source are saturated, and therefore this is a maximum flow.

2.4 Exact and Approximation Algorithms

To solve a computational problem we need to combine three building blocks essentially: an algorithm, a computational device and a data structure on which we apply our algorithm. The computational device nowadays is a computer or a network of computers linked together to increase their power. A *data structure* is a way to store and organize data in order to facilitate access and modifications. In our notion of network flows a data structure is a way to represent the nodes,

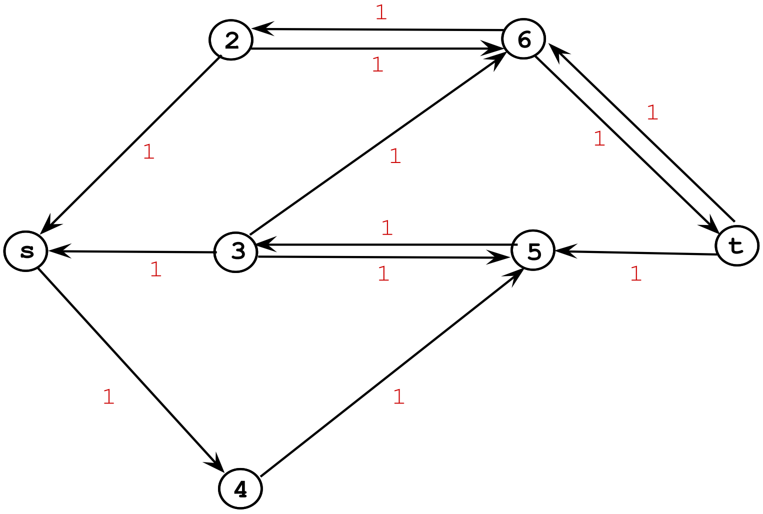
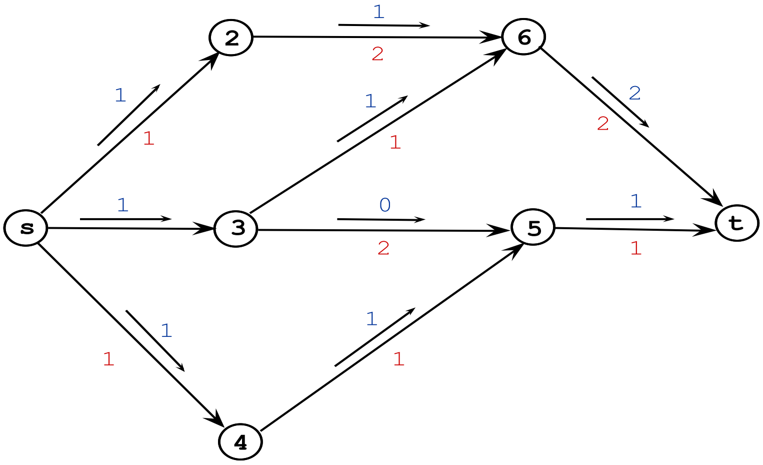


Figure 2.5: Residual Network



Type	Color
Capacity	Red
Flow	Blue

Figure 2.6: A Maximum Flow

edges, demands and flows. Although this area is very important in the general concept of algorithms it is beyond the scope of this thesis so we will not cover it in detail. The algorithms which we consider in this thesis use the *adjacency lists* representation of graphs and networks. The algorithms may use some auxiliary data structures, for example priority queues. We will not consider details of these data structures in this thesis.

Network flow problems fall into the broader category of Combinatorial Optimization Problems. A vast number of algorithms exist to solve problems in *Combinatorial Optimization.* These algorithms can, generally speaking, be categorized in two groups, Exact Algorithms and Approximation Algorithms. Exact Algorithms are the algorithms which when executed on a problem instance output an exact solution of a problem. Unfortunately many of the problems in Combinatorial Optimization appear to be difficult to be solved exactly in polynomial time. This means that getting an exact solution would be very expensive (in terms of time or computational power needed) and in many cases impossible. The network flow problems that we consider have polynomial-time algorithms, but only with a polynomial of high degree.

In computer science we are mostly interested in algorithms which return an optimal or near-optimal solution in polynomial time with respect to the input size. To achieve fast polynomial-time algorithms we need to relax some of our requirements. In general an algorithm is considered to be "good" if it returns an optimal solution, in polynomial time, for any instance. So if we are not willing to relax our polynomial-time requirement we have two options:

- Relax the any instance requirement. We can devise algorithms for various special instances that run in polynomial time,
- Relax the optimal solution requirement, that is find a solution which is not optimal, but is close enough.

The latter approach is the most common and it is examined deeply in this thesis. We examine approximation algorithms, that is algorithms which find a solution that is close to the optimal. We formalize this in the following definition.

Definition 11 ([63]). *Let $\mathcal{A}$ be an optimization (maximization or minimization) problem with positive cost function c, and let α be an algorithm which, given an instance I of $\mathcal{A}$ returns a feasible solution $f_\alpha(I)$. Let $\widetilde{f}(I)$ be an optimal solution of $\mathcal{A}$. Then α is called an ϵ-approximate algorithm for $\mathcal{A}$ for some $\epsilon > 0$ if and only if*

$$\frac{|c(f_\alpha(I)) - c(\widetilde{f}(I))|}{c(\widetilde{f}(I))} \leq \epsilon$$

for all instances I.

The solution $f_\alpha(I)$ of the ϵ-approximate algorithm α for the optimization problem $\mathcal{A}$ is called ϵ-approximate solution.

2.5 Efficiency and Complexity

To estimate the running time of an algorithm we need a performance guarantee. Typically this guarantee is expressed in terms of the input size parameters. There are three basic approaches to measure the complexity (running time) of an algorithm:

Empirical Analysis: Estimate how algorithms perform in practice. This is done by implementing the algorithm and running this implementation on a set of different problem instances.

Average-case Scenario: Estimate the expected number of steps an algorithm needs to terminate. This is executed by assigning a probability distribution for the problem instances and by deriving asymptotic expected running times using statistical analysis.

Worst-case Scenario: This analysis provides upper bounds for the number of steps an algorithm needs to terminate on any problem instance of given size. By providing upper bounds we ensure that the algorithm will terminate within the running time estimated.

In this thesis we use the worst-case to analyze the performance of our algorithms.

Let n, m and U be the input parameters indicating the size of an input network for a network optimization problem P. The n and m parameters are the number of nodes and edges, as used before. It is assumed that all input numbers (for example edge capacities) are given as rational numbers and the parameter U is the largest integer (in absolute value) among the denominators and numerators of these numbers. To estimate the complexity, as we stated above, we need to derive a bound on the running time in terms of the input parameters. Such a bound is given usually by some expression of the form

$$c \cdot g(n, m, U),$$

where c is a constant and g is a known function.

We say that the running time is $O(g(n, m, U))$ if the running time of an algorithm is bounded above asymptotically (up to a constant factor) by $g(n, m, U)$. We say that the running time is $\Omega(g(n, m, U))$, if it is bounded below asymptotically by $g(n, m, U)$. Finally, we say that it is $\Theta(g(n, m, U))$ if it is bounded both above and below asymptotically by $g(n, m, U)$.

If we can express $g(n, m, U)$ using a polynomial function in n, m and $\log U$, for example, $O(n^p \cdot m^q \cdot (\log U)^t)$ for some constants $p, q, t \in \mathbb{R}_+$, we say that the algorithm has *polynomial* running time. We say that an algorithm is *strongly polynomial*, if we can express the running time in terms of the parameters n and m only, for example, $O(n^p \cdot m^q)$. A *pseudopolynomial* running time depends polynomially on the input parameters n and m but exponentially on the length of the input, for example, $O(n^p \cdot m^q \cdot U)$.

2.6 Computational Models

The history of algorithms is tightly coupled with the history of computing architecture. As computers and computer networks evolved, algorithms were designed to make appropriate use of this environment. Algorithms were also designed for computational models which were yet to be realized in practise. In this section we briefly introduce the two basic computational models which are explored in this thesis. Parallel computing is also briefly discussed for completeness but it is

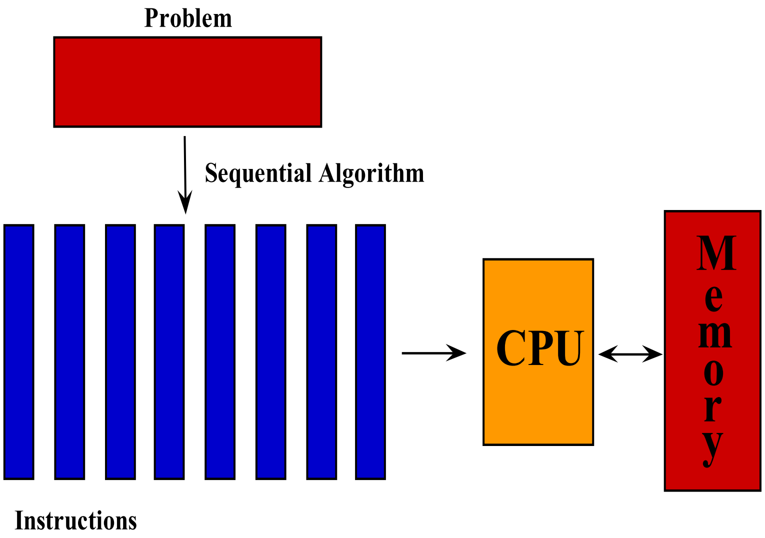


Figure 2.7: Sequential Computing

not within the scope of this thesis. More details are given when needed in the following chapters.

2.6.1 Sequential Computing

Early computers had only a single processor and could only accept programs that would be implemented in a sequential manner. They provided the platform for *sequential algorithms* which formed the basic theoretical framework for algorithmic problem solving. In sequential computing an algorithm has the following properties.

- An algorithm runs on a single computer having a single central processing unit (CPU).
- Instructions are executed one after another; only one instruction can be executed at a time.
- The CPU can read and write to any memory location with equal speed (RAM-Random Access Machine).

Figure 2.7 shows how sequential computing is executed. We consider sequential algorithms for solving the MCF Problem in Chapters 5, 6 and 7.

2.6.2 Parallel Computing

In the early 1980s the industry began developing computers with parallel processors [10]. In this setting processors worked together in a synchronized environment, normally under the control of a central processor. This motivated the design of *parallel algorithms* to harness this power. The properties of parallel algorithms [39] are:

- They are intended to be executed using multiple processors on a single computer.
- A parallel algorithm consists of separate parts that can be executed in parallel; with each part broken down into a set of instructions executed as a sequential model.
- Instructions from each part, where no dependencies between parts exist, are executed simultaneously by different processors.
- An overall central coordination is achieved by an appropriate synchronization mechanism.
- All processors have equal access to a single shared memory (PRAM-Parallel RAM).

Figure 2.8 shows how parallel computing is executed.

2.6.3 Distributed Computing

More recently and through the emergence of computer networks a new type of computing has been introduced, the distributed computing. Clusters of computers in different locations, with various speeds and capacities, are working together in an asynchronous environment, communicating with each other over the network using message passing or some middleware. *Distributed algorithms*, which utilize such a computing environment, have similar properties with parallel algorithms. Their main difference is that in distributed computing each processor has its own local memory, and information is passed along using messages, normally in asynchronous manner. Furthermore in contrast to parallel algorithms,

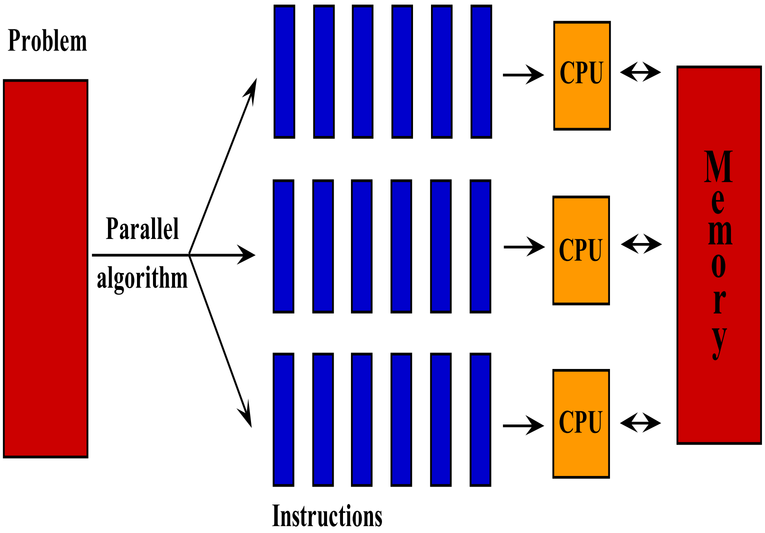


Figure 2.8: Parallel Computing

distributed algorithms do not have central control. Figure 2.9 illustrates how the two methods differ.

Distributed algorithms are widely used in cluster [84, 21] and grid computing [14, 22]. The Internet enables access to the power of thousands of computers in different locations to solve computationally intensive problems. There are a number of versions of distributed computing models. We consider distributed algorithms for the maximum concurrent flow problem in Chapters 8, 9 and 10, and give there further details for the model used by those algorithms.

2.7 Summary

In this chapter we have introduced the main definitions and terminology used in this thesis. We have described the main concepts in graph theory and network flow problems which are essential as a background for our work. We have also discussed the different types of algorithms used to solve combinatorial optimization problems and the different methods used to measure their running time. In the next chapter we are going to introduce the problems addressed in this thesis and give some of their main applications.

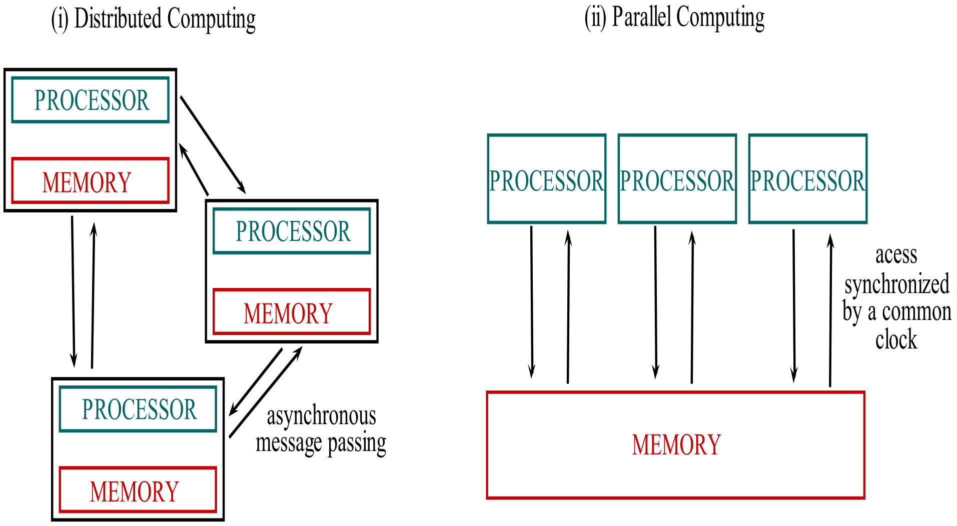


Figure 2.9: Distributed versus Parallel Computing

Chapter 3

Multicommodity Flows and the Maximum Concurrent Flow Problem

Contents

The multicommodity flow problem is a network flow problem which involves the collection of several independent network flow problems specified on the same network, each one with its own requirements and constraints, but also sharing the constraints of the common underlying network. Consider the case of a telecommunication network as an example. Multiple users download movies from different suppliers using the same network. Each commodity can be associated with the video traffic produced by each user with the demand being the size of the movie. The users have their own bandwidth constraints (imposed by the Internet providers) but since they also share the same network they are limited by the capacity of the links of the underlying network.

In Section 3.1 we give the formal definition of the general multicommodity flow problem and its special case called the maximum concurrent flow problem which is the main focus of the thesis. In Section 3.2 we provide a simple example to illustrate these definitions. In Section 3.3 we provide two distinct formulations of the maximum concurrent flow problem which form the basis for two approaches to deriving algorithms for the problem. Finally, in Section 3.4 we describe some of the applications of the multicommodity flow problem.

3.1 Definition

The multicommodity flow problem is a network flow problem with multiple commodities between different source-sink pairs. We are given a problem instance where we have a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with $n = |\mathcal{N}|$ nodes and $m = |\mathcal{E}|$ edges each with an associated capacity $c(u, v)$, and a specification of k commodities with source $s_i \in \mathcal{N}$, destination $t_i \in \mathcal{N}$ and demand d_i. The flow $f_i(e)$ of commodity i on edge e is a function $f_i : \mathcal{E} \to \mathbb{R}_+$.

The multicommodity flow problems is a class of network flow problems which have the following characteristics:

$$\sum_{v \in \mathcal{N}} f_i(u, v) - \sum_{v \in \mathcal{N}} f_i(v, u) = 0, \ \forall\, u \in \mathcal{N} \setminus \{s_i, t_i\}, \ \forall\, i = 1, 2, \ldots, k \tag{3.1}$$

$$f_i(u, v) \geq 0, \ \forall\, i = 1, 2, \ldots, k, \tag{3.2}$$

$$\sum_{i=1}^{k} f_i(u, v) \leq c(u, v), \ \forall\, (u, v) \ \in \ \mathcal{E}. \tag{3.3}$$

For the **maximum multicommodity flow problem** there are no specified demands. The objective is to send as much flow as possible from a set of sources to a set of destinations. The objective for this problem is the following:

$$\max \sum_{i=1}^{k} \sum_{v \in \mathcal{N}} f_i(s_i, v). \tag{3.4}$$

For the **minimum-cost multicommodity flow problem** a cost function exists which assigns a cost $w(u, v)$ to each edge. The specification of the network with edge costs and capacities is often abbreviated as $N = (\mathcal{G}, w, c)$, where $w : \mathcal{E} \to \mathrm{I\!R}$ is a vector of edge costs, and $c : \mathcal{E} \to \mathbb{R}_+$ is a vector of non-negative edge capacities. Considering these the objective of the **minimum-cost** multi-commodity flow problem is:

$$\min \sum_{(u,v)\in\mathcal{E}} w(u, v) \cdot \sum_{i=1}^{k} f_i(u, v) \tag{3.5}$$

subject to

$$\sum_{z\in\mathcal{N}} f_i(s_i, z) - \sum_{x\in\mathcal{N}} f_i(x, s_i) = d_i, \ \forall\, i = 1, 2, \ldots, k, \tag{3.6}$$

$$\sum_{z\in\mathcal{N}} f_i(t_i, z) - \sum_{x\in\mathcal{N}} f_i(x, t_i) = -\, d_i, \ \forall\, i = 1, 2, \ldots, k. \tag{3.7}$$

Equation (3.6) simply states that the flow of commodity i pushed in the network from its source s_i is equal to its demand d_i. Equation (3.7) states that the flow pulled out of the network from the sink of commodity i must be equal to its demand d_i.

In this thesis we examine in detail the **maximum concurrent flow problem** (MCF problem). The objective of this problem is to maximize the minimum fraction of the satisfied demand of each commodity, that is to satisfy the maximum possible proportion of all demands γ. We are given a network $N = (\mathcal{G}, c)$. The formulation of the problem is given below.

(P0)

$$\mathit{maximize}\ \gamma$$

subject to

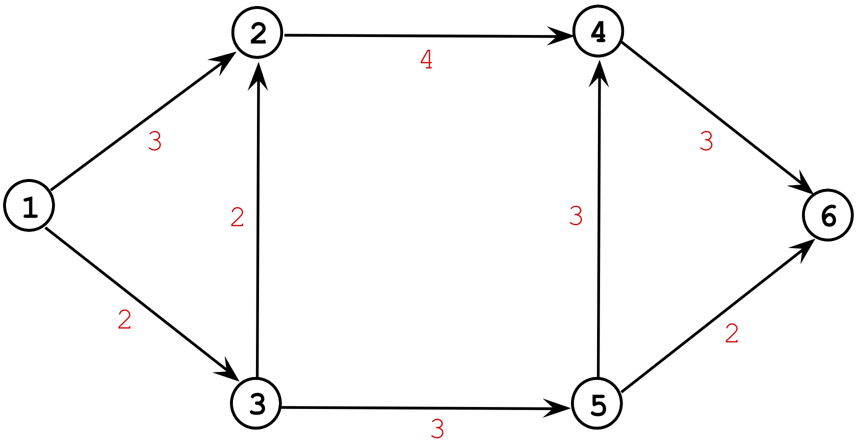
$$\sum_{x:\ (x,v)\in\mathcal{E}} f_i(x,v) - \sum_{z:\ (v,z)\in\mathcal{E}} f_i(v,z) \geq \begin{cases} 0 & \forall\ v \in \mathcal{N} \setminus \{s_i, t_i\} \\ -\gamma d_i & \text{if } v = s_i \\ \gamma d_i & \text{if } v = t_i \end{cases} ,\forall\ i = 1, 2, \ldots, k. \tag{3.8}$$

$$\sum_{i=1}^{k} f_i(u,v) \leq c(u,v), \forall\ (u,v) \in \mathcal{E}, \tag{3.9}$$

$$f_i(u,v) \geq 0, \forall i : i = 1, 2, \ldots, k.$$

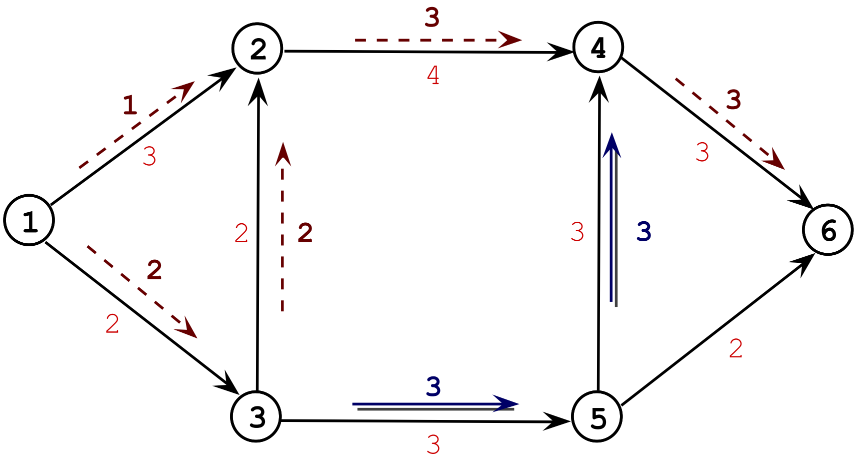
3.2 A Simple Example

In this section we provide a simple example to illustrate the concepts we have introduced in the previous section. Figure 3.1 shows an input network with two commodities. The network consists of $n = 6$ nodes and $m = 8$ edges with corresponding capacities. Each commodity has a source-sink pair, this is $1 - 6$ for commodity 1 and $3 - 4$ for commodity 2. The demands are 3 and 4 units of flow for commodity 1 and 2 respectively. Figure 3.2 shows a non-optimal solution of this problem. Under this solution we manage to send 3 units of flow of both commodities. This corresponds to a 100% of the demand of commodity 1 and 75% of the demand of commodity 2. In Figure 3.3 we show that a better solution exists. More specifically we show an optimal solution where we manage to send 100% of the demand of both commodities.



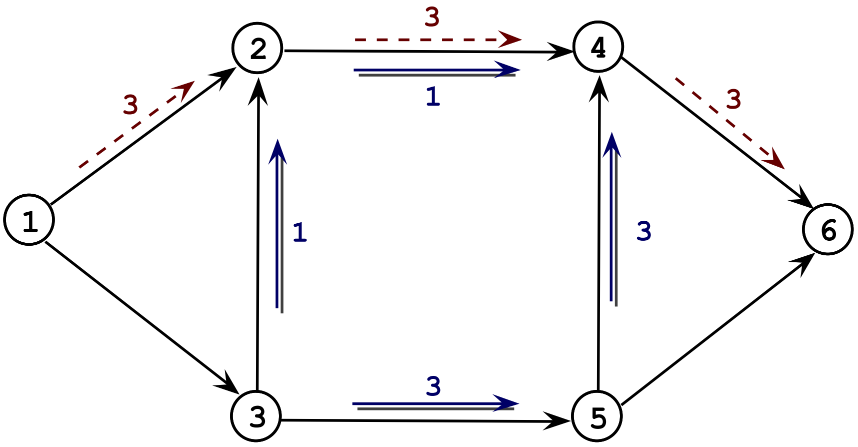
commodity	s	t	demand
1	1	6	3
2	3	4	4

Figure 3.1: An Example Input for the MCF problem



commodity	s	t	demand	symbol
1	1	6	3	⇢
2	3	4	4	⇒

Figure 3.2: Non-optimal Flow;$\gamma = 0.75$



commodity	s	t	demand	symbol
1	1	6	3	- - -➤
2	3	4	4	⇒

Figure 3.3: Optimal Flow;$\gamma = 1$

3.3 Two Formulations of the MCF Problem

We can formulate the maximum concurrent flow problem in two ways, the edge-based formulation (introduced as the **(P0)** problem in Section 3.1) and the path-flow formulation. Based on these two formulations different algorithms for solving the problem can be constructed. In this section we first give an edge-based formulation of the MCF problem, which is alternative(but equivalent) to **(P0)**, and the formulation of the dual problem. Then we present the path-based formulation. We will refer to these formulations later when we provide the solution methods for the maximum concurrent flow problem.

3.3.1 Edge based formulation

Recall from Section 3.1 that we have a network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with capacities $c(u, v)$ for all edges $(u, v) \in \mathcal{E}$, and the source s_i, destination t_i and demand d_i, for each commodity i, $i = 1, 2, \ldots, k$. Formulation **(P1)** is equivalent to formulation **(P0)**. The variables in **(P1)** are $f_i(u, v)$ for $(u, v) \in \mathcal{E}$ and $i = 1, 2, \ldots, k$ and

λ. The variable $f_i(u,v)$ stands for the flow of commodity i on edge (u,v), as in **(P0)**. The variable λ is the upper bound on the "relative" congestions on edges. To see the equivalence between **(P0)** and **(P1)**, observe that given values of the variables γ and $f_i(u,v)$ are feasible for **(P0)** if and only if the values $\lambda = 1/\gamma$ and $f_i(u,v)/\gamma$ are feasible for **(P1)**.

The formulation of **(P1)** can be obtained by substituting γ with $1/\lambda$ and $f_i(x,v)$ with $f_i(x,v)/\gamma$ in **(P0)**, resulting is the following congestion minimization problem

(P1)

$$\mathit{minimize}\ \lambda$$

subject to

$$\sum_{x:\,(x,v)\in\mathcal{E}} f_i(x,v) - \sum_{z:\,(v,z)\in\mathcal{E}} f_i(v,z) = \begin{cases} 0 & \forall\, v \in \mathcal{N} \setminus \{s_i, t_i\} \\ -d_i & \text{if } v = s_i \\ d_i & \text{if } v = t_i \end{cases} \quad , \forall\, i = 1, 2, \ldots, k. \tag{3.10}$$

$$\sum_{i=1}^{k} f_i(u,v) \leq \lambda c(u,v), \forall\, (u,v) \in \mathcal{E} \tag{3.11}$$

$$f_i(u,v) \geq 0, \forall i : i = 1, 2, \ldots, k$$

For given flows f_i of the commodities $i = 1, 2, \ldots, k$ (which satisfy (3.10)) let $f(e) = \sum_{i=1}^{k} f_i(e)$ be the total flow on edge e. Then, $\lambda_f(e) \equiv f(e)/c(e)$ is defined as the congestion of an edge and $\lambda_f = \max\{\lambda_f(e) : e \in \mathcal{E}\}$ is called the congestion of the flow.

The dual of **(P1)** has a variable $l(u,v) \geq 0$ associated with each of the capacity constraints (3.11) and an unconstrained variable $z_{i,v}$ for each of the demand constraints (3.10). We will talk more about the dual of a linear program and its importance in combinatorial optimization in Section 4.2. For each of the variables in the Primal **(P1)** there exist an associated constraint in the Dual. So, the dual of **(P1)** is the following

(D1)

$$maximize \sum_{i=1}^{k} d_i(z_{i,t_i} - z_{i,s_i})$$

subject to

$$\sum_{(u,v)\in\mathcal{E}} c(u,v)l(u,v) \le 1, \tag{3.12}$$

$$z_{i,v} - z_{i,u} \le l(u,v), \forall(u,v) \in \mathcal{E},\ \forall\, i = 1,2,\dots,k, \tag{3.13}$$

$$l(u,v) \ge 0.$$

Observe that if $z_{i,v}, l(u,v)$ are a feasible solution for **(D1)** then so is $z_{i,v}+\alpha_i, l(u,v)$ for any numbers $\alpha_1, \alpha_2, \dots, \alpha_k$, and the value of the objective function doesn't change. Therefore we can consider only the feasible solutions for **(D1)** such that $z_{i,s_i} \equiv 0$. For any feasible values $l(u,v), (u,v) \in \mathcal{E}$, the objective function is maximized for $z_{i,v} = dist_l(s_i, v)$ and it is equal to $\sum_{i=1}^{k} d_i \cdot dist_l(s_i, t_i)$, where $dist_l(s_i, v)$ is the shortest path distance from s_i to v with respect to the edge length $l(u,v)$. To see this, observe that for any feasible values of $z_{i,v}$ we must have

$$z_{i,v} \le dist_l(s_i, v)$$

from constraints (3.13). Indeed, by induction on the length of the shortest path from s_i to v, if $(s_i, \dots, x, v)$ is the shortest path and $z_{i,x} \le dist_l(s_i, x)$ (by induction) then

$$\begin{aligned} z_{i,v} &\le z_{i,x} + l(x,v) \\ &\le dist_l(s_i, x) + l(x,v) \\ &= dist_l(s_i, v), \end{aligned}$$

where the first inequality follows from (3.13). So, the value of the objective function is

$$\sum_{i=1}^{k} d_i \cdot z_{i,t_i} \le \sum_{i=1}^{k} d_i \cdot dist_l(s_i, t_i).$$

Now check that $z_{i,v} \equiv dist_l(s_i, v)$ is a feasible solution and its objective value is

$$\sum_{i=1}^{k} d_i \cdot dist_l(s_i, t_i).$$

Also, observe that the inequality (3.12) is in fact an equality in the optimal solution. This holds because if in the optimal solution the inequality is not tight, we can always increase the lengths without violating the constraints but achieving a better solution thus, contradicting the fact that the previous solution was optimal. Therefore, **(D1)** reduces to the following formulation

(D1')

$$maximize \textstyle\sum_{i=1}^{k} d_i \cdot dist_l(s_i, t_i)$$

subject to

$$\sum_{(u,v)\in\mathcal{E}} c(u,v)l(u,v) = 1 \tag{3.14}$$

$$l(u,v) \geq 0.$$

3.3.2 Path based formulation

In the path-based formulation we deal again with the same specification as before. We have a network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with capacities $c(e)$ for all edges $e \in E$,and demands d_i, $i = 1, 2, \ldots, k$, for each commodity i, source s_i and destination t_i.

Let $\mathcal{P}_i$ be the set of paths between vertices s_i and t_i and let $\mathcal{P} := \bigcup_i \mathcal{P}_i$. We assume that for any two distinct commodities $i \neq j$, $(s_i, t_i) \neq (s_j, t_j)$, so the set of paths $\mathcal{P}_i$ and $\mathcal{P}_j$ are distinct. (If two commodities had the same source and destination, then we would view them as one commodity). Let $f(P)$ denote the amount of flow sent along path $P, \forall P \in \mathcal{P}$. The path-based LP formulation of the maximum concurrent flow problem is the following

(P2)

$$maximize\ \gamma,$$

such that

$$\sum_{P:e\in P} f(P) \leq c(e), \forall\, e \in \mathcal{E}, \tag{3.15}$$

$$\sum_{P\in\mathcal{P}_i} f(P) \geq \gamma d_i, \forall\, i : i = 1, 2, \ldots, k, \tag{3.16}$$

$$f(P) \geq 0, \forall P \in \mathcal{P}$$

The sum on the left hand side of equation (3.15) is the total flow on edge e. In the edge based formulation the problem **(P0)** of maximizing the throughput γ is equivalent to the problem **(P1)** of minimizing the congestion, λ. We have a similar equivalence in the path based formulation. An equivalent formulation, which can be obtained by substituting γ with $1/\lambda$ and $f(P)$ with $f(P)/\gamma$ is:

(P3)

$$\textit{minimize}\ \lambda,$$

such that

$$\sum_{P:e\in P} f(P) \leq \lambda c(e), \forall\, e \in \mathcal{E}, \tag{3.17}$$

$$\sum_{P\in\mathcal{P}_i} f(P) \geq d_i, \forall\, i : i = 1, 2, \ldots, k, \tag{3.18}$$

$$f(P) \geq 0, \forall P \in \mathcal{P}$$

The dual LP for **(P2)** has a variable $l(e)$ for each capacity constraint (3.17) of the primal, called the length of edge e, and a variable z_i for each commodity demand constraint (3.18).

(D2)

$$\textit{minimize}\ \ \textstyle\sum_{e\in\mathcal{E}} c(e)l(e)$$

such that

$$\sum_{e \in P} l(e) \geq z_i, \quad \forall i, \forall P \in \mathcal{P}_i, i = 1, 2, \ldots, k,$$

$$\sum_{i=1}^{k} d_i z_i \geq 1,$$

$$l(e) \geq 0, \forall e \in \mathcal{E},$$

$$z_i \geq 0, \forall i : i = 1, 2, \ldots, k.$$

In each optimal solution of **(D2)**, $z_i = dist_l(s_i, t_i)$ and $\sum_{i=1}^{k} d_i dist_l(s_i, t_i) = 1$. We can argue that the constraint $\sum_{i=1}^{k} d_i z_i \geq 1$ is an equality. The reason is that if the constraint is not tight we can divide all the edge lengths by some number $\beta > 1$ so that all the constraints are satisfied but the objective function is smaller, contradicting the fact that it is optimal, i.e. the minimum. Hence the problem formulation can be restated as

(D2')

$$minimize \ \ \sum_{e \in \mathcal{E}} c(e) l(e)$$

such that

$$\sum_{i=1}^{k} d_i \cdot dist_l(s_i, t_i) = 1,$$

$$l(e) \geq 0, \forall e \in \mathcal{E},$$

which is equivalent to the dual of the edge based formulation **(D1)**. We will refer later also to the dual of **(P3)**. The direct dual of **(P3)** is the following problem.

(D3)

$$maximize \ \ \sum_{i=1}^{k} d_i z_i$$

such that

$$\sum_{e \in P} l(e) \geq z_i, \quad \forall i, \forall P \in P_i, i = 1, 2, \ldots, k, \tag{3.19}$$

$$\sum_{e \in \mathcal{E}} c(e)l(e) \leq 1,$$

$$l(e) \geq 0, \forall e \in \mathcal{E},$$

$$z_i \geq 0, \forall i : i = 1, 2, \ldots, k.$$

In any optimal solution of **(D3)** we have $z_i = dist_l(s_i, t_i)$ and $\sum_{i \in \mathcal{E}} c(e)l(e) = 1$ (from the constraints (3.19)) and the objective function. Hence we can consider the dual in the following form.

(D3')

$$maximize \quad \sum_{i=1}^{k} d_i \cdot dist_l(s_i, t_i)$$

such that

$$\sum_{e \in \mathcal{E}} c(e)l(e) = 1,$$

$$l(e) \geq 0, \forall e \in \mathcal{E}.$$

The formulations we have described are exploited when designing algorithms to solve the maximum concurrent flow problem. Some algorithms have been developed using only the dual of the formulations and others use the primal dual relation. We will explain the derivation of the different algorithms used to solve the MCF problem using these formulations in Chapters 4 and 5.

3.4 Applications of Multicommodity Flow Problems

Multicommodity flow problems arise in a variety of applications, from distribution of goods and transportation problems to computer and communication networks. In this section we consider several application domains and indicate how we can formulate them to become problem instances of multicommodity flow problems. In most cases we do not get clear problems as MCF, but need to consider additional constraints/objectives.

3.4.1 The Sparsest Cut Problem

Consider an undirected network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with capacities $c(e)$ for all edges $e \in E$, and a source-sink-demand specification (s_i, t_i, d_i) for each commodity i with $i = 1, 2, \ldots, k$. For each subset $S \subseteq \mathcal{N}$, let $I(S) = \{i : |S \bigcap \{s_i, t_i\}| = 1\}$ denote the terminal pairs that are disconnected by removing the edges in $C(S) = \{(u, v) \in \mathcal{E} : u \in S, v \notin S\}$. The *sparsity ratio* $\rho(S)$ of the set S is given by the total capacity of the edges which connect S with $\mathcal{N} \setminus S$ divided by the total demand separated by this cut, i.e.:

$$\rho(S) = \frac{\sum_{e \in C(S)} c(e)}{\sum_{i \in I(S)} d_i}. \tag{3.20}$$

The objective in the Sparsest Cut Problem is to minimize the sparsity ratio $\rho(S)$ over all S such that $I(S) \neq \emptyset$. In other words the objective is to remove the maximum demand possible in the "cheapest" way with respect to capacity.

Finding a sparsest cut is NP-Hard [58]. The Sparsest Cut problem has numerous applications in designing approximation algorithms for optimization problems in the area of routing and embedding (see [71, 54]). It can be shown that the dual of Maximum Concurrent Flow Problem is in fact equivalent to the relaxation of the Sparsest Cut Problem (see [71]). This relation has given rise to many algorithms that solve the Sparsest Cut Problem by exploiting the properties of the MCF Problem (see [36, 54]).

3.4.2 VLSI Circuit

Very-large-scale-integration (VLSI) is the process of integrating thousands of electronic components into a small area (for example creating integrated circuits by combining transistors into a single chip). The transistors lay out in a grid each one placed at a node. The wires connecting the transistors may be viewed as edges. Certain transistor pairs must communicate with each other using these wires. The transistor pairs may be viewed as source-sink pair and the set of wires connecting them as a path. The objective is to find a path between each source-sink pair with the constraint that each transistor must be used only by one communication pair. This is required because information passing through the same wire may cause distortion. Extensive work on VLSI design can be found in [2, 15, 66, 68]. One can understand now that if we have thousands pairs of transistors we need to find such a path for, we could end up with thousands of layers of wires resulting in a huge cost. Therefore we need to design a communication layout which minimizes the number of layers. To solve the problem of minimizing the number of layers required to build a VLSI circuit we need to solve a special version of the Maximum Concurrent Flow Problem.

3.4.3 Transportation and Distribution Networks

In distribution networks multiple products need to be distributed from production points to consuming points using fleets of trucks, trains or tankers (see [67]). Middle points such as warehouses or rail yards may be present. Certain capacities exist related to the transportation fleets and to the storage fleets. Transportation networks arise in several contexts in our daily lives such as road, air, and rail networks (see [79]). They involve the transportation of commodities, including goods, people or services (see [17]). The network topology and the capacities of the means of transportation impose constraints on the cost and the amount of commodities that can be moved. The objective in this problem setting is to minimize the operational costs. The problem can be formulated by the minimum cost multicommodity flow problem.

3.4.4 Computer and Communication Networks

Computer and communication networks are one of the most frequent applications of the multicommodity flow problems. Computers and storage devices are linked together with transmission lines exchanging data between them. The devices can be considered as nodes and the transmission lines as edges with certain capacities. The data transmitted are represented by commodities with demands being the amount of data requested by the different computer devices. The data transmitted can be messages (see [59, 8]), packets (see [56]), videos and many more.

3.4.5 Other Applications

The multicommodity flow problems have applications in numerous other problem domains. An example is cryptography, Lai [53] converted the problem of simultaneously generating independent keys for independent users into a multicommodity flow in a multigraph. Their problem was to distribute a secret key between each pair of users in a wireless network so that a secure communication could be established. They denoted each user by a node and each key by a commodity. The problem now is to find a path between all the pairs so that the requested commodity can be sent. The links between two nodes have capacities equal to the rate of the mutual information (the amount of information exchanged per unit of time between the two nodes). Then the problem of simultaneously distributing keys for different users becomes a multicommodity flow problem (see [37]).

Finally, the road congestion problem [61] is another area that can be addressed using a solution of the multicommodity flow problem. The source-sink pair is accordingly the origin-destination of a traveler (commodity) with the demand being equal to 1, i.e. the unit (car) which moves in the network. The interesting part is that each user would like to minimize her own cost (travel time, widely known as the Wardrop's first principle [78], or travel cost) whereas in the greater context the objective is to minimize the total congestion in the network. To avoid "selfish" behavior one can impose a congestion charge, i.e. a fee is paid if a user moves in a certain area at some specific time. The fees for traveling along a particular edge can be obtained from the dual solution (edge lengths). The optimal solution then can be enforced if everyone chooses the cheapest way to transport. This problem domain can be formulated by the maximum concurrent flow problem.

3.5 Summary

In this chapter we have presented the multicommodity flow problems and introduced the maximum concurrent flow problem, which is the main problem addressed in this thesis. The main applications of the multicommodity flow problems have been briefly discussed. We have also provided both the edge and the path based formulation for the problem along with their variations. These formulations have given rise to solution algorithms for the MCF problem. In the following chapter we explore the main solution methods and give an overview of the previous work on the maximum concurrent flow problem.

Chapter 4

Solution Methods and Previous Results

Contents

The Multicommodity Flow Problem can be considered as a generalization of the single-commodity maximum flow problem. The Maximum Flow Problem (MF) was first formulated in the 1930s to study the Russian Railway System (see Schrijver [69]). Ford and Fulkerson [26] first solved the problem in 1950s using augmenting paths - a method known now as the Ford-Fulkerson algorithm. Numerous methods have been proposed to solve the MF problem since then, the most fundamental ones being the network simplex method [18, 34], the blocking-flow method [23] and the push-relabel method [32, 34]. The Maximum Flow Problem inspired Ford and Fulkerson [26] and Elias et al. [24] to independently prove the Max-Flow Min-Cut Theorem, one of the most important theorems in the area of Combinatorial Optimization and a basis for numerous algorithms for various network flow problems.

However, although the MF problem has been studied extensively and various algorithms have been proposed to solve it, its solution methods do not readily

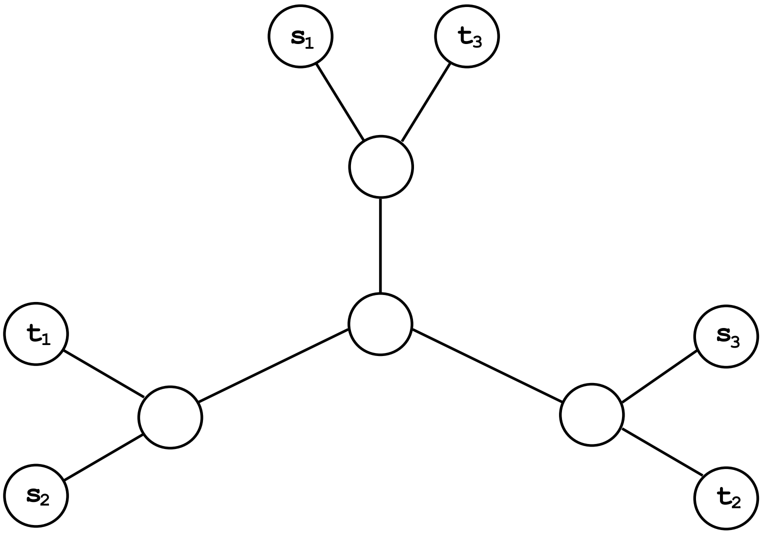


Figure 4.1: A three-commodity flow counterexample for the Max-Flow Min-Cut Theorem in an undirected network with uniform capacities and demands(all equal to 1). The minimum cut (separating all sources from their destinations) has capacity two, but the maximum value of flow is equal to 1.5

extend to the multicommodity flow problems. This is due to the set of bundle constraints, which limit the total flow of all commodities on each edge. Furthermore the Max-Flow Min-Cut theorem cannot extend to more than two commodities (see [38]) even for an undirected network. Fulkerson [29] gives an example of an undirected network with three commodities where the maximum flow is strictly less than the minimum cut (see Figure 4.1).

