APPROXIMATION ALGORITHMS FOR COVERING AND PACKING PROBLEMS ON PATHS

ARINDAM PAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
NOVEMBER 2012

APPROXIMATION ALGORITHMS FOR COVERING AND PACKING PROBLEMS ON PATHS

by

ARINDAM PAL

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of

Doctor of Philosophy

to the

Indian Institute of Technology Delhi

November 2012

To my family for their love, support and patience

Certificate

This is to certify that the thesis titled **Approximation Algorithms for Covering and Packing Problems on Paths** being submitted by Arindam Pal for the award of the degree of Doctor of Philosophy in Computer Science and Engineering is a record of original bonafide research work carried out by him under our guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree or diploma.

Naveen Garg
Professor
Department of
Computer Science and Engineering
Indian Institute of Technology Delhi

Amit Kumar
Professor
Department of
Computer Science and Engineering
Indian Institute of Technology Delhi

Acknowledgments

I am grateful to my doctoral research advisors Professor Amit Kumar and Professor Naveen Garg for being a constant source of inspiration. They gave me the freedom to work on problems that I like. They patiently listened to my ideas, even when some of them were not so great. They also gave valuable suggestions to improve the ideas and gave many ideas of their own. I am indebted to them for their help and advice during my graduate studies. This thesis would not have been possible without their cooperation.

I learned a great deal from my professors – Sandeep Sen, Amitabha Bagchi and Ragesh Jaiswal. The courses taught by them (along with my advisors) built the foundation of my research on different topics of mathematics and theoretical computer science. In addition, I had the good fortune of working with them on some research problems, which also shaped my thoughts on theoretical research.

I would also like to thank my coauthors Sambuddha, Venkat, Yogish, Prashant and Saurav. Without their help, many of the results in my thesis would not have seen the light of the day. Thank you very much for all those discussions and ideas that we had over the last few years.

I spent a great time at IIT Delhi with my friends Muralidhara, Rudra, Ayesha, Syamantak, Anamitra, Shibashis, Pravesh, Brojeswar, Manoj, Anuj, Chinmay, Swati and Mona. Sorry if I forgot anyone's name. You gave me company during both good and bad times. I will fondly remember and cherish those moments in the years to come.

I thank my good friends Dinesh, Vinay and Gopalda for giving me company. Special thanks to Dinesh for those intensive discussions at IBM Research and at my home. Thanks to Vinay for the good times we had in JNU. Thanks also to Gopalda for being such a good friend and mentor. I will always remember the good moments that I spent with all of you.

My special thanks to Roger Federer for playing such great game of tennis and providing quality entertainment over the last 10 years. You showed me by your own example, that hard work,

determination, dedication and discipline can help a man to reach great heights. I love you Roger!

Last but not the least, I would like to thank my family members – my uncle Dr Debi Prasad Pal (Jatha), late father Bani Prasad (Baba), mother Bani (Ma), wife Sushmita, sister Anindita (Didi), brother-in-law Kanai (Dada), and nephews Arkajyoti and Debajyoti (Bhagna). Without their moral support, encouragement and cooperation, this thesis would not have been possible. I affectionately dedicate this thesis to them.

Abstract

Routing and scheduling problems are fundamental problems in combinatorial optimization, and also have many applications. Most variations of these problems are NP-Hard, so we need to use heuristics to solve these problems on large instances, which are fast and yet come close to the optimal value. In this thesis, we study the design and analysis of approximation algorithms for such problems. We focus on two important class of problems. The first is the Unsplittable Flow Problem and some of its variants and the second is the Resource Allocation for Job Scheduling Problem and some of its variants. The first is a *packing* problem, whereas the second is a *covering* problem.

In the Unsplittable Flow Problem, we are given a path or a tree, each edge of which has a capacity. We are also given a set of requests, each of which has a start vertex, an end vertex, a demand and a profit. The objective is to select a subset of requests so as to maximize the total profit, subject to the condition that on every edge the total demand of the selected requests is at most it's capacity. We also study variants of this problem such as Unsplittable Flow Problem with Rounds and Unsplittable Flow Problem with Bag Constraints. We give constant factor approximation algorithms for all of these problem on paths and trees under the *no-bottleneck assumption*. We also give a constant factor competitive algorithm for the Online Interval Coloring problem.

In the Resource Allocation for Job Scheduling Problem, the timeline is divided into a set of discrete timeslots.. We are given a set of jobs, each of which has a start time, an end time and a demand requirement. We are also given a set of resources, each of which has a start time, an end time, a capacity and a cost. A feasible solution is a set of resources satisfying the constraint that at any timeslot, the sum of the capacities offered by the resources is at least the demand required by the jobs active at that timeslot, *i.e.*, the selected resources must cover the jobs. The objective is to select a subset of resources of minimum cost, which will cover all the jobs.

This is called the *resource allocation problem* (ResAll). We consider the partial covering version (PartialResAll) and the prize-collecting version (PrizeCollectingResAll) of this problem. We give an $O(\log(n+m))$-approximation algorithm for the PartialResAll problem, where n is the number of jobs and m is the number of resources respectively. We also give a 4-approximation algorithm for the PrizeCollectingResAll problem.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we study several important classes of covering and packing problems restricted to paths. In the class of covering problems, each edge (or a consecutive set of edges) of a path has a *demand*, and we would like to allocate resources to meet the demands under various constraints. We broadly call this class of problems *resource allocation for job scheduling*. In the packing scenario, we consider the problems where each edge has a *capacity*, and we would like to route demands under these constraints. We broadly call this class of problems *routing problems in communication networks*.

Many combinatorial optimization problems which are NP-Hard on general graphs remain NP-Hard on paths. A path is a natural setting for modeling many applications, where a limited resource is available and the amount of the resource varies over time. Many routing and scheduling problems fit into this framework. For packing problems, we can represent time instants as vertices, time intervals as edges and the amount of resource available in a time interval as the capacity of the corresponding edge. The requirement of a resource between two time instants can be represented as a demand between the corresponding vertices with a certain profit associated with it. Similarly for covering problems, we can think of the time interval between two time instants as jobs, whose demands must be satisfied on every time interval on their span by the resources.

1.1 Preliminaries

In this section, we define the notation and terminology that we will use throughout the thesis. We work with undirected graphs, unless stated otherwise. We begin with some definitions.

1.1.1 Approximation algorithms and approximation factors

Since almost all the problems considered in this thesis are NP-hard, it is unlikely that there exist polynomial-time algorithms to compute the optimal solution for them. So, our goal will be to compute an approximate solution, which is close to the optimal solution. An *α-approximation algorithm* for an optimization problem Π is a polynomial-time algorithm that for all instances of the problem produces a solution whose value is within a factor of α of the value of an optimal solution for that instance. If $\mathrm{ALG}(I)$ is the value of the solution computed by an algorithm and $\mathrm{OPT}(I)$ is the value of the optimal solution on input instance $I \in \Pi$ then, $\mathrm{OPT}(I) \leq \mathrm{ALG}(I) \leq \alpha \cdot \mathrm{OPT}(I)$ (for *minimization* problems) or $\mathrm{OPT}(I) \geq \mathrm{ALG}(I) \geq \alpha \cdot \mathrm{OPT}(I)$ (for *maximization* problems) for every instance I. The number α is called the *approximation factor* of the algorithm.

1.1.2 Online algorithms and competitive ratios

In the online setting, data arrives over time, and at each point of time the algorithm has to maintain a solution for the data that has already arrived. In contrast, an *offline algorithm* has the entire input available for processing. Often, we can't hope to compute the optimal solution without seeing the whole input in advance. Let σ be an input sequence and let $\mathrm{ALG}(\sigma)$ and $\mathrm{OPT}(\sigma)$ be the costs of the solution of the algorithm and the optimal offline solution on σ. An online algorithm is *ρ-competitive* if for every sequence σ, $\mathrm{OPT}(\sigma) \leq \mathrm{ALG}(\sigma) \leq \rho \cdot \mathrm{OPT}(\sigma)$ (for *minimization* problems) or $\mathrm{OPT}(\sigma) \geq \mathrm{ALG}(\sigma) \geq \rho \cdot \mathrm{OPT}(\sigma)$ (for *maximization* problems). The number ρ is called the *competitive ratio* of the algorithm.

1.2 Routing problems in communication networks

A *communication network* consists of *nodes* communicating with each other through a set of *links* interconnecting these nodes. We can think of these nodes as transmitters and receivers and the links as channels. Each channel has some *capacity* or *bandwidth*. A fundamental problem in communication networks is to allocate bandwidth and assign paths to connection requests. A *connection request* consists of two nodes called its *source* and *destination*. There is a *demand* associated with the request. The objective is to allocate bandwidth on some path from source to destination to satisfy the demand. Since there are several requests, it may not be possible to satisfy all demands without exceeding the capacities of some channels.

Most of these problems can be modeled as variants of the *multicommodity flow problem* in a graph. Here, we are given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges of the graph. Each edge $e \in E$ has a *capacity* $c_e \equiv c(e)$. We are also given a set of *requests* $\mathcal{R} = \{R_1, \ldots, R_k\}$. Each request R_i has a *source vertex* s_i, a *destination vertex* t_i and a bandwidth *demand* d_i. Sometimes, there is also a *profit* w_i associated with R_i. The goal is to route the requests without violating any edge capacity. This is the *feasibility* condition. The objective function that we want to optimize varies for different problems. Here are some natural objective functions.

1. What is the *maximum* number of requests that can be satisfied feasibly?

2. What is the *minimum* number of rounds required to satisfy *all* requests, so that in every round the set of requests that are satisfied are feasible?

1.2.1 Notations

We summarize the symbols we will use along with their meanings in Table 1.1 on page 24.

1.2.2 Problem definition and motivation

We now define the various problems that we will study.

Symbol	Explanation
$c_{\max}, c_{\min}$	Maximum and minimum capacities.
$d_{\max}, d_{\min}$	Maximum and minimum demands.
$w_{\max}, w_{\min}$	Maximum and minimum profits.
α	Expansion of a graph.
Δ	Maximum degree of a graph.
ω	Maximum clique size of a graph.
r	Maximum edge congestion of a graph.

Table 1.1: Notations used in the thesis

MAX-EDP (MAXIMUM EDGE-DISJOINT PATHS PROBLEM)

Input: Graph $G = (V, E)$, requests $\mathcal{R} = \{(s_i, t_i) : i = 1, \ldots, k\}$.

Output: A feasible subset of requests $S \subseteq \mathcal{R}$ along with a path P_i connecting (s_i, t_i) for all $i \in S$, such that P_i and P_j are edge-disjoint for $i \neq j$.

Objective: Maximizing the number of feasible requests $|S|$.

MAX-UFP (THE UNSPLITTABLE FLOW PROBLEM)

Input: Graph $G = (V, E, c)$, requests $\mathcal{R} = \{(s_i, t_i, d_i, w_i) : i = 1, \ldots, k\}$.

Output: A feasible subset of requests $S \subseteq \mathcal{R}$ along with a path P_i connecting (s_i, t_i) for all $i \in S$.

Objective: Maximizing the total profit $\sum_{i \in S} w_i$.

ROUND-UFP (UNSPLITTABLE FLOW PROBLEM WITH ROUNDS)

Input: Graph $G = (V, E, c)$, requests $\mathcal{R} = \{(s_i, t_i, d_i) : i = 1, \ldots, k\}$.

Output: Partition $\mathcal{R}$ into a number of sets such that each set is feasible.

Objective: Minimizing the total number of sets.

BAG-UFP (UNSPLITTABLE FLOW PROBLEM WITH BAG CONSTRAINTS)

Input: Graph $G = (V, E, c)$, *bags* of requests $\mathcal{R}^1, \ldots, \mathcal{R}^p$, where each bag $\mathcal{R}^j = \{(s_i^j, t_i^j, d_i^j) : i = 1, \ldots, k\}$ has a profit w^j.

Output: A subset of bags B and at most one request from each bag along with the paths for all selected requests such that the set of requests are feasible.

Objective: Maximizing the total profit $\sum_{\mathcal{R}^j \in B} w^j$.

We will study these problems when the input graph is a path or a tree. Note that there is a unique path between any two vertices, and so we only need to figure out which requests to choose. These problems when restricted to a path can also be used for modeling a time-varying resource. For each time t, we have a vertex. The capacity of the edge $(t, t+1)$ denotes how much resource is available.

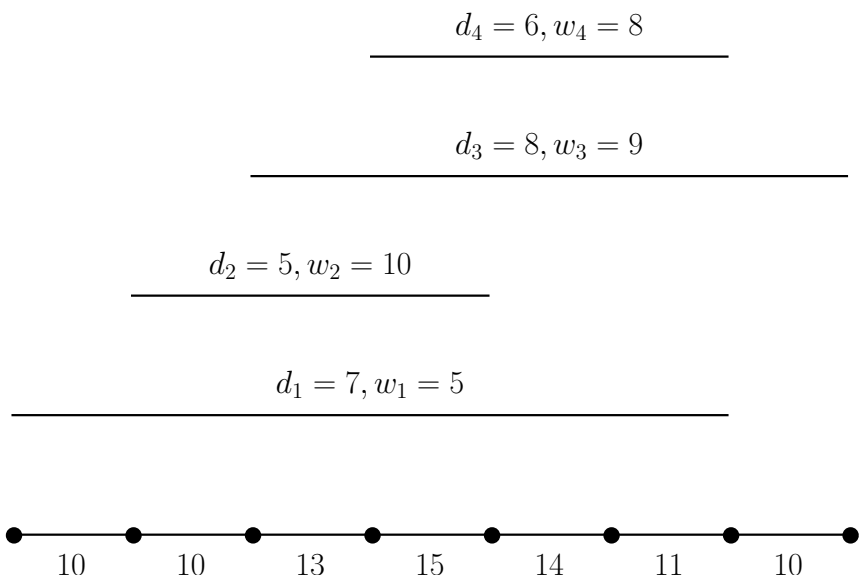


Figure 1.1: A sample Max-UFP instance

In Max-UFP, there are a set of users who want to use different amounts of this resource over different time intervals and are ready to pay for this. The goal is to select a subset of these users to maximize the profit, while satisfying the resource availability constraint at each instant, *i.e.*, the total demand of selected users at any instant does not exceed the resource available. An example of Max-UFP is shown in Figure 1.1.

The concept of *bag constraints* (at most one request can be selected from each bag) in Bag-UFP is quite powerful. Apart from handling the notion of release time and deadline, it can also work in a more general setting where a job can specify a set of possible time intervals where it can be scheduled. Moreover, it allows for different instances of the same job to have different bandwidth

requirements, processing times and profits.

In ROUND-UFP, we can model the number of *copies* of the time-varying resource needed to satisfy all requests. This can also model routing in optical networks, where each copy of the resource corresponds to a distinct frequency. As the number of distinct available frequencies is limited, minimizing the number of rounds for a given set of requests is a natural objective.

One important assumption we make is the *no-bottleneck assumption* (NBA), which states that the maximum demand requirement of any request is at most the minimum edge capacity, i.e., $\max_i d_i \leq \min_e c_e$. Note that this assumption is stronger than the *feasibility* requirement, which says that the demand of any request is at most the minimum edge capacity on its source-sink path. This is a standard assumption in these settings. From a practical perspective, it should hold. From an algorithmic perspective, it is needed to ensure that the integrality gap of the linear programming relaxation is small.

1.2.3 Related work

MAX-EDP is NP-hard, even for restricted classes of graphs like planar graphs. However, MAX-EDP can be solved optimally in polynomial time for some classes of graphs. When the graph is a path, it translates to finding the maximum number of pairwise disjoint intervals. This is equivalent to finding a maximum independent set in an interval graph, which can be done in linear time [35]. For trees, a polynomial time algorithm was given in [34]. In undirected rings, MAX-EDP can also be solved optimally in polynomial time [54]. For undirected graphs, MAX-EDP is known to be APX-hard [30] and there is no algorithm with approximation factor $\Omega((\log n)^{\frac{1}{2}-\epsilon})$ for any $\epsilon > 0$, unless $\mathbf{NP} \subseteq \mathbf{ZPTIME}(n^{\text{polylog}(n)})$ [2]. Here, $\mathbf{ZPTIME}(n^{\text{polylog}(n)})$ is the set of languages that have randomized algorithms that always give the correct answer and have expected running time $n^{\text{polylog}(n)} \equiv n^{O(\log^{O(1)} n)}$. For directed graphs, there is no algorithm with approximation factor $\Omega(m^{\frac{1}{2}-\epsilon})$ for any $\epsilon > 0$, unless $\mathbf{P} = \mathbf{NP}$ [36]. For directed graphs, MAX-EDP has a bounded-length greedy (BGA) $O(\sqrt{m})$-approximation algorithm [42].

MAX-EDP has also been studied for special type of graphs. For bounded-degree expander graphs, Kleinberg and Rubinfeld [43] showed that the bounded-length greedy algorithm gives an

$O(\log n \log\log n)$-approximation. Kolman and Scheideler [45] gave an improved $O(\log n)$-approximation by using the fact that routing number for expanders is $O(\log n)$. For two-dimensional meshes, Kleinberg and Tardos [44] gave a randomized polynomial-time algorithm that achieves a constant-factor approximation with high probability. For hypercubes, Kolman and Scheideler [46] gave an $O(\log n)$-approximation by exploiting the fact that hypercubes have flow number $O(\log n)$.

Round-UFP is NP-Hard, since it contains the Bin Packing problem as a special case, where the graph is just a single edge. Bin Packing is known to be APX-hard, so a *polynomial time approximation scheme* (PTAS) is not possible. However, it has an *asymptotic polynomial time approximation scheme* (APTAS). There are also simple greedy algorithms like *first-fit* and *best-fit*, which give constant-factor approximations [13, 37, 53, 55]. When all capacities and demands are 1, Round-UFP reduces to the interval coloring problem on paths, for which a simple greedy algorithm gives the optimal coloring in linear time.

The Round-UFP problem for paths has been well-studied in the context of online algorithms. Here the demands (intervals) arrive in arbitrary order, and we need to assign them a color on their arrival so that all intervals with one color form a feasible packing, *i.e.,* total demand on any edge does not exceed its capacity. In this context, it is also called the *interval coloring* problem. When all capacities and demands are 1, *i.e.*, when no two intersecting intervals can be given the same color, the first-fit algorithm achieves a constant competitive ratio. Kierstead [40] first proved that first-fit requires at most 40ω colors to color an interval graph with clique size ω. Later Kierstead and Qin [41] improved it to 26ω. Subsequently, Pemmaraju et al. [51] improved it to 8ω, which is currently the best known upper bound. Chrobak and Slusarek [22] showed that first-fit uses at least 4.4ω colors in the worst case. Kierstead and Trotter [39] gave a different online algorithm which uses at most $3\omega - 2$ colors. They also proved that any deterministic online algorithm in the worst case will require at least $3\omega - 2$ colors, so this algorithm is the best possible one can hope for.