For directed graphs, the Max-Flow Min-Cut Theorem does not even hold for the two-commodity case. Kennington [46] gives an example of such a case (see Figure 4.2). In this example the minimum cut is two but the maximum flow is 1.5. Leighton and Rao [54] prove an approximate Max-Flow Min-Cut Theorem. They showed that for a uniform capacity Multicommodity Flow problem the minimum cut ratio is within an $O(\log n)$ factor of the maximum concurrent flow. Their groundbreaking work inspired a line of research on the Approximate Max-Flow Min-Cut Theorem. Klein et al. [49] extended this result to the general MCF problem and proved an $O(\log D \log C)$ approximate Max-Flow Min-Cut Theorem, where D is the sum of the demands and C is the sum of the capacities. Their work was improved to a $O(\log^2 k)$ ratio by Tragoudas [75], Garg et al. [31] and

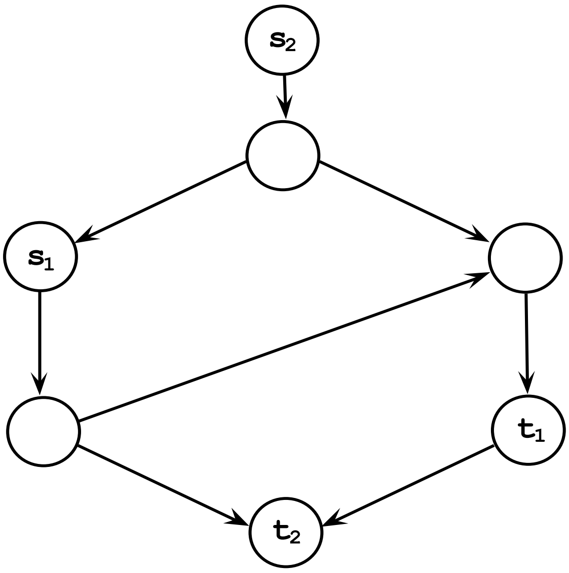


Figure 4.2: A two-commodity flow counterexample for the Max-Flow Min-Cut Theorem in a directed network with uniform capacities and demands (all equal to 1).

Plotkin and Tardos [64]. The running time was further improved to a factor of $O(\log k)$ by Aumann and Rabani [3].

The literature on the Multicommodity Flow Problem has been substantial since the publications of the work of Ford and Fulkerson [27] and Hu [38]. The MCF-problem is very important not only for its direct applications but also for its use in solving NP-Hard problems. Leighton and Rao [54] show how an approximate solution of the MCF-problem can be used to find an approximate solution for the sparsest cut problem. They use this result to design approximate algorithms for a variety of NP-Hard problems such as graph partitioning, min-cut linear arrangement and minimum feedback arc set. Klein et al. [49] generalized these results and used them to develop an $O(\log^3 n)$-approximate algorithm for the NP-hard problem of minimum deletion of clauses of a 2 CNF-formula.

For the remainder of this chapter we present the main solution methods used to solve the MCF-problem. Firstly, in Section 4.1 we present the exact solution algorithms for solving the MCF-problem and subsequently, in Section 4.2 we explore the approximation algorithms for this problem.

4.1 Exact Solution Algorithms

The scope of this thesis is to examine approximation algorithms for the MCF-problem. However, to cover the whole spectrum we provide some literature review on exact solution algorithms. Since the MCF-problem can be formulated as a linear program it can be solved using linear programming (LP) techniques. The first linear programming method dates back to the times of World War II. Kantorovich [41] developed linear programming problems to plan expenditures and returns in order to reduce costs for the army. Dantzig [19] proposed the simplex method to solve this problem, which on average is very efficient, but was proved by Klee and Minty [48] to have an exponential running time in the worst case.

Up to 1979 the research community could not prove that the general LP problem could be solved polynomially. Khachiyan [47] first showed that linear programming could be solved in polynomial time by proposing the ellipsoid method. The method terminates in $O(n^6 B^2)$ running time, where n is the dimension of the problem (the number of variables) and B is the total number of bits in the input. The algorithm is inferior to the simplex algorithm proposed by Dantzig [19] in most cases encountered in practice but unlike the simplex algorithm, it is always polynomial. Later Karmarkar [43] proposed the interior point method which improved the running time to $O(n^{3.5} B^2)$. Many improvements have been proposed since then.

The multicommodity flow problem can be solved using general linear programming techniques. Some authors have tried to exploit its structure specifically and proposed exact algorithms which run significantly faster for the multicommodity problem. Up to recently the fastest multicommodity flow algorithm for finding an exact solution has been the one by Vaidya [76] which has a complexity bound of $O(k^{2.5} m^{0.5} n^2 B)$ time. Kamath and Palmon [40] have also proposed an algorithm with faster running time when $m < k^{0.9} n^{0.9}$. Their algorithm, which is based on the interior point method, terminates in $O((k^{0.5} m^{2.7} + km^{1.5} n^{1.2} + km^{2.5})B)$.

4.2 Approximation Algorithms

The maximum concurrent flow problem can be solved by LP solvers, applied to LP formulations given in Section 3.3 but the complexity increases rapidly as the number of commodities in the network increases. Since solving the problem using LP solvers may become inefficient for large problem instances we need to approach the problem by approximation algorithms. Using formulation **(P1)**, the feasible flows f_i and a feasible value λ must satisfy

$$\lambda \geq \frac{\sum_{i=1}^{k} f_i(u,v)}{c(u,v)}, \forall\ (u,v) \in \mathcal{E}. \tag{4.1}$$

Since we are minimizing λ we can set:

$$\lambda = \max_{(u,v)\in\mathcal{E}} \left\{ \frac{\sum_{i=1}^{k} f_i(u,v)}{c(u,v)} \right\} \tag{4.2}$$

Recall that the λ value in (4.2) is referred to as the congestion of the *concurrent flow* $\mathbf{f} = (f_1, f_2, ..., f_k)$. At this point we recall the notion of an ϵ-approximate algorithm given in Definition 11 (see subsection 2.4). Note that in the context of the maximum concurrent flow problem we measure the cost by its congestion value λ. Given a parameter $\epsilon \geq 0$ we say that a flow is an ϵ-approximate flow if its congestion λ is at most $(1+\epsilon)$ times the optimal congestion λ^*, that is, $\lambda \leq (1+\epsilon)\lambda^*$. Thus, following Definition 11, an ϵ-approximate algorithm for the maximum concurrent flow problem computes ϵ-approximate flows.

In this section we introduce the main combinatorial algorithmic methods for designing approximate multicommodity flow algorithms. We describe their main features and how they can be used to design approximation algorithms. We describe the main algorithms used to solve the multicommodity flow problem based on these methods in the next chapter.

4.2.1 The Primal-Dual Approach

The primal-dual method is one of the many linear programming techniques that have been devised to solve combinatorial optimization problems. The method was first proposed by Dantzig et al. [20] who extended the "Hungarian Method" proposed by Kuhn [52] for the assignment problem to a general algorithm for linear programming problems. In this section we are going to describe the classical primal-dual method and its use in approximation algorithms. For a more in depth review on this method see the textbooks of Papadimitriou and Steiglitz [63] and Williamson [80].

Consider the linear program

(LP)

$$\min \sum_{j=1}^{n} c_j \cdot x_j$$

subject to

$$\sum_{j=1}^{n} a_{i,j} x_j \geq b_i, \text{ for } i = 1, 2, \ldots, m,$$
$$x_j \geq 0, \text{ for } j = 1, 2, \ldots, n.$$

Its dual is

(DLP)

$$\max \sum_{i=1}^{m} b_i \cdot y_i$$

subject to

$$\sum_{i=1}^{m} a_{i,j} y_i \leq c_j, \text{ for } i = 1, 2, \ldots, n,$$
$$y_i \geq 0, \text{ for } i = 1, 2, \ldots, m.$$

For the linear programs given above we have a set of complementary slackness conditions that must be satisfied to have an optimal solution. The conditions are given below:

Primal complementary slackness condition:

C1: for each j, either $x_j = 0$ or $\sum_{i=1}^{m} a_{i,j} y_i = c_j$,

Dual complementary slackness condition:

C2: for each i, either $y_i = 0$ or $\sum_{j=1}^{n} a_{i,j} x_j = b_i$.

Theorem 1 ([63]). *Vectors $\mathbf{x}$ and $\mathbf{y}$ are optimal if and only if the conditions above are satisfied.*

The primal-dual methods starts from a feasible dual solution, usually we set $\mathbf{y} = \{y_1, y_2, \ldots, y_n\} = 0$. Then, in each iteration it tries to find a feasible primal solution which satisfies the complementary slackness conditions. Either it finds a feasible primal solution $\mathbf{x}$ that obeys the complementary slackness conditions, so both $\mathbf{x}$ and $\mathbf{y}$ are optimal, or it finds a modified $\mathbf{y}$ which improves the objective function. We keep trying to modify $\mathbf{y}$ in a certain fashion until we can find a primal feasible solution $\mathbf{x}$, and hence an optimal solution for the problem.

In approximation algorithms instead of using the tight complementary slackness conditions we relax them so that we can obtain an approximate optimal solution. Finding an approximate solution is much faster than finding the exact solution of the problem and with the right measure guarantee we can ensure that our solution is close enough to the optimal one. Consider the following set of relaxed complementary slackness conditions.

Primal relaxed complementary slackness condition:

$$\text{CR1: for each } j\text{, either } x_j = 0 \text{ or } c_j \geq \textstyle\sum_{i=1}^{m} a_{i,j} y_i \geq c_j/\alpha,$$

Dual relaxed complementary slackness condition:

$$\text{CR2: for each } i\text{, either } y_i = 0 \text{ or } b_i \leq \textstyle\sum_{j=1}^{n} a_{i,j} x_j \leq \beta b_i,$$

where $\alpha, \beta > 1$ (see, for example [72]). It can be proved that using these conditions we can get a solution close to the optimal.

Lemma 1. *If $\mathbf{x}$ and $\mathbf{y}$ are feasible solutions to the primal and dual problem and satisfy the relaxed optimality conditions stated above then $\mathbf{x}$ is an $(\alpha\beta - 1)$-approximate solution for the primal problem.*

Proof. Consider the objective function of the linear programming formulation.

We have

$$\sum_{j=1}^{n} c_j x_j \leq \alpha \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{i,j} y_i) x_j \tag{4.3}$$

$$= \alpha \sum_{i=1}^{m} y_i \sum_{j=1}^{n} (a_{i,j} x_j) \tag{4.4}$$

$$\leq \alpha\beta \sum_{i=1}^{m} b_i y_i \tag{4.5}$$

Hence by relaxing the complementary slackness conditions we are able to have a solution within a factor of $\alpha\beta$ of the optimal one, that is $\mathbf{x}$ is an $(\alpha\beta - 1)$-approximate solution for the primal problem. □

The following pseudocode gives a description of a generic way of computing an approximate solution using the primal-dual approach.

```
Algorithm: Primal-Dual Approximation
Initialization Round: Start with a dual feasible solution
$\mathbf{y} = \{y_1, y_2, \ldots, y_n\}$(Usually $\mathbf{y} = 0$) and find the primal solution
$\mathbf{x} = \{x_1, x_2, \ldots, x_m\}$, so that the pair $\mathbf{x}, \mathbf{y}$ satisfies the relaxed
Complementary Slackness Conditions. ;
while Primal solution $\boldsymbol{x}$ is infeasible do
    1. Increase one or more values $y_i$ until some dual constraints go tight,
       i.e. $\sum_{i=1}^{m} a_{i,j} y_i = c_j/\alpha$ for some $j$, while maintaining feasibility of the
       dual vector $\mathbf{y}$
    2. Select some subset of tight dual constraints and increase the values
       of the corresponding primal variables;
end
Output: Approximate optimal solution
```

For the maximum concurrent flow problem the following Complementary Slackness Conditions must be satisfied for an optimal solution. These conditions are derived directly from the Primal-Dual formulations **(P3)-(D3)** (Subsection 3.3.2):

Primal complementary slackness condition:

MCF.C1: for each P, either $f(P) = 0$ or $l(P) \equiv \sum_{e \in P} l(e) = dist_l(s_i, t_i)$.

Dual complementary slackness condition:

MCF.C2: for each edge e, either $l(e) = 0$ or $f(e) = \lambda c(e)$,

Many approximation algorithms for the MCF problem are based on relaxations of the conditions MCF.C1 and MCF.C2, obtaining conditions similar to CR1 and CR2, though often technically more complicated. For example, the relaxed complementary slackness conditions for MCF would also relax the conditions "$x_i = 0$" and "$y_i = 0$" in CR1 and CR2, allowing both the primal and dual variables to take on some appropriately small values. An important part of developing an approximation algorithm based on relaxed complimentary slackness conditions is to prove that if those conditions are satisfied, then the solution has the desired approximation precision. There are also different ways of arranging the progress of a primal-dual algorithm. For example, some MCF algorithms that we review later, maintain the flow and the edge lengths both feasible and satisfy one of the complimentary slackness conditions. Such an algorithm makes progress during the computation towards satisfying also the other relaxed complementary slackness condition. We discuss this further in the following chapters.

4.2.2 Previous Work

We summarize here the main algorithmic results for the approximate MCF problem. The details of the two main approaches - rerouting and incremental - are discussed in the next chapters.

The Maximum Concurrent Flow Problem has been introduced by Shahrokhi and Matula [70] in early 1990s. They described an approximation algorithm for the uniform capacity case based on an exponential length function. Their algorithm first determines a feasible solution using an initial length function, it then calculates the shortest paths for each commodity and reroutes the flow where appropriate. The algorithm terminates in $O(\epsilon^{-5} nm^7)$ time. This algorithm started the "rerouting" framework for the MCF problem and a number of improvements followed. Klein et al. [50] gave a faster approximation algorithm for the unit capacity concurrent flow problem by using a different length function. The new length function made it possible to make bigger improvements in each iteration thus reducing the total running time. The algorithm proposed in [50] runs in $\widetilde{O}(km(\epsilon^{-1}n+\epsilon^{-3}m))$. Leighton et al. [55] generalize the results in Shahrokhi and Matula [70] and [50] for the case of arbitrary capacities and propose an algorithm which runs in $\widetilde{O}(\epsilon^{-2}k^2mn)$ time. Radzik [65] improved this time by a factor of k by providing a deterministic round-robin selection of the commodities instead of a randomized one.

On a departure from this line of research Garg and Koenemann [30] describe the first approximation algorithm based on an iterative method for the MCF problem that terminates in $\widetilde{O}(\epsilon^{-2}m(m+k)+knm^2)$ based on Primal-Dual methods. The algorithm proceeds in phases each one consisting of k iterations. In each phase d_i units of flow of each commodity are routed from its source to its destination. Then the length of each edge is updated and the algorithm proceeds to the next phase. A shortest path is calculated for a commodity using the current length function and flow is sent along this path so that it is saturated. Fleischer [25] improved the bound of the algorithm by noticing that it is sufficient to compute approximate maximum flows for each commodity instead of exact ones. This removed the bottleneck on the running time of [30] of calculating k maximum flows and thus reduced the running time to $\widetilde{O}(\epsilon^{-2}m(m+k))$. Subsequently Karakostas [42] managed to improve the running time of the algorithm by grouping together all the commodities with the same source. This way the shortest path tree for all these commodities can be computed by just one call of the Dijkstra's algorithm. Flow is then routed along these paths in ratio of the demands of the commodities. Hence the number of iterations in each phase is now bounded by the number of sources, which is at most $min(n,k)$. The running time of Karakostas' algorithm [42] is $\widetilde{O}(\epsilon^{-2}m^2)$. Recently Madry [57] managed to improve this running time by randomizing the selection of paths in each iteration and providing a suitable data structure to manage the paths. Madry's algorithm [57] terminates in $\widetilde{O}(\epsilon^{-2}(m+k)n)$ time.

The Flow Deviation Method for the MCF problem introduced by Fratta et al. [28] is another example of the incremental method. This method is applied to the problem **(P2)** of maximizing the throughput γ. The method starts with a feasible flow satisfying strict capacity constraints $(f(e) < c(e))$ and gradually increases the throughput while maintaining these constraints. The difference between the flow deviation method and the incremental algorithms mentioned above is the type of edge lengths (dual variables) used. The edge lengths in the algorithms which follow the Garg-Koenemman's [30] incremental scheme are generally of the form

$$l(e) = \exp(\alpha f(e)),$$

for some α which depends on ϵ and the size of the input network. On the other hand, in the flow deviation method, the edge lengths can be viewed as having the form

$$l_f(e) = \frac{c(e)}{(c(e) - f(e))^2}.$$

The objective of the dual **(D2)** is to minimize $\sum_{e \in \mathcal{E}} l(e)c(e)$. This translates in Fratta et al. [28] to minimizing the function $\Psi(f) = \sum_{e \in \mathcal{E}} l_f(e)c(e)$, which is referred to as a "barrier" function. The edge length function $l_f(e)$ ensures that the strict capacity constraints are maintained. The algorithm proposed by Fratta et al. [28] converges in $O(\epsilon^{-3}km^5)$ shortest path computations. To reduce the barrier function they perform a Frank-Wolfe procedure based on shortest paths. Bienstock and Raskina [12] give an algorithm based on this method which terminates in $(\epsilon^{-2}km^3 + km^3 \log m)$ minimum-cost flow computations. To achieve this they solve minimum-cost flow problems for each commodity to update its flow. They allow several Frank-Wolfe iterations within the improvement of the barrier function, in contrast to [28] who allow only one iteration.

Recently significant improvements have been made for the MCF problem in undirected graphs. Although this problem is out of the scope of this thesis we summarize the main results. Kelner et al. [44] give an algorithm which computes an ϵ-approximate flow in $\widetilde{O}(\epsilon^{17/3}k^4m^{4/3} + \epsilon^4k^{13/2}m)$. They adapt the algorithm from [55] to use flows with electrical capacity constraints associated with the k commodities instead of minimum cost flow as its oracle call. Kelner et al. [44] use an algorithm that is a direct extension of the electrical flow based maximum flow algorithm from [16] to minimize the maximum congestion of an edge. They reduce the problem of computing maximum flows subject to capacity constraints to the problem of computing electrical flows in resistor networks.

Subsequently, Kelner et al. [45] introduced a new framework for approximately solving flow problems in capacitated, undirected graphs and apply it to the MCF problem. The framework consists of two main parts: an iterative scheme that reduces the problem to the construction of a projection matrix with certain properties and, the construction of such a matrix. Kelner et al. [45] have improved the running time to approximately solve the MCF problem in undirected graphs to $O(\epsilon^2 k^2 m^{1+o(1)})$.

4.3 Summary

In this part we have introduced the main background and terminology for network flow problems. We have defined the Maximum Concurrent Flow problem and provided its different formulations and applications. Different algorithms for solving the problem were introduced based on its formulations. We have briefly described the main solution methods used and outlined the previous work on the MCF problem. In the next part we are going to examine sequential algorithms for solving the MCF problem. First, we analyze and compare the main algorithms proposed in the literature and show the differences between the two main methods. We then modify the main algorithms proposed to solve the MCF problem and show that each one can be viewed as a variation of the other. In the last part we are examining the main distributed algorithms for solving this problem.

Part II

Analysis of Sequential Algorithms

Chapter 5

The Main Solution Algorithms

Contents

In this chapter we consider the two main combinatorial algorithmic schemes for the Maximum Concurrent Flow Problem. In Section 5.1 we describe the Rerouting Method and in Section 5.2 we examine the Incremental Method. We explain the computation and discuss the main properties, including running times. The aim of this chapter is to elucidate the relationship between the main algorithms proposed to solve the MCF problem and to explicate the effect of their differences on the gradual improvement of the running time to solve the problem. In Chapter 6 we propose an exponential length function for the Incremental Method and show how it fits in the original analysis in [30]. This functional relationship between the length function and the flow on an edge converts the Incremental Method to an instance of the Rerouting Method. In Chapter 7 we propose a new way of calculating minimum cost flows for the Rerouting Method based on the Successive Shortest Path algorithm. This change converts a special case of the Rerouting Method to an instance of the Incremental Method. Essentially, in this part of the thesis, we show that each solution method can be considered as an instance of the other, which shows that the two frameworks are more closely related than how they have been traditionally perceived.

5.1 The Rerouting Method

The Maximum Concurrent Flow problem defined in Chapter 3, was first introduced by Shahrokhi and Matula [70]. The problem was motivated by a variation of the routing problem in packet-switched telecommunication networks. In their paper Shahrokhi and Matula [70] describe a fully polynomial-time approximation scheme (FPTAS) for the Unit Capacity MCF problem (UC-MCF). The problem is a restriction of the Maximum Concurrent Flow problem with all edge capacities equal to 1. Let $|f| = \max_{e\in\mathcal{E}}\{f(e)\}$ denote the maximum flow over all edges e (that is, the congestion of the flow, in the unit capacities case) and $l : \mathcal{E} \to \mathbb{R}_+$ be a nonnegative length function. The length of a path is denoted by $l(P) = \sum_{e\in P} l(e)$. The following theorem gives the optimality conditions for a multicommodity flow.

Theorem 2 ([70]). *If all the edge capacities are equal to* 1, *then for a multicommodity flow f satisfying the demands d_i and a length function l,*

$$\begin{aligned} |f|\sum_{e\in\mathcal{E}} l(e) = |f|\sum_{e\in\mathcal{E}} l(e) \geq \sum_{e\in\mathcal{E}} l(e)f(e) = \sum_{i=1}^{k}\sum_{e\in\mathcal{E}} f_i(e)l(e) \\ \geq \sum_{i=1}^{k} dist_l(s_i,t_i)d_i. \end{aligned} \tag{5.1}$$

Moreover, a multicommodity flow f minimizes $|f|$ if and only if there exists a nonzero length function l for which all the above terms are equal.

This theorem can be viewed as the duality theorem for the LP formulations **(P1)** and **(D1')** given in Section 3.3.1. Assuming that we consider only the normalised lengths, that is $\sum_{e\in\mathcal{E}} l(e) = 1$ as required in **(D1')**, the L.H.S. of (5.1) is the objective function of the formulation **(P1)** and the R.H.S. of (5.1) is the objective function of the formulation **(D1')**. Theorem 2 is important because it can be used to derive conditions for an ϵ-approximate flow.

Corollary 1. *For a multicommodity flow f satisfying the demands d_i and a length function l if*

$$|f| \leq (1+\epsilon)\frac{\sum_{i=1}^{k} dist_l(s_i, t_i) d_i}{\sum_{e \in \mathcal{E}} l(e)}, \tag{5.2}$$

then f is ϵ-approximate.

Proof. From Theorem 2 for an optimal flow f_{opt} the following inequality holds

$$|f_{opt}| \geq \frac{\sum_{i=1}^{k} dist_l(s_i, t_i) d_i}{\sum_{e \in \mathcal{E}} l(e)}. \tag{5.3}$$

Then from (5.2)

$$|f| \leq (1+\epsilon)\frac{\sum_{i=1}^{k} dist_l(s_i, t_i) d_i}{\sum_{e \in \mathcal{E}} l(e)} \leq (1+\epsilon)|f_{opt}|. \tag{5.4}$$

□

5.1.1 Description of Shahrokhi and Matula

The general approach is to keep updating flows while maintaining appropriate length function to keep making progress towards satisfying condition (5.2). The remarkable idea introduced by Shahrokhi and Matula [70] was to use a length function which is an exponential function of the edge flows:

$$l(e) = \exp(cm^2 f(e)/\epsilon), \tag{5.5}$$

for some constant c. This function is used to decide how the flow is rerouted from highly utilized edges to low utilized ones. The algorithm maintains the set of active paths $\mathcal{P}_i$ (paths with non-zero flow) for each commodity i. The initial flow used in this algorithm is the flow obtained by supplying the demand for each commodity i from its source s_i to its destination t_i through the path with the minimum number of edges. That is, initially there is one active path for each commodity and all demand of this commodity is sent along this path. Then the algorithm proceeds in iterations. At the beginning of each iteration it compares the maximum edge flow $|f|$ with the ratio of $\sum_{i=1}^{k} dist_l(s_i, t_i)d_i$ to $\sum_{e \in \mathcal{E}} l(e)$ (see (5.3)). If $|f|$ satisfies (5.2) then the algorithm terminates and the resulting flow is ϵ-approximate (Corollary 1). If (5.2) is not satisfied, then the algorithm calculates the shortest path P_i^{SP} from s_i to t_i for each commodity i using the exponential length function (5.5). The algorithm also chooses the longest active path $\overline{P}_i \in \mathcal{P}_i$ for each commodity i. It then chooses the commodity i which maximizes the quantity

$$l(\overline{P}_i) - l(P_i^{SP}),$$

and reroutes a small fraction of the flow of commodity i from $\overline{P}_i$ to P_i^{SP} (P_i^{SP} is added to the set of active paths $\mathcal{P}_i$, if not already there). This procedure continues until the stopping condition for approximate optimality is met. The algorithm runs in $O(\epsilon^{-5}m^8)$ time, because it can be shown that there are at most $O(\epsilon^{-2}m^3)$ active paths at any iteration and thus each iteration can be implemented in $O(\epsilon^{-2}m^4)$ time, with at most $O(\epsilon^{-3}m^4)$ iterations. Shahrokhi and Matula [70] showed that this bound can be improved to $O(\epsilon^{-5}nm^7)$.

5.1.2 Klein's Proposal

Later this algorithm has been improved in a number of ways. Klein et al. [50] improved the running time by introducing a different length function:

$$l(e) = \exp(c \log(m\epsilon^{-1}) f(e)/\epsilon|f|), \tag{5.6}$$

where c is some constant. Instead of using absolute edge flow values as in (5.5), this length function uses edge flows relative to the maximum congestion. The crucial change however is the change of the factor m^2 in (5.5) to $\log m$. This enables rerouting a higher fraction of flow resulting in considerably fewer iterations. Klein et al. [50] also proposed a new framework of approach to the problem. The notion of an ϵ-good path for a commodity i is introduced. A path $P \in \mathcal{P}_i$ is ϵ-good if the following condition holds

$$l(P) - dist_l\{s_i, t_i\} \leq \epsilon l(P) + \epsilon \frac{\max_{e \in \mathcal{E}}\{f(e)\}}{\min\{D, kd_i\}}, \tag{5.7}$$

where D is the sum of the demands d_i. The intuition behind this definition is that a path for a commodity is ϵ-good if its length is not much greater than the shortest path length. A path that is not ϵ-good is said to be ϵ-bad. They also introduce relaxed optimality conditions based on the notion of ϵ-good paths which enable them to check if the current flow is ϵ-approximate. If it is not, then an appropriate ϵ-bad path is chosen for improvement/rerouting. These conditions are given below, where f denotes a flow and l denotes a length function:

R1.1: $\forall e \in \mathcal{E}$ either $l(e) \leq (\epsilon/m) \sum_{e \in \mathcal{E}} l(e)$ or $f(e) \geq \max_{e \in \mathcal{E}}\{f(e)\}/(1+\epsilon)$

R1.2: $\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i \; s.t. \; P \epsilon-bad} f_i(P) l(P) \leq \epsilon \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} f_i(P) l(P)$.

The approximate optimality conditions are a relaxation of the complementary slackness conditions MCF.C1 and MCF.C2 for the MCF problem introduced in Section 4.2.1. The ϵ-good condition and R1.2 together are a relaxation of MCF.C1. The condition MCF.C1 requires that all flow goes only on the shortest paths ($f(P) > 0$ only if $l(P) = dist_l(s_i, t_i)$). The condition R1.2 relaxes this by requiring *most* of the flow to be on *approximate* shortest paths (ϵ-good paths). If a small fraction of the flow is on other paths (the ϵ-bad paths) then its cost (the LHS in R1.2) cannot be more than an ϵ fraction of the total. Condition R1.1 is a direct relaxation of MCF.C2. The condition MCF.C2 requires that for each edge either the length of this edge is zero or the edge is saturated ($f(e) = \lambda c(e)$). The condition R1.1 relaxes this by requiring that for each edge either the length of this edge is *relatively small* (the first part of R1.1) or the edge is *almost saturated* (the second part of R1.1). It can be shown that these two conditions imply that f is 7ϵ-approximate.

The particular exponential length function proposed by Klein et al. [50] always satisfies condition R1. The algorithm then tries to improve the flow so that eventually condition R2 is also satisfied. The algorithm iteratively finds an active path (it maintains active paths, as in [70]) that is ϵ-bad and reroutes a fraction of flow from this path to the shortest path from s_i to t_i. This way it gradually reroutes flow to less congested paths until both relaxed optimality conditions are satisfied.

The bottleneck procedure of the algorithm is the identification of an ϵ-bad active path. For this procedure they propose various deterministic and randomized schemes. The basic deterministic scheme is essentially similar to the one proposed by Shahrokhi and Matula [70]. To find an ϵ-bad path they find the shortest path P_i^{SP} and the longest active path $\overline{P}_i$ from s_i to t_i for each commodity i and check whether the condition (5.7) for an ϵ-good path holds. Thus their improvement does not come from the way they choose the paths to reroute flow. They achieve better running times than [70] because the new length function (5.6) enables them to make better improvement in each iteration, thus reducing the number of iterations of their algorithm. The notion of an ϵ-good commodity and the new approximate optimality conditions proposed are exploited in the randomized scheme. For the randomized algorithm they note that an ϵ-bad path contributes at least an ϵ fraction on the total length $\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} f_i(P) l(P)$. To find an ϵ-bad path they select an edge with probability proportional to $l(e)f(e)$ and choose an active path P_e through this edge with probability proportional to its flow. It can be shown that the probability of choosing this path is proportional to its contribution to the sum $\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} f_i(P) l(P)$. So in this randomized process the probability of selecting a bad path is ϵ, so we get an ϵ-bad path in expected $1/\epsilon$ iterations if such a path exists. To implement efficiently this randomized selection a dedicated data structure is proposed. The running time for the randomized algorithm is

$$O((k\epsilon^{-2} + m\epsilon^{-4})(m + n\log n)\log n) = \widetilde{O}((k\epsilon^{-2} + m\epsilon^{-4})(m+n)).$$

The running time of the simple deterministic algorithm is

$$\begin{aligned} &O((k\epsilon^{-1} + m\epsilon^{-3})(k^* n\log n + m(\log n + min\{k, k^* \log d_{max}\}))\log n)) \\ &= \widetilde{O}(k(k\epsilon^{-1} + m\epsilon^{-3})(m+n)), \end{aligned}$$

where k^* is the number of different sources s_i and, d_{max} is the maximum among all demands d_i.

5.1.3 Leighton's Proposal for Arbitrary Capacities

The algorithms of Shahrokhi and Matula [70] and Klein et al. [50] are only for the Unit-Capacity Maximum Concurrent Flow problem. Leighton et al. [55] propose the first combinatorial polynomial approximation algorithm for the MCF problem with arbitrary capacities. Their approach is similar to Shahrokhi and Matula [70] and Klein et al. [50] with the main differences being that they manage to reroute an entire commodity in each iteration, instead of a flow path and they use minimum cost flow calculations instead of shortest path. In each rerouting step they calculate a minimum cost flow for a commodity i and reroute a small fraction of the total flow of this commodity onto the minimum cost flow paths calculated. They generalize the length function proposed in [50] to the case of arbitrary capacities:

$$l(e) = exp\left\{c\frac{\ln(m\epsilon^{-1})}{\epsilon\lambda_0}\frac{f(e)}{c(e)}\right\}, \tag{5.8}$$

where c is a constant and λ_0 is the current upper bound on the maximum congestion. Observe that if all capacities are the same, then (5.8) becomes (5.6). We are going to describe the approach based on the scheme provided by Leighton et al. [55] in more detail.

Recall that $\lambda(e) = f(e)/c(e)$, with $f(e)$ the flow on edge e and $c(e)$ its capacity. Let l be a non-negative length function on the edges, f a multicommodity flow, and $\lambda = max_{e\in\mathcal{E}}\lambda(e)$. Let C_i be the cost of the current flow of commodity i under the length function l and denote by $C_i^*(\lambda)$ the minimum cost flow of commodity i, subject to costs $l(e)$ and capacity constraints $\lambda \cdot c(e)$. The following theorem and the complementary slackness conditions given by linear programming were used by Leighton et al. [55] to construct the relaxed optimality conditions needed for this algorithm.

Theorem 3. *For a multicommodity flow f satisfying capacities $\lambda c(e)$ and a length function l,*

$$\lambda\sum_{e\in\mathcal{E}} l(e)c(e) \;\geq\; \sum_{i=1}^{k}\sum_{e\in\mathcal{E}} f_i(e)l(e) \;=\; \sum_{i=1}^{k} C_i \geq \sum_{i=1}^{k} C_i^*(\lambda). \tag{5.9}$$

A multicommodity flow f minimizes λ if and only if there exists a non-zero length function l for which all the above terms are equal.

Corollary 2. *For the optimal flow f^* and any length function l*

$$\lambda^* \geq \frac{\sum_{i=1}^{k} C_i^*(\lambda^*)}{\sum_{e\in\mathcal{E}} l(e)c(e)}. \tag{5.10}$$

Proof. This simply follows from Theorem 3 applied to an optimal flow f^*

$$\lambda^* \sum_{e\in\mathcal{E}} l(e)c(e) \ \geq \ \sum_{i=1}^{k}\sum_{e\in\mathcal{E}} f_i^*(e)l(e) \ = \ \sum_{i=1}^{k} C_i \geq \sum_{i=1}^{k} C_i^*(\lambda^*).$$

□

The algorithms by Leighton et al. [55] are based on the following version of the complementary slackness conditions for the MCF problem.

Theorem 4 (Complementary Slackness conditions)**.** *A multicommodity flow f has a minimum congestion λ if and only if there exists a non-negative length function l such that:*

MCF.C1 *For each edge $e \in E$, either* **(a)** $l(e) = 0$ *or* **(b)** $f(e) = \lambda \cdot c(e)$

MCF.C2' *For each commodity i, $C_i = C_i^*(\lambda)$*

Given these conditions we can identify when a flow f and a length function l are optimal. As defined in Section 4.2 a flow f is ϵ-*approximate* if it has a congestion at most $(1 + \epsilon)$ times the optimal congestion λ^*. We would like to have conditions for checking whether a current solution is ϵ-*approximate* thus we are going to use a relaxed version of Theorem 3 and of the complementary slackness conditions to obtain this. Let f be a multicommodity flow and l a length function. Instead of classifying flow paths as ϵ-good or ϵ-bad as in [50], here the flows of whole commodities are classified as ϵ-good or ϵ-bad. For $\epsilon > 0$ we say that a commodity i is ϵ-*good* if

$$C_i - C_i^*(\lambda) \leq \epsilon C_i + \epsilon\frac{\lambda}{k}\sum_{e\in E} l(e)c(e). \tag{5.11}$$

The intuition behind this notion is that a commodity with cost flow equal to its minimum cost flow, with an ϵ error, or with a cost that corresponds to a very small fraction of the total cost of all commodities is considered to be ϵ-good. Such flows will sum up to an ϵ-approximate flow. Otherwise a commodity is said to be ϵ-*bad*. Leighton et al. [55] define the relaxed optimality conditions as follows:

R2.1 $\forall e \in E$ either
(a) $c(e)l(e) \leq (\epsilon/m) \sum_{e' \in \mathcal{E}} l(e')c(e')$ or
(b) $(1+\epsilon)f(e) \geq \lambda c(e)$.

R2.2 $\sum_{i\ \epsilon-bad} C_i \leq \epsilon \sum_{i=1}^{k} C_i$.

These condition are a relaxation of the Complementary Slackness Conditions MCF.C1 and MCF.C2'. The first condition R2.1 is a direct generalization of R1.1 to the case of arbitrary capacities, and a direct relaxation of the complementary slackness condition MCF.C1. For all edges either the length function $l(e)$ is close to zero, more precisely $l(e) \leq (\epsilon/mc(e)) \sum_{e' \in \mathcal{E}} l(e')c(e')$, or flow $f(e)$ is within a $(1+\epsilon)$ factor of the capacity of the edge $\lambda \cdot c(e)$, i.e. $f(e)$ close to $\lambda \cdot c(e)$. The second condition R2.2 ensures that all ϵ-bad commodities in total have only small contribution to the total cost. This condition, together with condition (5.11), indicating which commodity flows are ϵ-good, form the relaxed version of complementary slackness condition MCF.C2'. The following theorem sets the framework under which the algorithm is constructed. The two relaxed optimality conditions R2.1 and R2.2 will give us a 5ϵ-approximate flow.

Theorem 5. *A multicommodity flow f together with a length function l and an error parameter $0 < \epsilon \leq 1/5$ which satisfy the relaxed optimality conditions R2.1 and R2.2 is 5ϵ-approximate.*

If both the relaxed optimality conditions are satisfied the flow is 5ϵ-approximate. When this occurs the algorithm terminates and we get our required flow. We first prove for completeness, since it is not included in the previous literature, that the given relaxed optimality conditions will generate a 5ϵ-approximate flow then we are going to give some details of the algorithm and its running time.

The proof of Theorem 5 uses the following claim.

Claim 1. *A multicommodity flow which satisfies the relaxed optimality condition R2.1 also satisfies*

$$(1-\epsilon)\lambda \sum_{e\in\mathcal{E}} l(e)c(e) \leq (1+\epsilon) \sum_{e\in\mathcal{E}} f(e)l(e) \tag{5.12}$$

Proof. To prove this we need to consider the two cases

- $(1+\epsilon)f(e) \geq \lambda c(e)$
- $(1+\epsilon)f(e) < \lambda c(e)$,

Let $\mathcal{E}_1$ denote the set of edges which satisfy the first case and $\mathcal{E}_2$ denote the set of edges which satisfy the second case. For the first case the proof is trivial since it is obvious that if $(1+\epsilon)f(e) \geq \lambda c(e)$ then

$$\lambda \sum_{e\in\mathcal{E}_1} l(e)c(e) \leq (1+\epsilon) \sum_{e\in\mathcal{E}} f(e)l(e) \tag{5.13}$$

For the second case we have

$$c(e)l(e) \leq (\epsilon/m) \textstyle\sum_{e'\in\mathcal{E}} l(e')c(e')$$

Hence the sum over all the edges $e \in \mathcal{E}_2$ is at most

$$\textstyle\sum_{e\in\mathcal{E}_2} c(e)l(e) \leq \sum_{e\in\mathcal{E}} (\epsilon/m) \sum_{e'\in\mathcal{E}} l(e')c(e')$$

and thus

$$\lambda \sum_{e\in\mathcal{E}_2} c(e)l(e) \leq \lambda\epsilon \sum_{e'\in\mathcal{E}} l(e')c(e') \tag{5.14}$$

Adding the inequalities for the two cases (5.13) and (5.14) together gives us the required result.

□

Now we are ready to prove that the two relaxed optimality conditions imply a 5ϵ-approximate flow.

Proof of Theorem 5. We want to show that

$$\lambda \textstyle\sum_{e\in\mathcal{E}} l(e)c(e) \leq (1+5\epsilon) \sum_{i=1}^{k} C_i^*(\lambda)$$

and apply Corollary 2 to conclude that $\lambda = (1+5\epsilon)\lambda^*$. From now on C_i^* stands for $C_i^*(\lambda)$ Consider the term

$$\sum_{i=1}^{k} \sum_{e\in\mathcal{E}} f_i(e)l(e) \tag{5.15}$$

We split the sum in (5.15) into two sums

$$\mathbf{A} = \sum_{i\ \epsilon-good} \sum_{e\in\mathcal{E}} f_i(e)l(e) \tag{5.16}$$

$$\mathbf{B} = \sum_{i\ \epsilon-bad} \sum_{e\in\mathcal{E}} f_i(e)l(e). \tag{5.17}$$

Then from (5.11) we get

$$\mathbf{A} = \sum_{i\ \epsilon-good} C_i \le \sum_{i\ \epsilon-good} C_i^* + \epsilon \sum_{i\ \epsilon-good} C_i + \epsilon \sum_{i\ \epsilon-good} \frac{\lambda}{k} \sum_{e\in\mathcal{E}} l(e)c(e)$$

and hence

$$\begin{aligned}(1-\epsilon) \sum_{i\ \epsilon-good} C_i &\le \sum_{i\ \epsilon-good} C_i^* + \epsilon \sum_{i\ \epsilon-good} \frac{\lambda}{k} \sum_{e\in\mathcal{E}} l(e)c(e) \\ &\le \sum_{i=1}^{k} C_i^* + \epsilon \sum_{i=1}^{k} \frac{\lambda}{k} \sum_{e\in\mathcal{E}} l(e)c(e) \\ &= \sum_{i=1}^{k} C_i^* + \epsilon\lambda \sum_{e\in\mathcal{E}} l(e)c(e).\end{aligned}$$

For the second sum, we have

$$\mathbf{B} = \sum_{i\ \epsilon-bad} C_i \le \epsilon \sum_{i=1}^{k} C_i \le \epsilon\lambda \sum_{e\in\mathcal{E}} l(e)c(e)$$

Hence

$$\mathbf{A} + \mathbf{B} \le \sum_{i=1}^{k} C_i^* + 2\epsilon\lambda \sum_{e\in\mathcal{E}} l(e)c(e)$$

On the other hand, from (5.12)

$$\mathbf{A} + \mathbf{B} \geq \frac{(1-\epsilon)}{(1+\epsilon)} \lambda \sum_{e \in \mathcal{E}} l(e)c(e)$$

Thus,

$$\frac{(1-\epsilon)}{(1+\epsilon)} \lambda \sum_{e \in \mathcal{E}} l(e)c(e) \leq \sum_{i=1}^{k} C_i^* + 2\epsilon\lambda \sum_{e \in \mathcal{E}} l(e)c(e)$$

which implies that

$$\lambda \sum_{e \in \mathcal{E}} l(e)c(e) \leq \frac{(1+\epsilon)}{(1-3\epsilon-2\epsilon^2)} \sum_{i=1}^{k} C_i^* \leq \frac{(1+\epsilon)}{(1-3\epsilon-2\epsilon^2)} \sum_{i=1}^{k} C_i^*(\lambda)$$

From Corollary 2 we have

$$\lambda^* \geq \frac{\sum_{i=1}^{k} C_i^*(\lambda^*)}{\sum_{e \in \mathcal{E}} l(e)c(e)}.$$

Thus,

$$\lambda \leq \frac{(1+\epsilon)}{(1-3\epsilon-2\epsilon^2)} \lambda^* \leq (1+5\epsilon)\lambda^*.$$

The last inequality follows from the assumption that $\epsilon \leq 1/5$. □

Formally the Leighton et al. [55] algorithm proceeds as follows. We start from a flow that has congestion $\lambda_0 \leq k\lambda^*$. To obtain such a flow we find the maximum flow separately for each commodity i and then combine their flows to obtain a concurrent flow. It can be shown that for the used edge lengths (5.8) the relaxed optimality condition R1.1 is always satisfied. By rerouting flow towards cheaper paths, according to the edge lengths, we try to satisfy also condition R2.2. In each iteration we choose an ϵ-bad commodity i and form an auxiliary min-cost flow problem. Given an optimal solution f_i^* to this auxiliary problem we reroute $\sigma = \epsilon/8\alpha\lambda$ fraction of the flow f_i towards the paths of f_i^* by setting $f_i = (1-\sigma)f_i + \sigma f_i^*$. We repeat until either condition R2.2 is satisfied or the flow f has a maximum congestion $\lambda \leq \lambda_0/2$. If R2.2 is satisfied then we terminate. If $\lambda \leq \lambda_0/2$, then we reset $\lambda_0 = \lambda_0/2$ and continue.

Leighton et al. [55] give two algorithms, one deterministic and one randomized.