Adamy and Erlebach [1] introduced the *interval coloring with bandwidth* problem. In this problem, all edge capacities are 1 and each interval has a demand in $(0, 1]$. They gave a 195-competitive algorithm for this problem. Later, the competitive ratio was improved to 10 by Narayanaswamy [49]

and Azar et al. [3]. Epstein et al. [28] further generalized the problem by allowing arbitrary edge capacities and arbitrary demands. They gave a 78-competitive algorithm for this problem satisfying the no-bottleneck assumption (NBA). Without NBA, they gave a $O\left(\log\left(\frac{d_{\max}}{c_{\min}}\right)\right)$-competitive algorithm. They also showed that without this assumption, there is no deterministic online algorithm for interval coloring with nonuniform capacities and demands, that can achieve a competitive ratio better than $\Omega(\log\log n)$ or $\Omega\left(\log\log\log\left(\frac{c_{\max}}{c_{\min}}\right)\right)$. Here, $c_{\max}$ and $c_{\min}$ are the maximum and minimum edge capacities of the path respectively.

Round-UFP has been studied on trees and meshes ($n \times n$ two-dimensional grids) for the special case when all capacities and demands are 1. Bartal and Leonardi [9] gave an online algorithm for trees with competitive ratio $O(\log n)$. The also showed that any online algorithm for trees cannot have competitive ratio better than $\Omega\left(\frac{\log n}{\log\log n}\right)$. For meshes, they gave matching upper and lower bounds of $O(\log n)$.

Max-UFP and Bag-UFP are *weakly* NP-Hard, since they contain the Knapsack problem as a special case, where the graph is just a single edge. For Knapsack, an FPTAS is known, and it has a simple greedy 2-approximation algorithm [53, 55]. When all capacities, demands and profits are 1, Max-UFP specializes to Max-EDP. Recently, it has been proved that the problem is *strongly* NP-hard, even for the restricted case where all demands are chosen from $\{1, 2, 3\}$ and all capacities are uniform [12]. However, the problem is not known to be APX-hard, so a *polynomial time approximation scheme* (PTAS) may still be possible.

With NBA, Chakrabarti et al. [18] gave the first constant factor approximation algorithm for Max-UFP on the path and the approximation ratio was subsequently improved to $(2+\epsilon)$ for any constant $\epsilon > 0$ by Chekuri et al. [21]. They also gave a constant factor approximation algorithm for Max-UFP on trees. These algorithms are based on the idea of rounding a natural LP relaxation of the Max-UFP problem. Without NBA, Bonsma et al. [12] gave a polynomial time $(7+\epsilon)$-approximation algorithm for any $\epsilon > 0$, and a 25.12-approximation algorithm with running time $O(n^4 \log n)$. Their algorithm divides the demands into three classes: small, medium and large. For small and medium demands, they use LP rounding to get a $(3+\epsilon)$-approximation algorithm. For large demands, they model this as a maximum weight independent set problem for a set of

rectangles. Using a dynamic programming based algorithm, they give a 4-approximation algorithm for large demands.

MAX-UFP has also been studied for other graph classes. We mention some of the results for cycles and trees. Under NBA, it can be shown that MAX-UFP on a cycle can be reduced to two instances of MAX-UFP on a path, by splitting the cycle at a carefully selected edge. From this, if we have a ρ-approximation for MAX-UFP on a path, we can get a $(\rho + 1)$-approximation for MAX-UFP on a cycle [18]. Hence, by using the $(2 + \epsilon)$-approximation for MAX-UFP on a path given by [21], we can immediately get a $(3 + \epsilon)$-approximation for MAX-UFP on a cycle. By directly modeling the problem as an LP, one can get an improved $(2 + \epsilon)$-approximation [21]. For trees, under NBA, there is a 4-approximation for unit demands and a 48-approximation for arbitrary demands, the profits being arbitrary in both the cases [21]. Without NBA, there is an $O(\log n)$-approximation for unit profits and an $O(\log^2 n)$-approximation for arbitrary profits, the demands being arbitrary in both cases [20].

For general graphs, Kolman and Scheideler [46] gave an $O\left(\frac{1}{\alpha}\Delta\frac{c_{\max}}{c_{\min}}\log n\right)$-approximation for MAX-UFP with NBA, when profit of a request is equal to its demand. Without NBA, under the same assumption they gave an $O(\sqrt{m})$-approximation algorithm. Azar and Regev [4] gave a combinatorial $O(\sqrt{m})$-approximation algorithm with NBA. Chakrabarti et al. [18] gave an LP-based $O\left(\frac{1}{\alpha}\Delta\log n\right)$-approximation with uniform edge capacities and an $O\left(\frac{1}{\alpha}\Delta\log^2 n\right)$-approximation with arbitrary edge capacities. Azar and Regev [4] showed that for directed graphs, there is no algorithm for MAX-UFP with approximation factor $\Omega(m^{1-\epsilon})$ for any $\epsilon > 0$, unless $\mathbf{P} = \mathbf{NP}$.

The BAG-UFP problem was introduced by Chakaravarthy et al. [17], who gave an $O\left(\log\left(\frac{c_{\max}}{c_{\min}}\right)\right)$-approximation algorithm. Chakaravarthy et al. [15] gave the first constant factor approximation algorithm for the BAG-UFP problem on paths – the approximation ratio is 120. A related problem is the job interval selection problem for which Chuzhoy et al. [26] gave an $\left(\frac{e}{e-1}\right)$-approximation algorithm. See also Erlebach et al. for some additional results [31].

The round version of BAG-UFP is hard to approximate, because scheduling jobs with interval constraints is a special case of this. Recall that here, we have a collection of n jobs where each job is associated with a set of intervals on which it can be scheduled. The goal is to minimize the

total number of machines needed to schedule all jobs subject to these interval constraints. In the continuous version, the intervals associated with a job form a continuous time segment, described by a release date and a deadline. Chuzhoy et al. [24] gave an $O\left(\sqrt{\frac{\log n}{\log\log n}}\right)$-approximation algorithm for this version. This was subsequently improved by Chuzhoy and Codenotti [23] to an $O(1)$-approximation algorithm. They also showed that the linear programming formulation for the problem has an integrality gap of $\Theta\left(\frac{\log n}{\log\log n}\right)$. In the discrete version, where the set of allowed intervals for a job is given explicitly, Raghavan and Thompson [52] gave an $O\left(\frac{\log n}{\log\log n}\right)$-approximation algorithm using randomized rounding. Chuzhoy et al. [25] proved that it is $\Omega(\log\log n)$-hard to approximate the discrete version.

1.2.4 Our contributions

There has been lot of recent work on obtaining constant factor approximation algorithms for these NP-Hard problems. Obtaining constant factor approximation algorithms for these problems without NBA remains a challenging task; the only exception being the recent result of Bonsma et al. [12] which gives a constant factor approximation algorithm for Max-UFP on the line. We will assume that NBA holds in subsequent discussions.

Linear Programming formulation for Max-UFP

A natural linear programming formulation for Max-UFP on a path is given below. Here x_i denotes the fraction of the demand i that is satisfied and I_i is the unique path between s_i and t_i.

$$\begin{aligned} \textbf{maximize} \quad & \sum_{i=1}^{k} w_i x_i && \textbf{(UFP-LP)} \\ \textbf{such that} \quad & \sum_{i:e\in I_i} d_i x_i \leq c_e && \forall e \in E \\ & 0 \leq x_i \leq 1 && \forall i \in \{1,\ldots,k\} \end{aligned}$$

If we replace the constraints $x_i \in [0,1]$ by the constraints $x_i \in \{0,1\}$ we get an integer program, which precisely models Max-UFP.

Convex decomposition of a fractional LP solution

Suppose x is a feasible fractional solution for a maximization LP and $z_1, \ldots, z_k$ are feasible integral solutions for the LP, such that $x = \sum_{i=1}^{k} \lambda_i z_i$ and $\sum_{i=1}^{k} \lambda_i = \alpha$. Then the value of the best solution, say $z_{\max}$ among $z_1, \ldots, z_k$ is at least $\frac{1}{\alpha}$ fraction of the value of x. We can think of this as approximate convex decomposition of a fractional solution.

Our results

Starting with a simple algorithm for ROUND-UFP on paths, we give a unified framework for these problems. We round natural LP relaxations for MAX-UFP and BAG-UFP. The rounding algorithm essentially shows that one can express a fractional solution to the LP as an approximate convex combination of integer solutions. We show how to do this using our algorithm for ROUND-UFP. This leads to improved approximation algorithms for several of these problems. More specifically, our results are:

- We give a 24-approximation algorithm for the ROUND-UFP problem on paths. This is much simpler than the 78-competitive algorithm of [28], and gives an improved approximation ratio.
- We give a 17-approximation algorithm for the MAX-UFP problem on paths. Although a $(2+\epsilon)$-approximation is known for this problem, our approach using convex decompositions may be of independent interest.
- We give a 65-approximation algorithm for the BAG-UFP problem on paths, thus improving the constant approximation factor of 120 given by Chakaravarthy et al. [15].
- For trees, we give the first constant factor approximation algorithm for the ROUND-UFP problem – our approximation factor is 64.

These results have appeared in [27].

For the online version of the ROUND-UFP problem on paths, we have the following result.

- We give a 58-competitive algorithm for the online version of the ROUND-UFP problem on paths. This is simpler than the 78-competitive algorithm of [28], and gives a better competitive ratio.

This result appears in [48].

1.3 Resource allocation for scheduling jobs

We consider the problem of allocating resources to schedule jobs. As before, we are give a path G, and a set of jobs. Each job j is specified by a triplet (s_j, t_j, d_j), where $[s_j, t_j]$ denotes the interval corresponding to the job (also denoted by I_j), and d_j is its demand requirement. We shall assume that d_j values are 1. Further, we are also given a set of resources. Each resource is specified by its starting and ending vertex, and the capacity it offers and its associated cost. A feasible solution is a set of resources satisfying the constraint that for any edge, the sum of the capacities offered by the resources containing this edge is at least the demand required by the jobs containing that edge, i.e., the selected resources must cover the jobs. We call this the Resource Allocation problem (RESALL).

The above problem is motivated by applications in cloud and grid computing. Consider jobs that require a common resource such as network bandwidth or storage. The resource may be available under different plans; for instance, it is common for network bandwidth to be priced based on the time of the day to account for the network usage patterns during the day. The plans may offer different capacities of the resource at different costs. Moreover, it may be possible to lease multiple units of the resource under some plan by paying a cost proportional to the number of units.

Bar-Noy et al. [6] presented a 4-approximation algorithm for the RESALL problem. We consider two variants of this problem. The first variant is the partial covering version. In this problem, the input also specifies a number k and a feasible solution is only required to cover k of the jobs. The second variant is the prize collecting version wherein each job has a penalty associated with it; for every job that is not covered by the solution, the solution incurs an additional cost, equivalent to

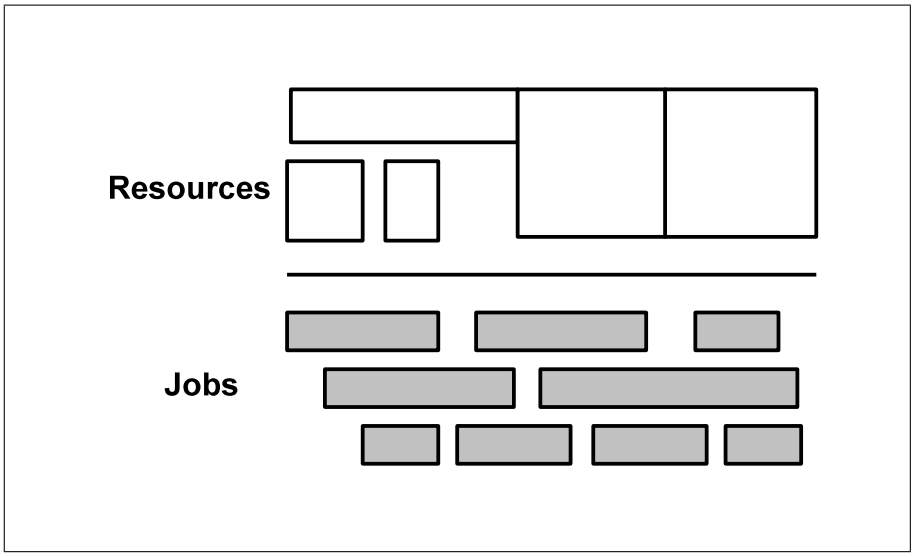


Figure 1.2: Illustration of the input

the penalty corresponding to the job. These variants are motivated by the concept of service level agreements (SLA), which stipulate that a large fraction of the client's jobs are to be completed. We study these variants for the case where the demands of all the jobs are uniform (say 1 unit) and a solution is allowed to pick multiple copies of a resource by paying proportional cost. We now define our problems formally.

1.3.1 Problem definition

We consider the graph $G = (V, E)$ which is a path with vertices numbered $1, 2, \ldots, |V|$ from left to right. An input instance consists of a set of *jobs* $\mathcal{J}$, and a set of *resources* $\mathcal{R}$. The number of jobs is n and the number of resources is m.

Each job $j \in \mathcal{J}$ is specified by an interval $I_j = [s_j, t_j]$ in the path. Recall that each job has demand requirement of 1. Each resource $i \in \mathcal{R}$ is specified by an interval $I_i = [s(i), e(i)]$ in the path, capacity w_i and cost c_i. We shall assume that the capacities w_i are integers. We interchangeably refer to the resources as *resource intervals*. We shall also refer to the interval I_j (or I_i) as the *span* of the job j (or resource i). A typical scenario of such a collection of jobs and resources is shown in Figure 1.2. We say that a job j (or resource i) *contains* an edge e if the associated interval I_j

(or I_i) contains e; we denote this as $j \sim e$ ($i \sim e$). We define a *profile* $P : E \to \mathbb{N}$ to be a mapping that assigns an integer value to every edge of the path. For two profiles, P_1 and P_2, P_1 is said to *cover* P_2, if $P_1(e) \geq P_2(e)$ for all $e \in E$. Given a set J of jobs, the profile $P_J(\cdot)$ of J is defined to be the mapping determined by the cumulative demand of the jobs in J, i.e. $P_J(e) = |\{j \in J : j \sim e\}|$. Similarly, given a multiset R of resources, its profile is: $P_R(e) = \sum_{i \in R : i \sim e} w_i$ (taking copies of a resource into account). We say that R *covers* J if P_R covers P_J. The cost of a multiset of resources R is defined to be the sum of the costs of all the resources (taking copies into account).

We now formally define the problems.

- ResAll: In this problem, a feasible solution is a multiset of resources R, which covers the set of *all* jobs $\mathcal{J}$. The cost of the solution is the sum of the costs of the resources in R (taking copies into account). The problem is to find a feasible solution of minimum cost.
- (0-1)-ResAll: This is similar to the ResAll problem, except that a resource can be used at most once to cover any job.
- PartialResAll: In this problem, the input also specifies a number k (called the *partiality parameter*) that indicates the number of jobs to be covered. A feasible solution is a pair (R, J) where R is a multiset of resources and J is a set of jobs such that R covers J and $|J| \geq k$. The cost of the solution is the sum of the costs of the resources in R (taking copies into account). The problem is to find a feasible solution of minimum cost.
- PrizeCollectingResAll: In this problem, every job j also has a penalty p_j associated with it. A feasible solution is a pair (R, J) where R is a multiset of resources and J is a set of jobs such that R covers J. The cost of the solution is the sum of the costs of the resources in R (taking copies into account) and the penalties of the jobs not in J. The problem is to find a feasible solution of minimum cost.

1.3.2 Related work

Our work belongs to the class of *partial* covering problems, which are a natural variant of the corresponding full cover problems. There is a significant body of work that consider such problems

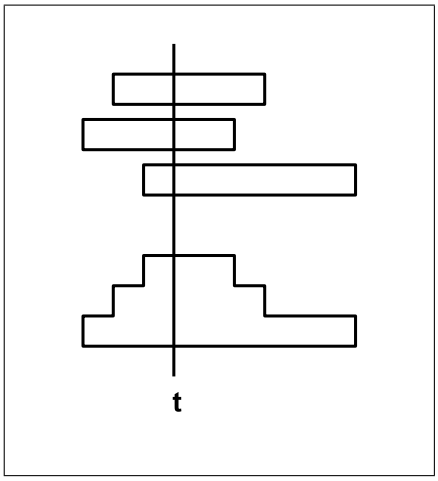


Figure 1.3: A Mountain M

in the literature, for instance, see [33, 8, 38, 47, 32].

In the setting where resources and jobs are embodied as intervals, the objective of finding a minimum cost collection of resources that fulfill the jobs is typically called the *full cover* problem. Full cover problems in this context have been dealt with in various earlier works [6, 11, 19]. Partial cover problems in the interval context have been considered earlier in [14].

The work in existing literature that is closest in spirit to our result is that of Bar-Noy et al. [6], and Chakaravarthy et al. [14]. In [6], the authors consider the full cover version, and present a 4-approximation algorithm. In this case, all the jobs have to be covered, and therefore the demand profile to be covered is fixed. The goal is to find the minimum cost set of resources, for covering this profile. In our setting, we need to cover only k of the jobs. A solution needs to select k jobs to be covered in such a manner that the resources required to cover the resulting demand profile has minimum cost.

In [14], the authors consider a scenario, wherein the edges have demands and a solution must satisfy the demand for at least k of the edges(PartialMultiResAll). They give a 16-approximation algorithm for the PartialMultiResAll problem. They also give a 4-approximation algorithm for the (0-1)-ResAll problem, where each resource can be used at most once. This is a generalization of the ResAll problem, where each resource can be used any number of times. In contrast, in our

setting, a solution needs to satisfy k *jobs*, wherein each job can span multiple edges. A job may not be completely spanned by any resource, and thus may require *multiple* resource intervals for covering it.

Jain and Vazirani [38] provide a general framework for achieving approximation algorithms for partial covering problems, wherein the prize collecting version is considered. In this framework, under suitable conditions, a constant factor approximation for the prize collecting version implies a constant factor approximation for the partial version as well. However, their result applies only when the prize collecting algorithm has a certain strong property, called the *Lagrangian Multiplier Preserving* (LMP) property. While we are able to achieve a constant factor approximation for the PrizeCollectingResAll problem, our algorithm does not have the LMP property. Thus, the Jain-Vazirani framework does not apply to our scenario.

1.3.3 Our contributions

A collection of jobs M is called a *mountain*, if there exists a edge e such that all the jobs in this collection contain the edge e; (see Figure 1.3; jobs are shown on the top and the profile is shown below). The justification for this linguistic convention is that if we look at the profile of such a collection of jobs, the profile forms a bimodal sequence, increasing in height until the peak, and then decreasing. The *span* of a mountain is the set of edges which are contained in one of the jobs in the mountain. A collection of jobs $\mathcal{M}$ is called a *mountain range*, if the jobs can be partitioned into a sequence $M_1, M_2, \ldots, M_r$ such that each M_i is a mountain and the spans of any two mountains are non-overlapping (see Figure 1.4).

We show that the input set of jobs can be partitioned into a logarithmic number of mountain ranges. Then we give a constant factor approximation algorithm for the special case of the PartialResAll problem, where the input set of jobs form a single mountain range $\mathcal{M}$. Using these two results along with dynamic programming, we get an approximation algorithm for the PartialResAll problem.

We give a approximation factor preserving reduction from the PrizeCollectingResAll problem to a certain full-cover problem and then use the approximation algorithm for that problem to

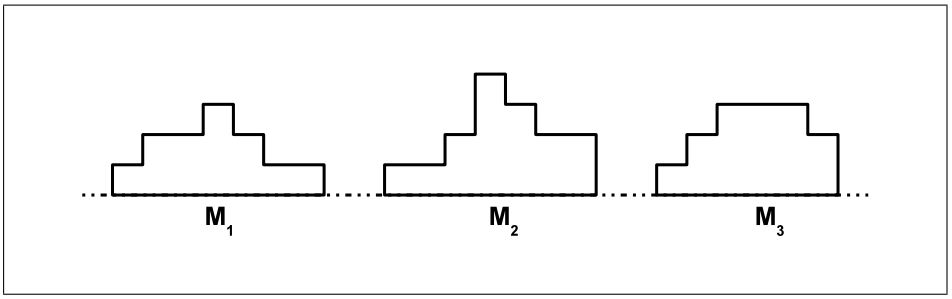


Figure 1.4: A Mountain Range $\mathcal{M} = \{M_1, M_2, M_3\}$

derive an approximation algorithm for the PrizeCollectingResAll problem.

Our results

- We present an $O(\log(n + m))$-approximation algorithm for the PartialResAll problem, where n is the number of jobs and m is the number of resources respectively.
- We give a 4-approximation algorithm for the PrizeCollectingResAll problem, by reducing it to the (0-1)-ResAll problem.

These results have appeared in [16].

1.4 Organization of the thesis

In Chapter 2, we study the Round-UFP problem and give constant factor approximation algorithms for this problem on paths and trees. Building on this, we give constant factor approximation algorithms for the Max-UFP and the Bag-UFP problems in Chapter 3. In Chapter 4, we study the online version of the Round-UFP problem, also known as the Online Interval Coloring problem, and give an improved constant factor competitive algorithm. We discuss the PartialResAll and the PrizeCollectingResAll problems in Chapter 5 and give $O(\log(n+m))$-approximation and 4-approximation algorithms for these two problems respectively. We conclude the thesis in Chapter 6 and discuss possible future directions on these problems along with some open problems.

Chapter 2

The Round-UFP Problem

We define the ROUND-UFP problem and give a constant factor approximation algorithm under NBA. We also give improved algorithms for some special cases of this problem.

2.1 Preliminaries

We are given a graph $G = (V, E)$, which is either a path or a tree, with edge capacities c_e for all edges $e \in E$. We are also given a set of requests $R_1, \ldots, R_k$. Request R_i has an associated source-sink pair (s_i, t_i) and a demand d_i. We shall use I_i to denote the associated unique path between s_i and t_i in G. A subset of demands will be called *feasible* if they can be routed without violating the edge capacities. The goal is to partition the set of demands into minimum number of colors, such that demands with a particular color are feasible.

Definition 2.1. *The* load *on an edge e, $l_e = \sum_{i:e \in I_i} d_i$, i.e., the total demand passing through the edge e.*

Definition 2.2. *The* congestion *of an edge e, $r_e = \left\lceil \frac{l_e}{c_e} \right\rceil$, i.e., the ratio of the load on the edge e to its capacity. Let $r = \max_{e \in E} r_e$ be the* maximum congestion *on any edge in the input graph.*

2.2 Approximation Algorithms for Round-UFP on Paths

2.2.1 A 3-approximation algorithm for uniform capacities and arbitrary demands

We consider the special case, where each edge of the path has a capacity c. We separate the demands into large and small demands. A demand d_i is called *large* if $d_i > \frac{1}{2}c$. Otherwise, it is called *small*. Let $\mathrm{OPT}(L)$ and $\mathrm{OPT}(S)$ be the optimum number of colors required for the instance containing only large demands and only small demands respectively. The algorithms for large and small demands are given below.

An optimal algorithm for large demands

We maintain several copies of the path, one copy for each color. We fill demands in the copies in an iterative manner. We sort the demands based on their left endpoints. Let $\mathcal{R}_i$ be the set of requests starting at $v_i, 1 \leq i \leq n-1$. We will pack the requests in $\mathcal{R}_1, \ldots, \mathcal{R}_{n-1}$ in this order. Starting with the requests in $\mathcal{R}_1$, we try to allocate the requests in $\mathcal{R}_i, 1 \leq i \leq n-1$ in one of the copies of the path, if it does not violate any edge capacities. Otherwise, we allocate a new copy and assign it there.

Lemma 2.1. *If χ is the number of colors required to pack all the large demands, then* $\mathrm{OPT}(L) \geq \chi$.

Proof. Note that if two large demands share any edge, they can't be given the same color, because the total load on the edge is more than c. Consider the demand d for which the last color χ was opened. Since d could not be assigned any one of the first $\chi-1$ colors, there are $\chi-1$ large demands, one for each color, which shared an edge with d. Since the demands have been considered in a left to right manner, all these $\chi - 1$ large demands will pass through the first edge e of d. Together with d, there are χ large demands passing through the edge e. Hence, the optimum has to give each of them a separate color, so it will also require at least χ colors. Hence, this algorithm uses the minimum number of colors. □

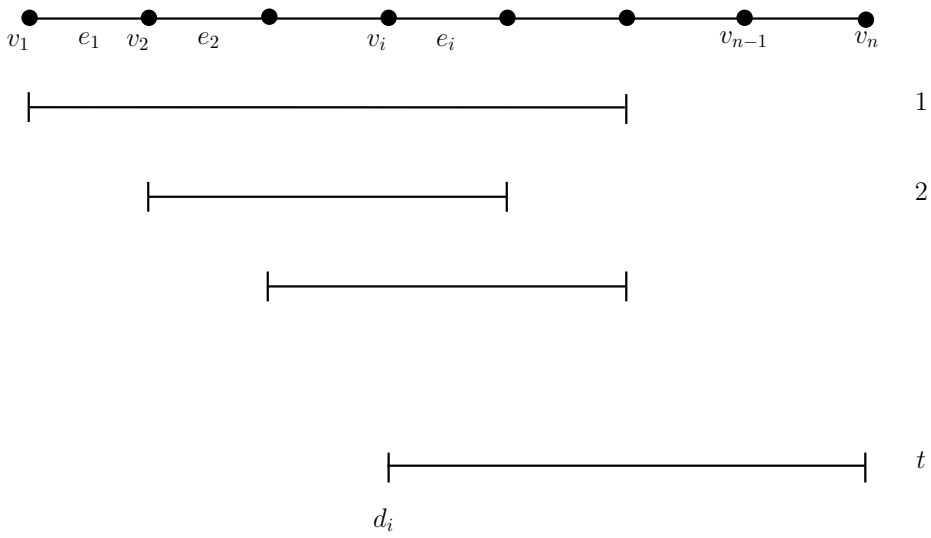


Figure 2.1: Illustration for the analysis of small demands.

A 2-approximation algorithm for small demands

The algorithm for small demands is exactly the same as the previous algorithm for large demands.

Let t be the number of copies of the path P required to assign all the requests in $\mathcal{R}_1, \ldots, \mathcal{R}_{n-1}$. Let l_i be the load on edge e_i, which is the sum of all demands passing through e_i.

Lemma 2.2. *When all the requests in $\mathcal{R}_1, \ldots, \mathcal{R}_{n-1}$ have been packed, there is an edge e_i such that in at least $t-1$ copies of P, $l_i > \frac{1}{2}c$.*

Proof. Consider the demand $d_i \in \mathcal{R}_i$ (for some i) due to which the last color t was opened. At the time d_i was considered, all the requests started on or before v_i. Since d_i could not be assigned any of the previous $t-1$ colors, there are $t-1$ edges, one for each color, such that the total load put by the existing small demands on each of these edges is strictly more than $c - d_i \geq \frac{1}{2}c$ since, $d_i \leq \frac{1}{2}c$. Since the demands have been considered in a left to right manner, the load on the first edge e_i of d_i on each of these $t-1$ colors is at least as much. Hence, e_i is the edge such that $l_i > \frac{1}{2}c$. □

It follows from Lemma 2.2 that the total load put by requests in $\mathcal{R}_1, \ldots, \mathcal{R}_{n-1}$ on e_i is greater than $\frac{1}{2}c(t-1)$. Hence, the congestion on edge e_i is more than $\frac{1}{2}(t-1)$, since the edge capacity is c. Thus, $r \geq r_{e_i} > \frac{1}{2}(t-1)$. Hence, $t < 2r+1$, which implies that $t \leq 2r$, since t is an integer.

Since, we can assign all the requests in $\mathcal{R}_1, \ldots, \mathcal{R}_{n-1}$ using $t \leq 2r \leq 2 \cdot \text{OPT(S)}$ copies, this is a 2-approximation algorithm.

A 3-approximation algorithm

We solve the instance containing only large demands and the instance containing only small demands separately. We have, $\text{ALG}(L) = \text{OPT}(L)$ and $\text{ALG}(S) \leq 2 \cdot \text{OPT}(S)$. Moreover, $\text{OPT} \geq \max\{\text{OPT}(L), \text{OPT}(S)\}$. Hence the total number of colors required by the algorithm is

$$\begin{aligned} \text{ALG} &= \text{ALG}(L) + \text{ALG}(S) \\ &\leq \text{OPT}(L) + 2 \cdot \text{OPT}(S) \\ &\leq 3 \cdot \text{OPT}. \end{aligned}$$

Number of colors in terms of the congestion bound r

For large demands, the total load on the edge e, $l_e > \frac{1}{2}c \cdot \chi$. On the other hand, $l_e \leq rc$. Hence, $rc > \frac{1}{2}c \cdot \chi$ and so $\text{ALG}(L) = \chi < 2r$, which implies that $\chi \leq 2r - 1$, since χ is an integer. For small demands, $\text{ALG}(L) \leq 2r$. Hence, $\text{ALG} \leq 2r - 1 + 2r = 4r - 1$.

Running time

Since we are only sorting the demands based on their left endpoints and maintaining a set of copies, the running time of the algorithm is polynomial.

2.2.2 A 24-approximation algorithm for arbitrary capacities and arbitrary demands

We consider an instance $\mathcal{I}$ of the Round-UFP problem given by a path G on n points, and a set of requests $R_1, \ldots, R_m$. Let OPT denote an optimal solution, and $\texttt{col}(\text{OPT})$ denote the number of colors used by OPT. We begin with a few definitions.

Definition 2.3. *The* bottleneck capacity b_i *of a request* R_i *is the smallest capacity of an edge in the interval between* s_i *and* t_i *– such an edge is called the* bottleneck edge *for request* R_i*. A demand* d_i *is said to be* small *if* $d_i \leq \frac{1}{4}b_i$*, else it is a* large *demand. More generally, a demand* d_i *is said to be* δ-small *if* $d_i \leq \delta b_i$*. Otherwise, it is a* δ-large *demand.*

Clearly, $\texttt{col}(\text{OPT}) \geq r$. We give an algorithm $\mathcal{A}$ which uses $O(r)$ colors. This will give a constant factor approximation algorithm for this problem. We first consider the case of large demands. We will use the following result of Nomikos et al. [50].

Lemma 2.3. *Consider an instance of* ROUND-UFP *where all capacities are integers and all demands* D_i *have bandwidth requirement* $d_i = 1$*. Then, one can color these demands with* r *colors.*

Lemma 2.4. *We can color all large demands with at most* $8r$ *colors.*

Proof. We first scale all capacities and demands such that the minimum capacity $c_{\min}$ becomes 1. Now, we round all capacities down to the nearest integer, and we increase all the demands d_i to 1. Note that this will affect the congestion of an edge e by a factor of at most 8. Since $c_e \geq c_{\min} = 1$, rounding c_e down to the nearest integer will reduce it by a factor of at most 2 (which will happen for a real number less than but arbitrarily close to 2). Since all demands are of size at least $\frac{1}{4}$ (because they are large demands, so $d_i > \frac{1}{4}b_i \geq \frac{1}{4}c_{\min} = \frac{1}{4}$), we may increase the requirement of a demand by a factor of at most 4. Thus, the value of r will increase by a factor of at most 8. Now, we invoke the result in Lemma 2.3. This proves the lemma. □

We now consider the more non-trivial case of small demands. We divide the edges into classes based on their capacities. We say that an edge e is of *class* l if $2^l \leq c_e < 2^{l+1}$. We use $\texttt{cl}(e)$ to denote the class of e. For a demand D_j, let l_j be the smallest class such that the interval I_j contains an edge of class l_j. The *critical edge* of demand D_j is defined as the first edge (as we go from left to right from s_j to t_j) in I_j of class l_j. Note that the critical edge could be different from the bottleneck edge, though both of them would be of class l_j.

Lemma 2.5. *The small demands can be colored with at most* $16r$ *colors.*

Proof. We maintain $16r$ different solutions to the instance $\mathcal{I}$, where a solution routes a subset of the demands. We will be done if we can assign each demand to one of these solutions. Let us call these solutions $\mathcal{S}_1, \ldots, \mathcal{S}_K$, where $K = 16r$. We first describe the routing algorithm and then show that it has the desired properties.

We arrange the demands in order of their left end-points – let this ordering be $D_1, \ldots, D_m$. Let e_j be the critical edge of D_j. When we consider D_j, we send it to a solution $\mathcal{S}_l$ for which the total requirements of demands containing e_j is at most $c_{e_j}/16$. At least one such solution must exist, otherwise $r_e > \frac{16r \cdot c_{e_j}/16}{c_{e_j}} = r$, a contradiction. This completes the description of how we assign each demand to one of the solutions. We now prove that each of the solutions $\mathcal{S}_l$ is feasible.

Fix a solution $\mathcal{S}_l$ and an edge e. Suppose e is of class i. Let $\mathcal{D}(\mathcal{S}_l)$ be the demands routed in $\mathcal{S}_l$ which contain the edge e. Among such demands, let D_u be the last demand for which the critical edge is to the left of e (including e) – let e' be such an edge. Clearly, $\texttt{cl}(e') \geq i$. For an integer $i' \leq i$, let $e^{(i')}$ be the first edge of class i' to the right of e (so, $e^{(i)}$ is same as e).

First consider the demands in $\mathcal{D}(\mathcal{S}_l)$ which are considered before (and including D_u). All of these demands go through e' (because all such demands begin before D_u does and contain e). So, the total requirement of such demands, excluding D_u, is at most $c_{e'}/16$ – otherwise we would not have assigned D_u to this solution. Because D_u is a small demand and $\texttt{cl}(e') \geq i$, the total requirements of such demands (including D_u) is at most

$$\frac{2^{i+1}}{16} + \frac{c_e}{4} \leq \frac{c_e}{8} + \frac{c_e}{4} = \frac{3c_e}{8}.$$

Now consider the demands in $\mathcal{D}(\mathcal{S}_l)$ whose critical edges are to the right of e – note that, such an edge must be one of $e^{(i')}$ for some $i' < i$. Similar to the argument above, the total requirements of such demands is at most

$$\sum_{i'=0}^{i-1} \left(\frac{2^{i'+1}}{16} + \frac{2^{i'+1}}{4} \right) = \frac{5}{16} \sum_{i'=0}^{i-1} 2^{i'+1} \leq \frac{5 \cdot 2^{i+1}}{16} = \frac{5 \cdot 2^i}{8} \leq \frac{5c_e}{8}.$$

Here, we have used the fact that $c_e \geq 2^i$. Thus, we see that the total requirements of demands in

$\mathcal{D}(\mathcal{S}_l)$ is at most

$$\frac{5c_e}{8} + \frac{3c_e}{8} \leq c_e.$$

Hence the solution is feasible. This proves the lemma. □

Combining the above two lemmas, we get the following theorem.

Theorem 2.6. *Given an instance of* ROUND-UFP*, there is an algorithm for this problem which uses at most* $24 \cdot \texttt{col}(\text{OPT})$ *colors, and hence it is a* 24*-approximation algorithm. Further, if all demands are small, then one can color the demands using at most* $16 \cdot \texttt{col}(\text{OPT})$ *colors.*

Running time

For large demands, we are using the algorithm by Nomikos et al. [50], which is polynomial-time. Scaling the capacities and demands requires polynomial-time. For small demands, sorting the demands and maintaining several copies of the path can be done in polynomial-time. The critical edge of a demand can also be found in polynomial-time. Hence, the overall algorithm runs in polynomial-time.

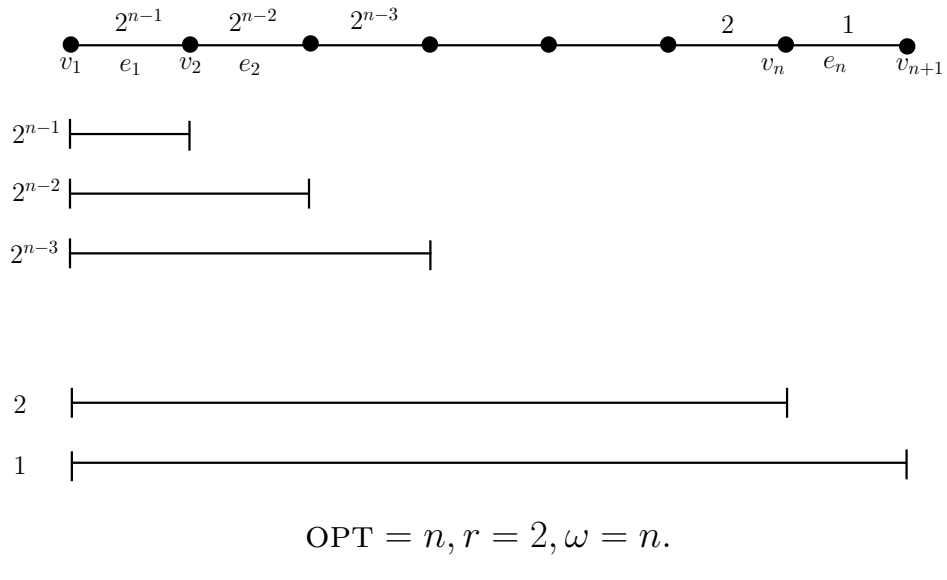
2.2.3 How bad can the congestion bound be?