The deterministic algorithm runs in $O(k^2(\log k \log n + \epsilon^{-2} \log \frac{n}{\epsilon}))$ minimum cost flow computations. To find an ϵ-bad commodity deterministically, all they need to do is to compute the costs $\sum_{e\in\mathcal{E}} f_i(e)l(e)$ for each commodity i and compare them to the minimum cost flows for these commodities. This way they need to check at most k commodities and thus an iteration can be implemented by computing at most k minimum cost flows. For the randomized algorithm they use an approach similar to Klein et al. [50]. To find an ϵ-bad commodity, they compute the cost C_i for each commodity i and then choose a commodity randomly with the probability proportional to its cost. They expect to find an ϵ-bad commodity within ϵ^{-1} trials, thus the expected number of minimum cost flow calculations is ϵ^{-1}. The expected time for the termination of the randomized algorithm is $O(k(\log k \log n + \epsilon^{-3} \log \frac{n}{\epsilon}))$ minimum cost flow computations.

5.1.4 Goldberg's Proposal

Goldberg [33] improved the running time of Leighton et al. [55] by a factor of ϵ^{-1}, following the same framework as in [55] with the only difference being the way a commodity is chosen to reroute flow. Goldberg [33] removes the notion of an ϵ-bad commodity and introduces a new relaxed optimality condition in the place of R2.2 which together with R2.1 imply an ϵ-approximate solution. The new condition is:

R3.2 $(1-2\epsilon)\sum_i C_i \leq \sum_{i=1}^{k} C_i^*$.

This way instead of trying to find an ϵ-bad commodity by randomly choosing commodities he introduces a more natural randomization strategy. A commodity is chosen uniformly at random and it updates its flow only if this gives progress towards satisfying condition R3.2. It is proved that we can expect significant progress even with this simpler method and that the running time decreases by an ϵ^{-1} factor to $O(k(\log k \log n + \epsilon^{-2} \log \frac{n}{\epsilon}))$.

5.1.5 Radzik's Proposal

Radzik [65] improved the running time of the deterministic algorithm proposed by Leighton et al. [55] by a factor of k so that it matched the running time of the best known randomized algorithm for the MCF problem. In his paper Radzik [65] finds a way to essentially replace the random selection of the commodities by a deterministic round-robin. In each iteration all commodities are examined one by one. The cost of their flow is compared to the minimum-cost flow and if a commodity is "bad" it reroutes some flow towards the minimum-cost flow computed. More specifically a commodity reroutes a fraction of its flow if the difference between its current cost C_i and its minimum cost C^* is greater than an ϵ factor of its current cost, i.e. it reroutes if $C_i - C_i^* > \epsilon C_i$. A commodity is therefore considered to be ϵ-good if the cost of its current flow is close to its minimum cost flow. Essentially Radzik [65] removes the additional condition used in Leighton et al. [55], where a commodity is also ϵ-good if its contribution to the total cost is small. The algorithm is proved to run in $O(k \lg n(\lg k + \epsilon^{-2}))$ minimum cost flow computations.

The table below summarizes the running times of the algorithms proposed in the literature for solving the MCF problem based on the rerouting framework.

Table 5.1: Comparison of Running times of Rerouting Algorithms

Author	**Problem**	**Deterministic Running time**	**Randomized Running Time**
Shahrokhi and Matula [70]	UC-MCF	-	$O(\epsilon^{-5}nm^7)$
Klein et al. [50]	UC-MCF	$\widetilde{O}((\epsilon^{-3}km^2)$	$\widetilde{O}(\epsilon^{-4}m^2)$
Leighton et al. [55]	MCF	$\widetilde{O}(\epsilon^{-2}k^2nm)$	$\widetilde{O}(\epsilon^{-3}knm)$
Goldberg [33]	MCF	$\widetilde{O}(\epsilon^{-1}k^2nm)$	$\widetilde{O}(\epsilon^{-2}knm)$
Radzik [65]	MCF	$\widetilde{O}(\epsilon^{-2}knm)$	-

We assume that m is greater than n and k. $\widetilde{O}$ hides polylogarithmic factors.

5.1.6 A Rerouting Example

To illustrate the rerouting method consider the network in Figure 5.1 with two commodities. The capacities of the edges and the source-sink pair $s_i - t_i$ for each

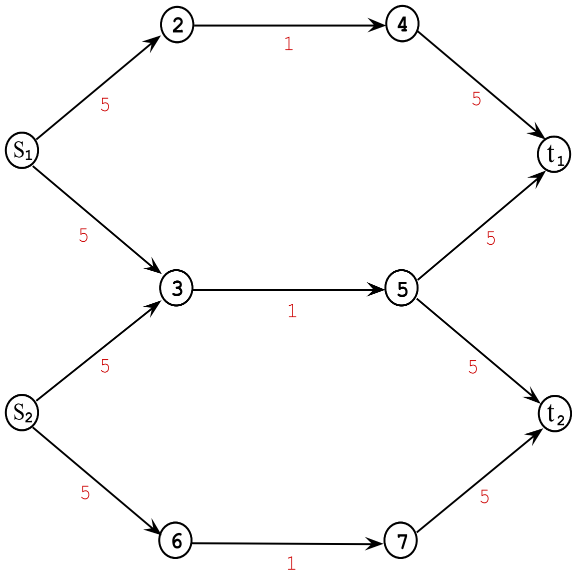
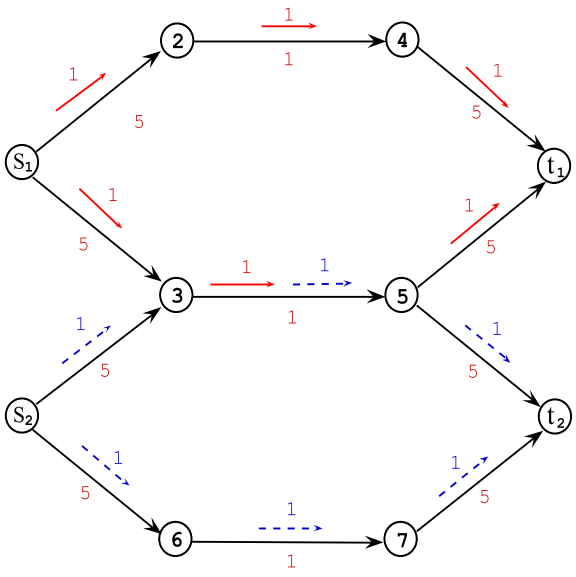
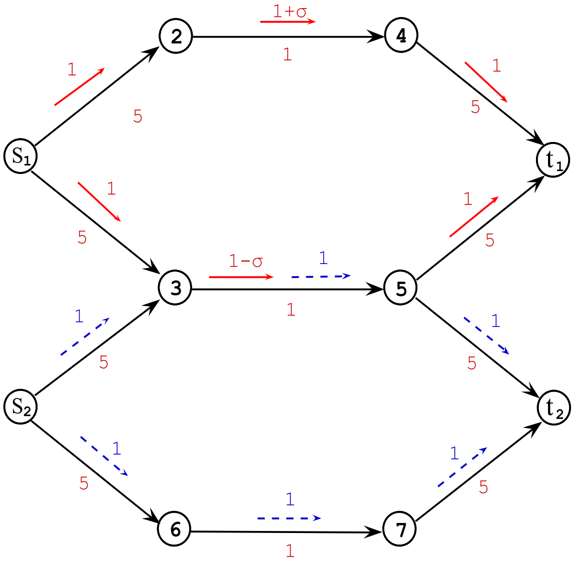


Figure 5.1: Network with Two Commodities



commodity	demand	flow symbol
1	2	→
2	2	⇢

Figure 5.2: Maximum Flow for each Commodity Routed Independently



commodity	demand	flow symbol
1	2	→
2	2	⇢

Figure 5.3: One Iteration of the Rerouting Algorithm

commodity i can be observed in this figure. Each commodity i has a demand $d_i = 2$. The rerouting method starts with each commodity computing and routing independently a maximum flow. In the network example in Figure 5.1 it is easy to verify that the value of the maximum flow for each commodity is two. The exact flow in the network for each commodity can be observed in Figure 5.2. This flow has a maximum congestion of $\lambda = 2$. In this example it is easy to verify that the optimal congestion is $\lambda^* = 4/3$. This can be achieved when each commodity sends 4/3 amount of flow on their independent path - the paths that the commodities have restricted access ($s_1 - 2 - 4 - t_1$ and $s_2 - 6 - 7 - t_2$ for commodities 1 and 2 respectively) and 2/3 of flow on the respective paths with the shared edge ($s_1 - 3 - 5 - t_1$ for commodity 1 and $s_2 - 3 - 5 - t_2$ for commodity 2). For this example assume that we set the approximation parameter to be $\epsilon = 1/3$.

After the initial routing we check whether the relaxed optimality conditions are met (see subsection 5.1.3). We can observe that the current solution is not ϵ-approximate. In one iteration of the rerouting algorithm then an ϵ-bad commodity is chosen and a small fraction σ is rerouted towards the minimum cost flow paths. In our example commodity 1 reroutes a σ fraction from the shared edge path ($s_1 - 3 - 5 - t_1$) to the independent path (see Figure 5.3). The procedure terminates when the relaxed optimality conditions are met.

5.2 The Incremental Method

5.2.1 Young's Proposal

Departing from the line of research which focused on rerouting methods for the Maximum Multicommodity Flow problem, Young [82] proposed a new technique to solve the MMF problem which builds the flow from scratch. Young [82] proposes an algorithm in the general context of packing and covering problems which applies directly to the multicommodity flow problem. A solution for the Maximum Multicommodity Flow problem can be viewed as a collection of flow paths $\mathcal{P}$ "packed" into the network. Each path P is from s_i to t_i for some commodity i and carries some flow $f(P)$. Moreover, for each edge $e \in \mathcal{E}$, the total flow must be within the capacity of this edge that is, $\sum_{P \in \mathcal{P}} f(P) \leq c(e)$. His technique, called oblivious rounding, deviates significantly from randomized rounding which involves solving first the relaxed linear program and then apply randomized rounding to find approximate solutions. In terms of the maximum multicommodity flow problem, the algorithm chooses a path that has short length (the length of a path is measured using a length function associated with the edge flows) and routes a unit of flow along this path, updating the edge lengths afterwards. This process continues for a number of iterations and the resulted accumulated flows satisfy cd_i demands for some constant $c > 1$. The algorithm builds the flow from scratch and only scales it down in the end so that it satisfies the problem constraints.

5.2.2 Garg and Koenemann's proposal

Young's method was later further developed by Garg and Koenemann [30] who provide a simpler and faster algorithmic framework for the MMF-problem based on incremental routing. Their algorithm is much faster because, unlike Young's method which pushes one unit of flow to shortest paths in each iteration, their method allows pushing as much flow as the minimum capacity of the shortest path, thus saturating it. This way their improvement in each iteration is much larger than Young [82] and thus fewer iterations are needed for the algorithm to terminate. Garg and Koenemann [30] also adapt this framework to the Maximum Concurrent Flow problem. We are going to describe their algorithm below since it provides the core model for the incremental framework for the MCF-problem. We are going to discuss future improvements below.

Garg and Koenemann [30] propose an algorithm which is based on the dual of the path-based formulation **(D2)** given in Section 3.3.2. Informally, the algorithm proceeds in phases, each one consisting of k iterations, one for each commodity. In each phase each commodity i transports d_i units of flow from its source to its destination. A length function, which is based on the current flow, is maintained for each edge. Flow is sent along shortest paths with respect to this length function. Each iteration i consists of steps, and in each step the current commodity i tries to push as much flow as possible from its source s_i to its destination t_i by saturating the shortest path from s_i to t_i calculated based on the current length function. Upon termination of the algorithm the final flow is scaled down by the number of phases so that it becomes feasible.

Formally the quantity we want to minimize is (see the dual problem **(D2)** in Section 3.3.2)

$$D(l) = \sum_{e \in \mathcal{E}} c(e) l(e).$$

The flow is set to $f(P) \equiv 0$ and the length function for each edge e to $l(e) = \zeta / c(e)$ for an appropriate parameter ζ derived from the analysis of the algorithm. Initially the dual variables z_i for each commodity are set to be the shortest path distance from s_i to t_i under the current length function, i.e. $z_i = min_{P \in P_i} l(P)$. The algorithm proceeds in phases, with each phase partitioned into k iterations. In iteration i the objective is to send d_i units of commodity i from source s_i to the destination t_i. This is done in steps. In phase p, iteration i, at the beginning of step s (that is, at the end of step $s-1$) we have the following values:

d_i^{s-1} is the demand remaining to be routed ($d_i^0 = d_i$)

$\Delta f_{p,i}^s(e)$ is the flow sent on edge e at step s of iteration i, phase p ($\Delta f_{1,i}^0(e) = 0$)

$l_{p,i}^s(e)$ is the length of edge e at step s of iteration i, phase p ($l_{1,0}^0(e) = \delta$)

In the current step the algorithm computes the shortest path P_i^{SP} from s_i to t_i based on the current length function. Then the capacity $c = \min\{c(e) : e \in P_i^{SP}\}$ of the path is calculated and the minimum of the remaining demand and this capacity is sent along the path, that is, we send

$$u^s = \min\{c, d_i^s\}.$$

At the end of the step the length and the remaining demand are updated as follows

$$l_{p,i}^s(e) = l_{p,i}^{s-1}(e)(1 + \epsilon \Delta f_{p,i}^s(e)/c(e)),$$

$$d_i^s = d_i^{s-1} - u^{s-1}.$$

The dual variable z_j is set to the shortest path under the updated length function. The algorithm proceeds until the objective value is at least one, that is $D(l) \geq 1$, in which case it is proven that the resulting (scaled) flow is ϵ-approximate.

Garg and Koenemann [30] introduce $a(l) = \sum_i d_i dist_l(s_i, t_i)$ where $dist_l(s_i, t_i)$ denotes the shortest path distance from s_i to t_i under the current length function. Then the problem formulation **(D2)** in Section 3.3.2 can be written as

$$minimize \;\; \beta := D(l)/a(l)$$

subject to

$$\sum_{e \in P} l(e) \geq dist_l(s_i, t_i), \quad \forall i, \forall P \in P_i, i = 1, 2, \ldots, k,$$

$$l(e) \geq 0, \forall e \in \mathcal{E}.$$

The problem can now be viewed as a problem of finding assignments for the lengths $l(e)$ such that β is minimized. Garg and Koenemann [30] provide an analysis for the case when $\beta \geq 1$ and show that the algorithm terminates in $\frac{\beta}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon}$ phases. To bound the number of phases they propose a two stage method to estimate β. First they find suitable lower and upper bounds for β and then they show how to reduce the number of phases if the value of β is large. Their analysis depends crucially on β to be greater than one. Garg and Koenemann [30] show how to handle the case when the optimal value $\beta < 1$ with an appropriate scaling technique which can be explained in the following way. Let f_i^{max} be the maximum flow of commodity i if routed independently and let

$$|f| = \min \frac{f_i^{max}}{d_i}.$$

Since at best all commodities can be routed simultaneously using single commodity maximum flows, $|f|$ is an upper bound on the value of the optimal solution. We remind the reader that we use the version of MCF problem formulation which maximizes the proportion of the demands. Note also that by sending $1/k$ fraction of f_i^{max} for each commodity i we get a feasible solution. Thus $|f|/k$ is a lower bound on the optimal solution, that is,

$$\frac{|f|}{k} \leq \beta \leq 1.$$

Scaling the demands by $\frac{|f|}{k}$ ensures that $\beta \geq 1$. But now β can be as large as k. To deal with this Garg and Koenemann [30] have to take extra steps throughout the algorithm. First they start the algorithm and proceed for $T = 2\frac{1}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon}$ phases. If the algorithm does not terminate then $\beta \geq 2$. Thus, double the demands by 2, causing β to drop to half, and proceed again for T more phases. This way we ensure that the algorithm will terminate in $T \log k$ such phases. The overall running time of the algorithm is $\widetilde{O}(\epsilon^{-2}m(m+k) + T_{MaxFlow})$, where $T_{MaxFlow}$ is the time to compute a maximum flow for a single commodity and the current best running time is $\widetilde{O}(nm)$ due to [62].

5.2.3 Fleischer's Proposal

Fleischer [25] improve the running time of the algorithm proposed by Garg and

Koenemann [30] using a different technique to find an upper bound on β. Instead of calculating k maximum flows at the beginning notice that it is sufficient to calculate an $O(m)$ approximation to the value of f_i^{max} since it is only used to calculate an initial estimate of β. Thus, instead of finding the exact value of f_i^{max} we can look for a value $\widehat{f}_i^{max} > \frac{1}{m} f_i^{max}$. Of course now β is bounded by,

$$\frac{|f|}{km} \leq \beta \leq 1.$$

Therefore, if we scale the flows by $\frac{|f|}{k}$ the value of β can be as large as km. However, using a technique similar to the one in [30], the algorithm will terminate in $T \log(km)$ phases. But now, the time to compute an approximate maximum flow for each commodity is significantly less than the time needed to compute a maximum flow for this commodity. Since any flow can be decomposed into at most m paths this value can be computed simply by sending flow across a maximum capacity path. Such a path can be found in $O(m \log m)$ time, for example using binary search over the range of capacities. Thus, the time needed to find a good upper bound approximation for β is now $O(Tm \log m \log(km))$. The running time of [30] is no longer dominated by the time needed to compute k maximum flows as in [30] and thus it is reduced to $\widetilde{O}(\epsilon^{-2}(m^2 + km))$.

Subsequently Karakostas [42] managed to improve the running time for the Maximum Concurrent Flow problem to $\widetilde{O}(\epsilon^{-2}m^2)$. To do this, instead of considering commodities one by one he considers their sources. Note that the number of sources could be less than the number of commodities since some of them could use the same source. Hence the number of iterations of the algorithm now is k', where k' denotes the number of different sources. This is a crucial improvement since now all the shortest paths of commodities sharing the same source can be calculated by one call of Dijkstra's algorithm. To handle the contention between the commodities sharing the same source, in each step instead of saturating the shortest path for each commodity (as previous algorithms) the flows of the commodities are scaled so that the sum of the flows do not exceed the capacity of any edge of the shortest paths calculated for these commodities. We formalize this below.

Consider the set of commodities Ω sharing the same source s_q. In phase p, iteration q at the beginning of step s we have:

d_i^s is the demand remaining to be routed ($d_i^0 = d_i$), $i \in \mathcal{Q}$

$\Delta f_{p,q,i}^s(e)$ is the flow sent on edge e at step s of phase p for commodity $i \in \mathcal{Q}$ ($\Delta f_{1,q,i}^0(e) = 0$)

$l_{p,q}^s(e)$ is the length of edge e ($l_{1,0}^0(e) = \delta$).

The algorithm calculates the shortest path P_i^{SP} for each commodity $i \in \mathcal{Q}$ and sets the amount of flow to be sent to $f_{p,q,i}^s = d_i^s$. Then it finds how much flow can actually be sent in this step by calculating the level of violation σ of the capacities in paths $P_i^{SP}, i \in \mathcal{Q}$

$$\sigma = \max\left\{1, \max_{e \in P_i : i \in \mathcal{Q}} \frac{\sum_{i : e \in P_i^{SP}} \Delta f_{p,q,i}^s}{c(e)}\right\},$$

and scales the flows to be sent to $f_{p,q,i}^s/\sigma$. This fraction of the remaining demand of each commodity is sent at this step. The remaining demands, edge flows and edge lengths are updated at the end of the step. The iteration terminates when the remaining demand of all commodities sharing the same source is zero.

The above scheme does not keep track of the edge flows of individual commodities. It maintains only the total edge flows and it is a fast way to compute an ϵ-approximate value for the congestion. In many applications we are interested in the exact flow of each commodity on each edge. This case is referred to by Karakostas [42] as the implicit representation of flows and he shows how it can be calculated by a more careful distribution of flows along shortest paths in each step. Instead of considering all the commodities that share the same source together in each step and routing the scaled flow of each one appropriately he considers the commodities one by one within the step. Thus the algorithm now is similar to the one described in [30] with the difference being that commodities that share the same source are considered together when calling Dijkstra's algorithm to compute a shortest path tree. The total running time of Karakostas' [42] implicit algorithm is $\widetilde{O}(\epsilon^{-2}(m^2 + kn))$.

5.2.4 Madry's Proposal

Recently, Madry [57] managed to provide an even faster approximation scheme for the Maximum Concurrent Flow problem. To do this he introduces ideas from dynamic graph algorithms. He observed that the shortest path subproblems that previous algorithms repeatedly solved are closely related, that is, the underlying graph is the same with only some of the edge lengths changing. Hence it is suboptimal to treat these subproblems as independent in each step and calculate a shortest path from scratch. By maintaining a suitable data structure this problem could be circumvented. Indeed, using ideas from the decremental all-pair shortest path problem (DAPSP) he manages to reduce the running time of previous algorithms. The DAPSP problem is the problem of maintaining shortest paths among all pairs on nodes in a graph that deletion of edges can occur. The data structure proposed maintains the length of the shortest path under the current length function and the set of paths with length within an $(1+\epsilon)$ factor of the shortest path. It also supports fast operations of retrieving the paths and their lengths.

The problem in this setting was that the data structures used for the decremental all-pair shortest path problem so far could not provide the appropriate bounds for the Multicommodity Flow Problems. Madry [57] shows how to modify the data structure so that it fits to the MCF problem. He introduces a subset $\widehat{\mathbf{P}}$ of paths in $\mathcal{P}$. Remember that $\mathcal{P}$ is the set of all paths in graph $\mathcal{G}$. The paths in the subset $\widehat{\mathbf{P}}$ are chosen in the following way. For $j = 1, 2, \ldots, \log n$ random sets S_j are obtained by sampling each node in $\mathcal{N}$ with probability $p_j = min\{\frac{10 \ln n}{2^j}, 1\}$. Then $P(S_j, 2^j)$ is the set of all paths in $\mathcal{P}$ which pass through at least one node in S_j and have exactly 2^j edges. The subset $\widehat{\mathbf{P}}$ is the union of all the sets $P(S_j, 2^j)$. This random set of paths $\widehat{\mathbf{P}}$, which can be seen as a "sparsification" of the set of all paths in $\mathcal{G}$, is proved in [57] that with high probability it contains all the set of paths of a given concurrent flow. As a consequence of this fact Madry [57] proves that such a structure contains approximate shortest paths with a high probability and thus can be used in the data structure maintained. Essentially Madry [57] transforms the framework provided by Garg and Koenemann [30] and Fleischer [25] for solving the MCF problem to a Monte-Carlo algorithm. The data structure maintained is used to find approximate shortest paths replacing Dijkstra's algorithm and it is updated each time flow is augmented. Madry [57] shows that maintaining this data structure does not cost too much time. Under this setting the algorithm terminates in expected $\widetilde{O}(\epsilon^{-2}(m + k)n \log M)$ time, where M denotes the upper bound on the size of binary representation of any number used in the input instance.

Table 5.2 summarizes the running times of the algorithms proposed in the literature for solving the MCF problem.

Table 5.2: Comparison of Running times of Incremental Algorithms for the MCF problem

Author	**Running time**
Garg and Koenemann [30]	$\widetilde{O}(\epsilon^{-2}m^2 + knm^2)$
Fleischer [25]	$\widetilde{O}(\epsilon^{-2}(m^2 + km))$
Karakostas [42]	$\widetilde{O}(\epsilon^{-2}m^2)$, $\widetilde{O}(\epsilon^{-2}(m^2 + kn))$
Madry [57]	$\widetilde{O}(\epsilon^{-2}(m + k)n)$

5.2.5 An Incremental Example

To illustrate the incremental method consider the network in Figure 5.4 with two commodities. The capacities of the edges and the source-sink pair $s_i - t_i$ for each commodity i can be observed in this figure. Each commodity i has a demand $d_i = 4$. One phase of the incremental method is split into $k = 2$ iterations. In each iteration each of the k commodities incrementally sends its demand through the network. The flow is sent in steps. In each step a shortest path is calculated under the current length function. Then the minimum of the capacity of the path and the remaining demand to be sent for the commodity is routed through the path. In this example assume that the approximation parameter ϵ is set to $\epsilon = 1/3$.

In the network example in Figure 5.5 we can observe the flow after one phase. The resulting flow was routed in the following way. Commodity 1 starts the computation. It computes a shortest path and sends the maximum among its

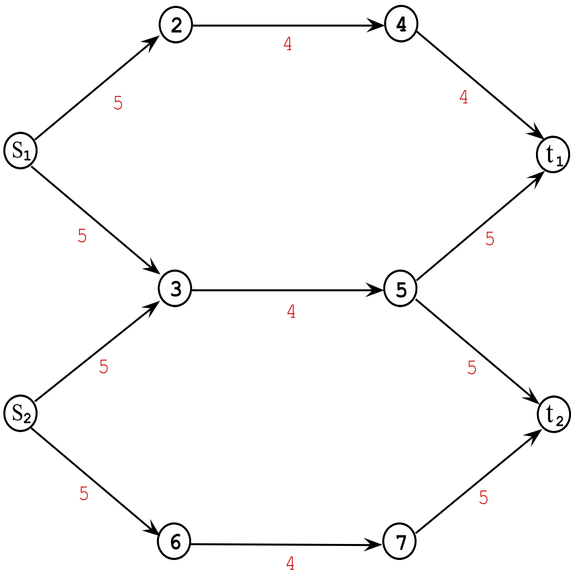


Figure 5.4: Two Commodities Network

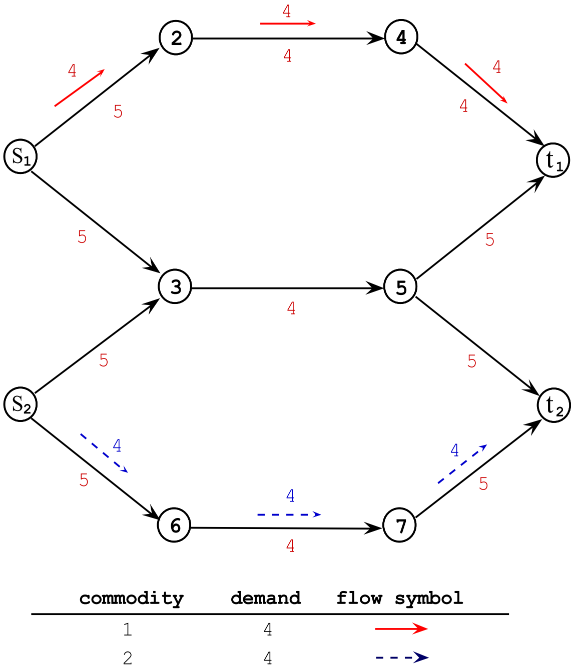


Figure 5.5: First Phase of Incremental Algorithm

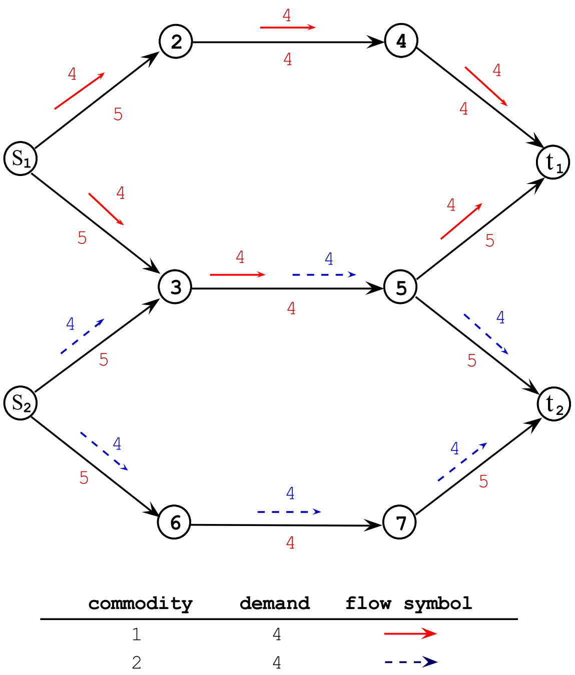


Figure 5.6: Second Phase of Incremental Algorithm

demand and the capacity of the computed shortest path. In the first iteration both available paths for commodity 1 have the same length. Assume that the commodity chooses path $s_1 - 2 - 4 - t_1$. Then the whole commodity can be routed in one step. The algorithm then proceeds to the next commodity. For commodity 2 again two paths are both shortest paths under the current length function. Then, assuming that the commodity picks the path $s_2 - 6 - 7 - t_2$ to send its flow, the whole demand can be sent in one step. The first phase then terminates.

Before the next phase starts the stopping criteria are checked. In this example the computed flow is still not ϵ-approximate since the optimal congestion is 2/3 (Flow of each commodity must be split in the ratio 2/3 : 1/3 among the available paths). Commodity 1 then will again compute the shortest path. In the second phase the shortest path is obviously $s_1 - 3 - 5 - t_1$ since it has no flow yet. The whole commodity can be routed in one step. Commodity 2 then can pick any of the two available paths since both have the same length (both paths had one length update sending four units of flow). Assuming that commodity 2 chooses

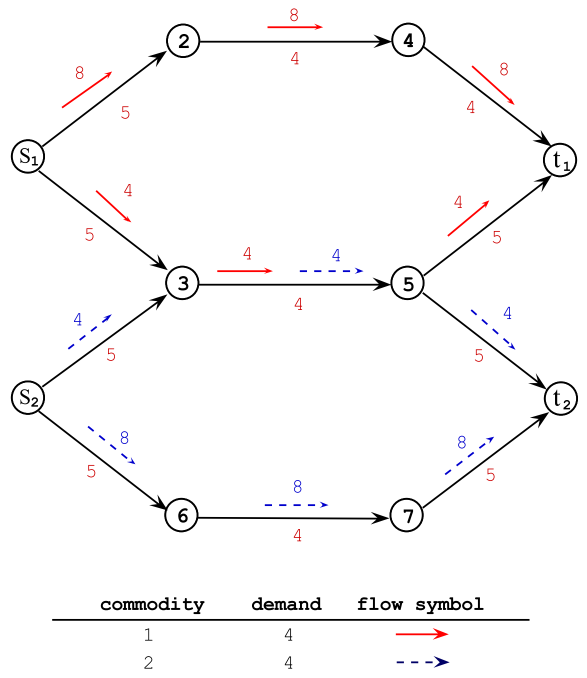


Figure 5.7: Termination of Incremental Algorithm

path $s_2 - 3 - 5 - t_2$ the resulting flow is given in Figure 5.6. For the given example the algorithm will terminate in one more phase. Both commodities in the next phase will choose the independent path and route their whole demand (the independent path is the shortest path in this phase). Then the resulting flow can be observed in Figure 5.7. This flow has to be scaled to meet the demands (this is the last iteration of the algorithm when the computed flows meet the stopping criteria). Observe that if we scale the current flow by three the resulting flow is indeed ϵ-approximate.

5.3 Summary

In this chapter we have presented the two main methods of solving the approximate maximum concurrent flow problem, the rerouting method and the incremental method. We have demonstrated how the approximation algorithms under each of the two methods are linked together. The two methods outlined in this chapter have traditionally been perceived as distinct. In the next two chapters we show that the two methods can be modified so that each one can be viewed as an instance of the other.

Although it is tempting to use the ideas based on these approximation algorithms to solve the maximum concurrent flow problem exactly unfortunately that does not lead to polynomial algorithms for the exact case. To get an exact solution of the MCF problem based on one of these algorithms we have to keep scaling down the parameter ϵ until we are very close to the optimal solution. Thus, parameter ϵ has to be exponentially small to get an exact optimal solution. However, since the running time of the algorithms described in this section depends on ϵ^{-2}, the result would be an exponential running time for finding an exact solution.

Chapter 6

The Incremental Method with an Exponential Length Function

Contents

For the incremental method, both the flow and the length function are updated incrementally, according to related but somewhat separate processes. The new length of an edge is equal to the previous one plus a small fraction which depends on the amount of flow routed through this edge at the current iteration. This means that two edges with the same capacity and current flow might have different lengths because the current value of the length of an edge depends on the history of flow updates on this edge. On the other hand, in the rerouting method the length $l(e)$ of an edge e is always a direct function of the current flow: two edges with the same flow have always the same length. In terms of the flow the incremental method builds the flow from scratch. In each iteration it sends a flow of value equal to the demand of each commodity. Upon termination it scales the flow down (if needed) so that it does not exceed the capacities of the edges. The rerouting method on the other hand starts with a flow (usually a maximum flow routed independently for each commodity) and then tries to redistribute the flow.

In this chapter we show that we can employ in the incremental method an exponential length function similar to the rerouting algorithm, which is a direct function of the current flow. This modification essentially converts the incremental method into an instance of the rerouting method.

6.1 Exponential Length Function

As described in Section 5.2, the update of the length function in the original incremental method [30] at the end of step s, iteration i of phase p is given by the formula

$$l_{p,i}^{s}(e) = l_{p,i}^{s-1}(e)(1 + \epsilon \Delta f_{p,i}^{s}(e)/c(e)),$$

where $\Delta f_{p,i}^{s}(e)$ is the flow sent on edge e in this step, $c(e)$ is the capacity of edge e and ϵ is the approximation parameter of the method.

We change this length function to the following function:

$$l_{p,i}^{s}(e) = \frac{\delta}{c(e)} \exp(\epsilon f_{p,i}^{s}(e)/c(e)), \tag{6.1}$$

where δ is a constant that will be given later and $f_{p,i}^{s}(e)$ is the *total* flow on edge e at the end of this step, and $c(e)$ and ϵ are as above. The new edge length function we propose has a functional relation with the total flow $f(e)$ on that edge

$$l(e) = \frac{\delta}{c(e)} \exp(\epsilon f(e)/c(e)). \tag{6.2}$$

This length function is similar to the one used in the rerouting method up to some constants (see Section 5.1). The length of an edge directly depends on the flow on this edge and is monotonically increasing with the flow. The maximum value is achieved when $f(e) = c(e)$, i.e. when the edge is saturated. This means that the cost of an edge is high when it is close to saturation. The incremental algorithm with the exponential length function is given below as Algorithm 1. Recall that $D(l) = \sum_{e \in \mathcal{E}} c(e)l(e)$ and note that the only modification we have made is replacing the length function. Other than that the algorithm is exactly the same as proposed by Garg and Koenemann [30]. Our analysis follows the structure of their original analysis but changes had to be worked out to adopt the derivations to the new length function.

The algorithm proceeds in phases, each one consisting of k iterations, one for each commodity. In each phase each commodity i transports d_i units of flow from its source to its destination. A length function, which is based on the current flow (6.2), is maintained for each edge. Flow is sent along shortest paths with respect to this length function. Each iteration i consists of steps, and in each step the current commodity i tries to push as much flow as possible from its source s_i to its destination t_i by saturating the shortest path from s_i to t_i calculated based on the current length function. Upon termination of the algorithm the final flow is scaled down by the number of phases so that it becomes feasible.

Algorithm 1: Incremental algorithm using an exponential length function

```
Input: Network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, capacities $c(e)$, commodities $i$ with source $s_i$,
       destination $t_i$, demand $d_i$, $i = 1, 2, \dots, k$
Initialization: Set $f(e) = 0$ for each edge $e$;
                Set $l(e) = \delta/c(e)$ for each edge $e$;
while $D(l) < 1$ do
    for commodities $i = 1$ to $k$ do
        Set $d_i^{'} = d_i$;
        while $D(l) < 1$ and $d_i^{'} > 0$ do
            1. Compute the shortest path $P_i^{SP}$ from $s_i$ to $t_i$ under the
            current length function;
            2. Define $u = \min\{c(e) : e \in P_i^{SP}\}$;
            3. Route $\Delta = \min\{u, d_i^{'}\}$ units of flow of commodity $i$ along
            path $P_i^{SP}$:
                $f_i(e) = f_i(e) + \Delta$ for each $e \in P_i^{SP}$;
            4. Update $l(e) = (\delta/c(e)) \exp(\epsilon f(e)/c(e))$ and $d_i^{'} = d_i^{'} - \Delta$;
        end
    end
end
Termination: Scale down the flow of each commodity $i$ so that its value
is exactly $d_i$
```

In the next Section we prove the correctness of our algorithm. In Section 6.3 we prove its running time.

6.2 Correctness of the Algorithm

In what follows we show that Algorithm 1 returns a 5ϵ-approximate flow. One *phase* in Algorithm 1 is one iteration of the outer "while" loop. One *iteration* in one phase is an iteration of the "for" loop: one iteration for each commodity. One *step* is one iteration of the innermost loop. Let $l_{p,i}^s$ denote the length function l at the end of step s in iteration i of phase p, and let $l_{p,i}$ be the length function at the end of iteration i in phase p. Similarly let $f_{p,i}^s(e)$ denote the total flow on edge e at the end of step s in phase p, iteration i. Thus $D(l_{p,i}^s)$ is the value of $D(l)$ at the end of step s, iteration i, phase p, and we set

$$\alpha(l_{p,i}) := \sum_{i=1}^{k} d_i dist(s_i, t_i; l_{p,i}),$$

where $dist(u, v; l) \equiv dist_l(u, v)$, denotes the shortest path from a node u to a node v under the length function l. Let $\Delta f_{p,i}^s(e)$ denote the additional flow sent on edge e at step s in iteration i of phase p. We prove the correctness of our algorithm below.

Theorem 6. *Algorithm 1 returns a 5ϵ-approximate flow.*

Proof. As in Section 5.2, let

$$\beta := min_l D(l)/\alpha(l) \tag{6.3}$$

be the optimal dual objective value, where the minimization is taken over the length function l such that $l(e) > 0, \sum_{e\in\mathcal{E}} l(e) > 0$.

At the end of step s in iteration i of phase p using (6.1) we have

$$\begin{aligned}
D(l^s_{p,i}) &:= \sum_{e\in\mathcal{E}} c(e) l^s_{p,i}(e) = \sum_{e\in\mathcal{E}} c(e)\frac{\delta}{c(e)} e^{\epsilon f^s_{p,i}(e)/c(e)} \\
&= \sum_{e\in\mathcal{E}} c(e)\frac{\delta}{c(e)} e^{\epsilon(f^{s-1}_{p,i}(e)+\Delta f^s_{p,i}(e)/c(e))} \\
&= \sum_{e\in\mathcal{E}} c(e)\frac{\delta}{c(e)} e^{\epsilon f^{s-1}_{p,i}(e)/c(e)} e^{\epsilon\Delta f^s_{p,i}(e)/c(e)} \\
&\leq \sum_{e\in\mathcal{E}} c(e) l^{s-1}_{p,i}(e)\left(1+\epsilon\frac{\Delta f^s_{p,i}(e)}{c(e)} + \epsilon^2\left(\frac{\Delta f^s_{p,i}(e)}{c(e)}\right)^2\right) \\
&= \sum_{e\in\mathcal{E}} l^{s-1}_{p,i}(e)c(e) + \epsilon\sum_{e\in\mathcal{E}} l^{s-1}_{p,i}(e)\Delta f^s_{p,i}(e) + \epsilon^2\sum_{e\in\mathcal{E}} l^{s-1}_{p,i}(e)\Delta f^s_{p,i}(e)\left(\frac{\Delta f^s_{p,i}(e)}{c(e)}\right) \\
&\leq \sum_{e\in\mathcal{E}} l^{s-1}_{p,i}(e)c(e) + \epsilon(1+\epsilon)\sum_{e\in\mathcal{E}} l^{s-1}_{p,i}(e)\Delta f^s_{p,i}(e) \\
&= D(l^{s-1}_{p,i}) + \epsilon(1+\epsilon)\Delta^s_{p,i} dist(s_i, t_i; l^{s-1}_{p,i}).
\end{aligned}$$

The inequality in the fourth line comes from the fact that $e^x \leq 1 + x + x^2$ for $0 \leq x \leq 1/5$ and the conditions that $\Delta f^s_{p,i}(e) \leq c(e)$ and assuming $\epsilon < 1/5$. The inequality in the sixth line follows again from the fact that the flow sent cannot exceed the edge capacities. The last equality follows from the fact that $\Delta f^s_{p,i}(e) = \Delta^s_{p,i}(e)$ on the edges of the shortest path (and zero on the rest of the edges) at each step. The variable $\Delta^s_{p,i}$ is the minimum between the capacity of the shortest path calculated and the remaining demand at this step (see Algorithm 1).

Summing up over all steps of iteration i in phase p we get (observe that $l^s_{p,i}(e) \leq l_{p,i}(e)$, because the edge lengths can only increase, and $\sum_s \Delta^s_{p,i} = d_i$)

$$D(l_{p,i}) \leq D(l_{p,i-1}) + \epsilon(1+\epsilon) d_i dist(s_i, t_i; l_{p,i}). \tag{6.4}$$

For simplicity of further derivations, let $D(p) = D(l_{p,k}) = D(l_{p+1,0})$ and $\alpha(p) = \alpha(l_{p,k}) = \sum_{i=1}^k d_i dist(s_i, t_i; l_{p+1,0})$, that is, $D(p)$ and $\alpha(p)$ are the values $D(l)$ and $\alpha(l)$ at the end of phase p. Inequality (6.4) summed up over all iterations in phase

p gives (observe that $l_{p,i}(e) \leq l_{p+1,0}(e)$)

$$\begin{aligned} D(p) &\leq D(p-1) + \epsilon(1+\epsilon)\sum_{i=1}^{k} d_i dist_l(s_i, t_i; l_{p,i}) \\ &\leq D(p-1) + \epsilon(1+\epsilon)\alpha(p). \end{aligned}$$

Now, since $D(p)/\alpha(p) \geq \beta$ (so $\alpha(p) \leq D(p)/\beta$), we have

$$D(p) \leq \frac{D(p-1)}{1-\epsilon(1+\epsilon)/\beta}. \tag{6.5}$$

We analyze now the change of the parameter $D(l)$ when the computation progresses over the phases. Since $D(0) = m\delta$, using (6.5) we have

$$\begin{aligned} D(p) &\leq \frac{m\delta}{(1-\epsilon(1+\epsilon)/\beta)^p} \\ &= \frac{m\delta}{1-\epsilon(1+\epsilon)/\beta}\left(1+\frac{\epsilon(1+\epsilon)}{\beta-\epsilon(1+\epsilon)}\right)^{p-1} \\ &\leq \frac{m\delta}{1-\epsilon(1+\epsilon)/\beta} exp\left\{\frac{\epsilon(1+\epsilon)(p-1)}{\beta(1-\epsilon(1+\epsilon)/\beta)}\right\} \\ &\leq \frac{m\delta}{1-\epsilon(1+\epsilon)} exp\left\{\frac{\epsilon(1+\epsilon)(p-1)}{\beta(1-\epsilon(1+\epsilon))}\right\}, \end{aligned} \tag{6.6}$$

where in the second inequality we have used the fact that $(1+x) \leq e^x$ for every x and in the last inequality we have used the assumption that $\beta \geq 1$.

The algorithm terminates at the first phase when $D(p) \geq 1$. Thus using (6.6) we conclude that if the algorithm terminates at phase p, we have

$$1 \leq D(p) \leq \tfrac{m\delta}{1-\epsilon(1+\epsilon)} exp\left\{\tfrac{\epsilon(1+\epsilon)(p-1)}{\beta(1-\epsilon(1+\epsilon))}\right\},$$

which implies

$$\frac{\beta}{p-1} \leq \frac{\epsilon(1+\epsilon)}{(1-\epsilon(1+\epsilon))\ln\left((1-\epsilon(1+\epsilon))/m\delta\right)}. \tag{6.7}$$

In the first $(p-1)$ phases we have routed $(p-1)d_i$ units of flow of each commodity i. Now we need to show by how much we need to scale the flow down to get a feasible flow (the flow within the edge capacities).