With NBA

We show an example, where even the optimal coloring requires $2r$ colors. Suppose there is a single edge of capacity 1. There are $2k$ copies of a (large) demand $\frac{1}{2} + \epsilon$, where $\epsilon \ll \frac{1}{k}$. Since, no two demands can be given the same color, the optimal coloring requires $2k$ colors, while the congestion bound r is $k + 1$. Hence, OPT $\approx 2r$, for large k.

Without NBA

We show an example where even the optimal coloring requires n colors, whereas the congestion bound is 2. In Figure 2.2, the capacities are geometrically decreasing and $c(e_i) = 2^{n-i-1}$ for $1 \leq i \leq n-1$. The demands D_j are between v_1 and v_j for $2 \leq j \leq n$ and $d_j = 2^{n-j}$. Since, no two



$$\text{OPT} = n, r = 2, \omega = n.$$

Figure 2.2: An example where $\text{OPT} = n, r = 2, \omega = n$.

demands can be given the same color, the optimal coloring requires n colors, while the congestion bound r is 2. So the ratio $\frac{\text{OPT}}{r}$ is $\Omega(n)$. Note that in this example, NBA is not satisfied. Here, ω is the maximum number of intervals that can't be assigned the same color, which is also the maximum clique size in the corresponding interval graph. Note that ω is a lower bound on OPT, so any solution requires at least ω colors.

2.3 Approximation Algorithms for Round-UFP on Trees

We now consider the ROUND-UFP problem on trees. Consider an instance $\mathcal{I}$ of this problem as described in Section 2.1. We consider the case of large and small demands separately. Let $\mathcal{D}^{\mathtt{l}}$ be the set of large demands and $D^{\mathtt{s}}$ be the set of small demands.

Lemma 2.7. *There is a 32-approximation algorithm for the* ROUND-UFP *problem on trees, when we only have demands in* $\mathcal{D}^{\mathtt{l}}$.

Proof. Chekuri et. al. [21] gave a 4-approximation algorithm for coloring a set of demands when all demands have requirement 1, and the capacities are integers. In fact, their algorithm uses at most $4r$ colors. In our case, first observe that if $D_i \in \mathcal{D}^{\mathtt{l}}$, then d_i lies between $\frac{1}{4}c_{\min}$ and $c_{\min}$. We create a new instance $\mathcal{I}'$, where we round-up the requirement of each demand D_i to $c_{\min}$. Further, we

round-down the capacity of each edge to the nearest multiple of $c_{\min}$. We claim that our algorithm uses at most $32 \cdot \texttt{col}(\text{OPT})$ colors, where $\texttt{col}(\text{OPT})$ denotes the number of colors used by the optimal solution for the large demands in $\mathcal{I}$. Indeed, by increasing the requirements of the large demands, and decreasing the capacities of the edges, we affect the congestion of an edge by at most $4 \cdot 2 = 8$. Now this is a uniform demands instance, which is the same as a unit demands instance by scaling the capacities and demands. We lose a further factor of 4 by using the 4-approximation algorithm of Chekuri et. al. Hence, the result follows. □

Lemma 2.8. *There is a* 32*-approximation algorithm for the* Round-UFP *problem on trees, when we only have demands in* $\mathcal{D}^{\mathsf{s}}$.

Proof. The proof is very similar to that of Lemma 2.5. We maintain $16r$ solutions. For a demand D_i, let a_i denote the least common ancestor of s_i and t_i. We consider the demands in a bottom-up order of a_i. For a demand D_i, we define two critical edges: the s_i-critical edge is the critical edge on the $a_i - s_i$ path, and the t_i-critical edge is the critical edge on the $a_i - t_i$-path. We send D_i to the solution in which both these critical edges have been used till $\frac{1}{16}$ of their total capacity only. Again it is easy to check that such a solution will exist. The rest of the argument now follows as in the proof of Lemma 2.5. □

Theorem 2.9. *There is a* 64*-approximation algorithm for the* Round-UFP *problem on trees.*

Proof. Follows from the two previous lemmas. □

Running time

For large demands, we are using the algorithm by Chekuri et. al. [21], which runs in polynomial-time. Scaling the capacities and demands requires polynomial-time. For small demands, sorting the demands and maintaining several copies of the tree can be done in polynomial-time. The critical edge of a demand can also be found in polynomial-time. Hence, the overall algorithm runs in polynomial-time.

Chapter 3

The Max-UFP and the Bag-UFP Problems

We define the MAX-UFP and the BAG-UFP problems and give constant factor approximation algorithms for both the problems under NBA. We are given a graph $G = (V, E)$, which is either a path or a tree, with edge capacities c_e for all edges $e \in E$. We are also given a set of requests $R_1, \ldots, R_k$. Request R_i has an associated source-sink pair (s_i, t_i), a demand d_i, and a profit w_i. We shall use I_i to denote the associated unique path between s_i and t_i in G. In order to route a request R_i, we send d_i amount of flow from s_i to t_i along the (unique) path between them in G. A subset of demands will be called *feasible*, if they can be simultaneously routed without violating the edge capacities.

In the MAX-UFP problem, we would like to find a feasible subset of demands of maximum total profit. In the BAG-UFP problem, we are given sets, which we will call *bags*, $\mathcal{D}^1, \ldots, \mathcal{D}^p$, where each set $\mathcal{D}^j$ consists of a set of requests $R_1^j, \ldots, R_{n_j}^j$. As before, each request R_i^j is specified by an interval I_i^j and a bandwidth requirement d_i^j. We are also given profits p^j associated with each of the bags $\mathcal{D}^j$. A feasible solution to such an instance picks at most one demand from each of the bags – the selected demands should form a feasible set of routable demands. The profit of such a solution is the total profit of the bags from which we select a demand. The goal is to maximize the total profit.

We require our instances to satisfy NBA. We use the notion of congestion, bottleneck capacity, large demands and small demands, as defined in chapter 2. We will use ideas from ROUND-UFP to give a constant factor approximation for MAX-UFP, and then extend it to BAG-UFP.

3.1 Linear Programming formulation for Max-UFP

A natural linear programming formulation for MAX-UFP on a path is given below. Here x_i denotes the fraction of the demand i that is satisfied and I_i is the unique path between s_i and t_i.

$$\begin{aligned} \textbf{maximize} \quad & \sum_{i=1}^{k} w_i x_i && \textbf{(UFP-LP)} \\ \textbf{such that} \quad & \sum_{i:e\in I_i} d_i x_i \leq c_e && \forall e \in E \\ & 0 \leq x_i \leq 1 && \forall i \in \{1,\ldots,k\} \end{aligned}$$

If we replace the constraints $x_i \in [0,1]$ by the constraints $x_i \in \{0,1\}$, we get an integer program, which precisely models MAX-UFP.

Definition 3.1. *The* integrality gap *of an integer program is the worst-case ratio over all instances of the problem of the value of an optimal solution to the integer programming formulation to the value of an optimal solution to its linear programming relaxation.*

3.1.1 Integrality gap of the UFP-LP without NBA

Chakrabarti et al. [18] showed that the integrality gap of the above LP is $\Theta\left(\log \frac{d_{\max}}{d_{\min}}\right)$. This can be as bad as $\Omega(n)$ without NBA, as the example in Figure 3.1 shows. In this example, $c(e_i) = 2^i$ for $i = 1,\ldots,n$. There is a demand of 2^i between v_i and v_{n+1} for $i = 1,\ldots,n$. For all such demands the profit is 1. Note that the optimum integral solution can route at most one demand, to get a profit of $OPT = 1$, while the optimal fractional LP solution can route each demand to the extent of $\frac{1}{2}$ ($x_i = \frac{1}{2}$), to get a profit of $OPT_f = \frac{n}{2}$. Hence, $\frac{OPT_f}{OPT} = \frac{n}{2} = \Omega(n)$. Note that in this example, NBA is not satisfied. Further, $d_{\max} = 2^n$ and $d_{\min} = 2$, so the bound $\Theta\left(\log \frac{d_{\max}}{d_{\min}}\right)$ is asymptotically

tight.

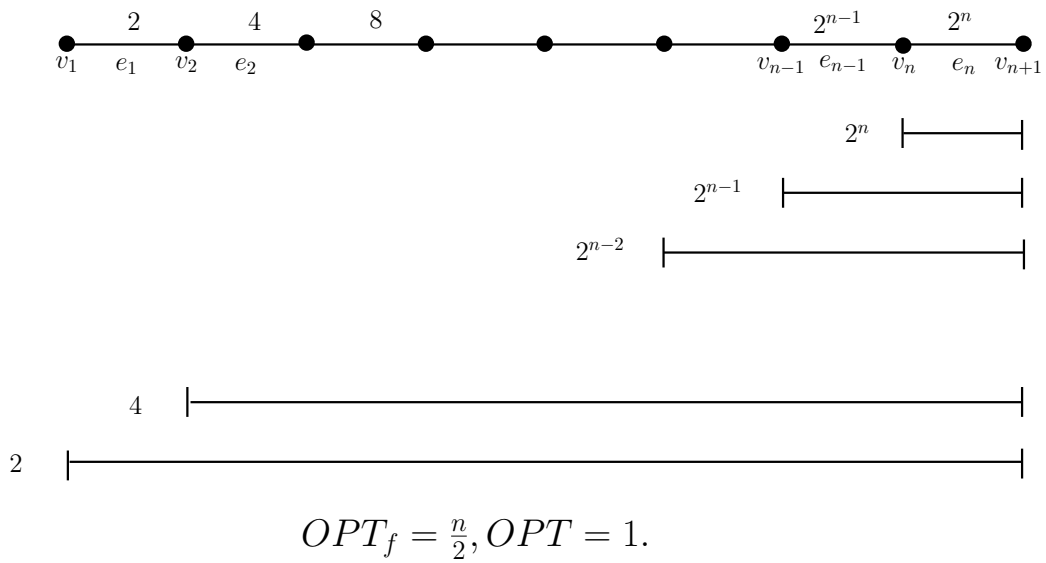


Figure 3.1: An example where $OPT_f = \frac{n}{2}, OPT = 1$.

3.1.2 Integrality gap of the UFP-LP with NBA

In Figure 3.2, the capacities and demands are as shown. All profits are 1. Here $d_1 = 2c, d_2 = d_3 = c + \epsilon$. The LP has a feasible solution given by $x_1 = \frac{1}{2}, x_2 = x_3 = \frac{c}{c+\epsilon}$. Hence, LP has a profit of $\frac{1}{2} + \frac{2c}{c+\epsilon} \approx 2.5$. Since routing any demand integrally will block the other demands, the IP can get a profit of at most 1. Hence, the integrality gap of the UFP-LP on this example is 2.5.

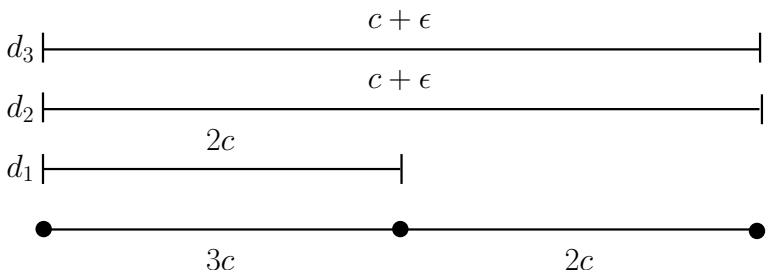


Figure 3.2: Integrality gap of 2.5 for paths.

3.2 Approximation Algorithm for Max-UFP

In this section we show how ideas from ROUND-UFP can be used to derive a constant factor approximation algorithm for MAX-UFP. Consider an instance $\mathcal{I}$ of MAX-UFP. As before, we divide the demands into small and large demands. For large demands, Chakrabarti et al. [18] showed that one can find the optimal solution by dynamic programming. For completeness, we include the result below.

Lemma 3.1. *The number of δ-large demands crossing any edge in a feasible solution is at most $\frac{2}{\delta}\left(\frac{1}{\delta}-1\right)$. Hence, an optimum solution can be found in $n^{O(1/\delta^2)}$ time using dynamic programming.*

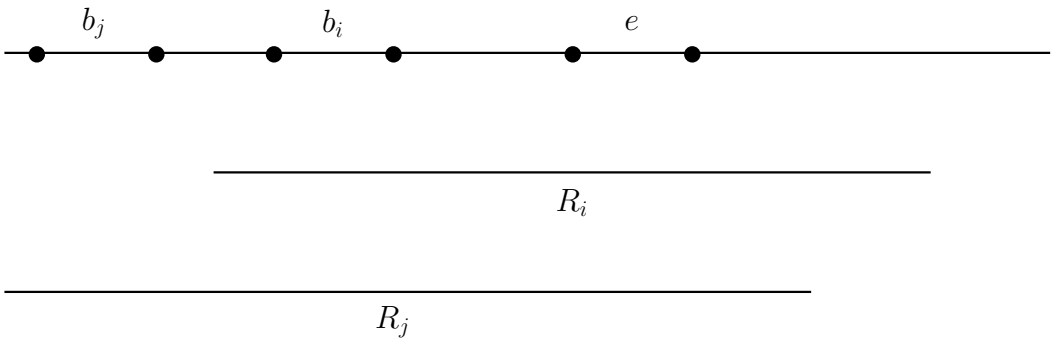


Figure 3.3: Illustration for the analysis of large demands.

Proof. Consider the set of requests S_e passing through the edge e in a feasible solution. The bottleneck edge of any such request will either be e or on its left or on its right. Let S_l be the set of requests whose bottleneck edge is e or on its left. Similarly, let S_r be the set of requests whose bottleneck edge is on the right of e. Among all requests in S_l, let R_i be the one whose bottleneck edge b_i is the rightmost. Then all requests in S_l will pass through b_i, since they pass through e and their bottleneck edge is on the left of b_i.

Consider any other demand $R_j \in S_l, j \neq i$. We know that $d_i > \delta c_{b_i}$, whereas $d_i \leq c_{\min} \leq c_{b_j}$, using NBA. Together this gives, $c_{b_j} > \delta c_{b_i}$. Hence, $d_j > \delta c_{b_j} > \delta^2 c_{b_i}$. The load put by R_i on b_i is $d_i > \delta c_{b_i}$. The load put by R_j for $j \neq i$ on b_i is $d_j > \delta^2 c_{b_i}$. The total number of such R_j is strictly less than $\frac{(1-\delta)c_{b_i}}{\delta^2 c_{b_i}}$. Together with the request R_i, the number of requests in S_l is at most $\frac{1}{\delta}\left(\frac{1}{\delta}-1\right)$. Similarly, $|S_r| \leq \frac{1}{\delta}\left(\frac{1}{\delta}-1\right)$. Hence, $|S_e| \leq |S_l| + |S_r| \leq \frac{2}{\delta}\left(\frac{1}{\delta}-1\right)$. □

Note that, according to our definition, large demands are $\frac{1}{4}$-large. Now we consider the small demands. The following lemma gives an approximation algorithm for small demands.

Lemma 3.2. *If there are only small jobs, then there is a* 16*-approximation algorithm for* Max-UFP.

Proof. We write the following natural LP relaxation for this problem – a variable x_i for demand D_i which is 1 if we include it in our solution, and 0 otherwise.

$$\begin{aligned} &\max \sum_i w_i x_i \\ &\sum_{i:e\in I_i} d_i x_i \le c_e \quad \text{for all edges } e \\ &0 \le x_i \le 1 \quad \text{for all demands } i \end{aligned} \tag{3.1}$$

Let $x^\star$ be an optimal solution to the LP relaxation. Let K be an integer such that all the variables $x_i^\star$ can be written as $\frac{\alpha_i}{K}$ for some integer α_i. Now we construct an instance $\mathcal{I}'$ of Round-UFP as follows. For each (small) demand D_i in $\mathcal{I}$, we create α_i copies of it. Rest of the parameters are same as those in $\mathcal{I}$. First observe that inequality (3.1) implies that $\sum_{i:e\in I_i} d_i\alpha_i \le Kc_e, \forall e \in E$. Thus, the congestion of each edge in $\mathcal{I}'$ is at most K. Using Lemma 2.5 for small demands, we can color the demands with at most $16K$ colors. It follows that the best solution among these $16K$ solutions will have profit at least $\frac{1}{16}\cdot\sum_i w_i x_i^\star$. □

Thus, we get the following theorem.

Theorem 3.3. *There is a* 17*-approximation algorithm for the* Max-UFP *problem.*

Proof. Given an instance $\mathcal{I}$, we divide the demands into large and small demands. For large demands, we compute the optimal solution using Lemma 3.1, whereas for small demands we compute a solution with approximation ratio 16 using Lemma 3.2. Then we pick the better of the two solutions.

Consider an optimal solution OPT with profit $\texttt{profit}(\text{OPT})$. Let $\texttt{profit}^l(\text{OPT})$ be the profit for large demands and $\texttt{profit}^s(\text{OPT})$ be the profit for small demands. If $\texttt{profit}^l(\text{OPT}) \ge$

$\frac{1}{17} \cdot \texttt{profit}(\text{OPT})$, then our solution for large demands will also be at least $\frac{1}{17} \cdot \texttt{profit}(\text{OPT})$. Otherwise, $\texttt{profit}^s(\text{OPT}) \geq \frac{16}{17} \cdot \texttt{profit}(\text{OPT})$. In this case, our solution for small demands will have value at least $\frac{1}{16} \cdot \frac{16}{17} \cdot \texttt{profit}(\text{OPT}) = \frac{1}{17} \cdot \texttt{profit}(\text{OPT})$. □

3.2.1 Running time

We can find the optimal solution for the instance containing only large demands using dynamic programming in polynomial-time. Indeed, since the large demands are $\frac{1}{4}$-large, we can compute the optimal solution in $O(n^{16})$ time using Lemma 3.1. For small demands, we have to find an optimal solution to the linear programming relaxation for Max-UFP. This can be done in polynomial-time using the ellipsoid method. Solving the Round-UFP instance $\mathcal{I}'$ using Lemma 2.5 can also be done in polynomial-time. We can make K polynomial in the input as follows. If the value of the variables $x_i^\star$ in the LP are less than $\frac{1}{k}$ (k is the number of demands), then we can ignore them. Otherwise, we can round them to the nearest multiple of $\frac{1}{k}$. This will cause a small error of at most $\frac{1}{k}$, which can be ignored. Since, K can be taken as the least common multiple of the denominators of the variables $x_i^\star$, this will make K polynomial in the input. Hence, the overall running time of the algorithm is polynomial.