Claim 2. *To obtain a feasible flow at the end of the computation of Algorithm 1 the computed flow is scaled down by a factor at most $\epsilon^{-1}\ln(1/\delta)$.*

Proof. Consider an edge e. For every $c(e)$ units of flow sent along edge e we increase its length by a factor of e^{ϵ} (see (6.1)). Initially the length $l(e) = \delta/c(e)$ for each edge e. Now we know that after $(p-1)$ phases $D(p-1) < 1$ and thus, $l_{p-1,0}(e) < 1/c(e)$. Therefore an edge e cannot be saturated more than $\epsilon^{-1}\ln(1/\delta)$ times because otherwise the length of edge e would raise to at least:

$$\frac{\delta}{c(e)} e^{\epsilon(\ln(1/\delta))/\epsilon} = \frac{1}{c(e)}.$$

Hence the amount of flow through edge e when the main loop of the algorithm terminates is less than $\epsilon^{-1}\ln(1/\delta)$ times its capacity. □

A simple consequence of this result is that if the computation ends in phase p, then the throughput is

$$\gamma \geq \epsilon(p-1)/\ln(1/\delta). \tag{6.8}$$

This follows from the fact that at least $d_i(p-1)$ units of flow of each commodity is sent at the end of the computation and the final flow is scaled down by at most $\epsilon^{-1}\ln(1/\delta)$.

Therefore the ratio of the dual to the primal solution from (6.7) and (6.8) is,

$$\frac{\beta}{\gamma} \leq \frac{(1+\epsilon)\ln(1/\delta)}{(1-\epsilon(1+\epsilon))\ln\frac{1-\epsilon(1+\epsilon)}{m\delta}}.$$

Setting $\delta = \left(\frac{m}{1-\epsilon(1+\epsilon)}\right)^{-1/\epsilon}$ we get

$$\begin{aligned}\frac{\beta}{\gamma} \leq& \frac{(1+\epsilon)\ln(1/\delta)}{(1-\epsilon(1+\epsilon))\ln\left(\frac{1-\epsilon(1+\epsilon)}{m\delta}\right)} \\ =& \frac{(1+\epsilon)}{(1-\epsilon(1+\epsilon))} \frac{\ln\left(\frac{m}{1-\epsilon(1+\epsilon)}\right)^{1/\epsilon}}{\ln\left(\frac{m}{1-\epsilon(1+\epsilon)}\right)^{-1+1/\epsilon}} \\ =& \frac{(1+\epsilon)}{(1-\epsilon(1+\epsilon))(1-\epsilon)}\end{aligned}$$

Assuming that $\epsilon < 1/5$, we get $\beta/\gamma \leq (1+5\epsilon)$, so the final flow is (5ϵ)-approximate.

□

6.3 Running Time

In this section we prove the running time of Algorithm 1. We state this result in the following theorem.

Theorem 7. *The total running time of Algorithm 1 is* $\widetilde{O}(\epsilon^{-2}(m^2 + km))$.

Proof. The running time of Algorithm 1 is proved by bounding the running time of the three main loops. The outer loop runs in p phases. Using the weak-duality theorem and (6.8) we have

$$1 \leq \frac{\beta}{\gamma} \leq \frac{\beta}{p-1}\frac{\ln(1/\delta)}{\epsilon}.$$

Therefore the number of phases is $p = O(\beta\epsilon^{-1}\ln(1/\delta))$. Using a technique proposed by Garg and Koenemann [30] and subsequently improved by Fleischer [25] we can find an upper bound for the value of β and use scaling. Using this technique, the number of scaling stages is $O(\log(km))$ and each stage has $O(\epsilon^{-1}\ln(1/\delta))$ phases (see Section 5.2 for details). Thus the total number of phases is $O(\epsilon^{-1}\ln(1/\delta)\log(km))$.

Each iteration consists of a number of steps each one involving a shortest path calculation. We need to bound the number of steps to get the overall running time of our algorithm. Recall that initially there is no flow in the network. Thus the length of each edge e initially is given by

$$l(e) = \frac{\delta}{c(e)} \exp(\epsilon f(e)/c(e)) = \frac{\delta}{c(e)}.$$

In each iteration except the last one, the length of some edge (the bottleneck edge of the path chosen to augment flow) increases by a factor of e^{ϵ}. Since each length has a value of $\delta/c(e)$ initially and, it is at most $1/c(e)$ at the final step (recall that $D(l) < 1$ just before the final step), the total number of steps r when a given edge e gives the bottleneck capacity satisfies the following inequality

$$\begin{aligned}
&\frac{\delta}{c(e)} e^{r\epsilon} \leq \frac{1}{c(e)} \\
&\Rightarrow \delta e^{r\epsilon} \leq 1 \\
&\quad \Rightarrow r\epsilon \leq \ln\left(\frac{1}{\delta}\right) \\
&\quad \Rightarrow r \leq \epsilon^{-1} \ln\left(\frac{1}{\delta}\right) \\
&\quad \Rightarrow r \leq \epsilon^{-2} \ln\left(\frac{m}{1-\epsilon(1+\epsilon)}\right).
\end{aligned}$$

Since there are m edges in the network, we conclude that the total number of steps charged to the edges is at most $O(\epsilon^{-2} m \ln\left(\frac{m}{1-\epsilon(1+\epsilon)}\right))$. We charge a flow increase to the commodities when the whole remaining demand of a commodity has been sent. This happens exactly once in each phase so the total number of steps charged to the commodities is $O(k)$ per phase. Thus the total number of steps charged to the commodities is $O(k\epsilon^{-1} \ln(1/\delta) \log(km)) = O(k\epsilon^{-2} \log(m) \log(km))$ and the total number of steps of the algorithm is

$$O(k\epsilon^{-2} \log(m) \log(km) + \epsilon^{-2} m \log(m)) = O(\epsilon^{-2} \log(m)(k \log(km) + m)).$$

Each step involves the calculation of a single source shortest path. The time to compute such a path using Dikstra's algorithm is $O(m + n \log n)$. Thus the total running time of the algorithm is

$$O((\epsilon^{-2}\log(m)(k\log(km)+m))(m+n\log n)) = \widetilde{O}(\epsilon^{-2}(m^2+km)).$$

This terminates the proof of Theorem 7.

□

6.4 Summary

In this chapter we have introduced a new exponential length function for the Incremental Method of solving the MCF problem. This length function has a functional relation with the flow on an edge. This means that at any instance we can retrieve the flow of an edge using its length. Under this setting we can consider the incremental method to be an instance of the rerouting method. The first few iterations of the outer loop can be considered as the initialisation phase. Subsequently we "reroute" flow to less congested paths by pushing flow on the shortest paths. The flow "rerouted" at any subsequent iteration is a small fraction of the demand. At any point of time, using the functional length function, we can stop and check how close we are to the optimal solution by a simple scaling. In the next chapter we show how we can use the Successive Shortest Path to calculate the minimum cost flows in the rerouting method. Using this result we essentially show that the rerouting method can be converted to an instance of the Incremental Method with a functional length function.

Chapter 7

Rerouting based on Shortest Paths

Contents

We have shown in the previous chapter how we can use an exponential length function (a length function used previously in the Rerouting Framework) in the Incremental Framework. In this chapter we show how we can use shortest paths to calculate minimum cost flows in the Rerouting Framework. This modification allows us to show that the Rerouting Method can be considered as an instance of the Incremental Method. More specifically, we are going to use the Successive Shortest Path algorithm (SSP) to find a minimum cost flow. Previously suggested minimum cost flow algorithms for the MCF Rerouting Framework, though theoretically efficient, are not practical [85, 13]. Theoretically the most efficient algorithm is described in [35]. This algorithm uses a technique of successive approximations based on cost scaling and sophisticated data structures. We modify the MCF algorithm developed by Radzik [65] to use the Successive Shortest Path algorithm to calculate a minimum cost flow in each iteration.

More specifically, we prove that approximate min-cost flows computed by the Successive Shortest Path algorithm suffice to find an ϵ-approximate solution to the MCF problem. Then, we present how to construct such a solution by introducing a modified version of the original network. We show how we use the Successive Shortest Path (SSP) algorithm for computing the min-cost flows of the commodities in each step and prove the running time of our algorithm. Our work builds directly on the analysis done in [65], but modifications have been made in order to enable an ϵ-approximate min-cost flow computation instead of an exact one. In Section 7.1 we describe the SSP algorithm and discuss its similarities with the Incremental algorithm. In subsection 7.1.1 we show how to modify the original network so that the SSP algorithm runs in polynomial time. Finally, in Section 7.2 we show how we can modify the round-robin algorithm proposed by Radzik [65] so that we can use the SSP algorithm to calculate the minimum cost flow in each step.

7.1 The Successive Shortest Path Algorithm

The SSP algorithm is used to calculate the minimum cost flow in a network. Its execution is very similar to the Incremental Method used to solve the MCF problem. The algorithm starts with zero flow and proceeds in iterations. In each iteration the shortest path from the source node v with excess flow to the destination node u with deficit flow is computed. Then the maximum amount that can be sent through this path at the current iteration is calculated and sent along the path. At the end of the iteration the capacities of the edges are recalculated and the procedure continues until no augmenting path can be found in the residual network $\mathcal{G}_{\mathcal{R}}$(see Section 2.3). The result is a minimum cost flow from node v to node u. The reader is referred to [1] for more details.

We will give the pseudocode for the SSP algorithm but first we recall some definitions and introduce some new notation. Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed network with n nodes and m edges. Each edge $(u, v) \in \mathcal{E}$ has an associated capacity $c(u, v)$. We focus here on one of the k commodities with source s_i, sink t_i and demand d_i. Let $exc(v)$ denote the excess flow of a node v, that is,

$$exc(v) = b(v) + \sum_{u \in \mathcal{N}} f(u, v) - \sum_{u \in \mathcal{N}} f(v, u),$$

where

$$b(v) = \begin{cases} 0 & \forall\, v \in \mathcal{N}/\{s_i, t_i\} \\ d_i & \text{if } v = s_i \\ -d_i & \text{if } v = t_i, \end{cases}$$

and f is the current flow of this commodity. Let $l : \mathcal{E} \to \mathbb{R}_+$ be an edge length (cost) function. Let $\pi(v)$ denote the *potential* of a node v and $l^\pi(u,v) = l(u,v) - \pi(u) + \pi(v)$ denote the length of an edge (u,v) with respect to node potentials. The lengths $l^\pi(u,v)$ are also referred to as reduced costs. The pseudocode of the SSP algorithm is given below.

Algorithm 2: The Successive Shortest Path Algorithm

```
**Input**: Network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, source-sink pair $(s_i, t_i)$, demand $d_i$
**Output**: Minimum cost flow of commodity $i$
1. Initialize $\pi(v) = 0,\ \forall\, v \in \mathcal{N},\ f(u,v) = 0,\ \forall\, (u,v) \in \mathcal{E}$
2. $exc(v) = b(v),\ \forall\, v \in \mathcal{N}$
3. $l^\pi(u,v) = l(u,v)$
**while** $e(s_i) > 0$ **do**
    1. Calculate the shortest path $P_i^{SP}$ from $s_i$ to $t_i$ and its length
       $dist_l(s_i, t_i)$ with respect to reduced costs $l^\pi(u,v)$
    2. Update $\pi(v) \leftarrow \pi(v) - dist_l(s_i, v)$
    3. Calculate $\delta = \min\{e(s_i), \min\{c_R(u,v) : (u,v) \in P_i^{SP}\}\}$
    4. Augment $\delta$ units of flow along path $P_i^{SP}$
    5. Update $f$, $\mathcal{G}_{\mathcal{R}}$ and $l^\pi(u,v)$
**end**
```

To adopt the SSP algorithm to fit in Radzik's Rerouting Framework [65] we propose an appropriate modification of the original network. Notice in line 3 of the "while" loop of Algorithm 2 that the amount of flow sent in each step is bounded by the smallest capacity of the shortest path computed. If these capacities are too small, then the running time of the algorithm will be exponential. This motivates us to construct a network that will eliminate this possibility. We describe how we can achieve this in the following section.

7.1.1 Approximation Algorithm for Minimum Cost Flow

To compute approximate minimum cost flows in polynomial time using the Successive Shortest Path algorithm we need to modify the original network. Consider the network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with n nodes and m edges with capacities $c(e)$. Now consider finding a minimum-cost flow of a single commodity with demand d, source s and destination t, and let the network $\mathcal{G}^\delta = (\mathcal{N}, \mathcal{E}^\delta)$ be the original network $\mathcal{G}$ but with the capacities of all edges rounded down to the units of $\delta = \epsilon d/m$

$$c^\delta(e) = \lfloor \tfrac{c(e)}{\delta} \rfloor \delta.$$

We assume without loss of generality that the capacities c are sufficient to have a flow which satisfies the demand d. The reason for this change is that we want to use a simpler algorithm to compute the minimum-cost flow, that is the Successive Shortest Path Algorithm, but we also want to keep the running time low. We need rounding so as to remove the possibility of very small residual capacities, which might cause exponential worst-case running time [83] of the Successive Shortest Path algorithm.

The difference of this ”rounded” network from the original one is that the capacity between the source and the destination may be reduced, but this reduction is at most $m\delta = \epsilon d$. Consider the network $\mathcal{G}$ and consider a cut $(\mathcal{C}, \overline{\mathcal{C}})$ with the source $s \in \mathcal{C}$ and destination $t \in \overline{\mathcal{C}}$. Let $c(\mathcal{C}, \overline{\mathcal{C}})$ be the capacity of the cut. The cut obviously contains at most m edges. Rounding the capacity of each edge down to a multiple of δ may result in removing at most δ units of flow from each edge. Hence, since there are at most m edges in the cut we conclude that we may decrease the capacity of the cut by at most $m\delta = \epsilon d$ units of flow. To deal with this, we search for a minimum-cost flow that satisfies the demand of $(1 - \epsilon)d$ in the ”rounded” network and then scale this flow by $(1 - \epsilon)$ to meet the original demand. Therefore, the final flow will be within the capacities $\frac{1}{1-\epsilon}c(e)$. Of course, the resulting flow would cost more but we prove that it is not too far from the minimum-cost flow. In fact we prove that it is within an ϵ fraction from it. In other words we will show that the obtained flow is an ϵ-approximate min-cost flow.

Let f^* be the minimum cost-flow of a single commodity with demand d in network $\mathcal{G}$ and let C^* be the cost of this flow. Let $f^{\delta *}$ be the minimum cost of the flow of this commodity satisfying demand $(1-\epsilon)d$ in network $\mathcal{G}^\delta$ and let $C^{\delta *}$ be the cost of this flow. Let $\widehat{f}^*$ be the scaled flow $\frac{1}{1-\epsilon} f^{\delta *}$ and let $\widehat{C}^*$ be the cost of $\widehat{f}^*$.

Lemma 2. *Flow $\widehat{f}^*$ satisfies the edge capacities $c(e)/(1-\epsilon)$ and its cost is within a factor of $1/(1-\epsilon)$ from the minimum cost of a flow in network $\mathcal{G}$.*

Proof. Consider flow f^* in network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and the network $\overline{\mathcal{G}}^\delta = (\mathcal{N}, \overline{\mathcal{E}})$ defined as follows:
For all $e \in \mathcal{E}$, with capacities $c(e)$ there exist two parallel corresponding edges $e', e'' \in \overline{\mathcal{E}}$, with capacities $\overline{c}(e'), \overline{c}(e'')$ respectively such that

$$\overline{c}(e') = c^\delta(e) \text{ and } \overline{c}(e'') = c(e) - c^\delta(e)$$

Each flow f in $\mathcal{G}$ has a corresponding flow f' in $\overline{\mathcal{G}}^\delta$ obtained in the following way

$$\text{if } f(e) \leq c^\delta(e), \text{ then } \begin{cases} f'(e') = f(e), \\ f'(e'') = 0 \end{cases}$$
$$\text{if } f(e) > c^\delta(e), \text{ then } \begin{cases} f'(e') = c^\delta(e), \\ f'(e'') = f(e) - c^\delta(e) \end{cases}$$

We can decompose f^* into at most $2m$ paths in $\overline{\mathcal{G}}^\delta$ with

$$f^* = f_1^* + f_2^*$$

where f_1^* is the sum of the flow paths which don't use "small" edges, i.e. edges with capacities less than δ, and f_2^* consists of the flow paths which do use "small" edges.

It follows that this flow in $\mathcal{G}^\delta$ satisfies

$$C^* = C(f^*) \geq C(f_1^*) \geq C^{\delta *}$$

We conclude that

$$\widehat{C}^* = \tfrac{C^{\delta *}}{1-\epsilon} \leq \tfrac{C^*}{1-\epsilon}.$$

□

Lemma 3. *The running time of the Successive Shortest Path algorithm on the network $\mathcal{G}^\delta$ is $\widetilde{O}(\epsilon^{-1} m^2)$.*

Proof. We route at least δ units in each iteration. Hence the number of iterations to route $(1-\epsilon)d$ units from the source s_i to the destination t_i is bounded above by $d/\delta = m/\epsilon$.
For each iteration we use Dijkstra's Shortest Path algorithm which runs in $O(m+n\log n)$ time.
Hence the total running time is

$$O(m(m+n\log n)\epsilon^{-1}) = \widetilde{O}(m^2\epsilon^{-1}).$$

□

7.2 A Modification of the MCF Round-Robin Algorithm

In the previous section we have shown how we can use the SSP algorithm to find an approximate minimum cost flow in polynomial running time. We are going to use this method to calculate approximate minimum cost flows in the algorithm proposed by Radzik [65] for the MCF problem (see Section 5.1). Note that to get a polynomial running time of the SSP algorithm we needed to modify the capacities of the network, search for a minimum cost flow within these capacities and then scale the flow by a small fraction. We need to prove that these modifications within the round-robin algorithm proposed by Radzik [65] will still give us an ϵ-approximate solution for the MCF problem. In this section we follow the analysis of [65] proving that our modifications will still give an ϵ-approximate solution in a polynomial running time.

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a network with n nodes and m edges with capacities $c(e), e \in \mathcal{E}$. Let $\mathcal{G}^\delta = (\mathcal{N}, \mathcal{E}^\delta)$ be the original network $\mathcal{G}$ with modified capacities

$$c^\delta(e) = \lfloor \frac{c(e)}{\delta} \rfloor \delta,$$

where $\delta = \frac{\epsilon d}{m}$. Let $\lambda(e) = f(e)/c(e)$, where $f(e)$ is the flow on edge e and $c(e)$ its capacity. Let l be a nonnegative length function on the edges, f a multicommodity flow, and $\lambda = max_{e\in\mathcal{E}}\lambda(e)$. Let c_i be the cost of the current flow of commodity i under the length function l and denote c_i^* with the minimum cost flow of commodity i, subject to costs l and capacity constraints $\lambda \cdot c(e)$. If f_i is a flow of commodity i in network $\mathcal{G}$, for $i = 1, 2, \ldots, k$, then $f = (f_1, f_2, \ldots, f_k)$ is a *concurrent flow* of commodities 1 through k in network $\mathcal{G}$.

For a length function l, define the potential to be

$$\Phi_l = \sum_{e \in \mathcal{E}} l(e)c(e).$$

The cost of a flow f with respect to a length function l is

$$C(l, f) = \sum_{e \in \mathcal{E}} l(e)f(e).$$

The cost of a commodity i is

$$C_i = \sum_{e \in \mathcal{E}} l(e)f_i(e).$$

Let $C_i^*(\lambda, l)$ (also abbreviated as C_i^*) denote the minimum cost flow of commodity i under capacities $\lambda c(e)$. We also define:

$$C^*(\lambda, l) = \sum_{i=1}^{k} C_i^*(\lambda, l).$$

An ϵ-approximate minimum-cost flow $\widehat{f}_i^*$ of a commodity i with demand d_i from source s_i to destination t_i is defined as the flow with cost

$$\widehat{C}_i^* \leq \frac{1}{(1-\epsilon)} C_i^*,$$

and satisfying capacities $\widehat{c}(e) = \frac{1}{(1-\epsilon)} c(e)$.

As in [65] we define the edge length function to be

$$l_f(e) = \frac{e^{\alpha \lambda_f(e)}}{c(e)}, \quad \text{for each } e \in \mathcal{E}, \tag{7.1}$$

but we modify α to fit our analysis:

$$\alpha = \frac{1}{2}(1+\epsilon)\lambda^{-1}\epsilon^{-1}\ln(m\epsilon^{-1}),$$

where λ is an upper bound on the optimal congestion λ^* and $\lambda(e)$ is the congestion of the current flow on edge e.

The algorithm starts by computing a maximum flow for each commodity i independently and scaling these flows to the demands of the commodities. These flows give the initial concurrent flow. Then the termination conditions (in the while loop) are checked, as given in Algorithm 3 below. If they hold, the algorithm terminates and the current flow is returned. We will prove later that this flow is actually ϵ-approximate. If the conditions are not satisfied, the algorithm proceeds in iterations. In each iteration each commodity is considered one by one. First, an approximate minimum cost flow is calculated using the Successive Shortest Path algorithm on network $\mathcal{G}^\delta$ (Algorithm 2). Then, the algorithm checks whether the current cost of the flow is close to its approximate minimum cost. If it is, nothing happens and the next commodity is considered. If not, a small fraction of the flow of commodity i is rerouted onto the minimum cost flow paths. This iterative procedure continues until the congestion has significantly dropped down or a significant improvement has been made on the potential function. If these conditions are not met when all commodities have been considered, the algorithm terminates and returns the current flow. The modified pseudocode of the round-robin algorithm proposed by Radzik [65] is given below. We refer to this algorithm as the MRR algorithm. The step value σ, which specifies the fraction of flow that is rerouted, is $(\epsilon^2/\log n)$.

```
Algorithm 3: Modified Round-Robin Algorithm (MRR)
  Input: Network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, source-sink pairs $(s_i, t_i)$, demands $d_i$
  Concurrent flow $f = (f_1, f_2, \dots, f_k)$, $\epsilon > 0$, $\lambda \leftarrow \lambda_f$ $\sigma = \epsilon/(4\alpha\lambda)$
  Output: Flow $f$ where either $\lambda(f) \leq (1-\epsilon)\lambda$ or $f$ is $(9\epsilon)$-approximate
  $f^{(0)} = f$  , $l^{(0)} = l_f$  ,$\Phi^{(k)} = \Phi_f$, $\Phi^{(0)} = 2\Phi_f$
  while $\lambda_f \geq (1-\epsilon/3)\lambda$ and $\Phi^{(0)} - \Phi^{(k)} \geq (\epsilon^2/8)\Phi^{(0)}$ do
      $\Phi^{(0)} = \Phi^{(k)}$
      for $i = 1$ to $i = k$ do
          $\widehat{f}_i^* \leftarrow$ Call SSP Algorithm 2 on network $\mathcal{G}^\delta$ to find an $\epsilon$-approximate
          minimum cost flow of commodity $i$ within capacities $(1+2\epsilon)\lambda c(e)$
          if $c_i - \widehat{c}_i^* \geq \epsilon c_i$ then
              $f_i' \leftarrow (1-\sigma)f_i + \sigma\widehat{f}_i^*$
          end
          else
              $f_i' \leftarrow f_i$
          end
          $f^{(i)} = (f_1', f_2', \dots, f_i', f_{i+1}, \dots, f_k)$
          $\Phi^{(i)} \leftarrow \Phi(f^{(i)})$
      end
      $f \leftarrow f^{(k)}$
  end
```

7.2.1 Analysis of the Modified Round-Robin Algorithm

To implement the SSP algorithm fast we need to search for an approximate minimum cost flow in the modified network $\mathcal{G}^\delta$. Recall from Section 7.1.1 that we search for a minimum cost flow that satisfies demand $(1-\epsilon)d_i$ and scale the flow up by $(1-\epsilon)$. Thus, we result in increasing the capacities by $1/(1-\epsilon)$ in the original network. In [65] the approximate minimum cost flow is calculated using capacities that are scaled by a factor of $(1+\epsilon/3)$. We need to modify this part of the algorithm so that the SSP algorithm fits correctly. In the modified version we search for an approximate minimum cost flow within capacities

$$c'(e) = \frac{1+\epsilon/2}{1-\epsilon}c(e) = (1+2\epsilon)c(e).$$

This modification has implications on the whole analysis of the algorithm in [65]. We therefore need to adjust the main theorems in [65] appropriately and show that our proposed method of calculating minimum cost flows still results in an ϵ-approximate solution. To prove our main theorems we inevitably need to replicate some parts of the proofs in [65]. Some lemmas in [65] can be used without modification but we include these parts here for completeness.

Lemma 4 ([65])**.** *Let f be a concurrent flow and let l be a length function. Then*

$$\lambda\Phi_l \;\geq\; C(l,f) \;\geq\; C^*(\lambda,l).$$

The following corollary is a simple consequence of Lemma 4.

Lemma 5 ([65])**.** *For any length function l and any $\lambda \geq \lambda^*$,*

$$\lambda^* \;\geq\; \frac{C^*(\lambda,l)}{\Phi_l}. \tag{7.2}$$

In the following theorem we prove that when the algorithm terminates we get a 9ϵ-approximate flow.

Theorem 8. *(Approximate Optimality Conditions)*
Let $\lambda \geq \lambda^*$ and $\epsilon \leq 1/12$. If a concurrent flow f and a length function l are such that the following inequalities hold

$$\lambda_f \;\leq\; (1+2\epsilon)\lambda \tag{7.3}$$

$$C^*(\lambda,l) \;\geq\; (1-4\epsilon)\lambda\Phi_l \tag{7.4}$$

then flow f is (9ϵ)-approximate.

Proof. Let f be a concurrent flow and l be a length function such that the two inequalities (7.3) and (7.4) hold. Then from (7.2) and (7.4) we get that

$$\lambda^* \;\geq\; (1-4\epsilon)\lambda.$$

Using this result we have

$$\lambda_f \;\leq\; (1+2\epsilon)\lambda \;\leq\; \frac{1+2\epsilon}{1-4\epsilon}\lambda^* \;\leq\; (1+9\epsilon)\lambda^*.$$

The last inequality follows from the assumption that $\epsilon \leq 1/12$. □

We have shown that if $C^*(\lambda, l) \geq (1-4\epsilon)\lambda\Phi_l$ and our edge flows are within $(1+2\epsilon)\lambda$ factor of the capacities, we get a (9ϵ)-approximate solution. All we need to show now to prove that when the algorithm terminates we get a (9ϵ)-approximate flow is to show that our stopping conditions imply Inequality (7.4). First, we need to measure the level of improvement in each iteration.

Lemma 6 ([65]). *For each iteration of the outer loop of MRR algorithm and for each i, $i = 1, 2, \ldots, k$, we have*

$$\Phi^{(i)} \;\leq\; \Phi^{(i-1)}, \tag{7.5}$$

$$\lambda_{f^{(i)}} \;\leq\; (1+2\epsilon)\lambda, \tag{7.6}$$

and if the flow of commodity i has changed during this iteration, then

$$\lambda\left(\Phi^{(i-1)} - \Phi^{(i)}\right) \geq \frac{\epsilon}{8}(C_i - \widehat{C}_i^*). \tag{7.7}$$

We omit the proof of this lemma as it would be an exact replication of the proof in [65]. Using Lemma 6 we can prove that if the algorithm terminates, the resulting flow is (9ϵ)-approximate. To show the correctness of our algorithm we show that if the potential function does not decrease substantially (by a factor of $(1-\Omega(\epsilon^2))$) during one iteration of the while loop, then the value of $C^*(\lambda, l)$ is close to $\lambda\Phi_l$. We prove this in the following lemma.

Lemma 7. *If at the end of one iteration of the outer loop of MRR algorithm $\lambda_f \geq (1-\epsilon/3)\lambda$ and $\Phi^{(0)} - \Phi^{(k)} \leq (\epsilon^2/8)\Phi^{(0)}$, then*

$$C^*(\lambda; l) \;<\; (1-4\epsilon)\lambda\Phi_l. \tag{7.8}$$

Proof. Consider one iteration of the outer loop and assume that at the end of this iteration the two conditions above hold. Inequality (7.8) follows from the following four inequalities

$$\begin{aligned} C^*(\lambda, l) &\geq (1-\epsilon/3)\sum_{i=1}^{k} C_i^*, && (7.9)\\ C^* &\geq (1-\epsilon)\sum_{i=1}^{k} \widehat{C}_i^*, && (7.10)\\ \sum_{i=1}^{k} \widehat{C}_i^* &\geq (1-\epsilon)\sum_{i=1}^{k} C_i \;-\; (4/3)\epsilon\lambda\Phi_l, && (7.11)\\ \sum_{i=1}^{k} C_i &\geq (1-(4/3)\epsilon)\lambda\Phi_l. && (7.12) \end{aligned}$$

Using these four inequalities we get

$$\begin{aligned} C^*(\lambda, l) &\geq (1-\epsilon/3)(1-\epsilon)\sum_{i=1}^{k} C_i^*\\ &\geq (1-\epsilon/3)(1-\epsilon)\sum_{i=1}^{k} C_i \;-\; (4/3)\epsilon\lambda\Phi_l\\ &\geq (1-\epsilon/3)(1-\epsilon)(1-(4/3)\epsilon)\lambda\Phi_l \;-\; (4/3)\epsilon\lambda\Phi_l\\ &\geq (1-4\epsilon)\lambda\Phi_l. \end{aligned}$$

We showed (7.10) in Section 7.1.1, Lemma 2. Inequalities (7.9), (7.11) and (7.12) have essentially the same proofs as in [65], so we do not repeat them here. This ends the proof of the lemma. □

Lemma 7 together with Theorem 8 prove that when the MRR algorithm terminates we have either improved the congestion by at least a factor of $(1-\epsilon/3)$, or we have computed an (9ϵ)-approximate flow.

7.2.2 Running time

Recall that we modify the way we compute the minimum cost flows in each iteration with respect to [65]. The rest of the computation of the MRR Algorithm is the same as the Round Robin Algorithm introduced in [65].

Theorem 9. *The running time of the MRR algorithm is $\widetilde{O}(\epsilon^{-3}km^2)$.*

Proof. The improvement within each iteration is exactly the same as in [65] so we can bound it using similar arguments. Radzik [65] proves that the round-robin algorithm terminates after $O(\epsilon^{-2}\log n)$ iterations of the outer loop. Each of these iterations is dominated by k computations of approximate minimum cost flows. The initial concurrent flow formed from the (scaled) maximum flows of individual commodities has a maximum congestion at most $k\lambda^*$. Since the MRR algorithm reduces the congestion at least by a factor of $(1-\epsilon/3)$ (if it does not, then we know that we have already a 9ϵ-approximate flow), then repeating this algorithm $O(\epsilon^{-1}\log k)$ times gives a 9ϵ-approximate flow. Thus using the MRR algorithm, we can compute a 9ϵ-approximate concurrent flow within the time of $O(k\epsilon^{-3}\log n \log k)$ single-commodity approximate minimum cost flow computations. This can be improved to $O(k \log n(\epsilon^{-2} + \log k))$ single-commodity approximate minimum cost flow computations using the scaling process proposed by Leighton et al. [55]. The SSP algorithm described in Section 7.1 computes a single-commodity minimum cost flow in $O(\epsilon^{-1}m^2)$ time. Hence using the SSP algorithm, we compute a 9ϵ-approximate concurrent flow in $\widetilde{O}(\epsilon^{-3}km^2)$ time.

□

This running time is comparable with the $\widetilde{O}(\epsilon^{-2}kmn)$ running time of the algorithm analyzed by Radzik [65], if the input network is sparse and ϵ is a constant. We believe that the analysis of the MRR algorithm could be improved by analyzing together the number of iteration in all k applications (for all k commodities) of the SSP algorithm in one iteration of the MRR algorithm.

7.3 Summary

We have presented the two main combinatorial methods for solving the Maximum Concurrent Flow problem. The two approaches, though different in design and execution, seem to have similar properties and produce analogous results. In both methods a length function is used to find paths for routing the flow. In the incremental method the length function is used to estimate shortest paths whereas in the rerouting method it is used to compute a minimum cost-flow path.

In terms of execution, the incremental method starts with a zero flow and builds the flow gradually assigning a length function on each edge which depends on the previous updates of the flow on this edge. After the first phase, the demand of each commodity is routed from its source to its destination, and the lengths are updated. The rerouting approach uses an exponential length which only depends on the current flow of an edge.

We have shown in Chapter 6 that we can use an exponential length function to track the improvement in the incremental framework. This length function is similar to the one used in the rerouting framework (monotonically increasing, direct interdependence with the flow). In fact, the incremental framework using the exponential length function can be viewed as an instance of the rerouting framework. The transportation of d_i units of flow for each commodity i can be regarded as a preprocessing phase corresponding to what the initial maximum flow estimation is for the rerouting approach. After this first calculation of a feasible flow, the incremental algorithm sends d_i units of flow again and again,through the shortest paths under the current exponential length function, until the termination condition is met. In the final step the flow is scaled down so that it is feasible. Essentially what the algorithm does is rerouting some flow through less costly paths since at each phase the amount of flow of each commodity sent in the network is a small fraction of the total current flow in the network.

We have also shown in Chapter 7 how we can use the SSP algorithm to calculate minimum cost flows in the rerouting framework. The execution of the SSP algorithm is similar to the execution of the incremental algorithm. To find a minimum cost flow, the SSP algorithm starts with zero flow and incrementally sends flow through shortest paths until the whole commodity has been sent. This is exactly the execution performed in one iteration of the incremental framework. The rerouting algorithm then sends a small fraction of its current flow to this minimum cost flow paths. Again, this is similar to the execution of the incremental algorithm where in each iteration the amount of flow sent for a commodity i is a small fraction of the total amount of flow sent in all the previous iterations.

Part III

Analysis of Distributed Algorithms

Chapter 8

Distributed Computing Models

Contents

The computing industry has evolved since the birth of the first computing machine. The first computing model of sequential processing was in time complemented with parallel computing models and then with distributed models of computation. Distributed computing arose from the integration of computer and networking technologies. This model has given the power to solve large and complex problems fast and precisely.

In this chapter we introduce the main concepts related to distributed computing, focusing on the synchronous model of computation. In Section 8.1 we give the definition of a distributed system and its distinct characteristics which differentiate it from other computational models. In Section 8.2 we describe the major features of a distributed system that need to be considered when designing computation. In Section 8.3 we compare the two main types of communication mechanisms, the Synchronous and the Asynchronous communications. Finally, in Section 8.4 we give an overview of previous work on the Multicommodity Flow

Problem algorithms for distributed computing models. For further reading on the foundations of the area of Distributed Computing we refer the reader to the textbooks *"Introduction to Distributed Systems"* by Thampi [73] and *"Distributed Computing: Principles, Algorithms and Systems"* by Kshemkalyani and Singhal [51].

8.1 Definition and Characteristics

The motivation for constructing and using distributed systems comes from the inherent availability of low-priced, high performance computers and network tools. Connecting a number of computers together to perform a common task may lead to capabilities much more powerful than a supercomputer.

Definition 12 ([51]). *A distributed system is a collection of independent entities that cooperate to solve a problem.*

A distributed system typically consists of a collection of autonomous computers which have their own memory and run their own operating system. The computers are loosely-coupled together via middleware. The middleware acts as a communication network through which the computers can cooperate to solve a specific task.

Generally, a system is considered to be distributed if it has the following characteristics [51]:

No shared memory: The processors of the system do not share their memory. They have their own memory and execute protocols without being able to coordinate with each other, except through the communication network. The processors of the system might share address space via the abstraction of distributed shared memory.

No common physical clock: This also follows from the notion of no shared memory. The processors of the system cannot access a Global Clock and cannot perfectly synchronize their executions. They rather perform their individual tasks asynchronously and communicate with each other through message passing at the end of their current execution stage.

Autonomy: Each component has its own memory and its own operating system. They may run at different speeds and with different power. The processors are not part of one dedicated system, they rather cooperate with each other to solve a common problem.

Physical Separation: Usually the processors are wide apart. They are often set in different locations and perform different tasks. The middleware is what connects the processors and what makes the distributed computation available. The processors do not need to be separated in distinct geographical locations. For example, they could be a part of a Local Area Network as well. The separation of processors, even if only over a small local area, means that communication between processors may be expensive, so it should be kept to a minimum.

8.2 Features

A distributed system has a number of properties which make it attractive to be used.

Fault-Tolerance: A system is **fault-tolerant** if it can mask the presence of faults. Distributed systems should recover from processor failures quickly without the need of restarting the whole procedure. This is desirable especially when designing distributed algorithms for the World Wide Web (WWW), because of the continuous changes of the network's topology and demand. Usually fault-tolerance is achieved by providing redundancies in hardware and/or software. By adding such extra processors we can ensure that the service will run uninterrupted since any fault in the system would be replaced immediately by the extra processors. Fault-tolerance of distributed algorithms means that their computation is correct even if some processors malfunction (though there would always be a limit on the type and number of failures which can be tolerated).

Scalability: The topology of the distributed system allows it to be scalable. New processors can be added without creating a bottleneck in the communication network. To ensure that the system can continue to operate properly when new processors are added, techniques like replication are employed. Multiple copies of resources are placed throughout the system so that the load is spread when new entities are added.

Enhanced Reliability: Replicating resources and executions in a distributed system can make it less vulnerable to crashes and malfunctions.

Self-Stabilization Regardless of the initial state of the system or the changes that could occur during the execution of processes, the distributed system should converge to a legitimate state without external intervention.

8.3 Distributed Models

Several different models of distributed systems exist depending on the architecture of the system. The most significant characteristics of such models are synchronous or asynchronous execution and communication based on message passing or shared memory. In this section we briefly examine and compare these characteristics.

Message-passing systems involve a set of processors which use their own memory and perform their own computation. The processors communicate with each other via **Send** and **Receive** operations. On the other hand in a **distributed shared memory system** the communication is performed via **Read** and **Write** operations on a shared address space. The two models are illustrated in Figure 8.1.

Distributed models can also be classified in terms of the level of synchronisation of the execution of individual processors. Full synchronisation would mean that all processors run the same set of operations at the same time in parallel. In each global time step all processors would perform the same simple operation. However, this kind of synchronisation is not attainable in a distributed system. The model used in the distributed algorithms studied in this thesis introduces the notion of a *round*. Processors execute tasks in asynchronous way but synchronise their execution at the beginning of each round by waiting for all processors to finish their previous round. This is in some sense a virtually synchronous process,

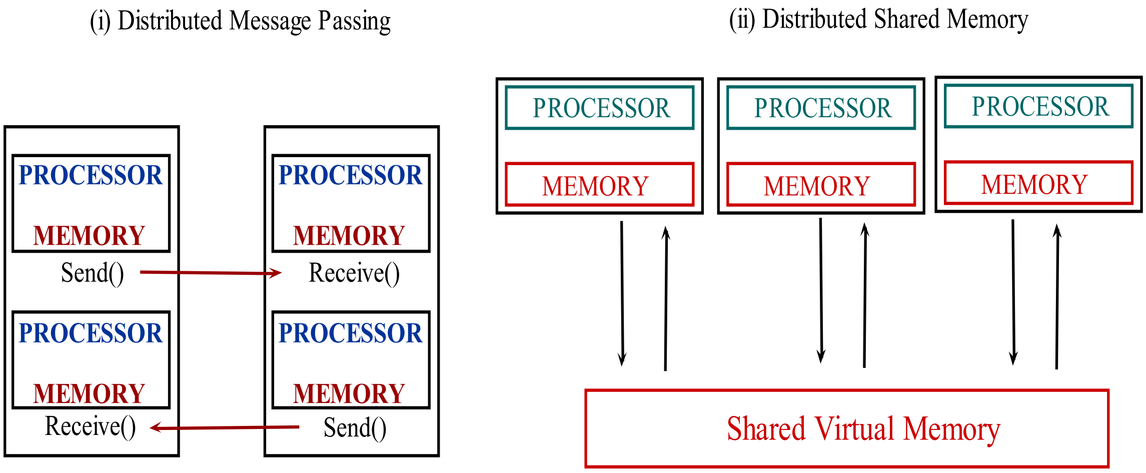


Figure 8.1: Message Passing System versus Shared Memory System

where the processors execute their tasks in an asynchronous manner for a period of time and then they synchronise their next set of instructions using the notion of the round. We note that the notion of a round should be viewed as a property of an algorithm.

There are similarities between this model (the details of the synchronisation process are given later in subsection 8.4.2) and the general characteristics of the Bulk Synchronous Parallel (BSP) model and its variants [74, 77]. The general idea of the BSP model is that the computation is organised in "supersteps" and the communication between the processors occurs only between these supersteps. The computation within one superstep is asynchronous. The running time in the BSP model combines the cost of the local computation, the number of supersteps and the cost of communication between the processors into one global running time. The analysis of algorithms for our distributed model is predominantly focused on minimizing the number of rounds. The analysis may also include bounding the total "sequential" running time, which is simply the running time of the local computation summed up over all processors. The bounding of the total running time sequentially is done so that it can be easily compared with the running time of existing sequential algorithms. We stress that the main objective is to minimize the number of rounds. Our model does not explicitly consider the communication cost of synchronising the processors at the end of a round because it is assumed to be negligible in comparison to the running time of one round.

The model we consider in this thesis is based on the Distributed Shared Memory Model with perfectly synchronised starting time of each round but asynchronous execution of the tasks within the rounds. There is no co-ordination between the processors within one round. The only co-ordination that exists is at the beginning of a round when processors are allowed to communicate by first "posting" data to the shared address space and then reading off some aggregate of the posted data. The type of aggregation of the data depends on a particular problem and would come from the restrictions of the assumed physical computational environment. For example, for the MCF problem, each processor is responsible for computing the flow of one commodity, posts the current flow of this commodity at the end of each round, but can read only the total (sum) of the flows of all commodities. This restriction is motivated by the assumption that a processor may have access to the information about the total traffic on any particular link in a (global) network, but might not know who contributes to this traffic. Further details are given in subsection 8.4.2.

8.4 Previous Work

By distributed algorithms for multicommodity flow problems we mean algorithms that route flow of all commodities in a parallel but uncoordinated manner. The framework is similar for all of the existing algorithms. Certain restrictions exist depending on the model used. Consider a network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where n is the number of nodes and m is the number of edges. We have k commodities with associated demands $d_i, i = 1, 2, ..., k$. Commodity i must be routed from its source s_i to its destination(sink) t_i.

Two types of distributed models for the maximum concurrent flow problem have been described depending on the way decisions of how to send the flow are made:

M1 The algorithms in which decisions are made in parallel at the nodes,

M2 The algorithms in which decisions are made in parallel for each commodity.

We next discuss the algorithmic framework for distributed algorithms for the MCF problem in the more traditional model **M1**. Then we introduce the details of the second model **M2**, which was proposed by Awerbuch et al. [7] and is referred to as the Billboard Model. The MCF algorithms for this model are the subject of Chapters 9 and 10.

8.4.1 Decisions at the Nodes

Awerbuch and Leighton [5] proposed the first distributed algorithm for solving the Maximum Concurrent Flow Problem. Their algorithm relied on the queues of the flows of different commodities on each edge. Informally, the algorithm tries to push one unit of flow along each edge in each round in an attempt to balance the flow on this edge. The algorithm simply tries to send flow through an edge $e = (u, v)$, if the queue of a commodity at u is greater than the queue of the same commodity at v. The queue of a commodity at a node v is defined as the amount of flow of that commodity stacked at node v. Contention between commodities is resolved by sending the commodity with the largest disparity between the ends of each edge. If the flow reaches the right sink, it is removed from the network. Awerbuch and Leighton [5] prove that the algorithm is stable by showing that the height of any queue at any round is bounded. The nature of the algorithm makes it appropriate for use in decentralized networks where link failures can occur. The algorithm terminates in $\widetilde{O}(\epsilon^{-3}Lm^3k^{5/2})$ steps where L denotes the size of the longest flow path. One round consists of the following phases.

Phase 1: Add $(1+\epsilon)d_i\delta_i^{-1}$ units of flow on each of the δ_i edges incident to each source s_i.

Phase 2: Push flow $f_i(u, v)$ of commodity i across each edge $(u, v) \in \mathcal{E}$, from the tail of the edge u to its head v such that the following term is maximized:

$$\sum_{i=1}^{k} f_i(u, v)(q_i(u) - q_i(v) - f_i(u, v))d_i^{-2}$$

subject to

$$\begin{aligned}
&f_i(u,v) \geq 0, \;\; \forall \;\; 1 \leq i \leq k, \\
&\sum_{1\leq i\leq k} f_i(u,v) \leq c(u,v), \\
&f_i(u,v) = 0 \;\text{ if }\; c(u,v) \leq \epsilon d_i/m.
\end{aligned}$$

$q_i(u)$ and $q_i(v)$ denote the height of the queue of commodity i at the tail and head of edge (u,v), respectively.

Phase 3: Remove flow from sinks.

Phase 4: Rebalance at nodes by reallocating flows so that the edge queues for each commodity are equal within each node.

To measure the progress of the algorithm the following potential function is introduced

$$\Gamma = \sum_{(u,v)\in\mathcal{E}} \sum_{1\leq i\leq k} \left(\frac{q_i(u)}{d_i}\right)^2 + \left(\frac{q_i(v)}{d_i}\right)^2 \tag{8.1}$$

Based on the same distributed computing model, Awerbuch and Leighton [6] managed to improve the running time to $\widetilde{O}(\epsilon^{-1}Lkm)$ steps by approaching the problem more carefully. Instead of using the potential function (8.1) they use an exponential potential function

$$\phi_i(q) = \exp(\alpha_i \cdot q), \tag{8.2}$$

where $\alpha_i \equiv \frac{\epsilon}{8Ld_i}$ and q denotes the height of the queue. It can be shown that maximizing (8.1) is equivalent to minimizing

$$\sum_{i=1}^{k} \left[(q_i(u) - f_i(u,v))^2 - (q_i(v) + f_i(u,v))^2\right],$$

so the objective of one round of the algorithm from [5] is, for each edge (u,v), to minimize the sum of the squares of the sizes of the queues associated with this edge. The objective of one round of the algorithm from [6] is to minimize the sum of the potentials of the queues associated with edge (u, v), that is, to send across edge (u, v) flows $f_i(u, v)$ such that the following sum is minimized:

$$\sum_{i=1}^{k} \left[\phi_i \left(q_i(u) - f_i(u, v) \right)^2 - \phi_i \left(q_i(v) + f_i(u, v) \right)^2 \right].$$

Awerbuch and Leighton [6] also introduce a change in the way queues are built. Instead of allowing all queues to have arbitrary sizes, they now fix the capacities of the queues at the source node s_i to some specified value Q_i and allow for an overflow buffer. They define the potential of an overflow buffer of size b to be

$$\sigma_i(b) = \phi'_i(Q_i) \cdot b = \alpha_i \cdot b \cdot \exp(a_i \cdot Q_i). \tag{8.3}$$

Using this refined method of approaching the problem they show that no overflow buffer ever gets too large. Hence, after a significant number of rounds the flow remaining in the queues is negligible with respect to the flow delivered at the sinks.

Muthukrishnan and Suel [60] propose a method, which they call second-order method, to improve the running time of the algorithms proposed in [5] and [6]. They show experimentally that this method gives better actual running times. They use a framework similar to [5] with the difference being in the amount of flow pushed across each edge. In [5], the flows $f_i(u, v)$ pushed across edge (u, v) are selected to maximize

$$\sum_{i=1}^{k} f_i(u, v)(\Delta_i(u, v) - f_i(u, v)),$$

where $\Delta_i(u, v) = q_i(u) - q_i(v)$. In [60], the selected values $f_i(u, v)$ maximize the expression

$$\sum_{i=1}^{k} f_i(u, v)(\Delta'_i(u, v) - f_i(u, v)),$$

where $\Delta'_i(u, v)$ is defined in the following way:

$$\Delta'_i(u, v) = \beta \cdot \Delta_i(u, v) + 2 \cdot (\beta - 1) \cdot f'_i(u, v), \tag{8.4}$$

and $f'_i(u, v)$ denotes the amount of flow of commodity i sent along edge (u, v) in the previous iteration and $1 \leq \beta \leq 2$. Note that when $\beta = 1$, the value of $\Delta'_i(u, v)$ is exactly the same as in [5]. They run their experiments for different inputs, iterating β by increments of 0.05. They show that when β is between 1.95 and 1.99 their method converges to within 16 digits of precision while the algorithm of Awerbuch and Leighton [5] is more than 10% away from the exact solution. For values of $\beta > 2$ the algorithm becomes unstable and does not converge.

8.4.2 The Billboard Model

The billboard distributed computation model for multicommodity flow problems was proposed by Awerbuch et al. [7]. Each commodity is associated with one agent. A "billboard" maintains the current total flows on the edges of the network. The agents are allowed to read the values from the billboard at the beginning of each round and decide how to reroute the flows of their commodities. Each agent has information about the total edge flows (from the billboard) and the edge flows of its commodity (from the local state), that is, they cannot distinguish between the flows of each commodity on a given edge. At the end of each round they submit (post) their flows to the billboard again. This model falls within the broad category of the distributed shared memory models, as discussed in Section 8.3, but the agents (processors) have access only to the "aggregate" of data (in our case they have access only to the total flow on edges).

The agents are not allowed to co-ordinate with each other in any other way than through the billboard, that is, they cannot read other agents' flows or have information about their current decisions. The only co-ordination that exists is that the agents synchronize the start of their executions using the notion of a round. Decisions are made in parallel at each round. Each agent executes the same local program and decides how to update the flow of its commodity subject to certain constraints depending on the algorithm used. More precisely, each agent knows the topology and parameters of the common underlying network (the edges of the network and their capacities) and the specification of its commodity (the source, the destination and the demand of this commodity). The agents, and the commodities, do not have their "identities". They perform the same program but on their own data. For example, they may start the computation finding the shortest path from the source to the destination. The agents know when the current round ends, from the parameters of the executed algorithm, the upper bound on the (local) time of one round of any agent.

Two maximum concurrent flow algorithms were proposed for this model [7, 4]. Both algorithms work in the general model outlined above, but the Greedy Distributed algorithm (GDR-MCF) of [4] uses a weaker (more restrictive) synchronization process. In the model used in [7] for the Approximate Steepest Descent (DGD-MCF) algorithm, the global clock counts the rounds starting from 1. That is, all agents start at the same time (round 1) and their computation can refer to the current round number. The computation terminates after a fixed number of rounds. In the model used in [4] for the Greedy Distributed algorithm the clock informs the agents only that the next round starts but does not provide the counter of the rounds. This means that the computation in each round cannot depend on the index of the round. The computation only terminates once a stabilisation is achieved (certain stopping criteria have been met). Distributed MCF algorithms which have this property are considered in [4] as *stateless* algorithms.

The first algorithm under the above framework was proposed by Awerbuch et al. [7]. They describe approximate distributed MCF algorithm which is based on the Steepest Descent method for global optimization and was inspired by the sequential MCF algorithm proposed by Garg and Koenemann [30]. In the distributed MCF algorithm of [7], a tiny amount of flow of all commodities on all edges needs to be placed initially and later it is increased in a multiplicative way. More precisely, initially the flow $f_i(e)$ for each edge e is set to $f_i(e) = \epsilon c(e)/k$. This might affect the flow conservation conditions but since the flow is at most $\epsilon c(e)$ on each edge the violation is not significant, and can be easily corrected at the end of the computation.

The computation is done in phases. In each phase an amount of $\epsilon^2 d_i / \log m$ is pushed into the network for each commodity i. A phase consists of rounds. In each round a blocking flow of commodity i for some small edge capacities $c_i(e)$ is computed by sending flows along ϵ-approximate shortest paths from s_i to t_i according to the length function given by

$$l(e) = \frac{(m^{1/\epsilon})^{\lambda(e)}}{c(e)}. \tag{8.5}$$

To avoid oscillations, a condition that the flow of each commodity on each edge increases at most by a multiplicative factor $(1 + \delta)$ for an appropriately small δ(or by a polynomially small additive amount) is introduced. This is done by allowing the flow of commodity i to increase on an edge e by at most

$$c_i(e) = \frac{\epsilon^2 f_i(e)}{\log m}$$

in each round. This condition is sufficient to avoid unpredictable oscillations because the total flow on each edge in one round increases by at most a factor of $(1 + \epsilon^2/\log m)$. The algorithm runs in $\widetilde{O}(L/\epsilon^4)$ rounds, each round running in $\widetilde{O}(m^2/\epsilon^2)$ time, giving the total of $\widetilde{O}(Lm^2/\epsilon^6)$ time.

The second algorithm based on the distributed model was proposed by Awerbuch and Khandekar [4]. They describe the first stateless greedy algorithm for the distributed multi-commodity concurrent flow problem with the number of rounds poly-logarithmic in the size of the input and linear in the parameter L (same as in [7]).

A routing metric is introduced to indicate the cost of an edge with respect to its congestion. This is given by the derivative of the potential function

$$\phi_{e,\mu}(f(e)) = m^{\frac{f(e)}{c(e)\cdot\mu}},$$

where μ is a value associated with the maximum congestion in the network. The agent of commodity i reads the flow values from the billboard at the beginning of each round. Then the costs of all edges are calculated. The shortest path B $\in P_i$, where P_i is the set of paths of commodity i, is then found. The cost of this path is compared with the average cost of the flow of commodity i in the network which can be found by adding the total cost of commodity i on all edges and dividing this by its demand d_i. If the cost of path B is smaller than $(1 - (\epsilon))$ times the average cost and the allowed push forward flow for this commodity is greater than zero the agent reroutes a portion of flow of commodity i from all paths to path B. The procedure proceeds until a desired fraction of the flow is rerouted or until no path can be found with cost significantly smaller than the average path cost. At this stage the commodity writes its new flow on the billboard and waits for the beginning of the next round (that is, waits until all agents complete the current round). The algorithm runs in $\widetilde{O}(L/\epsilon^8)$ rounds, each round running in $\widetilde{O}(m^2)$ time, giving a computation overhead of $\widetilde{O}(Lm^2/\epsilon^8)$ for each commodity.

To deal with the instability that can arise by routing all the flows simultaneously (an edge flow could repeatedly sharply increase and then sharply decrease) ”speed limits” are imposed. These limits control the maximal amount a flow can increase or decrease on an edge. This way oscillations are prevented and the system converges. The interesting aspect of the algorithm in [4] is that a more restrictive ”speed limit” is enforced on the way down rather than on the way up, which is opposite to the intuition that increase is more ”dangerous" than decrease. The flow on a path is allowed to increase much more rapidly than it is allowed to decrease.

The property of the GDR-MCF of being stateless has potential benefits. Such an algorithm is more robust because each decision is taken based entirely on information that comes at the current point of time, without reference to the past. The agents do not rely on the global clock and do not rely on initialization. This feature makes the GDR-MCF algorithm attractive for internet applications since the internet protocol itself is an example of stateless interaction. In contrast, in [7] a global clock needs to be maintained because the DGD-MCF algorithm crucially depends on maintaining a state and also depends on the appropriate initialization. If, for example, changes occur in the network topology or demands, the DGD-MCF algorithm proposed in [7] needs to be initialized (restarted) again, whereas the GDR-MCF algorithm proposed in [4] can adjust its flows locally without disrupting the flows that are not affected.

8.5 Summary

In this chapter we have introduced the distributed computation model and we have described its main characteristics. We have also described the previous literature on distributed algorithms for the MCF problem. Finally, we have introduced the billboard distributed model which is the model used by the algorithms we examine in more detail in this thesis. In the next two chapters we are analyzing the two main distributed algorithms (DGD-MCF and GDR-MCF) proposed for the MCF problem and we propose heuristic improvements to speed up their running time.

Chapter 9

The Approximate Steepest Descent Framework

Contents

Awerbuch et al. [7] describe a framework for approximate distributed algorithms for the multicommodity flow problem, which was based on the Steepest Descent method for global optimization and was inspired by Garg and Koenemann's [30] sequential algorithm for this problem. The model of distributed computing used by this framework is described in Section 8.4. The upper bound on the running time of the algorithm proposed in [7] depends linearly on the maximum path size L. The main open question here is to reduce the dependency of the running time on L, from linear to ideally logarithmic. The first question in this direction is whether the algorithms such as in [7] and [4] do achieve this upper bound in the worst case. The upper bound analysis in both [7] and [4] was based on the analysis of the flow change on one edge on the shortest path. In each iteration, there is one edge with an increase of flow by at least some small amount. These increases are totalled to get the flow change for the whole path.

The upper bound on the number of rounds then follows from an upper bound on flow on a simple path. However, it is natural to expect that those individual increases would be happening in parallel in the same round on many edges. This expectation is reasonable because of the distributed nature of the algorithm.

If those edge increases had to happen in parallel, then the derived upper bound would not be tight. In the following chapters we construct a worst case input which shows that the upper bounds of the algorithms in [7, 4] are actually tight, up to polylogarithmic factors. We then propose a heuristic where each agent balances its flow in each round among available shortest paths. We show that for our worst-case example network Υ_L, the first phase of the Approximate Steepest Descent algorithm runs in $O(polylog(L))$ rounds with our heuristic, but $\widetilde{\Omega}(L)$ rounds are needed by the original version of the algorithm. While this does not provide a full analysis of our proposed heuristic (the full analysis remains an open challenge), it does indicate potential for better performance than the previous algorithms.

The rest of the chapter is structured as follows. In Section 9.1 we review the DGD-MCF algorithm proposed by Awerbuch et al. [7] and analyze its computation. In Section 9.2 we give an input for which we prove that the algorithm runs in $\widetilde{\Omega}(L)$ rounds. Our worst-case input helped us to identify where the bottleneck in the computation may occur. This led to a heuristic to avoid the bottleneck of the Gradient Descent Framework. We describe this heuristic in Section 9.3 and prove that it significantly speeds up, at least the first phase, the computation on network Υ_L.

9.1 The Approximate Steepest Descent Algorithm

The distributed computation model for the Approximate Steepest Descent Framework proposed in [7] is described in subsection 8.4.2. The pseudo-code of the algorithm proposed by Awerbuch et al. [7] is given below. The agents may finish the current phase in different rounds. This means that the agents may have a few "idle" rounds at the end of a phase to make sure that all agents complete this phase and can start together the next phase. This is done by waiting until the number of rounds in the phase reaches $T_p = O(L(\log^2 m \log(k/\epsilon))/\epsilon^4)$. It is proven in [7] (see Lemma 8) that this number of rounds is sufficient to route $\epsilon^2 d_i/\log m$ units of flow for each commodity i.

Algorithm 4: DGD-MCF algorithm

```
Input: Network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, Capacities $c(e)$, Commodities $i$ with source $s_i$,
       destination $t_i$, demand $d_i$, $i = 1, 2, \ldots, k$
Initialization: Set $f_i(e) = \epsilon c(e)/k$ for each edge $e$, for each commodity $i$;
for $O((\log m)/\epsilon^2)$ phases do
    for each commodity $i$ in parallel
    while commodity $i$ has not yet routed $\epsilon^2 d_i/\log m$ flow do
        1. Define capacities $c_i(e) = \epsilon^2 f_i(e)/\log m$;
        2. Compute a blocking flow under capacities $c_i(e)$ from $s_i$ to $t_i$
        along $\epsilon$-approximate shortest paths;
        3. Route the blocking flow computed;
    end
    Synchronization: Wait until the total number of rounds in this phase
    is $T_p$
end
```

We now sketch the analysis of this algorithm given in [7]. For simplicity, assume that the demands are scaled so that the maximum congestion, λ, is 1. Recall from Section 3.3.1 that the congestion $\lambda(e)$ of an edge e is given by $\lambda(e) = \sum_i f_i(e)/c(e)$. The improvement is measured using the following potential function Φ which is exponential with respect to the edge congestion

$$\Phi = \sum_e (m^{1/\epsilon})^{\lambda(e)}. \tag{9.1}$$

It is shown in [7] (Lemma 2.1) that Φ is approximated within a factor of $m^{O(1)}$ at the end of the algorithm. This gives an $O(\epsilon)$ approximation of the maximum congestion. Indeed, if the optimum potential Φ_{opt} is approximated within a factor of $m^{O(1)}$ then

$$
\begin{aligned}
\lambda &\leq \frac{\epsilon}{\log m} \log \Phi && (9.2) \\
&\leq \frac{\epsilon}{\log m} \log(m^{O(1)} \Phi_{opt}) \\
&\leq O(\epsilon) + \frac{\epsilon}{\log m} \log(m \cdot (m^{1/\epsilon})^{\lambda^*}) \\
&\leq 1 + O(\epsilon)
\end{aligned}
$$

where $\lambda^* = 1$ denotes the optimal congestion. The inequality (9.2) holds because (9.1) implies that $\Phi \geq (m^{1/\epsilon})^{\lambda}$.

Lemma 8 (Awerbuch et al. [7]). *The number of rounds needed to route $\epsilon^2 d_i / \log m$ units of flow for each commodity i is $O(L(\log^2 m \log(k/\epsilon))/\epsilon^4)$.*

Proof. Consider a commodity i. A phase does not end until each commodity sends $\epsilon^2 d_i / \log m$ units of flow. We are going to argue that this phase ends after $O(L(\log^2 m \log(k/\epsilon))/\epsilon^4)$ rounds. During one round, each commodity computes ϵ approximate shortest paths under the current length function and finds a blocking flow. The blocking flow is bounded by the capacities $c_i(e) = \epsilon^2 f_i(e)/\log^2 m$. So, each blocking flow saturates at least one edge on each approximate shortest path, increasing its flow by a factor of $1 + \epsilon^2/\log m$. Since the initial flow on the edge is $\epsilon c(e)/k$ and can be increased to at most $(1 + O(\epsilon))c(e)$, an edge can be saturated at most $O(\log m \log(k/\epsilon)/\epsilon^2)$ times. Hence, since any path has at most L edges we know that after $O(L(\log m \log(k/\epsilon)/\epsilon^2))$ rounds the length of the shortest path has increased by a factor of $1 + \epsilon$. Since the shortest path length cannot increase by a factor greater than $m^{O(1)+1/\epsilon}$ during the entire algorithm, recall that such an increase would imply an optimum flow, we conclude that after $O(L(\log^2 m \log(k/\epsilon)/\epsilon^4))$ rounds commodity i has pushed at least $\epsilon^2 d_i / \log m$ units of flow. □

Lemma 8 gives the following overall number of rounds of the Distributed Gradient Descent algorithm.

Theorem 10 (Theorem 1.1 in [7]). *The overall number of rounds of the Distributed Gradient Descent algorithm is $O(L(\log^3 m \log(k/\epsilon)/\epsilon^6))$*

The overall complexity of the algorithm is $O(L(\log^3 m \log(k/\epsilon)/\epsilon^6)T_{bf})$, where $T_{bf} = \widetilde{O}(m^2)$ is the time needed to compute a blocking flow. This follows from the fact that in each phase we send $\epsilon^2 d_i/\log m$ units of flow and thus we need $\log m/\epsilon^2$ phases to route d_i units of flow.

9.2 A Worst Case Input

The analysis of the DGD-MCF algorithm given in [7] left the question whether the obtained upper bound on the running time was tight in the worst case. The analysis of the upper bound of the DGD-MCF algorithm was based on the flow updates on one edge on the shortest path. The upper bound analysis was only counting the increases of flow on the saturated edges. When the flow is updated on long paths (in terms of size), then in addition to (at least one) saturated edge, the flow would increase on all edges of the path. To show that the upper bound is tight, one has to show an input such that in most of the iterations, if flow is increased on a path P, then increase on most of the edges of P is negligible. Constructing such an input turned out to be a major challenge.

We have constructed a worst case example which shows that the upper bound in [7] is tight up to polylogarithmic factors. The difficulty in constructing such a network comes from the distributed nature of the algorithm: we have to take into account the changes of flow coming from all agents. We needed to create a network which would "force" the algorithm to behave in a sequential manner, i.e. saturate only one edge of the shortest path in each round while increasing the flow on other edges only by a small amount in relation to their current capacities. To get such a behaviour we needed to force the algorithm not only to block the shortest path in the current round but also block all the other shortest paths at the same time thus, not letting any other edges to be saturated during the same round.

Our graph has regular structure and unit edge capacities, implying that the high running time comes from the structure of the graph and not from the variation of the edge capacities.

For an integer $L \geq 3$ we construct a directed network $\Upsilon_L = (\mathcal{N}, \mathcal{E})$ with $|\mathcal{N}| = \Theta(L^2)$ nodes and $|\mathcal{E}| = \Theta(L^2)$ edges with unit capacities. In this network

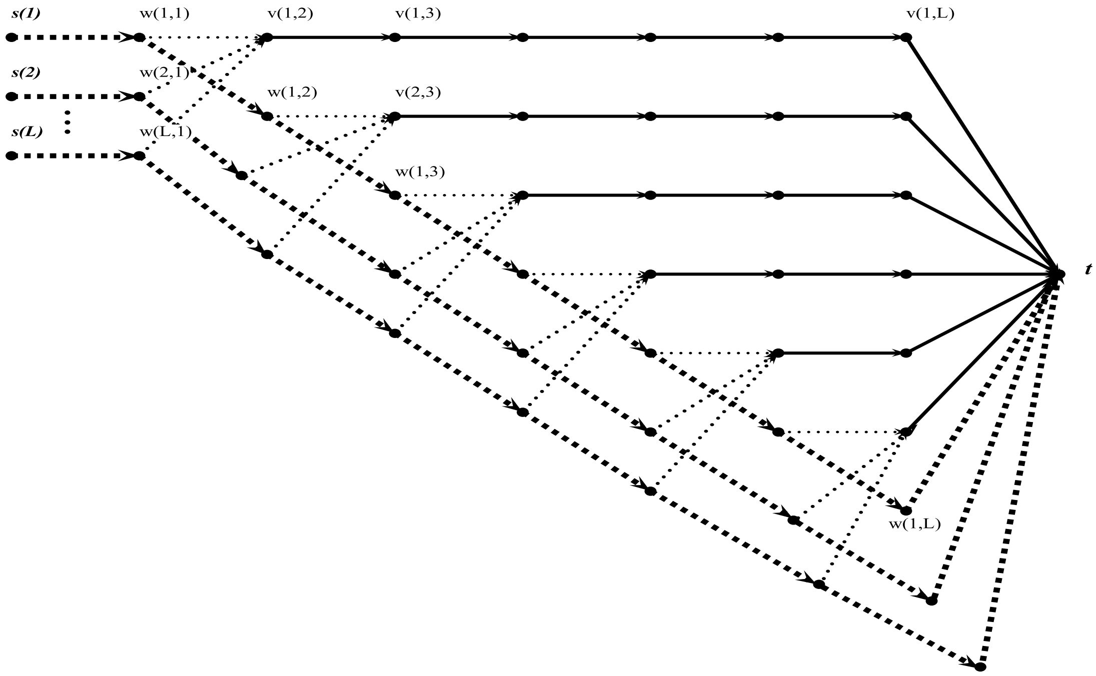


Figure 9.1: Example of Network Υ_L

we have L commodities with unit demand and source sink pair (s_i, t). The set $\mathcal{N}$ consists of the following nodes:

- t, the destination of all commodities;
- s_i, $i = 1, 2, \ldots, L$ (the source of commodity i);
- $v_{p,q}$, $p = 1, 2, \ldots, L-1$ and $q = p+1, p+2, \ldots, L$;
- $w_{i,p}$, $p = 1, 2, \ldots, L$ and $i = 1, 2, \ldots, L$.

The set of directed edges $\mathcal{E}$ is the union of the following six disjoint sets:

$$\mathcal{E}_1 = \{(v_{p,q}, v_{p,q+1}) : p = 1, 2, \ldots, L-2 \text{ and } q = p+1, p+2, \ldots, L \},$$

$$\mathcal{E}_2 = \{(v_{p,L}, t) : p = 1, 2, \ldots, L-1 \},$$

$$\mathcal{E}_3 = \{(s_i, w_{i,p}) : p = 1 \},$$

$$\mathcal{E}_4 = \{(w_{i,p}, t) : p = L \},$$

$\mathcal{E}_5 = \{(w_{i,p}, w_{i,p+1}) : p = 1, 2, \ldots, L-1\ \}$, and

$\mathcal{E}_6 = \{(w_{i,p}, v_{p,p+1}) : p = 1, 2, \ldots, L-1\ \}$.

Figure 9.1 shows the structure of network Υ_L. Observe that each source-to-destination path in Υ_L has exactly $L+1$ edges. Each commodity i has L paths which it can use. Among those, it has the "exclusive" path P_i:

$$P_i = \{(s_i, w_{i,1}), (w_{i,1}, w_{i,2}), (w_{i,2}, w_{i,3}), \ldots, (w_{i,L}, t)\}.$$

No edge of this path can be used by any other commodity. The exclusive paths are shown in Figure 9.1 by the dashed-line edges. The edges of the exclusive paths form the subset $\mathcal{E}_3 \bigcup \mathcal{E}_4 \bigcup \mathcal{E}_5$. Figure 9.2 shows such a path for commodity 1.

Commodity i can also use $(L-1)$ paths, $P_{i,j}$, for $j = 1, 2, \ldots, L-1$ defined in the following way:

$$P_{i,j} = \begin{cases} (s_i, w_{i,1}) & \text{connecting source } i \text{ to the network} \\ (w_{i,1}, w_{i,2}), \ldots, (w_{i,j-1}, w_{i,j}) & \text{the initial part of path } P_i \\ (v_{j,j+1}, v_{j,j+2}), \ldots, (v_{j,L}, t) & \text{the "shared" part} \end{cases}$$

Such a path consists of the edge connecting the source s_i to the network, namely $(s_i, w_{i,1})$. Then it has $j-1$ edges of path P_i and $L-j$ edges shared by all commodities. The edges of path $P_{i,j}$ form the subset $\mathcal{E}_1 \bigcup \mathcal{E}_2 \bigcup \mathcal{E}_3 \bigcup \mathcal{E}_5 \bigcup \mathcal{E}_6$.

The network is symmetric with respect to each commodity, meaning that each sub-network Υ_L^i in which each commodity i can send flow has the same structure as all other sub-networks Υ_L^i. Figure 9.2 shows the sub-network "visible" by commodity i. The optimal congestion is 1, and this can be achieved by saturating the exclusive paths for each commodity. This is not the only way of getting congestion 1 since some flows can be sent along shared paths as well. This network is difficult for the DGD-MCF algorithm because the agent i responsible for commodity i does not know that the exclusive path P_i cannot be used by other commodities, and is the best choice for commodity i. Using shared paths creates contention among the agents which may require a significant number of rounds to resolve.

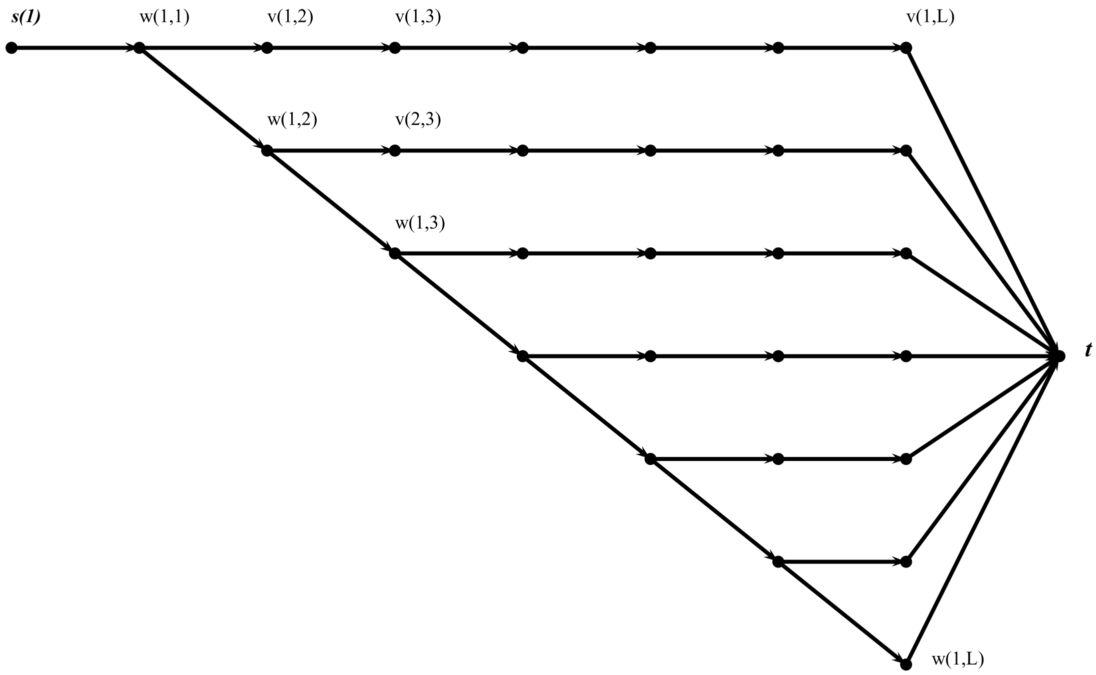


Figure 9.2: Network for One Commodity

9.2.1 Analysis of the DGD-MCF algorithm on network Υ_L

We consider only the first phase of the algorithm; when $\epsilon^2/\log m$ flow of each commodity is routed (recall $d_i = 1$). The algorithm starts by setting the flow on each edge to be $f_i(e) = \epsilon/k$ for all commodities i. Then a blocking flow is calculated using ϵ-approximate shortest paths. The computed blocking flow is added to the accumulated flow of commodity i until a total of $\epsilon^2/\log m$ flow is routed for each commodity.

Algorithm 4: Computation of phase 1 of DGD-MCF algorithm on Υ_L

```
Computation for Agent $i$ (controlling commodity $i$ );
**Initialization Round**: Set $f_i(e) = \epsilon/L$ ;
**while** *commodity $i$ has not yet routed $\epsilon^2/\log m$ flow* **do**
    1. Set the length of the edges as in (8.5);
    2. Define capacities $c_i(e) = \epsilon^2 f_i(e)/\log m$;
    3. Compute a blocking flow $g_i$ from $s_i$ to $t$ along $\epsilon$-approximate
    shortest paths, considering the paths in the order $P_{i,1}, P_{i,2}, \ldots, P_{i,j}$;
    4. Route the computed blocking flow $g_i$
**end**
```

Initially in our network all of the paths are shortest paths since they have the same flow, the same number of edges and the same(unit) capacities. Since each commodity computes a blocking flow without any coordination or co-operation with other commodities, the worst case would be such that all commodities pick the paths with the same shared part. In particular during the initial rounds we force all agents to use paths $P_{i,1}$. We force the agents to augment on paths $P_{i,1}$ for as long as these paths are available, that is, their length has not yet increased by a factor more than $(1+\epsilon)$. Saturating these paths gives us blocking flows, so no other paths will be used at this initial stage.

Note that at the time when paths $P_{i,1}$ have increased their length by a $(1+\epsilon)$ factor all other paths have increased their length by the same amount. That is, all other paths available for each commodity are the shortest paths with exactly the same length. When paths $P_{i,1}$ are not available, we force the agents to take the next available approximate shortest path $P_{i,j}$. In particular, they choose paths $P_{i,2}$ and follow the same pattern: saturating paths $P_{i,2}$ gives blocking flows, so no other paths will be used. Then the agents move to the paths $P_{i,3}$, and so on, until they route $\epsilon^2/\log m$ units of flow, which marks the termination of a phase.

In the remainder of this section, we prove the lower bound on the running time of the DGD-MCF algorithm on network Υ_L. We view the computation as a sequence of stages, each stage consists of a number of rounds. The next stage begins when the next shared path is picked up by the agents. We assume from now on that $\epsilon \leq \log m$ and $L \geq 7$.

Definition 13. *The Stage h for $h = 1, 2, \ldots, L$ begins at the round when the path $P_{i,h}$ is used for the first time for augmentation.*

To establish the running time of the DGD-MCF algorithm, we need a bound on the number of rounds required to terminate a Stage h, which is proven in Lemma 12. However, in order to show the main ideas and to establish some formulas, we analyze separately Stage 1 (Lemma 9). Then we consider the general Stage h.

Let $\Pi_{i,j}^{(h,r)}$ and $\Pi_i^{(h,r)}$ denote the lengths of paths $P_{i,j}$ and P_i at Stage h after round r respectively.

Lemma 9. *In Stage 1 the length of path $P_{i,1}$ at the end of round r is*

$$\Pi_{i,1}^{(1,r)} \leq \left(1 + 2\log m \left((1 + \frac{\epsilon^2}{\log m})^r - 1\right)\right) \Pi_{i,1}^{(1,0)}, \tag{9.3}$$

and the number of rounds in this stage is $\Omega(\epsilon^{-1})$.

Proof. Consider the initial augmentation of the algorithm for all commodities i. They all pick path $P_{i,1}$ to augment, as described above. Note that by choosing this path, no other paths can be used during the round since augmentations along these paths saturate edges$(s_i, w_{i,1})$ giving blocking flows (All edge capacities on paths $P_{i,1}$ are the same, so all edges on these paths get saturated).

The following result (9.4) holds for any flow change in a path and we prove this separately because we are going to use it repeatedly in our derivations later. If the flow on an edge e increases by $\Delta f(e)$, and the congestion increases from $\lambda(e)$ to $\lambda'(e)$, then for any path P, the new length Π' of this path is

$$\begin{aligned}
\Pi' &= \sum_{e \in P} (m^{1/\epsilon})^{\lambda'(e)} \\
&= \sum_{e \in P} (m^{1/\epsilon})^{f(e)+\Delta f(e)} \\
&= \sum_{e \in P} (m^{1/\epsilon})^{\lambda(e)} m^{\Delta f(e)/\epsilon} \\
&\leq \sum_{e \in P} (m^{1/\epsilon})^{\lambda(e)} \left(1 + 2\frac{\log m}{\epsilon} \Delta f(e)\right).
\end{aligned} \tag{9.4}$$

Recall that initially $f_i(e) = \epsilon c(e)/k = \epsilon/L$, $\lambda(e) = \epsilon$ and $c_i(e) = \epsilon^2 f_i(e)/\log m = \epsilon^3/L \log m$.

We compute the change in the path length resulting from the initial augmentation of all commodities.

Initially

$$\Pi_{i,j}^{(1,0)} = \sum_{e \in P_{i,j}} (m^{1/\epsilon})^{\lambda(e)} = Lm. \tag{9.5}$$

During the first round, the total flow of all commodities sent along the shared part of the paths $P_{i,1}$ is $\epsilon^3/\log m$. Thus using (9.4) the length of paths $P_{i,1}$ increase to the following $\Pi_{i,1}^{(1,1)}$:

$$\begin{aligned}
\Pi_{i,1}^{(1,1)} &\leq \sum_{e \in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)} \left(1 + 2\frac{\log m}{\epsilon} \Delta f(e)\right) \\
&= \sum_{e \in P_{i,1}} \left((m^{1/\epsilon})^{\lambda(e)}\right) + 2(L-2)\frac{\log m}{\epsilon}\frac{\epsilon^3}{\log m} m + 4\frac{\epsilon^3}{L \log m} m & (9.6) \\
&\leq \sum_{e \in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)} + 2L\epsilon^2 m \\
&= \Pi_{i,1}^{(1,0)} + 2\epsilon^2 \Pi_{i,1}^{(0)} & (9.7) \\
&= (1 + 2\epsilon^2)\Pi_{i,1}^{(1,0)}. & (9.8)
\end{aligned}$$

In line (9.6), we consider separately the first two edges of path $P_{i,1}$, which have new flows of $\epsilon^3/(L \log m)$, and the remaining $L-2$ shared edges, which have new flows of $\epsilon^3/\log m$ (each commodity contributes flow $\epsilon^3/(L \log m)$). Equality (9.7) follows from (9.5).

Let $\Delta f^{(h,r)}(e)$ denote the total flow change on edge e at stage h at the end of round r. The flow of each commodity increases by a factor of $(1 + \epsilon^2/\log m)$ in each round in Stage 1 and thus at the end of any round r the total flow change on a shared edge e of path $P_{i,1}$ is

$$\begin{aligned}\Delta f^{(1,r)}(e) &= L\left(\left(1+\epsilon^2/\log m\right)^r \frac{\epsilon}{L} - \frac{\epsilon}{L}\right) \\ &= \epsilon\left((1+\frac{\epsilon^2}{\log m})^r - 1\right). \end{aligned} \tag{9.9}$$

Hence at any round r using (9.4) and (9.9),

$$\begin{aligned}\Pi_{i,1}^{(1,r)} &\leq \sum_{e\in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)}\left(1+2\frac{\log m}{\epsilon}\Delta f(e)\right) \\ &\leq \left(1+2\frac{\log m}{\epsilon}\Delta f^{(1,r)(e)}\right)\Pi_{i,1}^{(1,0)} \\ &= \left(1+2\frac{\log m}{\epsilon}\epsilon\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right)\Pi_{i,1}^{(1,0)} \\ &= \left(1+2\log m\left((1+\frac{\epsilon^2}{\log m})^r - 1\right)\right)\Pi_{i,1}^{(1,0)}. \end{aligned} \tag{9.10}$$

The above shows that the bound (9.3) holds.

A path is used as long as it is ϵ-approximate and the flow increase on a single edge has not exceeded $\epsilon^2/\log m$. Using the general formula for the increase in path length (9.3) and noting that initially the path is ϵ-approximate we can lower bound the number of rounds needed to increase its length by $(1+\epsilon)$ and thus make the path unavailable. Stage 1 has at least r rounds, where r is the largest integer such that

$$\Pi_{i,1}^{(1,r)} \leq (1+\epsilon)\Pi_{i,1}^{(1,0)}.$$

Thus, using (9.3), the number of rounds is at least the largest integer r such that

$$\left(1+2\log m\left((1+\frac{\epsilon^2}{\log m})^r - 1\right)\right)\Pi_{i,1}^{(1,0)} \leq (1+\epsilon)\Pi_{i,1}^{(1,0)}.$$

Thus the number of rounds needed to terminate stage S_1 is at least

$$\epsilon^{-2}\log\left(1+\frac{\epsilon}{2\log m}\right)\log m \geq \frac{\epsilon^{-1}}{4}. \tag{9.11}$$

The final bound follows from the assumption that $\epsilon \leq \log m$. □

In Lemma 9, we derived an upper bound on the length of path $P_{i,1}$. All other paths are also affected by augmenting flow on $P_{i,1}$, that is, all the paths $P_{i,j}$(for $j \neq 1$) and the exclusive path P_i. On these paths only the flow of one edge $(s_i, w_{i,1})$ is affected; see in Figure 9.2. Next lemma gives a bound on the length of these paths at the end of round r in Stage 1. This bound will be used in the proof of Lemma 12.

Lemma 10. *The lengths $\Pi_{i,j}^{(1,r)}$ of paths $P_{i,j}$(for $j \neq 1$) and the length $\Pi_i^{(1,r)}$ of the exclusive path P_i at the end of round r in stage 1 are*

$$\Pi_{i,j}^{(1,r)} = \Pi_i^{(1,r)} \leq \left(1+2\frac{\log m}{L^2}\left((1+\frac{\epsilon^2}{\log m})^r - 1\right)\right)\Pi_{i,j}^{(1,0)}. \tag{9.12}$$

Proof. In stage 1, the flow of commodity i is increased by a factor of $(1+\frac{\epsilon^2}{\log m})$ in each round. Since initially flow of commodity i is ϵ/L the total flow on the first edge $(s_i, w_{i,1})$ at the end of round r is $\epsilon(1+\frac{\epsilon^2}{\log m})^r/L$. Using (9.4), the lengths of paths $P_{i,j}$ and P_i increase by the end of round r to

$$\begin{aligned}\Pi_{i,j}^{(1,r)} = \Pi_i^{(1,r)} &\leq \sum_{e\in P_{i,j}} (m^{1/\epsilon})^{\lambda(e)}\left(1+2\frac{\log m}{\epsilon}\Delta f(e)\right)\\ &= \Pi_{i,j}^{(1,0)} + 2m\frac{\log m}{\epsilon}\cdot\left((1+\frac{\epsilon^2}{\log m})^r - 1\right)\frac{\epsilon}{L} &(9.13)\\ &= \Pi_{i,j}^{(1,0)} + 2m\frac{\log m}{L}\left((1+\frac{\epsilon^2}{\log m})^r - 1\right) &(9.14)\\ &= \left(1+2\frac{\log m}{L^2}\left((1+\frac{\epsilon^2}{\log m})^r - 1\right)\right)\Pi_{i,j}^{(1,0)} &(9.15)\end{aligned}$$

The equality (9.14) holds because the flow changes only on one edge $(s_i, w_{i,1})$ of the path, so $\Delta f(e) = 0$ for all other edges. The equality (9.15) holds because $\Pi_{i,j}^{(1,0)} = Lm$. □

To prove our main Lemma 12 we use induction. Since we need to maintain both the upper and lower bound on the path lengths we need to find the bounds for the first stage for both cases. The next lemma gives a lower bound on the path length and on the number of rounds needed to terminate stage S_1.

Lemma 11. *In Stage* 1 *the length of path* $P_{i,1}$ *at the end of round* r *is*

$$\Pi_{i,1}^{(1,r)} \geq \left(1 + \frac{L^2 - 2L + 2}{L^2} \log m \left((1 + \frac{\epsilon^2}{\log m})^r - 1\right)\right) \Pi_{i,1}^{(1,0)}, \tag{9.16}$$

and the number of rounds in this stage is

$$\Theta(\epsilon^{-1}),$$

and the lower bound on the length $\Pi_{i,j}^{(1,r)}$ *of the paths* $P_{i,j}$ *for* $j > 1$ *is*

$$\Pi_{i,j}^{(1,r)} = \Pi_{i}^{(1,r)} \geq \Pi_{i,j}^{(1,0)} + \tfrac{\log m}{L} \left((1 + \tfrac{\epsilon^2}{\log m})^r - 1\right),$$

Proof. The following result 9.17 holds for any flow change in a path and we prove this separately because we are going to use it repeatedly in our derivations later. If the flow on an edge e increases by $\Delta f(e)$, and the congestion increases from $\lambda(e)$ to $\lambda'(e)$, then for any path P, the new length Π' of this path is

$$\begin{aligned}
\Pi' &= \sum_{e \in P} (m^{1/\epsilon})^{\lambda'(e)} \\
&= \sum_{e \in P} (m^{1/\epsilon})^{f(e) + \Delta f(e)} \\
&= \sum_{e \in P} (m^{1/\epsilon})^{\lambda(e)} m^{\Delta f(e)/\epsilon} \\
&\geq \sum_{e \in P} (m^{1/\epsilon})^{\lambda(e)} \left(1 + \frac{\log m}{\epsilon} \Delta f(e)\right) \qquad (9.17)
\end{aligned}$$

Applying (9.17) to path $P_{i,1}$ and the first round of the first stage we get,

$$\begin{aligned}
\Pi_{i,1}^{(1,1)} &= \sum_{e\in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)} m^{\Delta f(e)/\epsilon} \\
&\geq \sum_{e\in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)} \left(1 + \frac{\log m}{\epsilon}\Delta f(e)\right) \qquad (9.18) \\
&= \sum_{e\in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)} + (L-2)\frac{\log m}{\epsilon}\frac{\epsilon^3}{\log m} m + 2\frac{\log m}{\epsilon}\frac{\epsilon^3}{L\log m} m \qquad (9.19) \\
&\geq \sum_{e\in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)} + \frac{(L^2-2L+2)}{L}\epsilon^2 m \\
&= \Pi_{i,1}^{(1,0)} + \frac{L^2-2L+2}{L^2}\epsilon^2\Pi_{i,1}^{(1,0)} \\
&= (1 + \frac{L^2-2L+2}{L^2}\epsilon^2)\Pi_{i,1}^{(1,0)} \qquad (9.20)
\end{aligned}$$

For (9.19), we use the changes of flow $\Delta f(e)$ as explained after the derivations (9.6)-(9.8).

Recall that $\Delta f^{(1,x)}(e)$ denotes the total change in flow at edge e up to round x in Stage 1. At any round r from (9.9) and (9.18) the path length change is given by

$$\begin{aligned}
\Pi_{i,1}^{(1,r)} &\geq \sum_{e\in P_{i,1}} (m^{1/\epsilon})^{\lambda(e)} \left(1 + \frac{\log m}{\epsilon}\Delta f^{(1,r)}(e)\right) \\
&= \left(1 + \frac{L^2-2L+2}{L^2}\frac{\log m}{\epsilon}\epsilon\left((1+\frac{\epsilon^2}{\log m})^r - 1\right)\right)\Pi_{i,1}^{(1,0)} \\
&= \left(1 + \log m\frac{L^2-2L+2}{L^2}\left((1+\frac{\epsilon^2}{\log m})^r - 1\right)\right)\Pi_{i,1}^{(1,0)} \qquad (9.21)
\end{aligned}$$

Thus (9.16) holds.