3.3 Approximation Algorithm for Bag-UFP

We now extend the above algorithm to the Bag-UFP problem. Consider an instance $\mathcal{I}$ of this problem. As before, we classify each of the requests R_i^j as either large or small. For each bag, $\mathcal{D}^j$, let $\mathcal{D}^{j,\mathtt{l}}$ be the set of large demands in $\mathcal{D}^j$ and $\mathcal{D}^{j,\mathtt{s}}$ be the set of small demands in $\mathcal{D}^j$. Again, we have two different strategies for large and small demands.

Lemma 3.4. *If there are only large jobs, then there is a* 48*-approximation algorithm for* Bag-UFP*.*

Proof. Suppose, we have the further restriction that the selected intervals need to be disjoint. From Lemma 3.1, we know that the number of $\frac{1}{4}$-large demands crossing any edge in a feasible solution is at most $2 \cdot 4 \cdot (4-1) = 24$. Hence, if the demands are disjoint, the value of the objective function will

reduce by a factor of at most 24. However, for the latter problem, we can use the 2-approximation algorithm of Berman et al. [10] and Bar-Noy et al. [7]. This gives a 48-approximation algorithm. □

Lemma 3.5. *If there are only small jobs, then there is a* 17*-approximation algorithm for* Bag-UFP.

Proof. As in the case of Max-UFP problem, we first write an LP relaxation, and then use an algorithm similar to the one used for the Round-UFP problem. We have a variable x_i^j for demand D_i^j, which is 1 if we include it in our solution and 0 otherwise, and a variable y^j which is 1 if we choose a demand from the bag $\mathcal{D}^j$ and 0 otherwise. The LP relaxation is as follows.

$$\max \sum_j p^j y^j$$

$$\sum_{i:e\in I_i^j} d_i^j x_i^j \leq c_e \quad \text{for all edges } e \tag{3.2}$$

$$\sum_i x_i^j \leq y^j \quad \text{for all bags } \mathcal{D}^j \tag{3.3}$$

$$0 \leq x_i^j \leq 1 \quad \text{for all demands } i$$

$$0 \leq y^j \leq 1 \quad \text{for all bags } \mathcal{D}^j$$

Let x, y be an optimal solution to the LP above. Again, let K be a large enough integer such that $y^j = \frac{\alpha^j}{K}, x_i^j = \frac{\beta_i^j}{K}$, where α_j and β_i^j are integers for all j and i. Now we consider an instance of Round-UFP where we have β_i^j copies of the demand D_i^j. The only further restriction is that no two demands from the same bag can get the same color. Inequality (3.2) implies that $\sum_{i:e\in I_i^j} d_i^j \beta_i^j \leq K c_e, \forall e \in E$. So the congestion bound is K. We proceed as in the proof of Lemma 2.5, except that now we have $17K$ different solutions. When we consider the demand D_i^j, we ignore the solutions which contain a demand from the bag $\mathcal{D}^j$. Inequality (3.3) implies that $\sum_i \beta_i^j \leq \alpha^j \leq K, \forall j$. Hence, there will be at most K such solutions. For the remaining $16K$ solutions, we argue as in the proof of Lemma 2.5. □

Theorem 3.6. *There is a* 65*-approximation algorithm for the* Bag-UFP *problem.*

Proof. This follows from the two previous lemmas. We argue as in the proof of Theorem 3.3. □

3.3.1 Running time

For the instance containing only large demands, we are using the 2-approximation algorithms of Berman et al. [10] or Bar-Noy et al. [7], both of which runs in polynomial-time. Hence, the instance containing only large demands can be solved in polynomial-time. For small demands, we have to find an optimal solution to the linear programming relaxation for Bag-UFP. This can be done in polynomial-time using the ellipsoid method. Solving the constructed Round-UFP instance using Lemma 2.5 can also be done in polynomial-time. We can make K polynomial in the input using the technique in subsection 3.2.1. Hence, the overall running time of the algorithm is polynomial.

3.4 Approximation Algorithm for Max-UFP on Trees

Consider an instance $\mathcal{I}$ of Max-UFP on trees. We will show how the approximation algorithm for the Round-UFP problem can be used to obtain a constant factor approximation algorithm for the Max-UFP problem.

Theorem 3.7. *There is a* 64*-approximation algorithm for the* Max-UFP *problem on trees.*

Proof. We write the following natural LP relaxation for this problem – a variable x_i for demand D_i which is 1 if we include it in our solution, and 0 otherwise.

$$\begin{aligned} \max \sum_i w_i x_i & \\ \sum_{i:e\in I_i} d_i x_i \leq c_e & \quad \text{for all edges } e \\ 0 \leq x_i \leq 1 & \quad \text{for all demands } i \end{aligned} \tag{3.4}$$

Let $x^\star$ be an optimal solution to the LP relaxation. Let K be an integer such that all the variables $x_i^\star$ can be written as $\frac{\alpha_i}{K}$ for some integer α_i. Now we construct an instance $\mathcal{I}'$ of Round-UFP as follows. For each demand D_i in $\mathcal{I}$, we create α_i copies of it. Rest of the parameters are

same as those in $\mathcal{I}$. First observe that inequality (3.4) implies that $\sum_{i:e\in I_i} d_i\alpha_i \leq Kc_e, \forall e \in E$. Thus, the congestion of each edge in $\mathcal{I}'$ is at most K. Using Theorem 2.9, we can color the demands with at most $64K$ colors. It follows that the best solution among these $64K$ solutions will have profit at least $\frac{1}{64} \cdot \sum_i w_i x_i^\star$. □

Although this is worse than the 48-approximation algorithm of Chekuri et al. [21], this illustrates the power of our approach. We can handle all these problems in a unified framework.

3.4.1 Running time

We can find an optimal solution to the linear programming relaxation for Max-UFP on trees in polynomial-time using the ellipsoid method. Constructing the Round-UFP instance $\mathcal{I}'$ can also be done in polynomial-time. The Round-UFP instance $\mathcal{I}'$ can be solved in polynomial-time as shown in section 2.3. We can make K polynomial in the input using the technique in subsection 3.2.1. Hence, the overall running time of the algorithm is polynomial.

Chapter 4

Online Algorithms for the Interval Coloring Problem

In this chapter, we consider the ROUND-UFP problem in an on-line setting. As before, we are given a path $G = (V, E)$ with edge capacities c_e on edge e. Requests arrive in an on-line manner. A request R_i is specified by a triplet (s_i, t_i, d_i), where s_i is the starting vertex, t_i is the destination vertex and d_i is the actual bandwidth requirement. We shall also use I_i to denote the interval $[s_i, t_i]$. The on-line algorithm needs to color the demand on its arrival, such that the set of demands with the same color can be routed feasibly in the path G. The goal is to minimize the number of colors. Again, we shall assume that the requests satisfy the no-bottleneck assumption (NBA). Indeed, without this assumption, it is known that any deterministic on-line algorithm will have competitive ratio of $\Omega\left(\max\left\{\log\log n, \log\log\log\left(\frac{c_{\max}}{c_{\min}}\right)\right\}\right)$, where $c_{\max}$ and $c_{\min}$ are the maximum and minimum edge capacities of the path respectively [28].

4.1 Preliminaries

We fix a time n, and consider the requests which have arrived till time n, i.e., $R_1, \ldots, R_n$. Recall that for an edge e, the load l_e on e is the total demand of requests which contain e, i.e., $\sum_{i:e\in I_i} d_i$. Also, the congestion on e, $r_e = \left\lceil \frac{l_e}{c_e} \right\rceil$. Let $r = \max_e r_e$ be the maximum congestion on any edge.

Clearly, r is a lower bound on the minimum number of colors required to color the requests. For a set of requests S, let $l_e(S) = \sum_{i:e\in R_i, R_i\in S} d_i$ be the load put by the requests in S on edge e.

We can assume without loss of generality that $c_{\min} = 1$. Since NBA is satisfied, this implies $d_{\max} \leq 1$. We now round down the edge capacities c_e to the nearest power of 2. Let $\hat{c}_e$ denote these rounded capacities. Note that $\hat{c}_{\min}$ remains 1.

The *bottleneck edge* b_i of a request R_i is an edge of minimum capacity (with respect to $\hat{c}$) in I_i, i.e., $b_i = \arg\min_{e\in I_i} \hat{c}_e$. The capacity of the bottleneck edge, i.e., $\hat{c}(b_i)$ is called the *bottleneck capacity* of the request R_i. The *class* of a request R_i is defined as $\ell_i = \log_2 \hat{c}(b_i)$. Note that the class of a request can be between 0 and $\log_2 \hat{c}_{\max}$.

For a request R_i in class $j \geq 1$, we shall call it a *small demand* if $d_i \leq \min(1, 2^{j-3})$. Since, $\hat{c}(b_i) = 2^j$, $d_i \leq \frac{\hat{c}(b_i)}{8}$. For a demand d_i in class 0, we call it a small demand if $d_i \leq \frac{1}{4}$. Since, $\hat{c}(b_i) = 1$, $d_i \leq \frac{\hat{c}(b_i)}{4}$. Otherwise, we shall call the request a *large demand*. Note that large demands can exist only in classes 0, 1 and 2 (see the table below).

Class	**Small demands**	**Large demands**	**Bottleneck capacity**
0	$\left(0, \frac{1}{4}\right]$	$\left(\frac{1}{4}, 1\right]$	1
1	$\left(0, \frac{1}{4}\right]$	$\left(\frac{1}{4}, 1\right]$	2
2	$\left(0, \frac{1}{2}\right]$	$\left(\frac{1}{2}, 1\right]$	4
3	$(0, 1]$	NONE	8
$\vdots$	$\vdots$	$\vdots$	$\vdots$
j	$(0, 1]$	NONE	2^j

Table 4.1: Schematic representation of classes and capacities of demands

4.2 Our algorithm

In this section, we give a 58-competitive algorithm for the online interval coloring problem. When a request comes, we determine whether it is small or large. We handle small demands and large demands separately. For small demands, we give a 32-competitive algorithm. For large demands, we give a 26-competitive algorithm. The details of the algorithms are given in the following sections.

4.2.1 Small demands

In this section, we give a 32-competitive algorithm for small demands. For this, we first consider a special case when all edges have the same capacity. We shall then show that for non-uniform capacities, we can derive several instances of uniform capacity instances. We can then apply our algorithm for uniform capacities to each of these instances.

Uniform Capacities

We consider the online Round-UFP problem for the special case when all edges have capacity 1 and each demand d_i is at most $1/4$. This is without any loss of generality, since we can always scale the demands with the common capacity c to make the capacity of each edge to be 1. We call it the Round-UFP-Uniform problem.

We shall assign each arriving request a *level*. Let S_l be the set of requests which have been assigned level l. We shall show that the set of requests in each level can be colored with one color. Suppose we have already processed requests $R_1, \ldots, R_{i-1}$. Suppose these requests have been partitioned into levels $S_1, \ldots, S_{k_i}$. When the request R_i arrives, we find the smallest index k such that for every edge $e \in I_i$, the total load of the requests in $\cup_{k'=1}^{k} S_{k'}$ (including R_i), i.e., $l_e(\cup_{k'=1}^{k} S_{k'} \cup \{R_i\})$, is at most $\frac{k}{4}$. If no such index is found, start a new level $k_i + 1$, and assign R_i to S_{k_i+1}. For an edge e and level k, we say that e is *critical* for R_i on level k, if $e \in I_i$ and $l_e(\cup_{k'=1}^{k} S_{k'} \cup \{R_i\}) > \frac{k}{4}$. Note that e is an edge which prevented R_i to be put on level k. The complete algorithm is given as algorithm 1.

We now analyze this algorithm. Let r denote the maximum congestion of an edge if we consider the requests $R_1, \ldots, R_n$. As argued earlier, r is a lower bound on the minimum number of colors needed to color these demands.

Lemma 4.1. *The number of levels k_n is at most $4r$.*

Proof. Consider the first request R which gets assigned to level S_{k_n}. It must be the case there is a critical edge e for R on level $k_n - 1$. So, $l_e\left(\left(\bigcup_{k'=1}^{k_n-1} S_{k'}\right) \cup \{R\}\right) > \frac{1}{4}(k_n - 1)$. On the other hand, since the maximum congestion on any edge is r, $l_e\left(\left(\bigcup_{k'=1}^{k_n-1} S_{k'}\right) \cup \{R\}\right) \leq r$. Together, this

```
1  Algorithm Color;
2  Input: Demands R_i = (s_i, t_i, d_i) coming online;
3  Output: A feasible coloring of demands;
   // k_i is the number of levels used.
4  k_i ← 1;
5  while there are still requests in the input do
6      let R_i = (s_i, t_i, d_i) be the next request;
7      find the smallest level k ∈ {1, ..., k_i} such that for every edge e ∈ I_i, the total load of the
       requests in ∪_{k'=1}^{k} S_{k'} (including R_i), i.e., l_e(∪_{k'=1}^{k} S_{k'} ∪ {R_i}), is at most k/4;
8      if no such level is found then
9          k_i = k_i + 1;
10         go to line 7.
11     end
       // assign R_i to level k.
12     S_k ← S_k ∪ {R_i};
13 end
```

Algorithm 1: An online algorithm for $\frac{1}{4}$-small demands and unit edge capacity

implies that $\frac{1}{4}(k_n - 1) < r$. Hence, $k_n < 4r + 1$, which implies that $k_n \leq 4r$, since k_n is an integer. This proves the desired result. □

Lemma 4.2. *For a level k and edge e, the load on e by demands in S_k is at most* 1.

Proof. First consider the case $k = 1$. For an edge e, let R_j be the last demand containing e which was added to S_1. Then, by the algorithm, total load on e (including R_j) is at most 1/4.

Now, consider $k > 1$. We call an edge e critical if it is critical for some demand in S_k on level $k - 1$. Note that each demand in S_k must contain at least one critical edge (otherwise it should have been added to level $k - 1$ or earlier). Fix a critical edge e. Let the demands containing e which get added to S_k (in the order of arrival) be $R_{k_1}, \ldots, R_{k_j}$. Note that e must be critical for R_{k_j} on level $k - 1$. Hence, $l_e\left(\cup_{k'=1}^{k-1} S_{k'}\right) \geq \frac{k-1}{4} - d_j$, where d_j is the demand of request R_{k_j}. Since R_{k_j} is added to S_k, it must be the case that $l_e\left(\cup_{k'=1}^{k} S_{k'}\right) \leq \frac{k}{4}$. Subtracting the second inequality from the first we get, $l_e(S_k) \leq \frac{1}{4} + d_j \leq \frac{1}{2}$, since $d_j \leq \frac{1}{4}$.

Now consider an edge e which is not critical. Let e_L and e_R be the nearest critical edges on its left and right respectively. Clearly, any request in S_k containing e must contain either e_L or e_R. But the total load on the latter edges is at most 1/2. Hence, the load on e is at most 1. This

proves the lemma. □

We now conclude with the main result of this section.

Lemma 4.3. *The number of colors required by our algorithm is at most* $4r$*. Hence, it is a* 4*-competitive algorithm for the* Round-UFP-Uniform *problem.*

Proof. From Lemma 4.1, we know that the number of levels is at most $4r$. Lemma 4.2 shows that we can color the requests in each level using one color. Hence, the number of colors required is at most $4r$. □

4.2.2 Algorithm for Small Demands

We now describe our algorithm for small demands, where we will use the 4-competitive algorithm for the Round-UFP-Uniform problem. For each class l, we create a new instance of the Round-UFP-Uniform problem $\mathcal{I}_l$, where all requests are of class l. We shall use the algorithm of the previous section to color the demands in $\mathcal{I}_l$. If $l = 0$, the path in $\mathcal{I}_0$ is the same as the path G, but we set all edge capacities to 1. Now consider the case $l \geq 1$. We first contract all edges e in G for which $\hat{c}_e < 2^l$. For the remaining edges, we set their capacity to 2^{l-1}. This gives the path in instance $\mathcal{I}_l$. Observe that in $\mathcal{I}_l$, the demands d_i are at most 2^{l-3}, and hence at most $1/4$ times the capacity of the edges (by definition of small demands), and so $\mathcal{I}_l$ is indeed an instance of the Round-UFP-Uniform problem. Note that if a demand of class l contains an edge e, then $\hat{c}_e \geq 2^l$, and so this edge will not get contracted in the path in $\mathcal{I}_l$.

We can now describe the algorithm for coloring small demands. When a demand of class l arrives, we color it using algorithm 1 on $\mathcal{I}_l$. Hence, the number of colors used by our algorithm is the maximum over all values of l of the number of colors needed for coloring $\mathcal{I}_l$.

Since a particular color may be present in several of the colorings for the instance $\mathcal{I}_l$, we need to show that we can indeed put together the requests which have been colored with this color in different instances $\mathcal{I}_l$.

Lemma 4.4. *For a color* c*, let* S_l^c *be the requests of class* l *which get colored with* c *(in* $\mathcal{I}_l$*). Then* $\cup_l S_l^c$ *form a feasible set of requests in* G *with edge capacities* c_e*.*

Proof. Fix an edge e with $\hat{c}_e = 2^k$. Then this edge appears in $\mathcal{I}_0, \ldots, \mathcal{I}_k$ with edge capacities $1, 2, \ldots, 2^{k-1}$. Further, no demand in $\cup_{l \geq k+1} S_l^c$ contains e. Hence, the total load on e due to the demands in $\cup_l S_l^c$ is at most $1 + 2 + \cdots + 2^{k-1} = 2^k \leq c_e$. □

Let r be the maximum congestion of any edge on the instance containing only the small demands.

Lemma 4.5. *The number of colors used by our algorithm is at most* $32r$.