For the first stage to terminate the length of path $P_{i,1}$ must increase to $(1+\epsilon)\Pi_{i,1}^{(1,0)}$, that is

$$\Pi_{i,1}^{(1,r)} \geq (1+\epsilon)\Pi_{i,1}^{(1,0)}.$$

Substituting the bound from (9.16)we get that the length of path $P_{i,1}$ increases by a factor $(1+\epsilon)$ in t rounds, where

$$t \leq \left(\epsilon^{-2}\log m \log\left(1+\frac{L^2\epsilon}{2(L^2-2L+2)\log m}\right)\right) \leq \frac{3}{4}\epsilon^{-1}. \tag{9.22}$$

The last inequality above holds because $L \geq 7$ and $\epsilon \leq \log m$. From (9.11) and (9.22) we deduce that the first stage needs $\Theta(\epsilon^{-1})$ rounds to terminate.

Similarly as before (see Lemma 10) this change in the length of path $P_{i,1}$ affects all paths $P_{i,j} : j > 1$, and paths P_i. The change in flow affects one edge of these paths and hence the total change in length is given by

$$\Pi_{i,j}^{(1,r)} = \Pi_i^{(r)} \geq \Pi_{i,j}^{(1,0)} + \frac{\log m}{L}\left((1+\frac{\epsilon^2}{\log m})^r - 1\right) \tag{9.23}$$

□

The next lemma is essential for proving the lower bound on the running time of the DGD-MCF algorithm on network Υ_L. It gives us both the upper and lower bound on the number of rounds of a general Stage h. It also gives us un upper bound on the total flow change during the execution of the algorithm. Recall that $\Pi_i^{(h,r)}$ and $\Pi_{i,j}^{(h,r)}$ denote the lengths of paths P_i and $P_{i,j}$ respectively at the end of round r of stage h.

Lemma 12. *At the beginning of Stage h for $1 \leq h \leq L/3$, the length $\Pi_{i,j}^{(h,0)}$ of any path $P_{i,j}$, for $h \leq j \leq L$ and the length $\Pi_i^{(h,0)}$ of path P_i, that is, the length of any path that has not yet been chosen to augment flow is*

$$\Pi_{i,j}^{(h,0)} = \Pi_i^{(h,0)} \leq \left(1 + 2\frac{h-1}{L}\epsilon\right)\Pi_{i,j}^{(1,0)}. \tag{9.24}$$

The total change in flow of commodity i up to stage h is given by

$$\Delta f_i^{(h,0)} \leq \frac{3}{4}\frac{h-1}{L}\frac{\epsilon^2}{\log m}. \tag{9.25}$$

Finally, the number of rounds r_h in Stage h is

$$\frac{1}{6\epsilon} \leq r_h \leq \frac{3}{4\epsilon}. \tag{9.26}$$

Proof. To prove Lemma 12 we are going to use induction. We are first going to use the assumptions at the beginning of stage h (9.24) and (9.25) to show that the total number of rounds is bounded from above and below as in (9.26). Then using these bounds we are going to prove that at the beginning of stage $h+1$ the path length and total flow sent are given by (9.24) and (9.25).

Recall that $\Delta f^{(h,x)}(e)$ denotes the change(increase) of the total flow on edge e during the first x rounds of Stage h. As in (9.9), we have for each shared edge e,

$$\Delta f^{(h,x)}(e) = \epsilon\left((1+\frac{\epsilon^2}{\log m})^x - 1\right).$$

This is because the capacities start at the initial ϵ/L for each commodity at each edge and then they are saturated in each round(as it happened in stage 1 for path $P_{i,1}$; see the proof of Lemma 9). Observe that in any given stage the saturation will come from the shared part of the path being used. This happens because of the construction of the network Υ_L. The shared part of a path not yet being used has no flow, besides the initial artificial flow, in contrast to the exclusive edges of the paths that have being used in the previous stages. Thus, the capacities are tighter on the shared part of the path, and so the saturation happens at this part. Since we have L commodities sharing each edge we get the above change in flow in each round.

For all other edges the flow increases at most $\epsilon\left((1+\epsilon^2/\log m)^x - 1\right)/L$. As we want to find an upper bound on the path length $\Pi_{i,h}^{(h,x)}$ we use the change in flow in the shared part as an upper bound for the change in flow at the exclusive part. Using (9.4) the length of path $P_{i,h}$ at the end of round x within Stage h is bounded by (here $\lambda_h(e)$ is the congestion at edge e at the beginning of stage h)

$$\begin{aligned}\Pi_{i,h}^{(h,x)} &\leq \sum_{e\in P_{i,h}} (m^{1/\epsilon})^{\lambda_h(e)} \left(1+2\frac{\log m}{\epsilon}\Delta f^{(h,x)}(e)\right) \\ &\sum_{e\in P_{i,h}} (m^{1/\epsilon})^{\lambda(e)} \left(1+2\frac{\log m}{\epsilon}\epsilon\left((1+\frac{\epsilon^2}{\log m})^x-1\right)\right) \qquad (9.27) \\ &= \Pi_{i,h}^{(h,0)}\left(1+2\log m\left((1+\frac{\epsilon^2}{\log m})^x-1\right)\right) \\ &\leq \left(1+2\frac{h-1}{L}\epsilon\right)\left(1+2\log m\left((1+\frac{\epsilon^2}{\log m})^x-1\right)\right)\Pi_{i,h}^{(1,0)}. \qquad (9.28)\end{aligned}$$

Stage h ends when the length of path $P_{i,h}$ increases to $(1+\epsilon)\Pi_{i,h}^{(1,0)}$. The total number of rounds r_h needed to terminate stage h is the largest integer t for which

$$\Pi_{i,h}^{(h,t)} \leq (1+\epsilon)\Pi_{i,h}^{(1,0)}. \qquad (9.29)$$

Hence r_h is at least the largest integer t such that

$$\begin{aligned}&\left(1+2\frac{h-1}{L}\epsilon\right)\left(1+2\log m\left((1+\frac{\epsilon^2}{\log m})^t-1\right)\right) \\ &\qquad\qquad \leq \left(1+\frac{L-2h+2}{L}\epsilon-\epsilon^2\right)\left(1+2\frac{h-1}{L}\epsilon\right).\end{aligned}$$

The above inequality is equivalent to

$$\left(1+2\log m\left((1+\frac{\epsilon^2}{\log m})^t-1\right)\right) \leq \left(1+\frac{L-2h+2}{L}\epsilon-\epsilon^2\right),$$

since the RHS is less than $(1+\epsilon)$. In the following derivations each line implies the next one.

$$\begin{aligned}
&\left(1+\frac{\epsilon^2}{\log m}\right)^t \leq 1+\frac{1}{2\log m}\left(\frac{L-2h+2}{L}\epsilon-\epsilon^2\right)\\
&t\log\left(1+\frac{\epsilon^2}{\log m}\right) \leq \log\left(1+\frac{1}{2\log m}\left(\frac{L-2h+2}{L}\epsilon-\epsilon^2\right)\right)\\
&t \leq \frac{1}{\log\left(1+\frac{\epsilon^2}{\log m}\right)}\log\left(1+\frac{1}{2\log m}\left(\frac{L-2h+2}{L}\epsilon-\epsilon^2\right)\right)\\
&t \leq \frac{1}{\frac{\epsilon^2}{2\log m}}\left(\frac{1}{2\log m}\left(\frac{L-2h+2}{L}\epsilon-\epsilon^2\right)\right)\\
&t \leq \left(\frac{L-2h+2}{L}-\epsilon\right)\frac{1}{\epsilon}
\end{aligned}$$

This gives the following lower bound on r_h provided that $h \leq L/3$ and $\epsilon \leq 1/6$,

$$r_h \geq \left(\frac{L-2h+2}{L}-\epsilon\right)\frac{1}{\epsilon} \geq \frac{1}{6\epsilon},$$

as in (9.26).

Using similar arguments we are going to prove the upper bound on the number of rounds r_h in Stage h. From the derivations in (9.17) we have

$$\begin{aligned}
\Pi_{i,j}^{(h,x)} &\geq \sum_{e\in P_{h,j}} (m^{1/\epsilon})^{\lambda_h(e)}\left(1+\frac{\log m}{\epsilon}\Delta f^{(h,x)}(e)\right)\\
&= \sum_{e\in P_{h,j}} (m^{1/\epsilon})^{\lambda_h(e)} + \frac{\log m}{\epsilon}\sum_{e\in P_{h,j}} (m^{1/\epsilon})^{\lambda_h(e)}\Delta f^{(h,x)}(e)\\
&\geq \Pi_{i,j}^{(h,0)} + \frac{\log m}{\epsilon}(L-h-1)\sum_{e\in P_{h,j}} (m^{1/\epsilon})^{\lambda_1(e)}\Delta f^{(h,x)}(e)\\
&\geq \Pi_{i,j}^{(1,0)} + (L-h-1)\frac{\log m}{\epsilon}\epsilon\left(\left(1+\frac{\epsilon^2}{\log m}\right)-1\right)m\\
&= \left(1+\frac{L-h-1}{L}\log m\left(\left(1+\frac{\epsilon^2}{\log m}\right)-1\right)\right)\Pi_{i,j}^{(1,0)}.
\end{aligned}$$

To find the upper bound on the number of rounds r_h in stage h we need to find the smallest integer x such that

$$\Pi_{i,j}^{(h,x)} \geq (1+\epsilon)\Pi_{i,j}^{(1,0)}.$$

Hence r_h is at most the smallest integer x such that

$$1 + \frac{L-h-1}{L}\log m \left(\left(1+\frac{\epsilon^2}{\log m}\right)^x - 1\right) \geq 1+\epsilon$$

As before each inequality implies the inequality in the subsequent line.

$$\begin{aligned}
&\frac{L-h-1}{L}\log m \left(\left(1+\frac{\epsilon^2}{\log m}\right)^x - 1\right) \geq \epsilon \\
&\left(\left(1+\frac{\epsilon^2}{\log m}\right)^x - 1\right) \geq \frac{L}{(L-h-1)\log m}\epsilon \\
&\left(1+\frac{\epsilon^2}{\log m}\right)^x \geq 1 + \frac{L}{(L-h-1)\log m}\epsilon \\
&x \geq \frac{1}{\log\left(1+\frac{\epsilon^2}{\log m}\right)}\log\left(1+\frac{L}{(L-h-1)\log m}\epsilon\right) \\
&x \geq \frac{1}{\frac{\epsilon^2}{\log m}}\frac{L}{2(L-h-1)\log m}\epsilon \\
&x \geq \frac{L}{2(L-h-1)}\frac{1}{\epsilon}.
\end{aligned}$$

This gives us the upper bound on the number of rounds r_h for $h \leq L/3$,

$$r_h \leq \frac{L}{2(L-h-1)}\frac{1}{\epsilon} \leq \frac{3}{4\epsilon}.$$

as in (9.26).

Using the upper bound on the number of rounds in stage h we can bound the total flow sent on paths $P_{i,j}$ for $j \geq h$. The flow sent during stage h is given by

$$\begin{aligned} f_i^h &= \frac{\epsilon}{L}\left((1+\frac{\epsilon^2}{\log m})^{r_h} - 1\right) \\ &< \frac{\epsilon}{L}\frac{3\epsilon}{4\log m} \\ &= \frac{3\epsilon^2}{4L\log m}. \end{aligned} \tag{9.30}$$

Adding this to the assumption (9.25) gives us the required upper bound on the total flow change up to Stage h.

$$\begin{aligned} \Delta f_i^{(h+1,0)} &= f_i^h + \Delta f_i^{(h,0)} \\ &\leq \frac{3}{4}\frac{h-1}{L}\frac{\epsilon^2}{\log m} + \frac{3\epsilon^2}{4L\log m} \\ &= \frac{3h\epsilon^2}{4L\log m}. \end{aligned}$$

To prove the upper bound on the path length of the exclusive path Π_i and of the shared the paths $P_{i,j}$ for $j \geq h+1$ we use the derivations (9.4),

$$\begin{aligned} \Pi_{i,j}^{(h+1,0)} &\leq \sum_{e\in P_{i,j}} (m^{1/\epsilon})^{\lambda_1(e)}\left(1 + 2\frac{\log m}{\epsilon}\Delta_i^{(h+1,0)}(e)\right) \\ &\leq \sum_{e\in P_{i,j}} (m^{1/\epsilon})^{\lambda_1(e)}\left(1 + 2\frac{\log m}{\epsilon}\frac{3h\epsilon^2}{4L\log m}\right) \\ \leq \left(1+2\frac{h}{L}\epsilon\right)\Pi_{i,j}^{(1,0)}. \end{aligned}$$

Thus (9.24) holds for $h+1$. This completes the proof of Lemma 12. □

Using Lemma 12 we now establish the number of rounds of the DGD-MCF algorithm on the network Υ_L.

Theorem 11. *One phase of the DGD-MCF algorithm applied to network Υ_L requires $\Omega(L\epsilon^{-1})$ rounds to terminate for each commodity i.*

Proof of Theorem 11. A phase ends when we send $\epsilon^2/\log m$ flow of commodity i or there is no approximate shortest path available. We are going to prove that up to Stage h, $h \leq L/3$ the stoping requirements have not yet been met.

Up to Stage $L/3$, using our upper bound on the length of the path $\Pi_{i,j}^{(h,0)}$, for all $j > L/3$ we have

$$\Pi_{i,j}^{(L/3,0)} \leq \left(1 + 2\frac{\frac{L}{3}}{L}\epsilon\right)\Pi_{i,j}^{(1,0)} \tag{9.31}$$

$$\leq \left(1 + \frac{2\epsilon}{3}\right)\Pi_{i,j}^{(1,0)}. \tag{9.32}$$

Hence all the paths with index $j > h$ are still approximate shortest paths and thus can be used for augmentation.

The flow sent in each stage up to stage h for $h \leq L/3$ is given by

$$\begin{aligned} f_i^h &= \frac{\epsilon}{L}\left(\left(1 + \frac{\epsilon^2}{\log m}\right)^{r_h}\right) \\ &\leq \frac{\epsilon}{L}\left(\left(1 + \frac{\epsilon^2}{\log m}\right)^{\frac{3}{4\epsilon}}\right) \\ &= \frac{3}{4}\frac{\epsilon^2}{L\log m}. \end{aligned}$$

Hence for h stages and for $h \leq L/3$ the total flow sent is

$$\begin{aligned} \Delta f_i^{(h,0)} &\leq h\frac{3}{4}\frac{\epsilon^2}{L\log m} \\ &\leq \frac{\epsilon^2}{4\log m}. \end{aligned}$$

Since there still exist approximate shortest paths to augment flow and we have not yet sent the required flow, we conclude that the phase after $L/3$ stages has not yet been terminated. It follows from the fact that in each stage the number of rounds is $\Omega(\epsilon^{-1})$ that the number of rounds to terminate a phase is at least $\Omega(L\epsilon^{-1})$.

□

We have proved above that the number of rounds needed to terminate phase 1 of the DGD-MCF agorithm on network Υ_L is at least $\Omega(L\epsilon^{-1})$. This result shows that the upper bound of the DGD-MCF algorithm proposed in [7] is actually tight, up to polylogarithmic factors. In the next section we propose a heuristic improvement of the DGD-MCF algorithm and show that it significantly reduces the number of rounds, at least on network Υ_L.

9.3 Balancing Distributed MCF algorithm

The running time of the worst-case computation of the DGD-MCF algorithm on Υ_L linearly depends on L, as proved in Section 9.2.1, even in the simple case of unit capacities. The bottleneck appears due to the fact that most of the approximate shortest paths cannot be used in one round since they are blocked by the flow sent on the prior paths. This causes a very slow increase in the flow augmented on these paths when they are visited later on and thus a direct linear dependence of the running time to the maximum path size L.

The analysis of the execution of the DGD-MCF algorithm on network Υ_L suggests that by spreading the flow evenly on all available approximate shortest paths we could avoid this bottleneck and thus decrease the dependence on L. Below we present a way of augmenting the flow under the Distributed Gradient Descent algorithm framework that improves the running time of the algorithm by removing the polynomial dependence on L.

The key to our improvement lies in the calculation of the blocking flow, which tries to distribute flow to all available approximate shortest paths. This results in a much more aggressive increase of the flow in the subsequent iterations of the algorithm, which leads to a reduced total running time of the algorithm on network Υ_L. We do not know whether this always improves the running time in general, so we treat this algorithm as a heuristic. However, as an indication of the potential of this heuristic, we show that the proposed distribution on flow reduces the running time from $\widetilde{O}(Lm^2)$ to $\widetilde{O}(m^2)$ for the first phase on network Υ_L. In other words, this heuristic reduces the linear dependence on the parameter L to poly-logarithmic for this part of the computation.

Note that if all available approximate shortest paths were to be considered explicitly, in the process of distributing the flow, then we might get an exponential running time due to the number of paths. We overcome this problem by computing only one "maximum capacity" approximate shortest path for each edge of the network. Then we route the flow along the computed paths, scaling it down by the number of shortest paths found for each commodity. This way we ensure that we have a feasible solution, that is within the capacities, and at the same time we send at least $c(P_i)/q$ units of flow, where $c(P_i)$ is the capacity of path P_i and $q \leq m$ is the number of computed approximate shortest paths. We show how our algorithm handles the approximate shortest paths available that were not discovered by our search. In fact we are going to argue that even in these paths we send at least $c(P_i)/q$ units of flow.

The properties of the blocking flows which are used in the analysis of the running time of the DGD-MCF algorithm also hold for the blocking flows selected in the BD-MCF algorithms. This means that the upper bound for the DGD-MCF algorithm given in Theorem 8 applies also for the BD-MCF algorithm. We also show that one round of the BD-MCF algorithm can be implemented to have the same running time bound (up to a logarithmic factor) as one round of the DGD-MCF algorithm.

The remainder of the section is divided into two main parts. In Subsection 9.3.1 we describe our Balancing Distributed Multicommodity Flow algorithm (BD-MCF algorithm), and provide the pseudocode for it. In Subsection 9.3.2 we analyze the running time of the BD-MCF algorithm on our worst-case network Υ_L.

9.3.1 Description of the BD-MCF algorithm

Our BD-MCF algorithm fits in the framework of the Distributed Gradient Descent algorithm in [7]. It differs only in the way the blocking flows are calculated.

The overall structure of the BD-MCF algorithm is the same as the DGD-MCF algorithm. Initially we route a small amount of flow of all commodities on all edges. Then the algorithm proceeds in phases each one consisting of a number of rounds. During each phase, each agent i has to route $\epsilon^2 d_i / \log m$ flow of commodity i. Each round starts by setting the capacities to be $c_i(e) = \epsilon^2 f_i(e) / \log m$, where $f_i(e)$ is the current flow of commodity i on edge e. Then we compute a blocking flow under the current capacities and route the computed flow.

To compute the blocking flow, we first call the Maximum Capacity Path algorithm (MCP-algorithm) to get the set of edges $\mathcal{F}$, and the set of paths $\mathcal{P} = \cup_{e\in\mathcal{F}}\{P_e^{SP}\}$, where $\mathcal{F}$ denotes the set of edges for which an approximate shortest path passing through them exists, and P_e^{SP} denotes an approximate shortest path passing through e. Let $c(P_e^{SP})$ denote the capacity of path P_e^{SP}. Then we let flow $\widetilde{g}_i$ be the sum of the flows of values $c(P_e^{SP})/|\mathcal{F}|$ along the paths P_e^{SP}. Scaling the flows down by a factor $|\mathcal{F}|$ ensures that $\widetilde{g}_i$ is a feasible flow for the edge capacities $c_i(e)$. If the calculated flow $\widetilde{g}_i$ is not a blocking flow under the current capacities, we increase the flow using available approximate shortest paths. We stop when we cannot find an approximate shortest path with a residual capacity. Below we give the pseudo-code of the BD-MCF algorithm.

Algorithm 5: Balancing Distributed MCF algorithm (One phase)

```
Computation for Agent $i$ (controlling commodity $i$ );
while commodity $i$ has not yet routed $\epsilon^2 d_i/\log m$ flow do
    1. Set the lengths of the edges as in (8.5);
    2. Set the capacities to be $c_i(e) = \epsilon^2 f_i(e)/\log m$;
    3. Compute a blocking flow ;
    3a. Call Maximum Capacity Path algorithm to get the set of edges
    $\mathcal{F} \subseteq \mathcal{E}$ and the set of paths $\mathcal{P} = \{P_e^{SP} : e \in \mathcal{F}\}$;
    3b. Let $c_i(P_e^{SP})$ be the capacity of path $P_e^{SP}$ ;
    3c. Let $\widetilde{g}_i$ be the updating flow from $s_i$ to $t_i$ obtained by sending
    $c(P_e^{SP})/|\mathcal{F}|$ flow along path $P_e^{SP}$, for each $e \in \mathcal{F}$ ;
    3d. Increase $g_i$ to a blocking flow $g_i$, which uses only $\epsilon$-approximate
    shortest paths: start with $g_i = \widetilde{g}$ and while there is an $\epsilon$-approximate
    shortest path $P$ with residual capacity, increase $g_i$ by saturating path
    $P$;
    4. Route the computed blocking flow $g_i$;
end
```

The MCP algorithm works in the following way. The input is a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with n nodes and m edges with capacity $c_i(e)$ and length $l(e)$. We initialize the algorithm by finding the shortest path P^{SP} from s_i to t_i and its cost Π^{SP} using Dijkstra's algorithm. Then for each edge $e = (u, v) \in \mathcal{E}$ we calculate the shortest path from s_i to t_i which passes through that edge. To do this we call Dijkstra's algorithm twice, once to calculate the shortest path from s_i to u and once from v to t_i. Let x denote the capacity of the computed path. If the length of the path is greater than $1 + \epsilon$ times the shortest path length Π^{SP}, that is, the path is not an approximate shortest path from s_i to t_i, then we move to the next edge. Otherwise we add e to $\mathcal{F}$ and find the maximum capacity approximate shortest path through this edge. To do this we perform a binary search between the capacity of the shortest path x and the upper bound $y = x \cdot m^{2/\epsilon+1}$ on the capacity of any path that can pass through e. One iteration of the binary search checks whether there is an ϵ-approximate shortest path with capacity greater than $z = \frac{x+y}{2}$ which passes through e. This is done in the following way: we remove all the edges with capacities less than z and calculate a shortest path from s_i to t_i through e (using the remaining edges). If such a path exists and it is ϵ-approximate shortest path, we set $x = z$ and P_e^{SP} to this path. If it does not exist we set $y = z$. We keep repeating this procedure until the interval between x and y becomes less than ϵx. Then we add P_e^{SP} to $\mathcal{P}$ and proceed to the next edge. The pseudocode for the Maximum Capacity Path algorithm is given below.

```
Algorithm 6: Maximum Capacity Path algorithm
Input: Graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $n$ nodes, $m$ edges with capacity $c_i(e)$ and length
        $l(e)$
Output: Set of edges $\mathcal{F} \subseteq \mathcal{E}$, and set of paths $\mathcal{P} = \{P_e^{SP} : e \in \mathcal{F} \subseteq \mathcal{E}\}$
1. Initialize $\mathcal{F} \leftarrow \emptyset$ and $\mathcal{P} \leftarrow \emptyset$;
2. Compute the shortest paths from $s_i$ to all other nodes and from all
nodes to $t_i$ (two Dijkstra's algorithm computations);
3. Let the shortest path $P^{SP}$ from $s_i$ to $t_i$;
4. Set $\Pi^{SP}$ to be the shortest path length from $s_i$ to $t_i$;
for each edge $e = (u, v) \in \mathcal{E}$ do
    Let $P_e^{SP}$ be the shortest path from $s_i$ to $t_i$ passing through $e$ (This is a
    shortest path from $s_i$ to $u$ and from $v$ to $t_i$);
    if $\Pi_e^{SP} \leq (1 + \epsilon)\Pi^{SP}$ then
        1. $F \leftarrow \mathcal{F} \cup \{e\}$;
        2. Set $x = c(P_e^{SP})$, $y = x \cdot m^{1/\epsilon+1}$;
        while $y - x > \epsilon x$ do
            3. Set $z = \frac{x+y}{2}$;
            4. Let $\widetilde{G}$ be the graph resulting from removing from $G$ all edges
            with capacity $< z$;
            5. Find a shortest path $\widetilde{P}_e^{SP}$ in $\widetilde{G}$ from $s_i$ to $t_i$ which passes
            through edge $e$;
            if such path exists and its length is $\leq (1 + \epsilon)\Pi^{SP}$ then
                1. $P_e^{SP} \leftarrow \widetilde{P}_e^{SP}$;
                2. $x \leftarrow z$;
            end
            else
                $y \leftarrow z$;
            end
        end
        5. $\mathcal{P} \leftarrow \mathcal{P} \cup \{P_e^{SP}\}$;
    end
end
```

We show below the validity of the upper bound and the stopping condition for the binary search.

Lemma 13. *Let the length of the shortest path P^{SP} through edge e be Π^{SP} and let its bottleneck capacity be x. Also let the length of the maximum capacity path P_e^{SP} passing through e be Π_e^{SP} and similarly let its capacity be y. Then the maximum capacity of path P_e^{SP} is bounded above by*

$$y \leq x \cdot m^{1/\epsilon+1}$$

Proof. From the definition of the shortest path P^{SP} and the maximum capacity path P_e^{SP} we have

$$\Pi^{SP} \leq \Pi_e^{SP}.$$

From the definition of the path length we get the following inequalities

$$\begin{aligned} \Pi^{SP} = \sum_{e \in P^{SP}} \frac{(m^{1/\epsilon})^{\lambda(e)}}{c(e)} &\geq \frac{(m^{1/\epsilon})^{\lambda(x)}}{x} \\ &\geq \frac{1}{x}. \end{aligned}$$

Also

$$\begin{aligned} \Pi_e^{SP} = \sum_{e \in P_e^{SP}} \frac{(m^{1/\epsilon})^{\lambda(e)}}{c(e)} &\leq m \cdot \frac{(m^{1/\epsilon})^{\lambda(y)}}{y} \\ &\leq \frac{m^{1/\epsilon+1}}{y}. \end{aligned}$$

From the above equations we get our bound for the maximum capacity path

$$\begin{aligned} \frac{1}{x} &\leq \frac{m^{1/\epsilon+1}}{y} \\ \Rightarrow y &\leq x \cdot m^{1/\epsilon+1}. \end{aligned}$$

□

Corollary 3. *If the stopping condition of the MCP algorithm is met the output maximum capacity path for each edge e is ϵ-approximate.*

Proof. The correctness of our stopping condition lies in the fact that the capacity of the shortest path through an edge e is less than or equal to the maximum capacity of any path passing through e. Moreover, our upper and lower bounds for the capacity from the MCP algorithm are within an ϵx additive factor from the optimal solution. Therefore the output capacity z lies in the range

$$y - \epsilon y \leq z \leq y + \epsilon y,$$

i.e. our solution is ϵ-approximate. □

Below we show that the running time of one round of the BD-MCF algorithm is not much higher than the running time of one round of the DGD-MCF algorithm. To show this we calculate the time needed to distribute a blocking flow across approximate shortest paths which determines one round.

Lemma 14. *One round of the BD-MCF algorithm terminates in $\widetilde{O}(\frac{m^2}{\epsilon})$ time.*

Proof. We consider the computation time of the MCP algorithm. Initially Dijkstra's algorithm is performed to calculate the shortest path from s_i to t_i. Let $e = (u, v)$ be the current edge considered during the computation of the MCP algorithm. One iteration of the binary search requires two calls to Dijkstra's algorithm, one to find a shortest path from the source s to u and one to find a shortest path from v to the destination t in graph $\widetilde{G}$. This computation takes $\widetilde{O}(m)$ time. The binary search is performed over the range $[x, x \cdot m^{1/\epsilon+1}]$ and stops when the search interval becomes less than ϵx. Thus the number of iterations of the binary search is $O(\log(\frac{x \cdot m^{2/\epsilon+1} - x}{\epsilon \cdot x})) = O(\epsilon^{-1} \log m + \log(\epsilon^{-1})) = O(\epsilon^{-1} \log m)$. Finally, since we consider each edge we need to perform at most m binary searches. Thus one round requires $O(\epsilon^{-1} m \log m)$ calls of Dijkstra's algorithm, so it terminates in $\widetilde{O}(m^2 \epsilon^{-1})$ overall time. □

9.3.2 Execution of the BD-MCF Algorithm on Υ_L

In this section we analyze the execution of our BD-MCF algorithm on the network Υ_L. We show that the $\widetilde{O}(L\epsilon^{-1})$ bound on the number of rounds in the first phase of the DGD-MCF algorithm improves to a polylogarithmic bound $\widetilde{O}(\epsilon^{-2} \log^2 m)$

in the BD-MCF algorithm. We don't know the running time of the BD-MCF algorithm in a general network so, we treat the algorithm as a heuristic. We show however that in our worst-case input it is significantly faster than the DGD-MCF algorithm.

The crucial improvement our algorithm achieves, with respect to the DGD-MCF algorithm, is that by distributing the flow across all available approximate shortest paths, we circumvent the bad case we had before that most of the available paths remain empty for a significant amount of time. This way we ensure that the increase of the flow in later stages will be much more aggressive, thus reducing the running time. It is essential for our proof to show that the flow keeps increasing on all paths in each round. Our algorithm ensures that we have a relatively balanced distribution of the blocking flow along all available approximate shortest paths. This way we ensure that we have two effects:

1. The more congested paths stay available for much longer than previously.

2. When the congested paths become unavailable the rest of the available approximate shortest paths have already a sufficient amount of flow to carry out a more aggressive increase of flow.

More specifically, we prove the bound on the number of rounds in the first phase given in the following lemma.

Lemma 15. *The first phase of the Balancing Distributed Multicommodity Flow Algorithm on Υ_L, which delivers $\Theta(\epsilon^2/\log m)$ fraction of each commodity, terminates in $O(\epsilon^{-2}\log m \log(L\epsilon/\log m))$ rounds.*

To prove this bound we proceed in a similar fashion to the previous section. First, we look at the first iteration only and establish a lower bound on the length of paths at the end of this iteration. Then we generalize this bound to the subsequent rounds in the first phase and use it to prove Lemma 15.

We can consider the flow of only one commodity i, since the network is symmetrical from the point of view of each commodity, meaning that the subnetworks for individual commodities look the same (see Figure 2.2). We do, however, need to take into account the flows of all commodities when we estimate the lengths of edges. Initially $f_i(e) = \epsilon c(e)/k = \epsilon/L$, $\lambda(e) = \epsilon$ and

$c_i(e) = \epsilon^2 f_i(e)/\log m = \epsilon^3/L \log m$. As in Section 9.2.1, $\Pi_{i,j}^{(1,r)}$ denotes the length of path $P_{i,j}$ after round r at Stage 1.

We first look at the change of flow and path length in the first round. This should be helpful in following more detailed derivations we will need when considering a general round r. The total number of available approximate shortest paths q in the first round is equal to

$$q = \sum_{j=2}^{j=L} j = \frac{(L-1)(L+2)}{2}, \tag{9.33}$$

which is the number of the edges in the part of the network available to this commodity (as shown in Figure 9.2).

In the first round, the $\widetilde{g}_i$ flow calculated in line 3c of Algorithm 7 is actually a blocking flow. This happens because initially all paths have the same capacity, they all are shortest paths and they all pass through the first edge (s_i, w_i) (see Figure 9.2). Thus each agent sends $\epsilon^3/(qL\log m)$ flow to each path. Observe that these q paths can be combined to form the L paths $P_{i,j}$ as defined in Section 9.2. It is easy to observe that the approximate shortest paths calculated for the shared edges of $P_{i,j}$ are in fact the same path $P_{i,j}$. We can assume that for each exclusive edge $(w_{i,j}, w_{i,j+1})$, the computed approximate shortest path is $P_{i,j+1}$. This is because by making this assumption we "force" the paths which congest faster, i.e. the "top" paths in Figure 9.2), to even faster congestion, obtaining the worst case behavior. The change in length of path $P_{i,j}$ at the end of round r is bounded by (see (9.17)):

$$\begin{aligned}\Pi_{i,j}^{(1,r)} &\geq \sum_{e\in P_{i,j}} (m^{1/\epsilon})^{\lambda(e)} \left(1 + \frac{\log m}{\epsilon} \Delta f^{(1,r)}(e)\right) \\ &= \Pi_{i,j}^{(1,0)} + \sum_{e\in P_{i,j}} m \left(\frac{\log m}{\epsilon} \Delta f^{(1,r)}(e)\right),\end{aligned} \tag{9.34}$$

where $\Delta f^{1,r}(e)$ denotes the total change in flow up to round r. We remind that $\Pi_{i,j}^{(1,0)} = Lm$, and initially $\lambda(e) = \epsilon$.

In the first round the capacity of each path P_e^{SP} computed by MCP algorithm is

$$c(e) = \frac{\epsilon^3}{L \log m} \tag{9.35}$$

Our algorithm scales this flow so that it becomes feasible by dividing by the total number of approximate shortest paths q. The path $P_{i,j}$ in network Υ_L consists of $L - j - 1$ "shared" edges and $j + 1$ "exclusive edges", as defined in Section 9.2. The flow change (increase) of one commodity on each edge e of the shared part of the path $P_{i,j}$ is

$$\Delta f_i^{(1,r)}(e) = \frac{L - j + 1}{q} c(e). \tag{9.36}$$

This holds because in the MCP algorithm the path $P_{i,j}$ is assigned to $L - j + 1$ edges: $L - j - 1$ shared edges of $P_{i,j}$ and 2 edges, the exclusive edge $(w_{i,j-1}, w_{i,j})$ and the connecting edge $(w_{i,j}, v_{j,j+1})$ from the exclusive part of $P_{i,j}$. For each edge e of the shared part of $P_{i,j}$, all L commodities send the same flow (9.35), so the total increase of the flow on e is equal to

$$\Delta f^{(1,r)}(e) = \frac{L(L - j + 1)}{q} c(e). \tag{9.37}$$

The path also has $j + 1$ "exclusive" edges in which the flow changes due to one commodity only. For the lower bound of the path length we can simply ignore this change. Hence from (9.34), (9.36) and (9.37) the total change in path length in the first round is bounded as

$$\begin{aligned}
\Pi_{i,j}^{(1,1)} &\geq Lm + (L - j + 1)(L - j - 1)\frac{\log m}{\epsilon}\frac{\epsilon^3}{L \log m}\frac{L}{q}m \\
&= \left(1 + \frac{(L - j + 1)(L - j - 1)}{L}\frac{\epsilon^2}{q}\right) Lm \qquad (9.38) \\
&= \left(1 + \frac{(L - j + 1)(L - j - 1)}{L}\frac{\epsilon^2}{q}\right) \Pi_{i,j}^{(1,0)}. \qquad (9.39)
\end{aligned}$$

To prove Lemma 15 we need to find a lower bound on the total flow sent in each round. This will enable us to find un upper bound on the total number of rounds needed to terminate a phase. We also need to find a lower bound on the path length increase so that we ensure that the blocking flow is given by the expression in Claim 3. Notice that to find the upper bound on the number of rounds needed for a phase to terminate we need to maintain both the upper and lower bounds on the flow change in the network (Lemma 4).

Before we proceed with our proof we need to set up some notation that we are going to use. We are going to drop the index i for convenience since our results for a single commodity hold for every commodity. Hence, P_j stands for the jth path $P_{i,j}$ and $f_j^{(r)}$ is the flow of one commodity on path P_j at the end of round r. Let $c^{(r)}(e)$ denote the capacity of edge e at the beginning of round r. The capacity of path P_j at the beginning of round r is denoted by $c^{(r)}(P_j)$ and is equal to the capacity $c^{(r)}(e)$ of all the "shared" edges e of path P_j. This holds because the capacity of path P_j comes from the capacity of the edges of the "shared" part which is the minimum over all the edges of the path. Let $\delta f_j^{(r)}$ denote the amount of flow of one commodity calculated by the MCP-algorithm to be sent on path P_j at time round r, and let $\delta f^{(r)}$ denote the total amount of flow of this commodity calculated by the MCP-algorithm to be sent at this round. Also let $\Delta f_j^{(r)}$ denote the actual amount of flow of one commodity sent on path P_j in this round after the flow is scaled to become a blocking flow. Observe that in round 1, $\delta f_j^{(1)} = \Delta f_j^{(1)}$ as discussed previously. For any round $r \geq 2$, $\delta f_j^{(1)} \leq \Delta f_j^{(1)}$. Finally let $f_{BF}^{(r)}$ be the value of blocking flow sent in round r.

Claim 3. *For as long as all paths in Υ_L are ϵ-approximate shortest paths, the blocking flow of one commodity computed in round r saturates edge $(s_i, w_{i,1})$, and its value is given by the following expression*

$$f_{BF}^{(r)} = \frac{\epsilon^3}{L \log m} \left(1 + \frac{\epsilon^2}{\log m}\right)^{r-1}. \tag{9.40}$$

Proof. Our claim states that the blocking flow at any round r is the flow that saturates edge $(s_i, w_{i,1})$. Consider the flow at the beginning of round r. Recall that $f_j^{(r)}$ is the flow of path P_j at the end of round r, and $F_j^{(r)}$ be the flow on the "exclusive" edge $(w_{i,j-1}, w_{i,j})$ at the end of round r. Initially the flow on every edge was ϵ/L, so

$$f_j^{(r)} = \frac{\epsilon}{L} + \sum_{s=1}^{r} \Delta f_j^{(s)} \tag{9.41}$$

and

$$F_j^{(r)} = \frac{\epsilon}{L} + \sum_{x=j}^{L-1} \sum_{s=1}^{r} \Delta f_x^{(s)} \tag{9.42}$$

The formula (9.42) holds because the flow on an edge of the exclusive part is equal to the total flow sent on the paths that pass through that edge by construction of the network Υ_L. The capacity of path P_j in round r is determined by the flow $f_j^{(r)}$ on the "shared" part of the path, that is for each j:

$$c^{(r)}(P_j) = \frac{\epsilon^2}{\log m} \min \left\{ f_j^{(r)}, F_j^{(r)}, F_{j-1}^{(r)}, \ldots, F_1^{(r)} \right\} = \frac{\epsilon^2}{\log m} f_j^{(r)}. \tag{9.43}$$

This is straightforward because for $x = 1, 2, \ldots, j$ the value $F_x^{(r)}$ is at least $f_j^{(r)}$; see (9.41) and (9.42).

A blocking flow saturates at least one edge on each $s - t$ path. Let C_j be a $s - t$ cut obtained by removal of the "exclusive" edge $(w_{i,j-1}, w_{i,j})$ and one (arbitrary) edge from the "shared" part of the each of the paths $P_1, P_2, \ldots, P_{j-1}$. The capacity of the cut C_j is :

$$\frac{\epsilon^2}{\log m} \left(F_j^{(r)} + \sum_{i=1}^{j-1} f_j^{(r)} \right). \tag{9.44}$$

A flow saturates at least one edge of each path if and only if it saturates one of the cuts $C_1, C_2, \ldots, C_L$. Hence the value of the blocking flow on any round r is given by

$$f_{BF}^{(r)} = \frac{\epsilon^2}{\log m} \min \left\{ \sum_{i=1}^{L-1} f_i^{(r)}, \sum_{i=1}^{L-2} f_i^{(r)} + F_{L-1}^{(r)}, \ldots, f_1^{(r)} + F_2^{(r)}, F_1^{(r)} \right\} \tag{9.45}$$

Then using (9.41) and (9.42) we have (setting $F_L^{(r)} \equiv 0$)

$$\begin{aligned}
f_{BF}^{(r)} &= \frac{\epsilon^2}{\log m} \min_j \left\{ F_j^{(r)} + \sum_{i=1}^{j-1} f_j^{(r)} \right\} \\
&= \frac{\epsilon^2}{\log m} \min_j \left\{ \sum_{i=1}^{j-1} \left(\frac{\epsilon}{L} + \sum_{s=1}^{r} \Delta f_i^{(s)} \right) + \frac{\epsilon}{L} + \sum_{i=j}^{L-1} \sum_{s=1}^{r} \Delta f_i^{(s)} \right\} \\
&= \frac{\epsilon^2}{\log m} \min_j \left\{ \frac{j\epsilon}{L} + \sum_{i=1}^{L-1} \sum_{s=1}^{r} \Delta f_i^{(s)} \right\} \\
&= \frac{\epsilon^2}{\log m} \left\{ \frac{\epsilon}{L} + \sum_{i=1}^{L-1} \sum_{s=1}^{r} \Delta f_i^{(s)} \right\} \\
&= \frac{\epsilon^2}{\log m} F_1^{(r)}. \qquad (9.46)
\end{aligned}$$

Hence the blocking flow at round r is the flow which saturates edge $(s_i, w_{i,1})$.

The flow on edge $(s_i, w_{i,1})$ is initially ϵ/L and since it is saturated in every round x up to round r, then the flow and the capacity of this edge increase in each round by a factor of $(1 + \frac{\epsilon^2}{\log m})$. Therefore, the capacity of edge $(s_i, w_{i,1})$ at the beginning of round r is

$$c^{(r)}(s_i, w_{i,1}) = \frac{\epsilon^3}{L \log m} \left(1 + \frac{\epsilon^2}{\log m}\right)^{r-1}.$$

Since the blocking flow saturates edge $(s_i, w_{i,1})$ its value at the beginning of round r is given by the above expression. □

Claim 3 gives us the exact increase of flow of one commodity in any round r. We need also the flows on individual paths P_j, or at least good lower and upper bounds on these flows, to estimate their lengths. These bounds are essential to prove our main Lemma 15.

Claim 4. *The flow of one commodity on any path P_j, $j = 1, 2, ..., L-1$ at the end of round r, while all paths are ϵ-approximate shortest paths and $r \leq \epsilon^{-2} \log m \log(L/4)$, has the following upper and lower bounds:*

$$f_j^{(r)} \leq \frac{\epsilon}{L} \left(1 + 2\frac{L}{q} \left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1 \right) \right) \qquad (9.47)$$

and

$$f_j^{(r)} \geq \frac{\epsilon}{L}\left(1+\frac{1}{4}\frac{L-j+1}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right). \tag{9.48}$$

Proof. We are going to prove these bounds by induction. Initially the flow is set so that $f_j^{(0)} = \epsilon/L$. Hence, the allowed increase of flow on any edge in the first round is

$$c^{(1)} = \frac{\epsilon^2}{\log m} f_j^{(0)} = \frac{\epsilon^3}{L \log m}.$$

Using the expression for the flow change in the first round (9.36) we get

$$\delta f_j^{(1)} = \frac{L-j+1}{q} c_j^{(1)} = \frac{L-j+1}{q}\frac{\epsilon^3}{L\log m}$$

In the first round the updating flow is actually the blocking flow(no scaling is needed, so $\delta f_j^{(1)} = \Delta f_j^{(1)}$). This happens because all the edges have initially the same capacity. We note that this will not hold for subsequent rounds. Later on the updating flow will have to be scaled up to become a blocking flow. Thus the total flow in path P_j at the end of the first round is

$$f_j^{(1)} = f_j^{(0)} + \delta f_j^{(1)} = \frac{\epsilon}{L}\left(1+\frac{L-j+1}{q}\frac{\epsilon^2}{\log m}\right)$$

This means that (9.47) and (9.48) hold for $r = 1$. Now assume that (9.47) and (9.48) hold for some $1 \leq r \leq \epsilon^{-2}\log m \log(L/4)$, that is, the total flow $f_j^{(r)}$ on path P_j is bounded as in (9.47) and (9.48).

Then the allowed increase of flow at round $r+1$ is given by

$$c^{(r+1)}(P_j) = \frac{\epsilon^2}{\log m} f_j^{(r)} \leq \frac{\epsilon^3}{L\log m}\left(1+2\frac{L}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right)$$

and

$$c^{(r+1)}(P_j) = \frac{\epsilon^2}{\log m} f_j^{(r)} \geq \frac{\epsilon^3}{L \log m} \left(1 + \frac{1}{4} \frac{L-j+1}{q} \left(\left(1 + \frac{\epsilon^2}{\log m} \right)^r - 1 \right) \right) \tag{9.49}$$

Hence, using similar arguments as in (9.36), the updating flow on path P_j computed by the BD-MCF algorithm in round $r+1$ is

$$\begin{aligned} \delta f_j^{(r+1)} &= \frac{L-j+1}{q} c^{(r+1)}(P_j) \leq \frac{L}{q} c^{(r+1)}(P_j) \\ &\leq \frac{\epsilon^3}{q \log m} \left(1 + 2\frac{L}{q} \left(\left(1 + \frac{\epsilon^2}{\log m} \right)^r - 1 \right) \right) \end{aligned} \tag{9.50}$$

and

$$\begin{aligned} \delta f_j^{(r+1)} &= \frac{L-j+1}{q} c^{(r+1)}(P_j) \\ &\geq \frac{L-j+1}{q} \frac{\epsilon^3}{L \log m} \left(1 + \frac{1}{4} \frac{L-j+1}{q} \left(\left(1 + \frac{\epsilon^2}{\log m} \right)^r - 1 \right) \right) \end{aligned}$$

Then

$$\begin{aligned} \delta f^{(r+1)} &= \sum_{j=1}^{L-1} \delta f_j^{(r)} \\ &\leq \sum_{j=1}^{L-1} \frac{L}{q} \frac{\epsilon^3}{L \log m} \left(1 + 2\frac{L}{q} \left(\left(1 + \frac{\epsilon^2}{\log m} \right)^r - 1 \right) \right) \qquad (9.51) \\ &\leq 2 \frac{\epsilon^3}{L \log m} \left(1 + \frac{4}{L} \left(\left(1 + \frac{\epsilon^2}{\log m} \right)^r - 1 \right) \right). \qquad (9.52) \end{aligned}$$

Inequality (9.52) follows from (9.33).

Similarly

$$\begin{aligned}
\delta f^{(r+1)} &= \sum_{j=1}^{L-1} \delta f_j^{(r)} \\
&\geq \sum_{j=1}^{L-1} \frac{L-j+1}{q} \frac{\epsilon^3}{L \log m} \left(1 + \frac{L-j+1}{4q} \left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) \\
&= \frac{\epsilon^3}{L \log m} \left(1 + \sum_{j=1}^{L-1} \frac{(L-j+1)^2}{4q^2} \left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) \\
&\geq \frac{\epsilon^3}{L \log m} \left(1 + \frac{1}{3L} \left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1\right)\right). \qquad (9.53)
\end{aligned}$$

Inequality (9.53) follows from the fact that $\sum_{i=1}^{L} i^2 = \frac{L^3}{3}(1 + o(1))$.

Now we have to scale the updating flow $\delta f^{(r+1)}$ to the blocking flow $\Delta f_j^{(r+1)}$. According to Claim 3 the value of the blocking flow computed in round $r+1$ is given by

$$f_{BF}^{(r+1)} = \frac{\epsilon^3}{L \log m} \left(1 + \frac{\epsilon^2}{\log m}\right)^r. \qquad (9.54)$$

It can be easily checked that the bound (9.52) is smaller than the blocking flow (9.54) and thus the algorithm scales the updating flow up by the factor of $\frac{f_{BF}^{(r+1)}}{\delta f^{(r+1)}}$. Hence we have

$$\Delta f_j^{(r+1)} = \frac{f_{BF}^{(r+1)}}{\delta f^{(r+1)}} \delta f_j^{(r+1)}, \qquad (9.55)$$

So the flow sent on path P_j at round $r+1$ is given by

$$\begin{aligned}
\Delta f_j^{(r+1)} &= \frac{f_{BF}^{(r+1)}}{\delta f^{(r+1)}} \delta f_j^{(r+1)} \\
&\leq \frac{\frac{\epsilon^3}{L\log m}\left(1+\frac{\epsilon^2}{\log m}\right)^r}{\frac{\epsilon^3}{L\log m}\left(1+\frac{1}{3L}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right)} \\
&\times \frac{\epsilon^3}{q\log m}\left(1+2\frac{L}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right) \\
&= \frac{\epsilon^3}{q\log m}\left(1+\frac{\epsilon^2}{\log m}\right)^r \times \frac{\left(1+2\frac{L}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right)}{\left(1+\frac{1}{3L}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right)} \\
&\leq \frac{\epsilon^3}{q\log m}\left(1+\frac{\epsilon^2}{\log m}\right)^r\left(1+\left(2\frac{L}{q}-\frac{1}{3L}\right)\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right) \tag{9.56} \\
&\leq \frac{\epsilon^3}{q\log m}\left(1+\frac{\epsilon^2}{\log m}\right)^r\left(1+\frac{11}{3L}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right)\right) \tag{9.57} \\
&\leq 2\frac{\epsilon^3}{L\log m}\frac{L}{q}\left(1+\frac{\epsilon^2}{\log m}\right)^r. \tag{9.58}
\end{aligned}$$

In (9.56) we have used the fact that for $x \geq y \geq 0$,

$$\begin{aligned}
\frac{1+xz}{1+yz} = \frac{1+xz+yz-yz}{1+yz} &= 1+\frac{xz-yz}{1+yz} \\
&\leq 1+(x-y)z,
\end{aligned}$$

substituting $x = 2L/q$, $y = 1/3L$ and $z = \left(1+\frac{\epsilon^2}{\log m}\right)^r - 1$. In inequality (9.57) we have used the fact that $\frac{L}{q} \leq \frac{2}{L}$, and

$$\frac{11}{3L}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r-1\right) \leq 1,$$

for $r \leq \epsilon^{-2}\log m \log(L/4)$.

Note that the upper bound (9.58) on the flow change on path P_j is less than the lower bound of the capacity of the path (9.49). This means that we can scale up all the flows to obtain the blocking flow by one scaling without saturating any individual path.

Assuming, by induction, that (9.47) holds for r, adding (9.58) we can bound the flow on path P_j at the end of round $r+1$

$$\begin{aligned}
f_j^{(r+1)} &= f_j^{(r)} + \Delta f_j^{(r+1)} \\
&\leq \frac{\epsilon}{L}\left(1 + 2\frac{L}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) + 2\frac{\epsilon^3}{q\log m}\left(1+\frac{\epsilon^2}{\log m}\right)^r \\
&= \frac{\epsilon}{L}\left(1 + 2\frac{L}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r + \frac{\epsilon^2}{\log m}\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) \\
&= \frac{\epsilon}{L}\left(1 + 2\frac{L}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^{r+1} - 1\right)\right) \qquad (9.59)
\end{aligned}$$

Using (9.50) and (9.52) we get a lower bound on the flow sent in round $r+1$ on path P_j

$$\begin{aligned}
\Delta f_j^{(r+1)} &= \frac{f_{BF}^{(r+1)}}{\delta f^{(r+1)}}\delta f_j^{(r+1)} \\
&\geq \frac{\epsilon^3}{L\log m}\frac{L-j+1}{q}\left(1+\frac{\epsilon^2}{\log m}\right)^r \frac{\left(1+\left(\frac{1}{4}\frac{L-j+1}{q}\right)\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right)}{2\left(1+\frac{4}{L}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right)} \\
&\geq \frac{\epsilon^3}{L\log m}\frac{L-j+1}{q}\left(1+\frac{\epsilon^2}{\log m}\right)^r \frac{1}{2\left(1+\frac{4}{L}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right)} \\
&\geq \frac{1}{4}\frac{\epsilon^3}{L\log m}\frac{L-j+1}{q}\left(1+\frac{\epsilon^2}{\log m}\right)^r \qquad (9.60)
\end{aligned}$$

Inequality (9.60) holds because $\frac{4}{L}(1+\frac{\epsilon^2}{\log m})^r \leq 1$, for $r \leq \epsilon^{-2}\log m \log(L/4)$. As before,using (9.48) and (9.60), we now get a lower bound on the flow sent in round $r+1$ on path P_j

$$
\begin{aligned}
f_j^{(r+1)} &= f_j^{(r)} + \Delta f_j^{(r+1)} \\
&\geq \frac{\epsilon}{L}\left(1+\frac{1}{4}\frac{L-j+1}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) \\
&+ \frac{1}{4}\frac{\epsilon^3}{L\log m}\frac{L-j+1}{q}\left(1+\frac{\epsilon^2}{\log m}\right)^r \\
&= \frac{\epsilon}{L}\left(1+\frac{1}{4}\frac{L-j+1}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^r + \frac{\epsilon^2}{\log m}\left(1+\frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) \\
&= \frac{\epsilon}{L}\left(1+\frac{1}{4}\frac{L-j+1}{q}\left(\left(1+\frac{\epsilon^2}{\log m}\right)^{r+1} - 1\right)\right) \qquad (9.61)
\end{aligned}
$$

Bounds (9.59) and (9.61) mean that (9.47) and (9.48) hold with $r+1$ substituted for r. □

We have proved that the upper bound of the flow increase at any time round r is given by the formula in Claim 4. Using this claim and the lower bound on the path length increase due to a change in flow given in (9.17) we can get a bound on the path length at any time round r. Using this bound we are going to prove our main Lemma 15.

Proof of Lemma 15. For each edge on the shared part of P_i,

$$\Delta f^{(r)}(e) = \sum_i f_{i,j}^{(r)}(e) - \epsilon = L\left(f_j^{(r)} - \frac{\epsilon}{L}\right). \qquad (9.62)$$

From (9.4) the length of path P_j at the end of round r is bounded by

$$\begin{aligned}
\Pi_j^{(r)} &\leq \sum_{e \in P_j} (m^{1/\epsilon})^{\lambda(e)} \left(1 + 2\frac{\log m}{\epsilon} \Delta f^{(r)}(e)\right) \\
&= \sum_{e \in P_j} (m^{1/\epsilon})^{\lambda(e)} \left(1 + 2\frac{\log m}{\epsilon} L \left(f_j^{(r)} - \frac{\epsilon}{L}\right)\right) \\
&= \sum_{e \in P_j} (m^{1/\epsilon})^{\lambda(e)} + 2\frac{\log m}{\epsilon} L \sum_{e \in P_j} (m^{1/\epsilon})^{\lambda(e)} \left(f_j^{(r)} - \frac{\epsilon}{L}\right) \\
&= \Pi_j^{(0)} + 2\frac{\log m}{\epsilon} Lm \sum_{e \in P_j} \left(f_j^{(r)} - \frac{\epsilon}{L}\right) \\
&= \Pi_j^{(0)} \left(2\frac{\log m}{\epsilon} \sum_{e \in P_j} \left(f_j^{(r)} - \frac{\epsilon}{L}\right)\right)
\end{aligned}$$

Since the top path($j = 1$) congests more rapidly than any other path we are interested in the number of rounds needed so that it is no longer an approximate shortest path. Remember that the value of the blocking flow is given by (3) for as long as all paths in Υ_L are ϵ-approximate shortest paths. From Claim 4 (Inequality (9.47)) the length of the top path is given by

$$\begin{aligned}
\Pi_1^{(r)} &\leq \left(2\frac{\log m}{\epsilon} L \left(f_j^{(r)} - \frac{\epsilon}{L}\right)\right) \Pi_1^{(0)} \\
&\leq \left(2\frac{\log m}{\epsilon} L (\frac{\epsilon}{L} \left(1 + 2\frac{L}{q}\left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) - \frac{\epsilon}{L})\right) \Pi_1^{(0)} \\
&= \left(4 \log m \frac{L}{q} \left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) \Pi_1^{(0)} \\
&\leq \left(8\frac{\log m}{L} \left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1\right)\right) \Pi_1^{(0)}.
\end{aligned}$$

If the length of P_1 has not increased by a factor $(1 + \epsilon)$ by the end of round r, then

$$8\frac{\log m}{L} \left(\left(1 + \frac{\epsilon^2}{\log m}\right)^r - 1\right) \leq 1 + \epsilon$$

Hence the total number of rounds needed for path P_1 to increase its length by a $(1+\epsilon)$ factor is

$$r = \Omega\left(\epsilon^{-2}\log m \log\left(\frac{L\epsilon}{\log m}\right)\right), \tag{9.63}$$

provided that $\epsilon^2 = O(\log m)$.

The total flow change at any round r is given by the value of the blocking flow and it is

$$f_{BF}^{(r)} = \frac{\epsilon}{L}\left(1 + \frac{\epsilon^2}{\log m}\right)^r \tag{9.64}$$

The top path is no longer ϵ-approximate after $\Omega\left(\epsilon^{-2}\log m \log\left(\frac{L\epsilon}{\log m}\right)\right)$ rounds. Within this number of rounds we have proven that the value of the blocking flow is given by (9.64). Substituting for r we get that until the time that the top path becomes unavailable we have sent at least $\epsilon^2/\log m$ flow. Hence the first phase terminates in $\Theta\left(\epsilon^{-2}\log m \log\left(\frac{L\epsilon}{\log m}\right)\right)$ rounds.

□

9.4 Summary

In this chapter we have shown that the number of rounds needed to terminate the first phase of the DGD-MCF algorithm on network Υ_L is at least $\Omega(L\epsilon^{-1})$. This shows that the upper bound of the DGD-MCF algorithm proposed in [7] is actually tight, up to polylogarithmic factors. To overcome the bottleneck of the DGD-MCF algorithm we have proposed the BD-MCF algorithm and proved that first phase on network Υ_L terminates in time polylogarithmic in L. This shows that our algorithm significantly improves the running time when implemented on our worst-case network Υ_L. More specifically, we reduce the running time from $\widetilde{O}(Lm^2)$ to $\widetilde{O}(m^2)$ for the first phase on network Υ_L, or in other words, we reduce the linear dependence on the parameter L to poly-logarithmic.

Chapter 10

The Distributed Rerouting Algorithm

Contents

In this chapter we are going to examine the Greedy Distributed Rerouting (GDR-MCF) algorithm proposed by Awerbuch and Khandekar [4]. In the first Section we are going to to describe the GDR-MCF algorithm and provide some main theorems from [4] to help us in the analysis of the algorithm. In Section 10.2 we propose a new upper limit on the flow change which is monotonically increasing with respect to the flow, unlike the previous upper limit on flow proposed in [4]. In Section 10.3 we show that the number of rounds needed for the algorithm to terminate using the current bounds on the flow change is proportional to L. In fact, we show that the number of rounds of the GDR-MCF algorithm is proportional to L when executed on our worst-case network Υ_L, even when removing the limit on the flow decrease.

10.1 The Greedy Distributed Rerouting Algorithm

Awerbuch and Khandekar [4] describe the first stateless greedy algorithm for the distributed multi-commodity concurrent flow problem with the number of rounds poly-logarithmic in size of the input and linear in the parameter L. We refer to this algorithm as GDR-MCF algorithm. Agents, each one representing the flow of a single commodity, operate in a cooperative but uncoordinated manner. The agents perform their computations in an asynchronous manner and only synchronize at the start of a round. The computation model used is the Distributed Shared Memory model described in Section 8.3. More details about the model used in [4] can be found in subsection 8.4.2.

A routing metric is introduced to indicate the cost of an edge with respect to its congestion. This is given by the derivative of the potential function

$$\Phi = \sum_{e \in \mathcal{E}} \phi_{e,\mu}(f(e)),$$

where

$$\phi_{e,\mu}(f(e)) = m^{\frac{f(e)}{c(e)\cdot\mu}},$$

and $\mu \approx \epsilon \cdot \lambda$. The number $c(e)$ is the capacity of edge $e \in \mathcal{E}$, $f(e)$ is the total flow on edge e and μ is parameter associated with the maximum congestion in the network. Observe that this potential function is exactly the same as the potential function of the DGD-MCF algorithm when $\lambda = 1$; see (9.1).

To deal with the instability problem that can arise by routing all the flows simultaneously ”speed limits” on the changes of edge flows are imposed. These limits control the maximum amount by which a flow can increase or decrease on an edge. This way oscillations are prevented and the flows will converge. It is remarkable that a more restrictive ”speed limit” is enforced on the decrease of the edge flows than on the increase which is opposite to the intuition that increase is more ”dangerous" for oscillations than decrease.

More specifically, the algorithm maintains an upper bound $\Delta_i^+(e)$ and a lower bound $\Delta_i^-(e)$ on the total flow change on an edge e within one round. The bounds at the beginning of the round, are set to:

$$\Delta_i^-(e) = f_i(e)\frac{\beta}{4L}, \tag{10.1}$$

$$\Delta_i^+(e) = \max\{\beta \cdot f_i(e), (1+\beta) \cdot \ddot{f}_i(e)\}, \tag{10.2}$$

where $\ddot{f}_i(e)$ is an initial additive limit set for edges with small or no flow. The bounds change throughout the round while the flow changes on the edges.

The main execution of the algorithm during one round proceeds in the following way. An agent i reads the total edge flow values from the billboard at the beginning of each round. Then the costs of all edges are calculated as:

$$\phi'_{e,\mu}(f(e)) = \frac{\log m}{c(e) \cdot \mu} m^{\frac{f(e)}{c(e)\cdot\mu}},$$

and remain fixed throughout this round. The shortest path P^{SP} from s_i to t_i and its cost are computed. This cost is compared with the average cost of the flow of commodity i in the network, which can be found by adding the total cost of commodity i on all edges and dividing this by its demand d_i:

$$(1+\alpha) \cdot \phi'_f(P) < \frac{d_i}{\sum_{e \in \mathcal{E}}} f_i(e)\phi'_{e,\mu}(e).$$

If the cost of path P^{SP} is significantly smaller than the average cost and the upper and lower flow limits are not violated a small amount of flow is rerouted from all paths to path P^{SP}. The procedure proceeds until no paths which satisfy the flow limit constraints can be found or until no path can be found with cost significantly smaller than the average path cost. At this point the agent i writes the new flow of commodity i on the billboard and waits for the next round to begin. Pseudocode of one round of the GDR-MCF algorithm is given below as Algorithm 7.

The following theorem, proved in [4], describes the performance of the GDR-MCF Algorithm .

Theorem 12 (Theorem 1 from [4]). *The GDR-MCF algorithm achieves an* ϵ-

Algorithm 7: Greedy Distributed Rerouting algorithm [4]

```
// One round, commodity $i$ //
Input: flow vectors $f_i$, $f$ and $\mu$
Output: flow vectors $f_i$
Part 1                                    // Calculation of cost function //
1. Calculate $\lambda_{max}$;
2. Calculate $\mu \leftarrow \min\{\mu, 2^{\lfloor \log_2(\epsilon \cdot \lambda_{max}) \rfloor}\}$ or $2^{\lfloor \log_2(\epsilon \cdot \lambda_{max}) \rfloor}$ if $\mu$ is not set;
3. Calculate $\phi'_{e,\mu}(f(e)) \equiv m^{\frac{f(e)}{c(e) \cdot \mu}}$ for all $e \in \mathcal{E}$ ;          /* Cost of $e$ */
4. Define $\phi^{'}_{f}(P_{i,j}) \equiv \sum_{e \in P_{i,j}} \phi^{'}_{e,\mu}(f(e))$;
Part 2                              */ Calculation of residual capacities*/
1. $\alpha \leftarrow \frac{\epsilon}{40 \log m}$ ;
2. $\beta \leftarrow \Theta(\alpha \cdot \frac{\epsilon}{logm})$;
3. $\ddot{f}_i(e) \leftarrow \frac{\mu}{\log_2 m} \cdot \frac{c(e)}{(1+\beta)k} \cdot \log_2(1 + \frac{\alpha}{8})$;
4. upon each round initialize for all $e \in \mathcal{E}$;
   (a) $\Delta_i^-(e) \leftarrow f_i(e)\frac{\beta}{4L}$;
   (b) $\Delta_i^+(e) \leftarrow \max\{\beta \cdot f_i(e), (1+\beta) \cdot \ddot{f}_i(e)\}$;
Part 3                                         // Updating the flow //
while $\sum_{\nu \in \mathcal{V}} \Delta_i^-(s_i, \nu) > 0$ and $\exists\, P \in P_i$ s.t.
      $min_{e \in P} \Delta_i^+(e) > 0,$ and
      $(1+\alpha) \cdot \phi^{'}_{f}(P) < \left(\sum_{e \in \mathcal{E}} f_i(e) \phi^{'}_{e,\mu}(e)\right)/d_i$ do
    1. Set $P^* \leftarrow argmin\ \phi^{'}_{f}(P)$ over set $P_{i,j} \in P$ satisfying the above
    conditions ;
    2. $\delta \leftarrow \min\left\{\min_{e \in P^*} \Delta_i^+(e), \sum_{\nu \in \mathcal{V}} \Delta_i^-(s_i, \nu)\right\}$ ;
    3. $f_i(e) \leftarrow f_i(e) - \delta \cdot \frac{f_i(e)}{d_i}$, for all $e \in \mathcal{E}$ ;
    4. $\Delta_i^-(e) \leftarrow \Delta_i^-(e) - \delta \cdot \frac{f_i(e)}{d_i}$, for all $e \in \mathcal{E}$ ;
    5. $f_i(e) \leftarrow f_i(e) + \delta$ and $\Delta_i^+(e) \leftarrow \Delta_i^+(e) - \delta$, for all $e \in P^*$ ;
end
```

approximation within $\widetilde{O}(L)$ rounds. The running time of each round of the GDR-MCF algorithm is $\widetilde{O}(m^2)$ for each commodity.

Below we prove a couple of lemmas which are not explicitly given in [4] but are important in the analysis of the algorithm. Let $f_i^{(r,j)}(e)$ denote the value of the flow of commodity i on edge e at the end of the jth step of round r. Also, let $\delta f_i^{(r,j)}$ denote the amount of flow of commodity i rerouted at the jth step of round r.

Lemma 16. *The flow of each commodity i on each edge e at the end of a round never decreases by more than a $\beta/4L$ fraction of this flow at the start of the round. That is,*

$$f_i^{(r+1,0)}(e) \geq \left(1 - \frac{\beta}{4L}\right) f_i^{(r,0)}(e). \tag{10.3}$$

Moreover, the upper bound of the amount of flow of commodity i rerouted at step j is given by the following expression, which is always non-negative,

$$\Delta_{i,j}^{-} = \sum_{\nu \in \mathcal{V}} \Delta_{i,j}^{-}(s_i, \nu) = \frac{\beta}{4L} d_i - \sum_{q=1}^{j} \delta f_i^{(r,q)}. \tag{10.4}$$

Proof. We first prove (10.4).

Let $\Delta_{i,j}^{-}$ be the value of $\sum_{\nu \in \mathcal{V}} \Delta_{i,j}^{-}(s_i, \nu)$ at the end of step j. To prove our claim we need to establish a formula for the upper bound on the flow change at any step.

We prove by induction that formula (10.4) holds and that is always non-negative. Initially,

$$\Delta_{i,0}^{-} = \sum_{\nu \in \mathcal{V}} \Delta_{i}^{-}(s_i, \nu) = \sum_{\nu \in \mathcal{V}} \frac{\beta}{4L} f_i(s_i, \nu) = \frac{\beta}{4L} d_i \geq 0.$$

Assume that (10.4) holds for some step $j \geq 0$ and it is non-negative. Then at step $j+1$, the change of flow is given by

$$\begin{aligned}\delta f_i^{(j+1)} &= \min\left\{\min_{e\in P^*} \Delta_i^+(e), \Delta_{i,j}^-\right\} \\ &= \min\left\{\min_{e\in P^*} \Delta_i^+(e), \frac{\beta}{4L}d_i - \sum_{q=1}^{j} \delta f_i^{(r,q)}\right\}. \qquad (10.5)\end{aligned}$$

Then, using the formula for updating $\Delta_{i,j}^-(e)$ and the inductive hypothesis, we have

$$\begin{aligned}\Delta_{i,j+1}^- &= \sum_{\nu\in\mathcal{V}}\left(\Delta_{i,j}^-(s_i,\nu) - \frac{\delta f_i^{(r,j+1)}}{d_i} f_i(s_i,\nu)\right) \\ &= \frac{\beta}{4L}d_i - \sum_{q=1}^{j}\delta f_i^{(r,q)} - \sum_{\nu\in\mathcal{V}}\frac{\delta f_i^{(r,j+1)}}{d_i} f_i(s_i,\nu) \\ &= \frac{\beta}{4L}d_i - \sum_{q=1}^{j}\delta f_i^{(r,q)} - \delta f_i^{(r,j+1)} \qquad (10.6) \\ &= \frac{\beta}{4L}d_i - \sum_{q=1}^{j+1}\delta f_i^{(r,q)}.\end{aligned}$$

The value (10.6) is non-negative because of (10.5), so (10.4) holds for $j+1$ and is non-negative.

Now we prove (10.3). The flow at any step $j \geq 1$ is given by

$$\begin{aligned}
f_i^{(r,j)}(e) &\geq f_i^{(r,j-1)}(e) - \frac{\delta f_i^{(r,j)}}{d_i} f_i^{(r,j-1)}(e) \\
&= \left(1 - \frac{\delta f_i^{(r,j)}}{d_i}\right) f_i^{(r,j-1)}(e) \\
&= \prod_{q=1}^{j} \left(1 - \frac{\delta f_i^{(r,q)}}{d_i}\right) f_i^{(r,0)}(e) \\
&\geq \left(1 - \sum_{q=1}^{j} \frac{\delta f_i^{(r,q)}}{d_i}\right) f_i^{(r,0)}(e) \\
&\geq \left(1 - \frac{\beta}{4L}\right) f_i^{(r,0)}(e)
\end{aligned}$$

□

The above lemma proves that at any round we reroute at most $\beta/4L$ fraction of the demand and at any edge the flow at the end of the round is not decreased by more than this fraction. We show in our next lemma that if we have not managed to reroute a $\beta/4L$ fraction of the demand then all approximate shortest paths have at least one saturated edge.

Lemma 17. *If the round ends and the amount of the rerouted flow of commodity i is less than $d_i\beta/4L$, then each ϵ-approximate shortest path P has at least one saturated edge, that is,*

$$f_i^{(r+1,0)}(e) = f_i^{(r,0)}(e) + \Delta_i^+(e) = (1+\beta) f_i^{(r,0)}(e), \tag{10.7}$$

for some $e \in P$.

Proof. We are going to prove this lemma by contradiction. Assume that at some step j within a round the rerouted flow is less than $d_i\beta/4L$, and there exists an ϵ-approximate shortest path P^{SP} that has no saturated edge.

If the rerouted flow $\sum_{q=1}^{j} \delta f_i^{(r,q)} < d_i\beta/4L$, then from Lemma 16, $\Delta_{i,j}^- > 0$, that is, the round does not end.

□

10.2 The Upper Limit on the Increase of Flow in One Round

Awerbuch and Khandekar [4] define speed limits for the allowed increase and decrease of the flow on one edge in one round. In our analysis of their algorithm we discovered a behavior in the function $\Delta_i^+(e)$ of the allowed increase which is not monotonic: it linearly decreases when $f_i(e)$ increases from 0 to $\ddot{f}(e)$, and then increases. Usually the function on the allowed flow sent in each round is monotonically increasing. In other words, in each round we are allowed to send more flow than the previous one. This happens because in each round we take advantage of the knowledge we gain while distributing the flow in the previous rounds, which results in more aggressive increase of the flow and thus faster termination of the algorithm. In [4] this is not the case. The function of the allowed increase is given by

$$\Delta_i^+(e) \leftarrow (1+\beta) \cdot \max\{f_i(e), \ddot{f}(e)\} - f_i(e) \tag{10.8}$$

where

$$\ddot{f}(e) = \frac{\mu}{\log_2 m} \cdot \frac{c(e)}{(1+\beta)k} \cdot \log_2(1+\frac{\alpha}{8}) \tag{10.9}$$

The allowed increase in the flow in one round is $\beta \cdot f_i(e)$ except in the cases where the flow is small, i.e. when $f_i(e) \leq \ddot{f}(e)$, in which case an additive flow $(1+\beta) \cdot \ddot{f}(e) - f_i(e)$ is allowed to be sent. In Figure 10.1 we can observe graphically this behavior. Note that if the flow is zero the allowed capacity of the edge starts at $(1+\beta) \cdot \ddot{f}_i(e)$ and in the subsequent rounds it decreases. It only starts increasing at the point when the flow on this edge is equal to its additive limit $\ddot{f}_i(e)$. This non-monotonic behavior could force the algorithm to perform a significant number of rounds when edge flow is less or near the threshold value of $\ddot{f}(e)$, i.e. before a significant amount of flow is sent along edges with small or zero flow initially so that the multiplicative increase starts.

We propose a new upper limit on the increase of the flow defined as follows

$$\Delta_i^+(e) = \beta \cdot f_i(e) + (1+\beta) \cdot \ddot{f}(e)$$

This definition would not affect any of the properties of the upper limit on the increase of the flow or the analysis of the Greedy Distributed Rerouting algorithm, but it would reduce the number of rounds needed for the algorithm to properly initialize and it is much simpler. We will analyze the main lemmas involving the additive upper limit in [4] below. In fact we are showing that the "gain" is even better using this definition.

By setting the additive increase to be equal to

$$\ddot{f}(e) \leftarrow \frac{\mu}{\log_2 m} \cdot \frac{c(e)}{(1+\beta)k} \cdot \log_2\left(1+\frac{\alpha}{8}\right),$$

they allow only a fraction $\alpha/8$ increase/decrease in the cost ϕ'. For all k commodities the total additive increase in the flow is at most

$$(1+\beta)k \cdot \ddot{f}(e) \leq \frac{\mu}{\log_2 m} \cdot c(e) \cdot \log_2\left(1+\frac{\alpha}{8}\right),$$

which gives a multiplicative increase in the cost of

$$m^{\frac{1}{c(e)\cdot\mu}\frac{\mu}{\log_2 m}\cdot c(e)\cdot\log_2\left(1+\frac{\alpha}{8}\right)} = 1+\frac{\alpha}{8}.$$

The multiplicative speed limit on the increase of the flow $\beta \cdot f_i(e)$ gives an extra $\alpha/8$ increase in the cost ϕ'.

Lemma 18. *The cost $\phi'_{e,\mu}(f(e))$ of each edge e increases (respectively decreases) within a round by at most $(1+\frac{\alpha}{8})^2$ (respectively by at most $(1+\frac{\alpha}{8})$) factor.*

Proof. The upper limit on the multiplicative flow increase (decrease) from [4] causes the cost to increase (decrease) by at most $(1+\frac{\alpha}{8})$. The additive increase gives an extra factor of at most $(1+\frac{\alpha}{8})$. □

We conclude that changing the allowed increase function would not affect the analysis of the algorithm in [4] but it would give us a more regular behavior of the upper limit function and a simpler definition than the previous one (10.8).

The graph below shows the behavior of the old upper limit function with respect to the one we propose in this thesis.

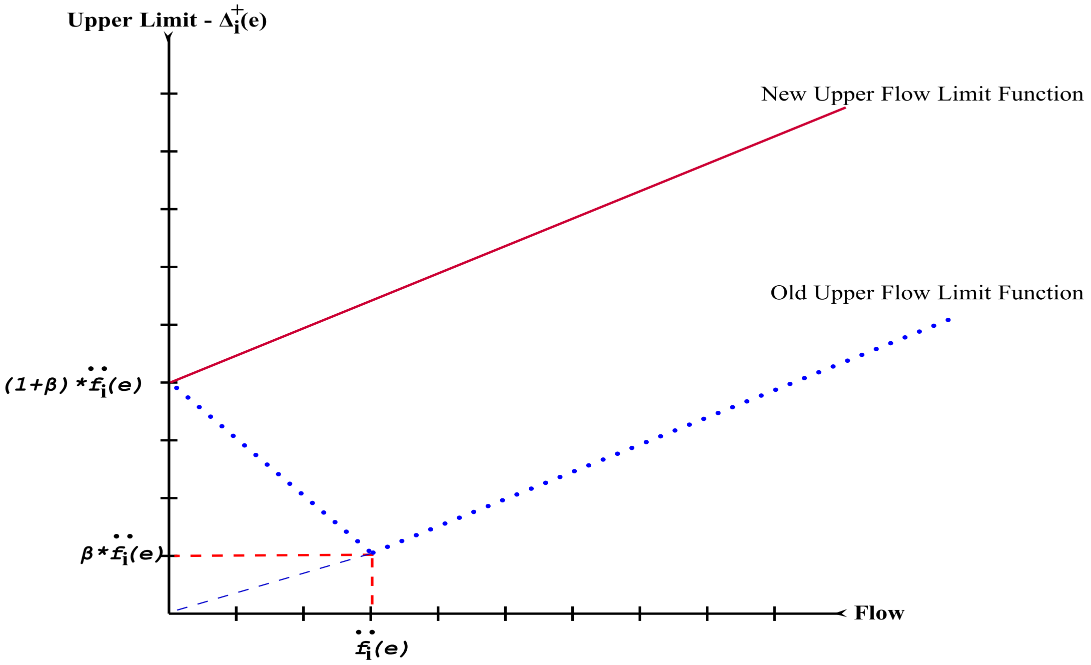


Figure 10.1: Old v New Upper Limit on the Increase of Flow

10.3 Running Time of Greedy Distributed Algorithm on Υ_L

10.3.1 The GDR-MCF Algorithm with the Flow-Decrease Constraints

The upper bound on running time of the GDR algorithm shown by Awerbuch and Khandekar [4] depends linearly on the maximum path size L. Awerbuch and Khandekar [4] claim that on the Internet only short paths are used (less than 10 hops) so the linear dependence of the running time on L is not a problem in practice. We would like to investigate this dependence further for the case when the maximum path size is actually large (in the worst case of the order of n). It is easy to show that the worst-case running time is linearly dependent on L if the decrease constraints are used. We show this result on network Υ_L shown in Figure 9.1. We use the notation for network Υ_L as introduced in Section 9.2.

Lemma 19. *The worst-case number of rounds during the computation of the*

Greedy Distributed Rerouting algorithm with the flow-decrease constraints is $\Omega(L \log^2 m/\epsilon^2)$.

Proof. From Lemma 16 the flow increase on an edge e at any round is given by

$$f_i^{(r+1,0)}(e) \geq \left(1 - \frac{\beta}{4L}\right) f_i^{(r,0)}(e). \tag{10.10}$$

Now consider the network in Figure 9.1 with k commodities. The demand for each commodity is now given by $d_i = 1$ and initially all the flow is on the top path $P_{i,1}$ for each commodity i. The capacity of all edges is $c(e) = 1$. Thus at the beginning the maximum congestion is $\lambda_{max} = k$ and the optimal congestion is $\lambda_{opt} = 1$. To achieve an ϵ-approximate solution, that is to achieve a congestion $\lambda < 1 + \epsilon$, a total flow of at least $k - (1 + \epsilon)$ needs to be rerouted from the top path to the lower paths. The flow of commodity i at round j in the top path is given by

$$f_i^{(j,0)}(e) \geq \left(1 - \frac{\beta}{4L}\right)^j > \frac{(1+\epsilon)}{k}.$$

To get an ϵ-approximate solution at round R the flow of commodity i at the top path must decrease to

$$f_i^{(R,0)}(e) \leq \left(1 - \frac{\beta}{4L}\right)^j \leq \frac{(1+\epsilon)}{k}.$$

Thus the number of rounds, R is bounded by

$$\begin{aligned} 1 - \frac{\beta}{4L} R &\leq \frac{(1+\epsilon)}{k} \\ \Rightarrow R &\geq \frac{4L}{\beta}\left(1 - \frac{(1+\epsilon)}{k}\right) \\ \Rightarrow R &\geq \frac{2L}{\beta}. \end{aligned}$$

Hence the number of rounds needed for the algorithm to terminate is $\Omega(L\epsilon^{-2} \cdot \log^2 m)$. □

10.3.2 The GDR Algorithm Without the Flow-Decrease Constraints

It is easy to see that the flow-decrease constraints will cause a linear dependence of the number of rounds on L because in any round only a fraction of $O(1/L)$ of flow is allowed to be rerouted. For this reason we would like to investigate the behavior of the algorithm on Υ_L without this restriction. No current upper bounds exist for this case. More specifically we modify the algorithm and remove all references to the flow-decrease bounds $\Delta_i^-(e)$. We abbreviate the new, modified algorithm without the flow-decrease constraints GDR^+ algorithm. Thus now if a path $P_{i,j}$ is chosen, the flow on this path will now increase by

$$\delta \leftarrow \min_{e \in P_{i,j}} \Delta_i^+(e) \tag{10.11}$$

To show the worst-case behavior of the GDR^+ algorithm on Υ_L we modify the network to fit to our analysis. We add an artificial edge of capacity $c(s_i,t)=1$ from each commodity source s_i to the destination t. We denote this path by $P_{i,0}$ to fit with the description of the other paths. The demand for each commodity is now $d_i = 2$.

The optimal congestion is $\lambda_{opt} = 1$ and each optimal flow f^* has $f_i^*(s_i,t) = 1$ and $f_i^*(s_i,w_{i,1}) = 1$. Recall that $f_{i,j}^{(r,0)}$ denotes the flow of commodity i on path $P_{i,j}$ at the beginning of round r. Initially all the flow of each commodity is concentrated on edge (s_i,t), that is $f_{i,0}^{(1,0)} = 2$ and $f_{i,j}^{(1,0)} = 0,\ j \geq 1$. We also alter the number of edges of each path by adding one edge on the "shared" part of each path. This way all of the paths $P_{i,j}$ have $L+2$ edges now instead of $L+1$. The exclusive path P_i remains unchanged. The reason for this is to force the commodities to choose the exclusive path for their initial augmentation. The topology and the capacities of all of the edges remain unchanged. Figure 10.2 shows the network Υ'_L from the point of view of one commodity.

Initially all of the flow of each commodity i is on edge (s_i,t). All of the remaining edges are empty. This way we have a similar setting to the one in Section 9.2.1. Our aim is to simulate the behaviour of the GDR algorithm with the Distributed

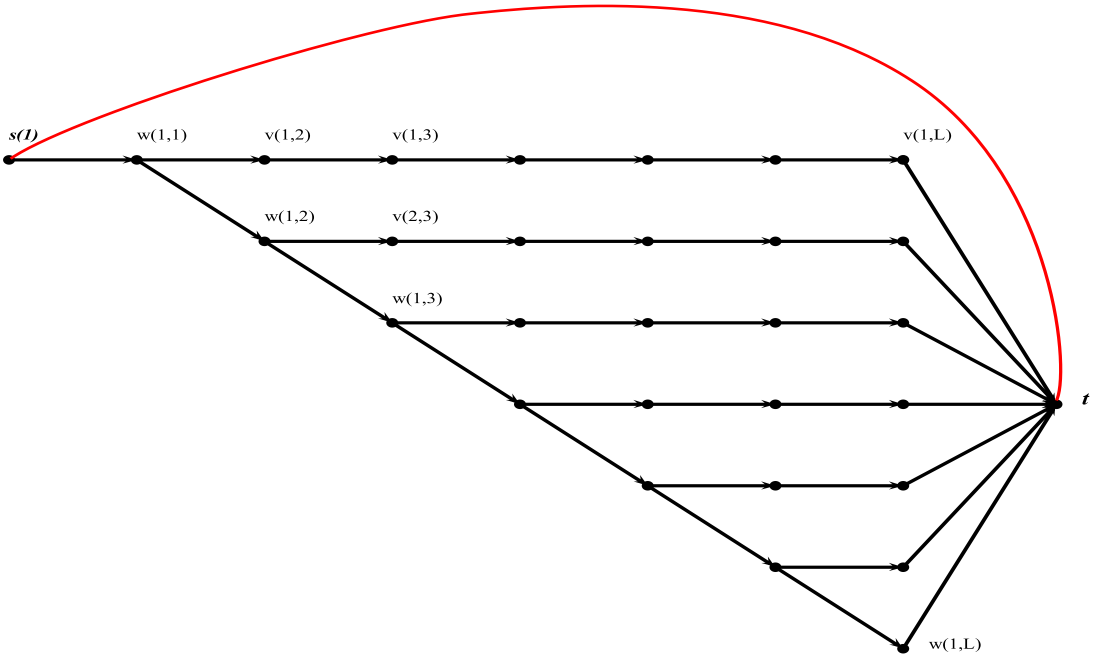


Figure 10.2: Network Υ'_L for One Commodity

Gradient Descent algorithm to be able to compare their execution. The flow in this situation, as before, is zero everywhere in the network Υ_L and it builds up gradually. The goal is to show that the algorithm needs to visit at least L paths before it achieves the required optimal solution.

The fact that the congestion initially is $\lambda_{init} = 2$ and the final congestion achieved by the algorithm is $\lambda_{final} \geq 1$ means that we only have one phase in the algorithm. Recall that the parameter

$$\mu \leftarrow \min\{\mu, 2^{\lfloor \log_2(\epsilon \cdot \lambda_{max}) \rfloor}\}$$

only changes when the congestion drops by a factor of 2, at which case the phase changes. Since the congestion only drops by a factor of 2 when we achieve the actual optimal solution we conclude that there is only one phase of the algorithm on the current setting of the network Υ'_L. Hence, assuming without loss of generality that $\epsilon = 2^{-i}$ for a positive integer i we have

$$\begin{aligned}\mu &= 2^{\lfloor \log_2(\epsilon \cdot \lambda_{max}) \rfloor} \\ &= 2^{\lfloor \log_2(\epsilon \cdot 2) \rfloor} \\ &= 2\epsilon.\end{aligned}$$

Informally the algorithm proceeds as follows. First it calculates the cost of all edges for a commodity i. Then it picks the shortest path and compares its cost with an average of the total cost. If the cost of the path is less than $(1-\alpha)$ the average cost, i.e. there exists an α benefit on the cost if this path is used, then the commodity is allowed to reroute on this path an amount of flow δ. After the augmentation the allowed flow that can be sent on all edges of the path chosen is recalculated. The chosen path might be using edges of other paths as well so the allowed flow that can be pushed can change. As long as there exists a path with positive minimum allowed increase of its edges and it is beneficial (its cost is significantly smaller than the average total cost) the algorithm keeps choosing this path and augments flow. The round terminates when no other path can be found satisfying this constraint. At this point the commodity writes its flow on the billboard and starts the procedure again until an approximate optimal flow is achieved. We are going to show that the algorithm in fact does not terminate before $\Omega(L)$ rounds have passed.

Theorem 13. *For the network Υ'_L and the initial flows $\boldsymbol{f}_0$ the Greedy Distributed Rerouting algorithm (GDR$^+$) needs at least $\Omega(L)$ rounds to reach an ϵ-approximate flow.*

First we are going to sketch our proof. This theorem will follow directly from Lemma 22, which follows from Lemmas 20 and 21. Agent i at the beginning of a round picks the shortest path from s_i to t_i and reroutes as much flow of commodity i as allowed by the push residual capacities $\Delta_i^+(e)$. At the beginning of the first round (round 0) all of the paths, besides the exclusive path P_i, in the network Υ'_L have the same length since there is no flow, they have the same size and all edges have the same capacity. The exclusive path has the smallest length because it is one edge shorter than the other paths. Each agent categorizes all of the paths as available but it chooses the exclusive one as the shortest path. The length of the shortest path is L and the lengths of the other paths are $L+1$. The length of the longest path $P_{i,0}$ is $m^{1/\epsilon}$ so all the paths are beneficial initially, that is, their cost is significantly smaller than the cost of the longest path (see the second line of the "while" loop of Algorithm 7). Since there is no flow on the shortest path P_i the allowed increase is given by

$$\delta = \min_{e \in P_i} \Delta_i^+(e) = (1+\beta) \cdot \min_{e \in P_i} \ddot{f}_i(e) = (1+\beta) \cdot \ddot{f}$$

No other paths are used in round 0 since the flow sent along path P_i saturates edge $(s_i, w_{i,1})$. The values $\ddot{f}_i(e)$ are equal for all the edges $e \in P_i$ because all edges have the same capacities. We abbreviate $\ddot{f} = \ddot{f}_i(e)$ to simplify notation.