Proof. Fix a class l. Let r_l be the maximum congestion of an edge in $\mathcal{I}_l$. Let r denote the maximum congestion of any edge in the original instance $\mathcal{I}$. We first argue that $r_l \leq 8r$. We define another instance $\hat{\mathcal{I}}_l$, which is the same as $\mathcal{I}_l$ except that the edge e has capacity $\hat{c}_e$. Let $\hat{r}_l$ be the maximum congestion of any edge in $\hat{\mathcal{I}}_l$. We first argue that $r_l \leq 2\hat{r}_l$. Indeed, if e is an edge in $\mathcal{I}_l$, then we know that $\hat{c}_e \geq 2^l$. We also know that any demand of class l must contain at least one edge e with $\hat{c}_e = 2^l$. So, for an edge $e \in \mathcal{I}_l$, either (i) $\hat{c}_e = 2^l$, in which case congestion on e is at most $\hat{r}_l$, or (ii) $\hat{c}_e > 2^l$. In the latter case, let e_L and e_R be the nearest edges on the left and the right of e with rounded capacities 2^l. Now, any demand in $\mathcal{I}_l$ which passes through e must contain either e_L or e_R. Hence, congestion on e is at most $2\hat{r}_l$.

Now, we argue that $\hat{r}_l \leq 4r$. Consider an edge e with $\hat{c}_e = 2^l$. In $\mathcal{I}_l$, we set the capacity of this edge to 2^{l-1}. Hence, the capacity of this edge in $\mathcal{I}_l$ is at least $c_e/4$. So, congestion of e in $\mathcal{I}_l$ is at least $r_e/4$. So, we get that $r \geq \frac{\hat{r}_l}{4}$. Equivalently, $\hat{r}_l \leq 4r$. Thus, we get $r_l \leq 2\hat{r}_l \leq 8r$. Using Lemma 4.3, our algorithm colors the demands in $\mathcal{I}_l$ using at most $4r_l \leq 32r$ colors. This proves the desired result. □

4.3 Large demands

We now describe our algorithm for coloring large demands. Recall that large demands exist only in classes 0, 1 and 2. We will color them using the algorithm given in [49]. The colorings for class 0 and class 2 will share colors, but these will be disjoint from the coloring for class 1. For convenience, we state the result below.

Theorem 4.6. *[49] Suppose all edges in the path have capacity 1 and let r be the maximum congestion of an edge. The number of colors required for coloring requests with demands in $\left(0, \frac{1}{4}\right]$, $\left(\frac{1}{4}, \frac{1}{2}\right]$ and $\left(\frac{1}{2}, 1\right]$ are at most $4r$, $3r$ and $3r$ respectively. Hence, coloring all requests requires at most $10r$ colors.*

We define three instances $\mathcal{I}_0, \mathcal{I}_1$ and $\mathcal{I}_2$. For $l \in \{0, 1\}$, we construct the instance $\mathcal{I}_l$ by contracting all edges e for which $\hat{c}_e \neq 2^l$. For $\mathcal{I}_2$, we contract all edges e for which $\hat{c}_e \neq 2^l$ and then reduce the capacity of edges by half. We first show that it is sufficient to color large demands of class l using $\mathcal{I}_l$ only.

Lemma 4.7. *Fix a class $l \in \{0, 1, 2\}$. Consider a coloring of demands of class l restricted to the instance $\mathcal{I}_l$. Then, coloring of class 1 is feasible. Further, for any color c, the set of demands in $\mathcal{I}_0 \cup \mathcal{I}_2$ which are colored with c is feasible.*

Note: It is possible that for a demand R_i of class l, we contracted some of the edges in I_i while constructing the path in $\mathcal{I}_l$. Hence, while considering the coloring in $\mathcal{I}_l$, we will consider only those edges of I_i which do not get contracted.

Proof. We would have contracted two types of edges in $\mathcal{I}_l$:

- **Edges e with $\hat{c}_e < 2^l$**: Since no class l demand passes through them, contracting these edges does not matter.
- **Edges e with $\hat{c}_e > 2^i$**: Consider such an edge e. So, $\hat{c}_e \geq 2^{l+1}$. Consider any coloring of class l requests in $\mathcal{I}_l$. Let e_L and e_R be the nearest edges with $\hat{c}$ values 2^l to the left and the right of e respectively (in the original graph). Then, any request of class l through e must contain either e_L or e_R. Hence, the total load on e due to such demands (of this color) is at most $\hat{c}_{e_L} + \hat{c}_{e_R} \leq 2^{l+1}$. Hence, demands of this color do not violate the edge capacity of e. □

We now show how to color class l demands in the instance $\mathcal{I}_l$. Let $r^{(l)}$ denote the maximum congestion of an edge in $\mathcal{I}_l$ (where the requests are all the class l demands).

Lemma 4.8. *We can color the demands of class l, for $l \in \{0, 1, 2\}$ using the following number of colors:*

- **Class 0 demands :** *These can be colored with at most* $6r^{(0)}$ *colors.*
- **Class 1 demands :** *These can be colored with at most* $7r^{(1)}$ *colors.*
- **Class 2 demands :** *These can be colored with at most* $3r^{(2)}$ *colors.*

Proof. First consider class 0 demands. So the demands lie in the range $\left(\frac{1}{4}, 1\right]$, and the capacity of each edge in $\mathcal{I}_0$ is 1. We now partition the demands into two parts: $\left(\frac{1}{4}, \frac{1}{2}\right]$ and $\left(\frac{1}{2}, 1\right]$. The claim now follows from Theorem 4.6. Now consider class 1 demands. They have demands in $\left(\frac{1}{4}, 1\right]$ and all edges have capacity 2. We scale down edge capacities and demands by a factor of 2. Now demands lie in the range $\left(\frac{1}{8}, \frac{1}{2}\right]$. We partition these into two parts based on their demands : $\left(\frac{1}{8}, \frac{1}{4}\right]$ and $\left(\frac{1}{4}, \frac{1}{2}\right]$. The result again follows from Theorem 4.6. Finally, we consider class 2 demands. These have demands in the range $\left(\frac{1}{2}, 1\right]$ and all edges have capacity 2. We scale down these values by a factor of 2. So now the demands lie in the range $[\left(\frac{1}{4}, \frac{1}{2}\right]$, and the result again follows from Theorem 4.6. □

Lemma 4.9. *We can color all the large demands in the original instance using at most* $26r$ *colors.*

Proof. Note that since the rounded edge capacities $\hat{c}_e \leq 2c_e$, $r^{(l)} \leq 2r$ for $l \in \{0, 1\}$. Since we halve the edge capacities, $r^{(2)} \leq 4r$. We will use Lemma 4.8. For class $\mathcal{I}_1$ at most $7r^{(1)} \leq 14r$ colors are required. Colors for $\mathcal{I}_0$ and $\mathcal{I}_2$ can be shared. These two classes can be colored using $\max\left(6r^{(0)}, 3r^{(2)}\right) \leq \max\left(6 \cdot 2r, 3 \cdot 4r\right) \leq 12r$ colors. Thus, the total number of colors needed for coloring large demands is at most $14r + 12r = 26r$. □

Our final algorithm now colors the small and the large demands separately. Combining Lemma 4.5 and Lemma 4.9, we get

Theorem 4.10. *Our algorithm for online* Round-UFP *is 58-competitive.*

Proof. We can color all the small demands using at most $32r$ colors and all the large demands using at most $26r$ colors. We know that $\mathrm{OPT} \geq r$. So, the total number of colors required to color any instance is at most $32r + 26r = 58r \leq 58 \cdot \mathrm{OPT}$. Hence, the result follows. □

4.3.1 Running time

Requests can be grouped based on their classes in polynomial-time. Finding the appropriate level of a request in Algorithm 1 (for uniform capacities) can be done in polynomial-time. Since, for small demands, we are using Algorithm 1 at most a polynomial number of times, the resulting algorithm runs in polynomial-time. For large demands, there are only three classes – 0, 1 and 2. For each class, we are using the algorithm in [49], which runs in polynomial-time. Hence, the algorithm for large demands also runs in polynomial-time. Hence, the overall algorithm runs in polynomial-time.

Chapter 5

Scheduling Resources for a Partial Set of Jobs

5.1 Introduction

We consider the problem of allocating resources to schedule jobs. We are given a path G, and a set of jobs. Each job j is specified by a triplet (s_j, t_j, d_j), where $[s_j, t_j]$ denotes the interval corresponding to the job (also denoted by I_j), and d_j is its demand requirement. We shall assume that d_j values are 1. Further, we are also given a set of resources. Each resource is specified by its starting and ending vertex, and the capacity it offers and its associated cost. A feasible solution is a set of resources satisfying the constraint that for any edge, the sum of the capacities offered by the resources containing this edge is at least the demand required by the jobs containing that edge, i.e., the selected resources must cover the jobs. We call this the Resource Allocation problem (ResAll).

We study two variants of the problem. The first variant is the partial covering version. The second variant is the prize collecting version. We study these variants for the case where the solution is allowed to pick multiple copies of a resource by paying proportional cost.

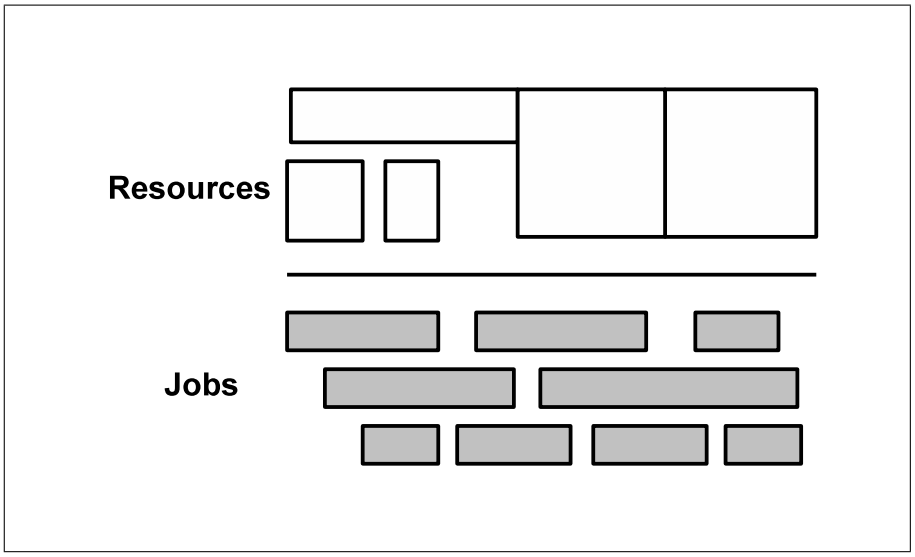


Figure 5.1: Illustration of the input

5.2 Problem Definition

We consider the graph $G = (V, E)$ which is a path with vertices numbered $1, 2, \ldots, |V|$ from left to right. An input instance consists of a set of *jobs* $\mathcal{J}$, and a set of *resources* $\mathcal{R}$.

Each job $j \in \mathcal{J}$ is specified by an interval $I_j = [s_j, t_j]$ in the path. Recall that each job has demand requirement of 1. Each resource $i \in \mathcal{R}$ is specified by an interval $I_i = [s(i), e(i)]$ in the path, capacity w_i and cost c_i. We shall assume that the capacities w_i are integers. We interchangeably refer to the resources as *resource intervals*. We shall also refer to the interval I_j (or I_i) as the *span* of the job j (or resource i). A typical scenario of such a collection of jobs and resources is shown in Figure 5.1.

We say that a job j (or resource i) *contains* an edge e if the associated interval I_j (or I_i) contains e; we denote this as $j \sim e$ ($i \sim e$). We define a *profile* $P : E \to \mathbb{N}$ to be a mapping that assigns an integer value to every edge of the path. For two profiles, P_1 and P_2, P_1 is said to *cover* P_2, if $P_1(e) \geq P_2(e)$ for all $e \in E$. Given a set J of jobs, the profile $P_J(\cdot)$ of J is defined to be the mapping determined by the cumulative demand of the jobs in J, i.e. $P_J(e) = |\{j \in J : j \sim e\}|$. Similarly, given a multiset R of resources, its profile is: $P_R(e) = \sum_{i \in R : i \sim e} w_i$ (taking copies of a resource into account). We say that R *covers* J if P_R covers P_J. The cost of a multiset of resources

R is defined to be the sum of the costs of all the resources (taking copies into account).

We now describe the two versions of the problem.

- PartialResAll: In this problem, the input also specifies a number k (called the *partiality parameter*) that indicates the number of jobs to be covered. A feasible solution is a pair (R, J) where R is a multiset of resources and J is a set of jobs such that R covers J and $|J| \geq k$. The cost of the solution is the sum of the costs of the resources in R (taking copies into account). The problem is to find a feasible solution of minimum cost.
- PrizeCollectingResAll: In this problem, every job j also has a penalty p_j associated with it. A feasible solution is a pair (R, J) where R is a multiset of resources and J is a set of jobs such that R covers J. The cost of the solution is the sum of the costs of the resources in R (taking copies into account) and the penalties of the jobs not in J. The problem is to find a feasible solution of minimum cost.

5.3 Outline of the Main Algorithm

In this section, we outline the proof of our main result:

Theorem 5.1. *There exists an $O(\log(n+m))$-approximation algorithm for the* PartialResAll *problem, where n is the number of jobs and m is the number of resources.*

The proof of the above theorem goes via the claim that the input set of jobs can be partitioned into a logarithmic number of *mountain ranges*. A collection of jobs M is called a *mountain* if there exists an edge e, such that all the jobs in this collection contain e; the specified edge where the jobs intersect will be called the *peak* edge of the mountain (see Figure 5.2; jobs are shown on the top and the profile is shown below). The justification for this linguistic convention is that if we look at the profile of such a collection of jobs, the profile forms a bimodal sequence, increasing in height until the peak, and then decreasing. The *span* of a mountain M is the set of edges which are contained in any of the jobs in the mountain, i.e., $\cup_{j \in M} I_j$. A collection of jobs $\mathcal{M}$ is called a *mountain range*, if the jobs can be partitioned into a sequence $M_1, M_2, \ldots, M_r$ such that each M_i is a mountain and the spans of any two mountains are non-overlapping (see Figure 5.3).

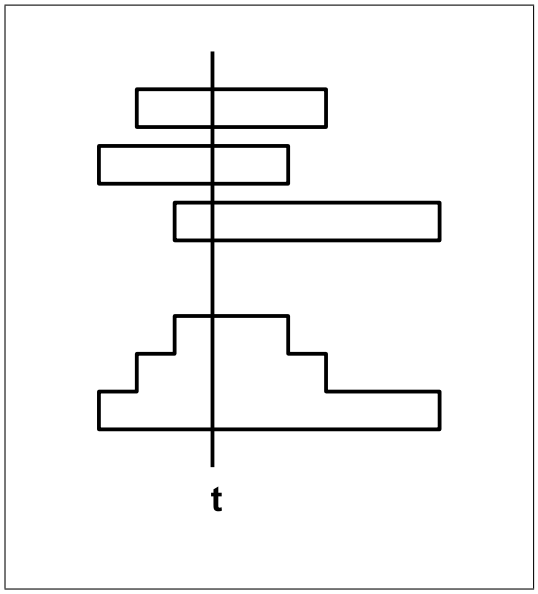


Figure 5.2: A Mountain M

We prove a decomposition lemma which shows that the input set of jobs can be partitioned into a logarithmic number of mountain ranges. Hence, our decomposition lemma implies that it is sufficient to get a good approximation for the case of a mountain range. It is not difficult to argue that one can extend this result to several mountain ranges by employing dynamic programming. We only need to know how many jobs to satisfy in each mountain range. For a single mountain range, we will prove the following result.

Theorem 5.2. *There exists a constant factor approximation algorithm for the special case of the* PartialResAll *problem, wherein the input set of jobs form a single mountain range* $\mathcal{M}$.

To prove Theorem 5.2, we need the following results.

1. A constant factor approximation for the case of a mountain.
2. Extending this result to a mountain range.

The first part is accomplished by the following theorem. The proof is given in Section 5.4.

Theorem 5.3. *There exists a* 8*-approximation algorithm for the special case of the* PartialResAll *problem wherein the input set of jobs form a single mountain* M.

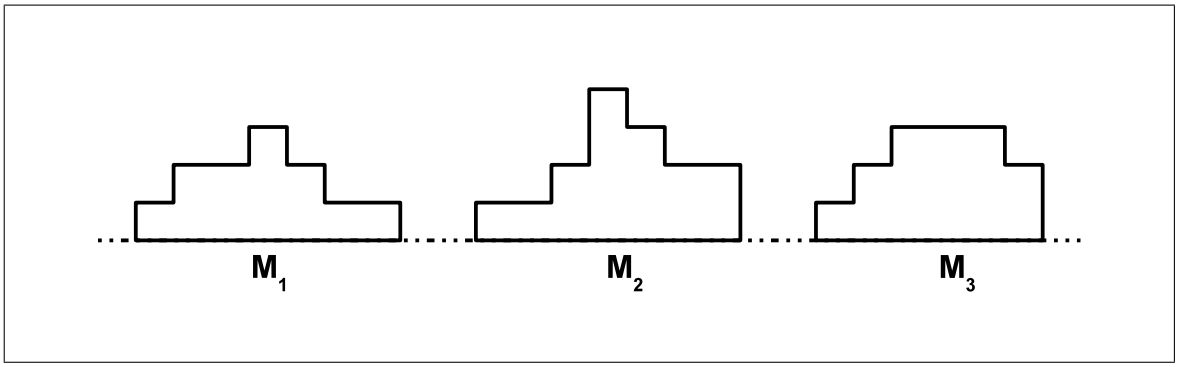


Figure 5.3: A Mountain Range $\mathcal{M} = \{M_1, M_2, M_3\}$

For the second part, we will collapse each mountain into a single edge. This can be done if resources are wide, i.e., they span the mountains which they intersect. But this may not always be the case. We need to solve a related problem.

*Problem Definition (*LSPC*):* We are given a demand profile over the set of edges E, which specifies an integral demand d_e for every edge e. The input resources are of two types, *short* and *long*. A short resource spans only one edge, whereas a long resource can span one or more edges. Each resource i has a cost c_i and a capacity w_i. The input also specifies a *partiality parameter* k. A feasible solution S consists of a multiset of resources S and a coverage profile: an integer k_e for each edge e satisfying $k_e \leq d_e$. The solution should have the following properties: (i) $\sum_e k_e \geq k$; (ii) at any edge e, the sum of capacities of the resource intervals from S containing e is at least k_e; (iii) for any edge e, at most one of the short resources containing e is picked (however, multiple copies of a long resource may be included). The objective is to find a feasible solution having minimum cost. See Figure 5.4 for an example (in the figure, short resources are shaded).

Theorem 5.4. *There exists a* 16*-approximation algorithm for the* LSPC *problem.*

5.4 Overview of Our Algorithm

In this section, we give an overview of our algorithm and describe the various results needed to prove the claimed approximation guarantee. We start with some notations.

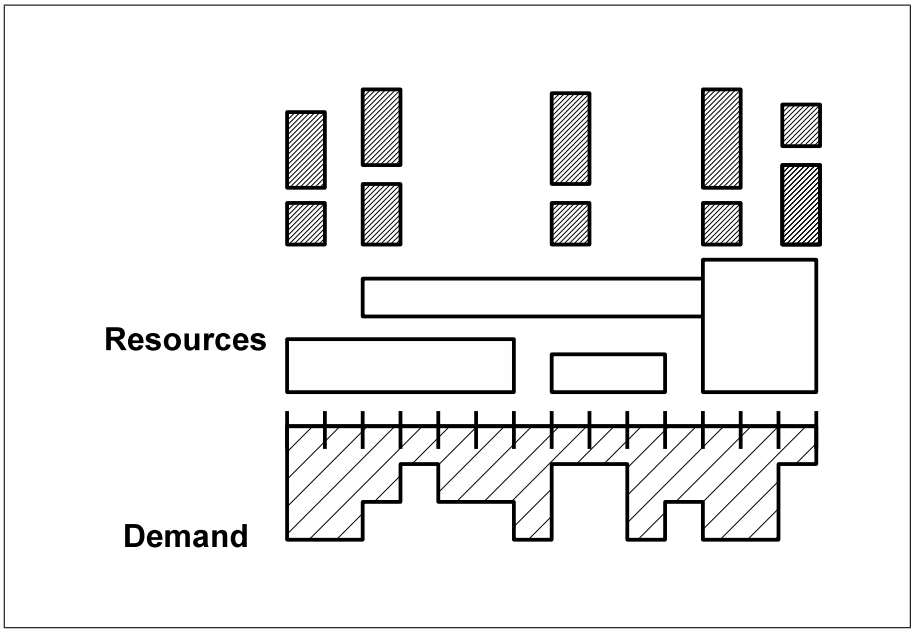


Figure 5.4: The LSPC problem

For a job j, let its *length* be $\ell_j = |I_j|$. Let $\ell_{\min}$ be the shortest job length, and $\ell_{\max}$ the longest job length. The proof of Lemma 5.5 is inspired by the algorithm for the MAX-UFP problem, due to Bansal et al. [5].

Lemma 5.5. *The input set of jobs can be partitioned into groups, $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_L$, such that each $\mathcal{M}_i$ is a mountain range and $L \leq 4 \cdot \lceil \log \frac{\ell_{\max}}{\ell_{\min}} \rceil$.*

Proof. We first categorize the jobs according to their lengths into r categories $C_1, C_2, \cdots, C_r$, where $r = \lceil \log \frac{\ell_{\max}}{\ell_{\min}} \rceil$. The category C_i consists of all the jobs with lengths in the range $[2^{i-1}\ell_{\min}, 2^i \ell_{\min})$. Thus all the jobs in any single category have comparable lengths: any two jobs j_1 and j_2 in the category satisfy $\ell_1 < 2\ell_2$, where ℓ_1 and ℓ_2 are the lengths of j_1 and j_2 respectively.

Consider any category C and let the lengths of the jobs in C lie in the range $[\alpha, 2\alpha)$. We claim that the category C can be partitioned into 4 groups G_0, G_1, G_2, G_3, such that each G_i is a mountain range. To see this, we divide the set of jobs in C into classes $H_1, H_2, \ldots, H_q, \ldots$ where H_q consists of the jobs containing the vertex $q \cdot \alpha$ (a job contains a vertex if the associated interval contains this vertex). Here q can possibly take any integer value. Note that every job belongs to some class

since all the jobs have length at least α; if a job belongs to more than one class, assign it to any one class arbitrarily. Clearly, each class H_q forms a mountain because any job in H_q contains the vertex $q\alpha$. For $0 \leq i \leq 3$, let G_i be the union of the classes H_q satisfying $q \equiv (i \mod 4)$. Since each job has length at most 2α, two classes H_j and H_{j+4} can't have an overlap, as they are separated by a distance 4α. Hence, each G_i is a mountain range. Thus, we get a decomposition of the input jobs into $4r$ mountain ranges. □

Assuming Lemma 5.5 and Theorem 5.2, we now outline the proof of Theorem 5.1. Let the optimal solution consists of k_i jobs from mountain range $\mathcal{M}_i, i = 1, \ldots, L$ (where L is given by Lemma 5.5), such that $k = \sum_i k_i$. Thus, if we knew $k_1, k_2, \ldots, k_L$, we could invoke Theorem 5.2 on each mountain range $\mathcal{M}_i$ (along with k_i as the partiality parameter) to determine a set of resources R_i having cost within a constant factor of the optimum for this mountain range. Taking the union of $R_1, R_2, \ldots, R_L$ yields a feasible solution R for the original problem instance. It is not difficult to argue that R is within a factor of cL of the optimum solution. The only issue in the above approach is that we do not know the values $k_1, k_2, \ldots, k_L$ (guessing them explicitly would take exponential time). However, this issue can be handled by using dynamic programming. The details are given below.

Proof of Theorem 5.1. Assuming Theorem 5.2, we prove Theorem 5.1. Let $\mathcal{J}$ be the input set of jobs, $\mathcal{R}$ be the input set of resources and k be the partiality parameter. Invoke Lemma 5.5 on the input set of jobs $\mathcal{J}$ and obtain a partitioning of $\mathcal{J}$ into mountain ranges $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_L$, where $L = 4 \cdot \lceil \log(\ell_{\max}/\ell_{\min}) \rceil$. Theorem 5.2 provides a c-approximation algorithm $\mathcal{A}$ for the PartialResAll problem wherein the input set of jobs form a single mountain range, where c is some constant. We shall present a (cL)-approximation algorithm for the PartialResAll problem.

For $1 \leq q \leq L$ and $1 \leq \kappa \leq k$, let $\mathcal{A}(q, \kappa)$ denote the cost of the (approximately optimal) solution returned by the algorithm in Theorem 5.2 with $\mathcal{M}_q$ as the input set of jobs, $\mathcal{R}$ as the input set of resources and κ as the partiality parameter. Similarly, let $\mathrm{OPT}(q, \kappa)$ denote the cost of the optimal solution for covering κ of the jobs in the mountain range $\mathcal{M}_q$. Theorem 5.2 implies that $\mathcal{A}(q, \kappa) \leq c \cdot \mathrm{OPT}(q, \kappa)$.

The algorithm employs dynamic programming. We maintain a 2-dimensional DP table DP$[\cdot,\cdot]$. For each $1 \leq q \leq L$ and $1 \leq \kappa \leq k$, the entry DP$[q,\kappa]$ would store the cost of a (near-optimal) feasible solution covering κ of the jobs from $\mathcal{M}_1 \cup \mathcal{M}_2 \cup \cdots \cup \mathcal{M}_q$. The entries are calculated as follows.

$$\text{DP}[q,\kappa] = \min_{\kappa' \leq \kappa}\{\text{DP}[q-1,\kappa-\kappa'] + \mathcal{A}(q,\kappa')\}.$$

The above recurrence relation considers covering κ' jobs from the mountain M_q, and the remaining $\kappa - \kappa'$ jobs from the mountains $M_1, \cdots, M_{q-1}$. Using this dynamic program, we compute a feasible solution to the original problem instance (i.e., covering k jobs from all the mountain ranges $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_L$); the solution would correspond to the entry DP$[L,k]$. Consider the optimum solution OPT to the original problem instance. Suppose that OPT covers k_q jobs from the mountain range $\mathcal{M}_q$ (for $1 \leq q \leq L$), such that $k_1 + k_2 + \cdots + k_L = k$. Observe that

$$\text{DP}[L,k] \leq \sum_{q=1}^{L} \mathcal{A}(q,k_q) \leq c \cdot \sum_{q=1}^{L} \text{OPT}(q,k_q),$$

where the first statement follows from the construction of the dynamic programming table and the second statement follows from the guarantee given by algorithm $\mathcal{A}$. However the maximum of OPT(q,k_q) (over all q) is a lower bound for OPT (we cannot say anything stronger than this since OPT might use the same resources to cover jobs across multiple subsets $\mathcal{M}_q$). This implies that DP$[L,k] \leq c \cdot L \cdot$ OPT. This proves the (cL)-approximation ratio.

It is easy to see that L is $O(\log(n+m))$ as argued below. It suffices if we consider only those vertices where some job or resource starts or ends; the other vertices can be ignored. Such a transformation will not affect the set of feasible solutions. Thus, without loss of generality, we can assume that the number of vertices is at most $2(n+m)$. Therefore, $\ell_{\max} \leq 2(n+m)$ and $\ell_{\min} \geq 1$. Hence, the overall algorithm has an $O(\log(n+m))$ approximation ratio. □

We now sketch the proof of Theorem 5.2. As mentioned earlier, there are two parts: single mountains and extension to mountain ranges via the LSPC problem.

For the case of a single mountain, we prove Theorem 5.3. The basic intuition is as follows. Given

the structure of the jobs, we will show that there is a *near-optimal* feasible solution that exhibits a nice property: the jobs discarded from the solution are extremal either in their left end-points or their right end-points. Let $\mathcal{J} = \{j_1, j_2, \ldots, j_n\}$ be the input set of jobs.

Lemma 5.6. *Consider the* PARTIALRESALL *problem for a single mountain. Let $S = (R_S, J_S)$ be a feasible solution such that R_S covers the set of jobs J_S with $|J_S| = k$. Let C_S denote its cost. Let $L =< l_1, l_2, \ldots, l_n >$ denote the jobs in increasing order of their left end-points. Similarly, let $R =< r_1, r_2, \ldots, r_n >$ denote the jobs in decreasing order of their right end-points. Then, there exists a feasible solution $X = (R_X, J_X)$ having cost at most $2 \cdot C_S$ such that*

$$\mathcal{J} \setminus J_X = \{l_i : i \leq q_1\} \cup \{r_i : i \leq q_2\} \tag{5.1}$$

for some $q_1, q_2 \geq 0$ where $|\mathcal{J} \setminus J_X| = n - k$.

Proof. We give a constructive proof to determine the sets J_X and R_X. We initialize the set J_X=$\mathcal{J}$. At the end of the algorithm, the set J_X will be the desired set of jobs covered by the solution. The idea is to remove the jobs that extend most to the right or the left from the consideration of J_X. The most critical aspect of the construction is to ensure that whenever we exclude any job from consideration of J_X that is already part of J_S, we do so in pairs of the leftmost and rightmost extending jobs of J_S that are still remaining in J_X. We terminate this process when the size of J_X equals the size of J_S, i.e., k. We also initialize the set $U = \phi$. At the end of the algorithm, this set will contain the set of jobs removed from $\mathcal{J}$ that belonged to J_S while constructing J_X.

We now describe the construction of J_X formally. We maintain two pointers *l-ptr* and *r-ptr*; *l-ptr* indexes the jobs in the sequence (L) of their left end-points and *r-ptr* indexes the jobs in the sequence (R) of their right end-points. We keep incrementing the pointer *l-ptr* and removing the corresponding job from J_X (if it has not already been removed) until either the size of J_X reaches k or we encounter a job (say *l-job*) in J_X that belongs to J_S; we do not yet remove the job *l-job*. We now switch to the pointer *r-ptr* and start incrementing it and removing the corresponding job from J_X (if it has not already been removed) until either the size of J_X reaches k or we encounter a job (say *r-job*) in J_X that belongs to J_S; we do not yet remove the job *r-job*. If the size of J_X

reaches k, we have the required set J_X.

Now suppose that $|J_X| \neq k$. Note that both *l-ptr* and *r-ptr* are pointing to jobs in S. Let *l-job* and *r-job* be the jobs pointed to by *l-ptr* and *r-ptr* respectively (note that these two jobs may be same).

We shall remove one or both of *l-job* and *r-job* from J_X and put them in U. We classify these jobs into three categories: *single*, *paired* and *artificially paired*.

Suppose that $|J_X| \geq k+2$. In this case, we have to delete at least 2 more jobs; so we delete both *l-job* and *r-job* and add them to U as *paired* jobs. In case *l-job* and *r-job* are the same job, we just delete this job and add it to U as a *single* job. We also increment the *l-ptr* and *r-ptr* pointers to the next job indices in their respective sequence. We then repeat the same process again, searching for another pair of jobs.

Suppose that $|J_X| = k+1$. In case *l-job* and *r-job* are the same job, we just delete this job and get the required set J_X of size k; We add this job to the set U as a *single* job. On the other hand, if *l-job* and *r-job* are different jobs, we remove *l-job* from J_X and add it to U as *artificially paired* with its pair as the job *r-job*; note that we do not remove *r-job* from J_X.

This procedure gives us the required set J_X. We now construct R_X by simply doubling the resources of R_S; meaning, that for each resource in R_S, we take twice the number of copies in R_X. Clearly $C_X = 2 \cdot C_S$. It remains to argue that R_X covers J_X. For this, note that $U = J_S - J_X$ and hence $|U| = |J_X - J_S|$ (because $|J_X| = |J_S| = k$). We create an arbitrary bijection $f : U \to J_X - J_S$. Note that J_X can be obtained from J_S by deleting the jobs in U and adding the jobs of $J_X - J_S$. We now make an important observation:

Observation 5.1. *For any* paired *jobs or* artificially paired *jobs j_1, j_2 added to U, all the jobs in J_X are contained within the span of this pair, i.e., for any j in J_X, $s_j \geq \min\{s_{j_1}, s_{j_2}\}$ and $t_j \leq \max\{t_{j_1}, t_{j_2}\}$. Similarly for any* single *job j_1 added to U, all jobs in J_X are contained in the span of j_1.*

For every *paired* jobs, j_1, j_2, Observation 5.1 implies that taking 2 copies of the resources covering $\{j_1, j_2\}$ suffices to cover $\{f(j_1), f(j_2)\}$. Similarly, for every *single* job j, the resources

covering $\{j\}$ suffice to cover $\{f(j)\}$. Lastly for every *artificially paired* jobs j_1, j_2 where $j_1 \in U$ and $j_2 \notin U$, taking 2 copies of the resources covering $\{j_1, j_2\}$ suffices to cover $\{f(j_1), j_2\}$. Hence the set R_X obtained by doubling the resources R_S (that cover J_S) suffices to cover the jobs in J_X. □

Recall that Bar-Noy et al. [6] presented a 4-approximation algorithm for the ResAll problem (full cover version). Our algorithm for handling a single mountain works as follows. Given a mountain consisting of the collection of jobs $\mathcal{J}$ and the number k, do the following for all possible pairs of numbers (q_1, q_2) such that the set J_X defined as per Equation 5.1 in Lemma 5.6 has size k. For the collection of jobs J_X, consider the issue of selecting a minimum cost set of resources to cover these jobs; note that this is a full cover problem. Thus, the 4-approximation of [6] can be applied here. Finally, we output the best solution across all choices of (q_1, q_2). Lemma 5.6 shows that this is an 8-factor approximation to the PartialResAll problem for a single mountain. This completes the proof of Theorem 5.3.

Theorem 5.4 is proved in Section 5.5. The reduction to the LSPC problem is given in Section 5.6.

5.5 LSPC Problem: Proof of Theorem 5.4

Finally, we complete the description of our algorithm by providing a 16-approximation algorithm for the LSPC problem. We extend the notion of profiles and coverage to intervals of the path. For an interval $[a, b]$, we say that an edge $e \in [a, b]$ if both of its end-points lie in $[a, b]$. Let $[a, b] \subseteq [1, |V|]$ be a *range*. By a profile over $[a, b]$, we mean a function Q that assigns a value $Q(e)$ to each edge $e \in [a, b]$. A profile Q defined over a range $[a, b]$ is said to be *good*, if for all edges $e \in [a, b]$, $Q(e) \leq d_e$ (where d_e is the input demand at e). In the remainder of the discussion, we shall only consider good profiles and so, we shall simply write "profile" to mean a "good profile". The *measure* of Q is defined to be the sum $\sum_{e \in [a,b]} Q(e)$.

Let S be a multiset of resources and let Q be a profile over a range $[a, b]$. We say that S is *good*, if for any edge e, it includes at most one short resource containing e. We say that S covers the profile Q, if for any edge $e \in [a, b]$, the sum of capacities of resources active in S and containing e

is at least $Q(e)$. Notice that S is a feasible solution to the input problem instance, if there exists a profile Q over the entire range $[1, |V|]$ such that Q has measure k and S is a cover for Q. For an edge e, let $Q_S^{\text{sh}}(e)$ denote the capacity of the unique short resource from S containing e, if one exists; otherwise, $Q_S^{\text{sh}}(e) = 0$.

Let S be a good multiset of resources and let Q be a profile over a range $[a, b]$. For a long resource $i \in S$, let $f_S(i)$ denote the number of copies of i included in S. The multiset S is said to be a *single long resource assignment cover* (SLRA cover) for Q, if for any edge $e \in [a, b]$, there exists a long resource $i \in S$ such that $w_i f_S(i) \geq Q(e) - Q_S^{\text{sh}}(e)$ (intuitively, the resource i can cover the residual demand by itself, even though there are other long resources in S containing e).

We say that a good multiset of resources S is an *SLRA solution* to the input LSPC problem instance, if there exists a profile Q over the range $[1, |V|]$ having measure k such that S is an SLRA cover for Q. The lemma below shows that near-optimal SLRA solutions exist.

Lemma 5.7. *Consider the input instance of the* LSPC *problem. There exists an SLRA solution having cost at most 16 times the cost of the optimal solution.*

To prove Lemma 5.7, we will use the following lemma, which is a reformulation of Theorem 1 in [14]. For a multiset of resources S, let $c(S)$ denote its cost.

Lemma 5.8. *[14] Let $\widehat{S}$ be a multiset of long resources covering a profile $\widehat{Q}$ over the range $[1, |V|]$. Then, there exists a multiset of long resources S' such that S' is a SLRA cover for Q and $c(S') \leq 16 \cdot c(\widehat{S})$.*

Proof of Lemma 5.7. Let OPT be the optimum solution and let Q be the profile of measure k covered by OPT. Let OPT_l and OPT_s be the multiset of long and short resources contained in OPT, respectively. Define Q_l to be the residual profile over $[1, |V|]$: $Q_l(e) = Q(e) - Q_S^{\text{sh}}(e)$. The multiset OPT_l covers the profile Q_l. Invoke Lemma 5.8 on OPT_l and Q_l (taking $\widehat{S} = \text{OPT}_l$ and $\widehat{Q} = Q_l$) and obtain a multiset of long resources S' which forms a SLRA cover for Q_l. Construct a new multiset S, by taking the union of S' and OPT_s. Notice that S is a SLRA solution. The cost of S' is at most 16 times the cost of OPT_l. So, S has cost at most 16 times the cost of OPT. □

Surprisingly, we can find the *optimum* SLRA solution S^* in polynomial time, as shown in Theorem 5.9 below. Lemma 5.7 and Theorem 5.9 imply that S^* is a 16-factor approximation to the optimum solution. This completes the proof of Theorem 5.4.