In the next round each commodity picks the top path $P_{i,1}$ first to augment flow. This happens because the top path is the shortest path in this round. The cost of a path is given by

$$\phi_f^{'}(P_{i,j}) \equiv \sum_{e \in P_{i,j}} \phi_{e,\mu}^{'}(f(e)) \tag{10.12}$$

All of the paths $P_{i,j}$ have a shared part and an exclusive part (and one edge connecting the two parts). Path $P_{i,j}$ has j edges on the exclusive part and $L-j$ edges on the shared part as defined in Section 9.2. Since only the exclusive edges have positive flow, then the path with the smallest number of exclusive edges will be the shortest, that is path $P_{i,1}$. Rerouting the flow onto path $P_{i,1}$ saturates edge $(s_i, w_{i,1})$, so no other path is used in this round. In round 2 the shortest path is $P_{i,2}$ for similar reasons as above, so flow is rerouted onto this path saturating edge $(w_{i,1}, w_{i,2})$. At this round some further flow can be rerouted also onto the path $P_{i,1}$ if the conditions are satisfied. This pattern continues during subsequent rounds: In round r first the path $P_{i,r}$ is chosen and rerouting of flow saturates the edge $(w_{i,r-1}, w_{i,r})$. Then some further flow can be also rerouted onto some of the paths. $P_{i,1}, P_{i,2}, \ldots, P_{i,r-1}$, but no flow will be rerouted onto any path $P_{i,j}$ for $j > r$. We show in Lemmas 20 and 21 that in each round only one new path can be used to reroute flow. Following from these two lemmas, Lemma 22 says that after $L/4$ rounds we still do not have an ϵ-approximate flow.

Lemma 20. *If path $P_{i,r}$ is used for the first time in round t, then paths $P_{i,j}, j > r$*

could not be used in the previous rounds $1, 2, \ldots, t-1$.

Proof. If any path $P_{i,j}, j > r$ was used in any of the previous rounds $1, 2, \ldots, t$ then by construction of the network Υ'_L path $P_{i,r}$ would also be used. The algorithm picks the shortest path first in each round to augment flow. Then if the constraints are satisfied it can augment flow in more paths. Now consider the two paths $P_{i,r}$ and $P_{i,j}, j > r$ at some round $s < t$ and assume that no flow has been sent yet to any of the two paths. Then the cost of each path is given by

$$\begin{aligned}\phi_f^{'}(P_{i,j}) &= \sum_{e \in P_{i,j}} \phi_{e,\mu}^{'}(f(e)) \\ &= \phi_{e,\mu}^{'}(s_i, w_{i,1}) + \phi_{e,\mu}^{'}(w_{i,1}, w_{i,2}) + \ldots \\ &+ \phi_{e,\mu}^{'}(w_{i,r-1}, w_{i,r}) + \ldots + \phi_{e,\mu}^{'}(w_{i,j-1}, w_{i,j}) + L - j.\end{aligned}$$

Comparing the two paths we can observe that the cost of path $P_{i,r}$ is broken down into the following parts

$$\phi_f^{'}(P_{i,r}) = \phi_{e,\mu}^{'}(s_i, w_{i,1}) + \sum_{l=1}^{r} \phi_{e,\mu}^{'}(w_{i,l-1}, w_{i,l}) + L - r.$$

and the cost of path $P_{i,j}$ is given by

$$\begin{aligned}\phi_f^{'}(P_{i,j}) = \phi_{e,\mu}^{'}(s_i, w_{i,1}) + \sum_{l=1}^{r} \phi_{e,\mu}^{'}(w_{i,l-1}, w_{i,l}) \\ + \sum_{l=r+1}^{j} \phi_{e,\mu}^{'}(w_{i,l-1}, w_{i,l}) + L - j.\end{aligned}$$

So in any round $s, s = 1, 2, \ldots, t$ path $P_{i,r}$ would be chosen first since its cost is less than the cost of path $P_{i,j}, j > r$. This holds because both paths have a common part of the same length and a part with the same size, but in contrast to path $P_{i,r}$ there are some edges in the second part of path $P_{i,j}$ which have positive flow. Thus if path $P_{i,r}$ is chosen for the first time in round t paths $P_{i,j}, j > r$ could not be used in the previous rounds by construction. □

Lemma 21. *Path* $P_{i,r}$ *is first chosen at round* r.

Proof. We are going to prove this result by induction. Consider the set of paths $\mathcal{P}_i = \{P_{i,1}, P_{i,2}, \ldots, P_{i,L}\}$ for each commodity i. At first the exclusive path P_i is chosen. For simplicity of our arguments we are going to define this round as round 0. In round 1 path $P_{i,1}$ is chosen for the reasons stated above. We assume that for $j \geq 1$ path $P_{i,s}$ is first chosen in round s. We need to show that in round $s+1$ path $P_{i,s+1}$ is chosen for the first time.

Lemma 20 implies that path $P_{i,s+1}$ is not used in round s. We are going to argue that path $P_{i,s+1}$ is the shortest path in round $s+1$. Remember that a path $P_{i,j}$ is given by its exclusive part and its shared part. Path $P_{i,j}, j \geq s+1$ has $L-j$ edges on the exclusive part and j edges on the shared part. Lemma 20 implies that no flow has yet been rerouted on any of the paths $P_{i,j}, j \geq s+1$ so the flow in the shared part is 0. Therefore the cost of any path $P_{i,j}, j \geq s+1$ is given by

$$\begin{aligned}
\phi_f^{'}(P_{i,j}) &= \sum_{e \in P_{i,j}} \phi_{e,\mu}^{'}(f(e)) \\
&= \phi_{e,\mu}^{'}(s_i, w_1) + \phi_{e,\mu}^{'}(w_{i,1}, w_{i,2}) + \ldots \\
&+ \phi_{e,\mu}^{'}(w_{i,s}, w_{i,s+1}) + \ldots + \phi_{e,\mu}^{'}(w_{i,j-1}, w_{i,j}) + L - j \\
&= \phi_f^{'}(P_{i,s+1}) + \phi_{e,\mu}^{'}(w_{i,s+1}, w_{i,s+2}) + \ldots \\
&+ \phi_{e,\mu}^{'}(w_{i,j-1}, w_{i,j}) - (L - s - 1).
\end{aligned}$$

Since for all paths $j > s+1$ the exclusive edges have positive flow it follows that $P_{i,j}$ is the shortest path if and only if $j = s+1$. Therefore path $P_{i,s+1}$ is first chosen in round $s+1$. □

Corollary 4. *Path $P_{i,r}$ is used for the first time in round r, and no paths $P_{i,j}, j > r$ are used in this round.*

Proof. This is straightforward since if path $P_{i,r}$ is used for the first time in round t it means that edge $(w_{i,s}, w_{i,s+1})$ is saturated. Thus by construction of the network Υ_L' no subsequent paths can be used. □

Lemma 22. *After $L/4$ rounds the congestion achieved is not yet ϵ-approximate, that is $\lambda_{max} > 1 + \epsilon$.*

Proof. Assume that after a number of rounds $L/4$ we have an optimal flow. This suggests that we have a congestion $\lambda \leq 1+\epsilon$. Lemma 20, Lemma 21 and Corollary 4 suggest that the flow is distributed in the top $L/4$ paths for all commodities i. Now, the capacity of the cut containing the top $L/4$ paths is equal to $L/4$ since the capacity of each path is equal to 1. Thus the total flow of each commodity i which is equal to 2 units is distributed among these paths. Then the best congestion that can be achieved is by rerouting x flow, where x is the solution to the following equation

$$2 - x = \frac{Lx}{\frac{L}{4}}$$

Thus we get a maximum congestion equal to $8/5$. The optimal congestion that can be achieved is less than or equal to $1+\epsilon$ so the congestion achieved is not ϵ-approximate, in fact we still need to decrease the congestion by a factor of $5/8$. □

Theorem 13 follows from Lemma 22 since the GDR^{+} needs at least $\Omega(L)$ rounds to reach an optimal solution on network Υ'_L.

10.4 The Greedy Balancing Distributed Algorithm

The analysis of the Greedy Distributed Algorithm on the network Υ'_L suggests that in order to reduce the number of rounds we need to find a way to avoid the bottleneck imposed by the "exclusive" path of Υ'_L and remove the flow decrease constraints (see Section 10.3.1). As in Chapter 9 a number of approximate shortest paths cannot be used to reroute flow in the same round because they are blocked by prior paths. Again we are going to avoid this bottleneck by spreading the flow among all available approximate shortest paths using the Maximum Capacity Path (MCP) algorithm algorithm.

In this section we are going to analyze the Balancing Distributed Rerouting Multicommodity Flow algorithm (BDR-MCF) on network Υ'_L. Our algorithm fits exactly in the framework of the GDR^{+} algorithm with the only change being the way we reroute flow. Instead of rerouting flow iteratively on shortest path we are distributing the flow among a number of approximate shortest paths, calculated using the MCP algorithm. Our analysis is similar to the analysis of the BD-MCF algorithm in Chapter 9.

10.4.1 Analysis of the Greedy Balancing Distributed Algorithm on Υ'_L

In this section we are going to prove that using the MCP algorithm to calculate and distribute flow among approximate shortest paths leads to reducing the number of rounds from $\widetilde{O}(L)$ to $\widetilde{O}(\log L)$. We are going to first show how the flow is distributed in the first round and then prove the general case.

Recall that on network Υ'_L the demand for each commodity is $d_i = 2$ and the whole flow of each commodity is initially on the edge (s_i, t). The maximum congestion is $\lambda_{init} = 2$ and the final congestion achieved by the algorithm is $\lambda_{final} \geq 1$. This means that we only have one phase in the algorithm. Remember that the parameter

$$\mu \leftarrow \min\{\mu, 2^{\lfloor \log_2(\epsilon \cdot \lambda_{max}) \rfloor}\}$$

Hence μ in all rounds, assuming without loss of generality that $\epsilon = 2^{-i}$ for a positive integer i is given by

$$\begin{aligned}\mu &= 2^{\lfloor \log_2(\epsilon \cdot \lambda_{max}) \rfloor}\\ &= 2^{\lfloor \log_2(\epsilon \cdot 2) \rfloor}\\ &= 2\epsilon.\end{aligned}$$

Initially, since $f_i(e) = 0$ for all the edges $e \in \mathcal{E}$ except the edge (s_i, t), the upper limit on flow increase $\Delta_i^+(e)$ is given by

$$\Delta_i^+(e) = (1+\beta)\ddot{f},$$

where

$$\ddot{f} = \frac{2\epsilon}{\log_2 m} \frac{1}{(1+\beta)L} \log_2(1 + \frac{\alpha}{8}).$$

All paths are approximate shortest paths at the beginning of the first round. Remember that the cost of a path is given by

$$\phi(P) = \sum_{e \in P} m^{\frac{f(e)}{c(e)\cdot\mu}} = (L+1)m.$$

For a path to be an approximate shortest path its cost must be smaller than the average cost, that is

$$(1+\alpha)\phi(P) < \left(\sum_{e \in \mathcal{E}} f_i(e)\phi'_{e,\mu}(e) \right) / d_i = 2m^{1/\epsilon}.$$

The inequality clearly holds for $\epsilon < 1/2$ since $L < m^{1/2}$. Thus the number of approximate shortest paths calculated using the MCP algorithm is

$$\sigma = \sum_{j=2}^{j=L} (j+1) = \frac{L(L+3)-4}{2}. \tag{10.13}$$

Flow is then rerouted according to the size of each path. As in Section 10.5 we are going to assume that for each exclusive edge $(w_{i,j}, w_{i,j+1})$, the computed approximate shortest path is $P_{i,j+1}$. By making this assumption we "force" the top paths that congest faster, i.e. the top paths in Figure 10.2, to even faster congestion, obtaining the worst case behaviour. This holds because we "force" more flow to be sent to the top paths in each round. Recall that in each round we calculate a shortest path for each edge and if the path is ϵ-approximate we send flow through the path. Hence, if an ϵ-approximate path has more edges it will get more flow.

Let $f_{i,j}^{(r)}(e)$ denote the amount of flow of commodity i on edge e at the end of round r on path $P_{i,j}$. The total flow rerouted in each path $P_{i,j}$ in the first round is given by

$$f_{i,j}^{(1)} = \frac{L+2-j}{\sigma}(1+\beta)\ddot{f}. \tag{10.14}$$

We can calculate whether the capacity of the edge $(s_i, w_{1,1})$ is violated by summing the flow over all the paths.

$$f^{(1)}(s_i, w_{1,1}) = \sum_{j=2}^{L} \frac{L+2-j}{\sigma}(1+\beta)\ddot{f} = (1+\beta)\ddot{f}, \tag{10.15}$$

where $f^{(r)}(u,v)$ denotes the amount of flow on edge (u,v) at the end of round r. Thus, in the first round edge $(s_i, w_{1,1})$ is saturated for each commodity i.

The flow of each commodity i is concentrated at edge (s_i, t) initially and it is rerouted from this edge to the lower paths in the subsequent rounds. Effectively, the execution of the BDR-MCF algorithm on network Υ'_L simulates the execution of the BD-MCF on network Υ_L. More specifically, the paths in the lower part of the network Υ'_L are initially empty and flow is gradually built up from zero (similar to the BD-MCF algorithm on network Υ_L). Thus, we expect the two algorithms to have exactly the same execution.

Flow rerouted from the top path $P_{i,0}$ to the lower part of network Υ'_L will saturate edge $(s_i, w_{1,1})$ for as long as the top path $P_{i,1}$ is an approximate shortest path, that is as long as all paths $P_{i,j}$ are approximate shortest paths. Since edge $(s_i, w_{1,1})$ is saturated in every round it follows that the total flow on edge $(s_i, w_{1,1})$ at the end of round r is

$$f^{(r)}(s_i, w_{1,1}) = (1+\beta)^r\ddot{f}. \tag{10.16}$$

The flow rerouted from the top path $P_{i,0}$ to the lower part of network Υ'_L saturates edge $(s_i, w_{1,1})$ for as long as the path $P_{i,1}$ is an approximate shortest path. To find a bound on the number of rounds we need to first find a bound on the flow change on path $P_{i,1}$.

Lemma 23. *The upper and lower bound on the flow of commodity i on path $P_{i,j}$ at the end of round r, as long as all paths $P_{i,j}$ are approximate shortest paths, is given respectively by*

$$f_{i,j}^{(r)} \geq \frac{L+2-j}{\sigma}(1+\beta)^r\ddot{f}. \tag{10.17}$$

and,

$$f_{i,j}^{(r)} \leq 2\frac{L+2-j}{\sigma}(1+\beta)^r\ddot{f}. \tag{10.18}$$

Proof. We prove the upper bound on the flow at the end of round r. We can prove the lower bound using similar arguments. For the first round the proof is trivial. At the end of round r the flow of paths $P_{i,j}$ is given by the expressions (10.17) and (10.18). This means that the upper bound on the upper limit on flow increase of path $P_{i,j}$ at round $r+1$ is

$$\Delta_{i,j}^{(r)} \leq 2\beta\frac{L+2-j}{\sigma}(1+\beta)^{r+1}\ddot{f}.$$

The total flow of commodity i sent on path $P_{i,j}$, calculated by the MCP algorithm, is

$$\delta f_{i,j}^{(r)} \leq 2\frac{(L+2-j)^2}{\sigma^2}\beta(1+\beta)^{r+1}\ddot{f}.$$

Summing over all paths $P_{i,j}$ we get the flow on edge $(s_i, w_{i,1})$

$$\begin{aligned} f^{(r)}(s_i, w_{i,1}) &\leq \sum_{j=1}^{L} 2\frac{(L+2-j)^2}{\sigma^2}(1+\beta)^{r+1}\ddot{f} \\ &\leq \frac{L(2L^2+9L+13)}{6\sigma^2}(1+\beta)^{r+1}\ddot{f} \\ &\leq \frac{2(2L^2+9L+13)}{3L(L+3)^2}(1+\beta)^{r+1}\ddot{f} \\ &\leq \frac{2}{L}(1+\beta)^{r+1}\ddot{f}. \end{aligned}$$

It can be easily verified using similar arguments that the lower bound of the flow on edge $(s_i, w_{i,1})$ is

$$f^{(r)}(s_i, w_{i,1}) \geq \frac{1}{L}(1+\beta)^{r+1}\ddot{f}.$$

Thus, the flow needs to be scaled by a factor of $\Theta(L)$ for edge $(s_i, w_{i,1})$ to be saturated. Scaling everything by $L/2$ we get that the upper bound on the flow of path $P_{i,j}$ at the end of round r is

$$\begin{aligned}
f_{i,j}^{(r)} &\leq \frac{L}{2} \cdot 2\frac{(L+2-j)^2}{\sigma^2}\beta(1+\beta)^{r+1}\ddot{f} \\
&\leq \frac{L}{2}\frac{L+2-j}{\sigma} \cdot 2\frac{(L+2-j)}{\sigma}\beta(1+\beta)^{r+1}\ddot{f} \\
&\leq 2\frac{(L+2-j)}{\sigma}\beta(1+\beta)^{r+1}\ddot{f}.
\end{aligned}$$

□

We can now find the time needed for the top path to no longer be an ϵ-approximate shortest path.

Lemma 24. *Path $P_{i,0}$ stops being an ϵ-approximate shortest path after $2\epsilon^{-2}\log^2 m \log\left(\epsilon^{-2}\log^2 m\right)$ rounds.*

Proof. The flow rerouted keeps saturating edge $(s_i, w_{i,1})$ for as long as the top path $P_{i,1}$ is available, that is, for as long as the length of path $P_{i,1}$ satisfies the following condition

$$(1+\alpha) \cdot \phi_f^{'}(P_{i,1}) < \left(\sum_{e\in\mathcal{E}} f_i(e)\phi_{e,\mu}^{'}(e)\right)/d_i. \qquad (10.19)$$

The cost $\phi_f^{'}(P_{i,1})$ of path $P_{i,1}$ is given by

$$\begin{aligned}
\phi_f^{'}(P_{i,1}) &= \sum_{e\in P_{i,1}} m^{\frac{f(e)}{\mu c(e)}} \\
&= \sum_{e\in P_{i,1}} m^{\frac{f(e)}{2\epsilon}} \\
&\leq (L+1)m^{\frac{L(L+1)}{\epsilon\sigma}\beta(1+\beta)^{r+1}\ddot{f}} \\
&\leq (L+1)m^{2\beta(1+\beta)^r\ddot{f}}. \qquad (10.20)
\end{aligned}$$

In the third line we have substituted the upper bound on the flow of edge e. Note that we multiply this flow by L since all commodities send flow at the shared part of path $P_{i,1}$.

Since in every round r we reroute $\beta(1+\beta)^r\ddot{f}$ flow from path $P_{i,0}$ to the rest of the paths, the total flow on path $P_{i,0}$ at the end of round r is

$$f(P_{i,0}) = 2 - (1+\beta)^r\ddot{f}.$$

Thus, the cost of path $P_{i,0}$ at the end of round r is

$$\phi'_f(P_{i,0}) = m^{\frac{2-(1+\beta)^r\ddot{f}}{2\epsilon}}.$$

Hence, as long as $(1+\beta)^r\ddot{f} \geq 1$

$$\begin{aligned}\left(\sum_{e\in\mathcal{E}} f_i(e)\phi'_{e,\mu}(e)\right)/d_i &\geq (2-(1+\beta)^r\ddot{f})\cdot m^{\frac{2-(1+\beta)^r\ddot{f}}{2\epsilon}} \\ &\geq m^{\frac{2-(1+\beta)^r\ddot{f}}{2\epsilon}}. \end{aligned} \tag{10.21}$$

We can find the number of rounds needed for the top path to stop being ϵ-approximate shortest path by substituting (10.20) and (10.21) in inequality (10.19). We get

$$\begin{aligned} 2\beta(1+\beta)^r\ddot{f} &\leq \frac{2-(1+\beta)^r\ddot{f}}{2\epsilon} \\ \Rightarrow r &= \frac{2}{\log(1+\beta)}\log\ddot{f} \\ \Rightarrow r &= 2\epsilon^{-2}\log^2 m \log\left(\epsilon^{-2}\log^2 m\right). \end{aligned}$$

□

Theorem 14. *The BDR-MCF algorithm terminates in* $O\left(\epsilon^{-2}\log^2 m\log\left(\epsilon^{-2}\log^2 m\right)\right)$ *rounds.*

Proof. For the algorithm to terminate we need to reroute one unit of flow from path $P_{i,0}$ to the rest of the paths. For as long as the top path $P_{i,0}$ is available the flow rerouted is equal to the flow of the edge $(s_i, w_{i,1})$ and is given by

$$f(s_i, w_{i,1}) = (1+\beta)^r\ddot{f}. \tag{10.22}$$

Thus, the flow rerouted is greater than one unit of flow in

$$r \geq \epsilon^{-2} \log^2 m \log \left(\epsilon^{-2} \log^2 m \right) . \tag{10.23}$$

Since the total time needed to reroute one unit of flow to the lower paths is less than the total time needed for the top path $P_{i,1}$ not to be ϵ-approximate shortest path we conclude that the algorithm terminates in $O\left(\epsilon^{-2} \log^2 m \log \left(\epsilon^{-2} \log^2 m \right) \right)$ rounds. □

10.5 Summary

In this part we have presented the main distributed algorithms for the Maximum Concurrent Flow problem. We have introduced the the main design concepts of distributed algorithms in general and described some of their applications. For the Maximum Concurrent Flow problem we have examined a special distributed model which uses a billboard for communication. More specifically, we analyzed the main algorithms for solving the MCF problem under this model, the DGD-MCF algorithm [7] and the GDR-MCF Algorithm [4]. We have constructed a worst case input for both cases and proved that the bounds on the number of rounds are tight. We have also proposed a heuristic to avoid the bottleneck of these algorithms, the Balancing Distributed MCF algorithm. The algorithm instead of calculating a path to distribute flow in each round it calculates a set of suitable paths and tries to balance the flow evenly among them. This enables more aggressive flow increase in the later stages of the algorithm, thus reducing the total running time.

Chapter 11

Conclusions

11.1 Summary

In this thesis we have examined the Maximum Concurrent Flow problem. Several approximation algorithms, both sequential and distributed, have been proposed in the literature to solve this problem. We have examined both computational models and contributed towards finding relations between sequential MCF algorithms and towards further analysis of distributed MCF algorithms.

Two frameworks have been developed to solve the Maximum Concurrent Flow problem using sequential algorithms: the rerouting and the incremental. These frameworks have been previously considered as distinct. In this thesis we have shown that the two methods are more closely related to each other than previously considered. We have shown that each method can be viewed as a variation of the other method. Moreover, the implementation of our proposed algorithms is more practical. For the incremental method we have proposed a new edge length function which is exponential in the edge flows. This length function is similar to the one used in the rerouting method. Using this length function we can view the incremental method as an instance of the rerouting method. We have proved that we can use this length function in the original incremental algorithm without increasing the running time of the algorithm. For the rerouting algorithm we have proposed a different method for computing minimum cost flows than in the original algorithm. We have shown how we can do this using the Successive Shortest Path algorithm. The Successive Shortest Path algorithm has an exponential worst case running time. We have shown how we can modify the network so that we can achieve a polynomial running time. The execution of the algorithm is similar to that of the incremental framework and its implementation is better in practise than the algorithms proposed in the literature to find minimum cost flows.

Recently two distributed algorithms were proposed to solve the Maximum Concurrent Flow problem. These algorithms deviate from the "classic" distributed computing. Decisions for flow changes are not taken locally in each node but more "globally" for each commodity. Commodities communicate with each other via a billboard instead of the classical method of message passing. The two algorithms proposed are similar in execution to the sequential algorithms proposed for solving the Maximum Concurrent Flow problem. They involve incremental building and rerouting of flows as in the sequential algorithms. In this thesis we have analyzed the two distributed MCF algorithms. We have constructed a worst case input for both algorithms which shows that their running time bounds are tight. We have shown that the bottleneck computation of the algorithms comes from the way they distribute the flow. We have proposed a heuristic improvement to overcome this bottleneck by distributing the flow more evenly. We have shown that our heuristic can improve the running time at least for our worst case network.

11.2 Future Work

We have proposed a heuristic improvement (the Balancing Distributed Algorithm) for the Distributed Gradient Descent Algorithm and the Greedy Distributed Algorithm which improves the number of rounds from $\widetilde{O}(L)$ to $\widetilde{O}(\log L)$ on networks Υ_L and Υ'_L respectively. Our heuristic speeds up the process for certain types of inputs but we would like to prove these bounds on general networks.

The bottleneck of the two distributed algorithms proposed in [7, 4] is due to the way they handle the distribution of flow on available paths. On our worst-case input both algorithms could not distribute flow to "lower" paths for a substantial number of rounds because the "top" paths were blocking the flow. As a consequence the build up of flow in subsequent rounds was slow. We managed to overcome this bottleneck by carefully balancing the flow through all available approximate shortest paths in each round. An interesting question is whether it is possible to achieve a well balanced blocking flows by some appropriate randomized process. Since the agents, associated with the commodities, are operating independently and in parallel, a randomized selection of commodities (as used in sequential MCF algorithms) does not seem feasible. However, a randomized selection of paths to distribute flow in each round might work.

We believe that the running time of the Greedy Distributed algorithm [4] could be improved by computing minimum cost flows instead of shortest paths to reroute flow in each round. A minimum cost flow computation might lead to bigger improvements in each round, thus reducing the total number of rounds needed for the algorithm to terminate. Finally, we believe that the analysis of sequential MCF algorithms could be further improved. For example, we do not know whether the $O(\epsilon^{-3}km^2)$ bound on the running time of the MRR algorithm presented in Chapter 7 is tight. We believe that this bound can be improved by considering the commodities which share the same source together and calling the SSP algorithm only once for all of them at the same time.

Notation

$\Delta f^{(h,r)}(e)$ total flow change on edge e at stage h at the end of round r *p. 129*

$\delta f^{(r)}$ total amount of flow of one commodity calculated by the MCP-algorithm to be sent at round r *p. 151*

$\Delta f_j^{(r)}$ actual amount of flow of one commodity sent on path P_j in round r *p. 151*

$\delta f_j^{(r)}$ amount of flow of one commodity calculated by the MCP-algorithm to be sent on path P_j at time round r *p. 151*

$\delta f_i^{(r,j)}$ amount of flow of commodity i rerouted at the jth step of round r *p. 166*

$\Delta f_i^{(h,0)}$ total change in flow of commodity i up to stage h *p. 134*

$\Delta f_{p,i}^{s}(e)$ flow sent on edge e at step s of iteration i, phase p *p. 70*

$\Delta_i^{+}(e)$ upper bound on the total flow change allowed on an edge e within one round *p. 164*

$\Delta_i^{-}(e)$ lower bound on the total flow change allowed on an edge e within one round *p. 164*

$\Delta_i(u,v)$ difference between the height of the queue of commodity i at the tail and at the head of the edge (u,v) *p. 113*

λ^* optimal congestion *p. 42*

λ_f congestion of a given flow f *p. 28*

$\lambda_f(e)$	congestion of edge e for a given flow f *p. 28*
$\mathbb{R}_+$	set of non-negative real numbers *p. 9*
$\mathcal{E}$	set of edges *p. 8*
$\mathcal{G}_{\mathcal{R}}$	residual network *p. 11*
$\mathcal{G}$	graph consisting of a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$ *p. 8*
$\mathcal{G}^\delta$	modification of network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with the capacities of all edges $e \in \mathcal{E}$ rounded down to the units of $\delta = \epsilon d/m$ *p. 93*
$\mathcal{N}$	set of nodes *p. 8*
$\mathcal{P}_i$	set of paths between vertices s_i and t_i *p. 30*
$\mathcal{P}$	union of all the set of paths $\mathcal{P}_i$ *p. 30*
$\overline{P}_i$	longest active path from s_i to t_i for commodity i *p. 55*
Φ_l	potential under the length function l *p. 96*
Π^{SP}	length of the shortest path P^{SP} through edge e *p. 146*
Π_e^{SP}	length of the approximate shortest path P_e^{SP} through edge e *p. 146*
$\Pi_{i,j}^{(h,r)}$	length of path $P_{i,j}$ at Stage h after round r *p. 128*
$\Pi_i^{(h,r)}$	length of path P_i at Stage h after round r *p. 128*
$\rho(S)$	sparsity ratio *p. 34*
$\mathbf{f}$	concurrent flow *p. 42*
Υ_L	worst-case network for the DGD-MCF algorithm *p. 123*
$\widehat{f}_i^*$	ϵ-approximate minimum-cost flow of commodity i *p. 96*
$c(P)$	capacity of path P *p. 142*
$c(u,v)$	capacity of edge (u,v) *p. 9*

$C_i^*(\lambda)$ minimum cost flow of commodity i, subject to costs $l(e)$ *p. 58*

$c^{(r)}(e)$ capacity of edge e at the beginning of round r *p. 151*

C_i cost of the current flow of commodity i under the length function l *p. 58*

$c_R(u, v)$ residual capacity *p. 11*

d_i demand of commodity i *p. 23*

$dist_l(s_i, t_i)$ shortest path distance from s_i to t_i with respect to the edge length $l(u, v)$ *p. 29*

$f(P)$ amount of flow sent along path P *p. 30*

$f(e)$ flow on edge e *p. 28*

$f(u, v)$ flow of edge (u, v) *p. 10*

f^* minimum cost-flow of a single commodity in network $\mathcal{G}$ *p. 93*

$f^{(r)}(u, v)$ amount of flow on edge (u, v) at the end of round r *p. 182*

$f_{BF}^{(r)}$ value of blocking flow sent in round r *p. 151*

$f_{i,j}^{(r)}(e)$ value of the flow of commodity i on edge e at the end of round r on path $P_{i,j}$ *p. 181*

$f_j^{(r)}$ flow of one commodity on path P_j at the end of round r *p. 151*

$f_i^{(r,j)}(e)$ value of the flow of commodity i on edge e at the end of the jth step of round r *p. 166*

$f^{\delta *}$ minimum cost-flow of a single commodity in network $\mathcal{G}^\delta$ *p. 93*

f_i^h total flow sent for commodity i at stage h *p. 139*

f_i flow of commodity i *p. 28*

$f_i(e)$ flow of commodity i on edge e *p. 28*

$f_i(u, v)$ flow of commodity i on edge (u, v) *p. 28*

f_i^{max}	maximum flow of commodity i if routed independently *p. 71*
$F_j^{(r)}$	value of the flow on the edge $(w_{i,j-1}, w_{i,j})$ at the end of round r *p. 152*
f_{opt}	optimal flow *p. 54*
$f_{p,i}^s(e)$	total flow on edge e at the end of step s in phase p, iteration i *p. 83*
k	number of commodities *p. 23*
k^*	number of different sources s_i *p. 57*
L	maximum path size *p. 4*
$l(e)$	length of edge e *p. 31*
$l(P)$	length of path P *p. 53*
$l(u,v)$	length of edge (u,v) *p. 29*
$l_{p,i}^s(e)$	length of edge e at step s of iteration i, phase p *p. 70*
m	number of edges *p. 8*
n	number of nodes *p. 8*
P_i^{SP}	shortest path from s_i to t_i for commodity i *p. 55*
P_i	exclusive path for commodity i on network Υ_L *p. 125*
P_e^{SP}	approximate shortest path passing through edge e *p. 143*
$P_{i,j}$	shared paths for commodity i on network Υ_L *p. 125*
$q_i(v)$	height of the queue of commodity i at node v *p. 112*
r_h	number of rounds in Stage h *p. 134*
s	source (origin) *p. 10*
s_i	source of commodity i *p. 23*
t	sink (destination) *p. 10*

t_i	destination of commodity i *p. 23*
$f_i(e)$	flow of commodity i on edge e *p. 23*

Glossary

ϵ-approximate . 16

ϵ-approximate flow . 42

ϵ-bad . 59

ϵ-good . 59

active paths . 54

blocking flow . 11

capacity of a path . 11

cut . 10

directed graph . 8

distributed shared memory system . 108

distributed system . 106

fault-tolerant . 107

flow . 10

maximum concurrent flow problem . 24

maximum multicommodity flow problem . 23

Message-passing systems . 108

minimum-cost multicommodity flow problem . 23

network . 9

path . 10

queue . 111

residual capacity . 11

residual network . 11

saturated edge . 11

stateless . 115

subgraph . 9

undirected graph . 8

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993. 8, 91

[2] C. Albrecht. Global routing by new approximation algorithms for multicommodity flow. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(5):622–632, 2001. 35

[3] Y. Aumann and Y. Rabani. An $\mathcal{O}(\log k)$ approximate min-cut max-flow theorem and approximation algorithm. *SIAM Journal on Computing*, 27(1): 291–301, 1998. 40

[4] B. Awerbuch and R. Khandekar. Greedy distributed optimization of multicommodity flows. In *Proceedings of the twenty-sixth annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 274–283. ACM, 2007. 3, 4, 5, 6, 115, 116, 117, 119, 120, 162, 163, 164, 165, 166, 169, 170, 171, 186, 188, 189

[5] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 459–468, 1993. 111, 112, 113, 114

[6] B. Awerbuch and T. Leighton. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. *Journal of Computer and System Sciences*, pages 487–496, 1994. 112, 113

[7] B. Awerbuch, R. Khandekar, and S. Rao. Distributed algorithms for multicommodity flow problems via approximate steepest descent framework. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007. 3, 4, 5, 6, 111, 114, 115, 116, 117, 119, 120, 121, 122, 123, 141, 142, 161, 186, 188

[8] C. Barnhart, C. A. Hane, E. L. Johnson, and G. Sigismondi. A column generation and partitioning approach for multi-commodity flow problems. *Telecommunication Systems*, 3(3):239–258, 1994. 36

[9] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows*. Wiley, 2011. 8

[10] K. A. Berman and J. L. Paul. *Algorithms: Sequential, Parallel, and Distributed*. Thomson/Course Technology, 2005. ISBN 9780534420574. 19

[11] D. P. Bertsekas. *Network Optimization: Continuous and Discrete Models*. Athena Scientific Belmont, 1998. 7, 8

[12] D. Bienstock and O. Raskina. Asymptotic analysis of the flow deviation method for the maximum concurrent flow problem. *Mathematical Programming*, 91(3):479–492, 2002. 49

[13] T. Brunsch, K. Cornelissen, B. Manthey, and H. Röglin. Smoothed analysis of the successive shortest path algorithm. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1180–1189. SIAM, 2013. 4, 90

[14] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002. 20

[15] R.C. Carden and C. Cheng. A global router using an efficient approximate multicommodity multiterminal flow algorithm. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 316–321. ACM, 1991. 35

[16] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S. Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 273–282. ACM, 2011. 49

[17] T. G. Crainic and J. Rousseau. Multicommodity, multimode freight transportation: A general modeling and algorithmic framework for the service network design problem. *Transportation Research Part B: Methodological*, 20(3):225 – 242, 1986. 35

[18] G. B. Dantzig. *Application of the simplex method to a transportation problem.* Wiley, 1951. 38

[19] G. B. Dantzig. *Maximization of a Linear Function of Variables Subject to Linear Inequalities, in Activity Analysis of Production and Allocation.* Wiley, New York, 1951. 41

[20] G. B. Dantzig, L. R. Ford, and D. R. Fulkerson. *A Primal-Dual Algorithm for Linear Programs.* Linear Inequalities and Related Systems, Annals of Mathematics Study No. 38. Princeton University Press, 1956. 43

[21] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 20

[22] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 11–20. Springer, 2004. 20

[23] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Math Doklady*, 11:1277–1280, 1970. 38

[24] P. Elias, A. Feinstein, and C.E. Shannon. A note on the maximum flow through a network. *Information Theory, IRE Transactions on*, 2(4):117–119, 1956. 38

[25] L. K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 13 (4):505–520, 2000. 48, 71, 74, 75, 87

[26] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956. 38

[27] L. R. Ford and D. R. Fulkerson. Flows in networks. *Princeton University Press*, 1962. 40

[28] L. Fratta, M. Gerla, and L. Kleinrock. The flow deviation method: An approach to store-and-forward communication network design. *Networks*, 3 (2):97–133, 1973. 48, 49

[29] D. R. Fulkerson. Flows in networks. *In Recent Advances in Mathematical Programming, McGraw-Hill*, pages 319–332, 1963. 39

[30] N. Garg and J. Koenemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37:630–652, 2007. 48, 49, 52, 68, 69, 70, 71, 72, 73, 74, 75, 81, 82, 87, 115, 119

[31] N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM Journal on Computing*, 25:698–707, 1993. 39

[32] A. V. Goldberg. *Efficient graph algorithms for sequential and parallel computers*. PhD thesis, Massachusetts Instutute of Technology, Dept. of Electrical Engineering and Computer Science, 1987. 38

[33] A. V. Goldberg. A natural randomization strategy for multicommodity flow and related algorithms. *Inf. Process. Lett.*, 42(5):249–256, July 1992. 64, 65

[34] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988. 38

[35] A. V. Goldberg and R. E. Tarjan. Solving minumum-cost flow problems by successive approximation. *Mathematics of Operations Research*, 15:430–466, 1990. 90

[36] M. T. Hajiaghayi and H. Räcke. An-approximation algorithm for directed sparsest cut. *Information Processing Letters*, 97(4):156–160, 2006. 34

[37] N. J. Harvey, R. D. Kleinberg, and A. R. Lehman. Comparing network coding with multicommodity flow for the k-pairs communication problem. Technical report, Massachussetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, November 2004. 36

[38] T. C. Hu. Multi-commodity network flows. *Operations Research*, 11(3): 344–360, 1963. 39, 40

[39] J. Jádá. *An introduction to parallel algorithms*. Addison Wesley, 1992. 19

[40] A. Kamath and O. Palmon. Improved interior point algorithms for exact and approximate solution of multicommodity flow problems. In *SODA*, volume 95, pages 502–511. Citeseer, 1995. 41

[41] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):pp. 366–422, 1960. 41

[42] G. Karakostas. Faster approximation schemes for fractional multicommodity flow problems. *ACM Transactions on Algorithms*, 4:13:1–13:17, 2008. 48, 72, 73, 75

[43] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 302–311. ACM, 1984. 41

[44] J. A. Kelner, G. L. Miller, and R. Peng. Faster approximate multicommodity flow using quadratically coupled flows. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1–18. ACM, 2012. 49

[45] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 217–226. SIAM, 2014. 49, 50

[46] J. L. Kennington. A survey of linear cost multicommodity network flows. *Operations Research*, 26(2):209–236, 1978. 39

[47] L. G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980. 41

[48] V. Klee and G. J. Minty. How good is the simplex algorithm. Technical report, DTIC, 1970. 41

[49] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Foundations of Computer Science, Proceedings of the 31st Annual Symposium*, volume 2, pages 726–737, 1990. 39, 40

[50] P. Klein, S. Plotkin, C. Stein, and E. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487, 1994. 47, 48, 55, 56, 57, 58, 59, 64, 65

[51] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008. 106

[52] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. 43

[53] S. Lai, L.and Ho. Simultaneously generating multiple keys and multicommodity flow in networks. In *Proceedings of the 2012 IEEE Information Theory Workshop (ITW 2012)*, Lausanne, Switzerland, Sep. 3-7 2012. 36

[54] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. 46(6), 1999. 34, 39, 40

[55] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50:228–243, 1995. 48, 49, 57, 58, 59, 60, 63, 64, 65, 102

[56] F. Y. Lin and J. R. Yee. A new multiplier adjustment procedure for the distributed computation of routing assignments in virtual circuit data networks. *INFORMS Journal on Computing*, 4(3):250–266, September 1992. 36

[57] A. Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of 42nd ACM symposium of Theory of Computing*, pages 121–130, 2010. 48, 74, 75

[58] D. W. Matula and F. Shahrokhi. Sparsest cuts and bottlenecks in graphs. *Discrete Applied Mathematics*, 27(1-2):113–123, 1990. ISSN 0166-218X. 34

[59] R. D. McBride and J. W. Mamer. Solving the undirected multicommodity flow problem using a shortest path-based pricing algorithm. *Networks*, 38 (4):181–188, 2001. 36

[60] S. Muthukrishnan and T. Suel. Second-order methods for distributed approximate single- and multicommodity flow. In *Proceedings of the 2nd Int. Workshop on Randomization and Approximation Techniques in Computer Science*, pages 369–384, 1998. 113

[61] A. Nagurney and W. Zhang. Mathematical models of transportation and networks. *In Mathematical Models in Economics, Encyclopedia of Life Support Systems, UNESCO*, 2007. 36

[62] J. B. Orlin. Max flows in $\mathcal{O}(nm)$ time, or better. *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, pages 765–774, 2013. 71

[63] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Courier Corporation, 1998. 16, 43, 45

[64] Serge A. Plotkin and É. Tardos. Improved bounds on the max-flow min-cut ratio for multicommodity flows. In *STOC*, pages 691–697, 1993. 40

[65] T. Radzik. Fast deterministic approximation for the multicommodity flow problem. *Mathematical Programming*, 78:43–58, 1997. 6, 48, 64, 65, 90, 91, 92, 95, 96, 97, 98, 99, 100, 101, 102

[66] P. Raghavan. Integer programming in vlsi design. *Discrete Applied Mathematics*, 1992. 35

[67] D. Ronen. Cargo ships routing and scheduling: Survey of models and problems. *European Journal of Operational Research*, 12(2):119–126, 1983. 35

[68] M. Sarrafzadeh and C.K. Wong. *An Introduction to VLSI Physical Design*. McGraw-Hill Higher Education, 1996. 35

[69] A. Schrijver. On the history of the transportation and maximum flow problems. *Mathematical Programming*, 91(3):437–445, 2002. 38

[70] F. Shahrokhi and D.W. Matula. The maximum concurrent flow problem. *Journal of the Association for Computing Machinery*, 37:318–334, 1990. 47, 48, 53, 54, 55, 56, 57, 58, 65

[71] D. B. Shmoys. Cut problems and their application to divide-and-conquer. *Approximation algorithms for NP-hard problems*, pages 192–235, 1997. 34

[72] K. Steiglitz and C. H. Papadimitriou. Combinatorial optimization: Algorithms and complexity. *Prentice Hall, New Jersey*, 5:231–246, 1982. 45

[73] S. M. Thampi. Introduction to distributed systems. *CoRR*, abs/0911.4395, 2009. 106

[74] A. Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196:109–130, 1998. 109

[75] S. Tragoudas. *VLSI Partitioning Approximation Algorithms Based on Multicommodity Flow and Other Techniques*. PhD thesis, University of Texas at Dallas, 1991. 39

[76] 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. 41

[77] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990. 109

[78] J. G. Wardrop. Some theoretical aspects of road traffic research-road paper. In *Proceedings of the Institution of Civil Engineers*, volume 1, pages 325–362. Road Engineering Division, 1952. 36

[79] W. W. White and A. M. Bomberault. A network algorithm for empty freight car allocation. *IBM Systems Journal*, 8(2):147–169, 1969. 35

[80] D. P. Williamson. The primal-dual method for approximation algorithms. *Mathematical Programming*, 91(3):447–478, 2002. 43

[81] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. 8

[82] N. E. Young. Randomized rounding without solving the linear program. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 170–178, 1995. 68, 69

[83] N. Zadeh. A bad network problem for the simplex method and other minimum cost flow algorithms. *Mathematical Programming*, 5(1):255–266, 1973. 93

[84] M. Zaharia, M.and Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010. 20

[85] Z.Király and P. Kovács. Efficient implementations of minimum-cost flow algorithms. *CoRR*, abs/1207.6381, 2012. 4, 90