Theorem 5.9. *The optimum SLRA solution S^* can be found in polynomial time.*

The rest of the section is devoted to proving Theorem 5.9. The algorithm goes via dynamic programming. The following notation is useful in our discussion.

- Let S be a good set of resources consisting of only short resources, and let $[a, b]$ be a range. For a profile Q defined over $[a, b]$, and an integer h, S is said to be an *h-free cover* for Q, if for any $e \in [a, b]$, $Q_S^{\text{sh}}(e) \geq Q(e) - h$. The set S is said to be an *h-free q-cover* for $[a, b]$, if there exists a profile Q over $[a, b]$ such that Q has measure q and S is a h-free cover for Q.

- Let S be a good multiset of resources and let $[a, b]$ be a range. For a profile Q defined over $[a, b]$, and an integer h, the multiset S is said to be an *h-free SLRA cover* for Q, if for any edge $e \in [a, b]$ satisfying $Q(e) - Q_S^{\text{sh}}(e) > h$, there exists a long resource $i \in S$ such that $w_i f_S(i) \geq Q(e) - Q_S^{\text{sh}}(e)$. For an integer q, we say S is an *h-free SLRA q-cover* for the range $[a, b]$, if there exists a profile Q over $[a, b]$ such that Q has measure q and S is a h-free SLRA cover for Q.

Intuitively, h denotes the demand covered by long resources already selected (and their cost accounted for) in the previous stages of the algorithm; thus, edges whose residual demand is at most h can be ignored. The notion of "h-freeness" captures this concept.

We shall first argue that any h-free SLRA cover S for a profile Q over a range $[a, b]$ exhibits certain interesting decomposition property. Intuitively, in most cases, the range can be partitioned into two parts (left and right), and S can be partitioned into two parts S_1 and S_2 such that S_1 can cover the left range and S_2 can cover the right range (even though resources in S_1 may contain some edges in the right range and those in S_2 may be contain edges in the left range). In the cases where the above decomposition is not possible, there exists a long resource spanning almost the entire range.

Lemma 5.10. *Let $[a, b]$ be any range, Q be a profile over $[a, b]$ and let h be an integer. Let S be a good multiset of resources providing an h-free SLRA-cover for Q. Then, one of the following three cases holds:*

- *The set of short resources in S form a h-free cover for Q.*
- *Vertex-cut: There exists a vertex v^*, $a \leq v^* \leq b-1$, and a partitioning of S into S_1 and S_2 such that S_1 is an h-free SLRA-cover for Q_1 and S_2 is an h-free SLRA-cover for Q_2, where Q_1 and Q_2 profiles are obtained by restricting Q to $[a, v^*]$ and $[v^* + 1, b]$, respectively.*
- *Interval-cut: There exists a long resource $i^* \in S$ such that the set of short resources in S forms a h-free cover for both Q_1 and Q_2, where Q_1 and Q_2 are the profiles obtained by restricting Q to $[a, s_{i^*} - 1]$ and $[t_{i^*} + 1, b]$ respectively.*

We first extend the notion of an SLRA cover to subsets of edges. Let $\mathcal{T} \subseteq E$ be a set of edges and let $\widehat{Q}$ be a profile over the set $\mathcal{T}$. A good multiset of resources S is said to be a SLRA cover for $\mathcal{T}$, if for any edge $e \in \mathcal{T}$, there exists a long resource $i \in S$ such that $w_i f_S(i) \geq Q(e) - Q_S^{\text{sh}}(e)$. We will use the following lemma, which is a reformulation of Lemma 4 in [14].

Lemma 5.11. *Let $\widehat{S}$ be a multiset consisting of only long resources. Let $\widehat{Q}$ be a profile over a non-empty set of edges $\mathcal{T}' \subseteq [a, b]$, for some a and b. Suppose $\widehat{S}$ is a SLRA cover for $\widehat{Q}$. Then one of the following properties is true:*

- *There exists a vertex $v^* \in [a, b-1]$ and a partition of $\widehat{S}$ into $\widehat{S}_1$ and $\widehat{S}_2$ such that $\widehat{S}_1$ is a SLRA cover for $\widehat{Q}_1$ and $\widehat{S}_2$ is a SLRA cover for $\widehat{Q}_2$, where $\widehat{Q}_1$ and $\widehat{Q}_2$ are the profiles obtained by restricting $\widehat{Q}$ to the edges in $\mathcal{T}' \cap [a, v^*]$ and $\mathcal{T}' \cap [v^* + 1, b]$, respectively.*
- *There exists a resource $i^* \in \widehat{S}$ spanning all the edges in $\mathcal{T}'$.*

Proof of Lemma 5.10. Consider a good multiset of resources S forming a h-free SLRA cover for a profile Q over a range $[a, b]$. Define the set of edges $\mathcal{T}'$:

$$\mathcal{T}' = \{e \in [a, b] \ : \ Q(e) - Q_S^{\text{sh}}(e) > h\}.$$

If $\mathcal{T}'$ is empty, then S is a h-free cover for Q; this corresponds to the first case of Lemma 5.10. So, assume $\mathcal{T}' \neq \emptyset$. Define a profile $\widehat{Q}$ over the edges in $\mathcal{T}'$: for any $e \in \mathcal{T}'$, let $\widehat{Q}(e) = Q(e) - Q_S^{\text{sh}}(e)$. Notice that S is a SLRA cover for the profile $\widehat{Q}$. Invoke Lemma 5.11 (with $\widehat{S} = S$). Let us analyze the two cases of the above lemma. Consider the first case in Lemma 5.11. In this case, there exists a vertex v^* and a partitioning of S into S_1 and S_2, with the stated properties. In this case, we see that S_1 and S_2 are h-free SLRA covers for $[a, v^*]$ and $[v^* + 1, b]$, respectively. This corresponds to the second case of Lemma 5.10. Consider the second case in Lemma 5.11. In this case, there exists a long resource $i^* \in S$ such that i^* spans all the edges in $\mathcal{T}'$. This means that for any $e \in [a, s_{i^*} - 1]$ or $e \in [t_{i^*} + 1, b]$, $Q(e) - Q_S^{\text{sh}}(e) \leq h$. Otherwise, $e \in \mathcal{T}'$ and i^* will contain e. This corresponds to the third case of Lemma 5.10. □

We now discuss our dynamic programming algorithm. Let $H = \max_{e \in E} d_e$ be the maximum of the input demands. The algorithm maintains a table M with an entry for each triple $\langle [a, b], q, h \rangle$, where $[a, b] \subseteq [1, |V|]$, $0 \leq q \leq k$ and $0 \leq h \leq H$. The entry $M([a, b], q, h)$ stores the cost of the optimum h-free SLRA q-cover for the range $[a, b]$; if no solution exists, then $M([a, b], q, h)$ will be ∞. Our algorithm outputs the solution corresponding to the entry $M([1, |V|], k, 0)$; notice that this is optimum SLRA solution S^*. Since we are computing $M([1, |V|], k, 0)$, the computation will depend only on k and not on $H = \max_{e \in E} d_e$, as $h = 0$. Computation of entries in both the tables M and A requires polynomial time, as is evident from the recurrence relations.

In order to compute the table M, we need an auxiliary table A. For a triple $[a, b]$, q and h, let $A([a, b], q, h)$ be the optimum h-free q-cover for $[a, b]$ (using only the short resources); if no solution exists $A([a, b], q, h)$ is said to be ∞. It is straightforward to compute the table A and this is explained in Section 5.5.2.

Based on the decomposition lemma (Lemma 5.10), we can develop a recurrence relation for a triple $[a, b]$, q and h. We compute $M([a, b], q, h)$ as the minimum over three quantities E_1, E_2 and E_3 corresponding to the three cases of the lemma. Intuitive description of the three quantities is given below and precise formulas are provided in Figure 5.5. In the figure, $\mathcal{L}$ is the set of all long resources[1].

[1]The input demands d_e are used in computing the table $A(\cdot, \cdot, \cdot)$

$$
\begin{aligned}
E_1 &= A([a,b],q,h).\\
E_2 &= \min_{\substack{c\in[a,b-1]\\ q_1\le q}} M([a,c],q_1,h)+M([c+1,b],q-q_1,h).\\
E_3 &= \min_{\substack{(i\in\mathcal{L},\alpha\le H)\ :\ \alpha w_i>h\\ q_1,q_2,q_3\ :\ q_1+q_2+q_3=q}} \begin{pmatrix}\alpha\cdot c_i + A([a,s_i-1],q_1,h)\\ +M([s_i,t_i],q_2,\alpha w_i)+A([t_i+1,b],q_3,h)\end{pmatrix}
\end{aligned}
$$

Figure 5.5: Recurrence relation for M

- *Case 1:* No long resource is used and so, we just use the corresponding entry $A([a,b],q,h)$ of the table A.

- *Case 2:* There exists a vertex-cut v^*. We consider all possible values of v^*. For each possible value of v^*, we try all possible ways in which q can be divided between the left and right ranges.

- *Case 3:* There exists a long resource i^* such that the ranges to the left of and to the right of i^* can be covered solely by short resources. We consider all the long resources i and also the number of copies α to be picked. Once α copies of i are picked, i can cover all edges with residual demand at most αw_i in an SLRA fashion, and so the subsequent recursive calls can ignore these edges. Hence, this value is passed to the recursive call. We also consider different ways in which q can be split into three parts – left, middle and right. The left and right parts will be covered by the solely short resources and the middle part will use both short and long resources. Since we pick α copies of i, a cost of αc_i is added.

We set $M([a,b],q,h) = \min\{E_1, E_2, E_3\}$. For the base case: for any $[a,b]$, if $q=0$ or $h=H$, then the entry is set to zero.

The order in which the entries of the table are filled is explained in Section 5.5.1. Computation of the entries in A is explained in Section 5.5.2. Using Lemma 5.10, we can argue that the above recurrence relation correctly computes all the entries of M. For the sake of completeness, a proof is included in Section 5.5.3.

5.5.1 DP Ordering

Define a partial order $\prec$ as follows. For pair of triples $z = ([a, b], q, h)$ and $z' = ([a', b'], q', h')$, we say that $z \prec z'$, if one of the following properties is true: (i)$[a', b'] \subseteq [a, b]$; (ii) $[a, b] = [a', b']$ and $q < q'$; (iii) $[a, b] = [a', b']$, $q = q'$ and $h > h'$. Construct a directed acyclic graph (DAG) D where the triples are the vertices and an edge is drawn from a triple z to a triple z', if $z \prec z'$. Let π be a topological ordering of the vertices in D. We fill the entries of the table M in the order of appearance in π. Notice that the computation for any triple z only refers to triples appearing earlier than z in π.

5.5.2 Computing the table A

We now describe how to compute the auxiliary table A. For a triple consisting of an edge e, $q \leq k$ and $h \leq H$, define $\gamma(e, q, h)$ as the cheapest cost of covering $q - h$ demand from the short resources containing e. This is a Knapsack problem and can be computed by dynamic programming. Time-complexity of the Knapsack problem is $O(n_e(q - h))$, where n_e is the number of short resources containing e.

Then, for a triple $\langle [a, b], q, h \rangle$, the entry $A([a, b], q, h)$ is governed by the following recurrence relation. Of the demand q that needs to be covered, the optimum solution may cover a demand q_1 from the edge e, and a demand $q - q_1$ from the range $[a, b - 1]$. We try all possible values for q_1 and choose the best:

$$A([a, b], q, h) = \min_{q_1 \leq \min\{q, d_b\}} A([a, b - 1], q - q_1, h) + \gamma(b, q_1, h).$$

It is not difficult to verify the correctness of the above recurrence relation.

5.5.3 Correctness of the Recurrence Relation (Figure 5.5)

We prove Theorem 5.9 by induction on the position in which a triple appears in the topological ordering π. The base case corresponds to triples that do not have a parent in D. Theorem 5.9 is trivially true in this case.

Consider any triple $z = ([a, b], q, h)$. Let S be the optimum h-free SLRA q-cover for $[a, b]$. There exists a profile Q over $[a, b]$ such that Q has measure q and S is a h-free SLRA cover for Q. Let us invoke Lemma 5.10 and consider its three cases.

Suppose the first case of the lemma is true. Let S_s be the set of short resources contained in S. Then, S_s is a h-free cover for Q. Therefore $E_1 = A([a, b], q, h) \leq c(S_s) \leq c(S)$.

Suppose the second case of the lemma is true. Let v^* be the vertex and S_1 and S_2 be the partition given by the lemma. Let Q_1 and Q_2 be the profiles obtained by restricting Q to the ranges $[a, v^*]$ and $[v^* + 1, b]$, respectively. Let the measures of Q_1 and Q_2 be q_1 and q_2, respectively. Then S_1 is a h-free q_1-cover for $[a, v^*]$ and S_2 is a h-free q_2-cover for $[v^* + 1, b]$. Therefore, by induction, $M([a, v^*], q_1, h) \leq c(S_1)$ and $M([v^* + 1, b], q_2, h) \leq c(S_2)$. In computing the quantity E_2, we try all possible ways of partitioning the range $[a, b]$ and dividing the number q. Hence, $E_2 \leq c(S_1) + c(S_2)$. Since $c(S) = c(S_1) + c(S_2)$, we see that $E_2 \leq c(S)$.

Suppose the third case of lemma is true. Let i^* be the long resource given by the lemma. Let S_1 be short resources in S that contain edges in $[a, s_{i^*} - 1]$. Similarly, let S_3 be the set of short resources in S that contain edges in $[t_{i^*} + 1, b]$. Let S_2 be the multiset of long resources in S and the set of short resources in S that contain edges in $[a, b]$. Let Q_1, Q_2 and Q_3 be the profiles obtained by restricting Q to the ranges $[a, s_{i^*} - 1]$, $[s_{i^*}, t_{i^*}]$ and $[t_{i^*} + 1, b]$, respectively. The lemma guarantees that S_1 and S_2 are h-free covers for Q_1 and Q_3 respectively. Let q_1, q_2 and q_3 be the measures of Q_1, Q_2 and Q_3, respectively. We see that $A([a, s_{i^*} + 1], q_1, h) \leq c(S_1)$ and $A([t_{i^*} + 1, b], q_3, h) \leq c(S_3)$. Let $\alpha^* = f_S(i^*)$ be the number of copies of i^* present in S. Notice that if $\alpha^* w_{i^*} \leq h$, then i^* is not a useful resource, because i^* will be covering only edges in $[s_{i^*}, t_{i^*}]$ with residual demands at most h; but all such edges are free and need not be covered. So, without loss of generality, assume that $\alpha^* w(i^*) > h$. Since i^* spans the entire range $[s_{i^*}, t_{i^*}]$, the resource i^* can cover all edges in the above range with residual demands at most $\alpha^* w(i^*)$. Let $S_2' = S_2 - \{i^*\}$. Notice that S_2' is a $(\alpha^* w_i)$-free SLRA cover for the profile Q_2. Therefore, S_2' is a $(\alpha^* w_i)$-free q_2-cover for the range $[s_{i^*}, t_{i^*}]$. Hence, by induction, $M([s_{i^*}, t_{i^*}], q_2, \alpha^* w(i^*)) \leq c(S_2')$. Therefore, $E_3 \leq c(S_1) + c(S_2) + c(S_3) = c(S)$.

The quantity $E = \min\{E_1, E_2, E_3\}$; so $E \leq c(S)$. The proof is now complete. □

Running time

Computing E_1 requires the table entry $A([a, b], q, h)$. Once we have filled the table A in polynomial time, this requires constant time. So, computing E_1 requires polynomial time. Computing E_2 requires the table entries $M([a, v^*], q_1, h)$ and $M([v^* + 1, b], q_2, h)$ for all possible values of $a \leq v^* \leq b$. So we need to compute $2(b - a + 1)$ entries of the table M which is polynomial. Hence, computing E_2 requires polynomial time. Computing E_3 requires the table entries $A([a, s_{i^*} + 1], q_1, h)$, $M([s_{i^*}, t_{i^*}], q_2, \alpha^* w(i^*))$ and $A([t_{i^*} + 1, b], q_3, h)$. Since each of them can be computed in polynomial time, computing E_3 requires polynomial time. Hence, the overall running time is polynomial.

5.6 Single Mountain Range: Proof of Theorem 5.2

In this section, we prove Theorem 5.2 via a reduction to LSPC. Recall that in the LSPC problem, we are given a demand profile over the set of edges E, which specifies an integral demand d_e for every edge e. The input resources are of two types, *short* and *long*. A short resource spans only one edge, whereas a long resource can span one or more edges. Each resource i has a cost c_i and a capacity w_i. The input also specifies a *partiality parameter* k. A feasible solution S consists of a multiset of resources S and a coverage profile: an integer k_e for each edge e satisfying $k_e \leq d_e$. The solution should have the following properties: (i) $\sum_e k_e \geq k$; (ii) at any edge e, the sum of capacities of the resource intervals from S containing e is at least k_e; (iii) for any edge e, at most one of the short resources containing e is picked (however, multiple copies of a long resource may be included). The objective is to find a feasible solution having minimum cost.

The reduction proceeds in two steps.

5.6.1 First Step

Let the input instance be $\mathcal{A}$, wherein the input set of jobs form a mountain range $\mathcal{M} = \{M_1, M_2, \cdots, M_r\}$. We will transform the instance $\mathcal{A}$ to an instance $\mathcal{B}$, with some nice properties: (1) the input set of jobs in $\mathcal{B}$ also form a mountain range; (2) every resource i in the instance $\mathcal{B}$ is either narrow or wide (see Section 5.3 for the definitions); (3) the cost of the optimum solution for the instance $\mathcal{B}$

is at most 3 times the optimal cost for the instance $\mathcal{A}$; (4) given a feasible solution to $\mathcal{B}$, we can construct a feasible solution to $\mathcal{A}$ preserving the cost.

Consider each resource i in $\mathcal{A}$ and let $M_p, M_{p+1}, \cdots, M_q$ (where $1 \leq p \leq q \leq r$) be the sequence of mountains that i intersects. Clearly, i fully spans the mountains $M_{p+1}, \cdots, M_{q-1}$. We will split the resource i into at most 3 new resources i_1, i_2, i_3; we say that i_1, i_2 and i_3 are *associated with* i. The resource i_2 will fully span the mountains $M_{p+1}, \cdots, M_{q-1}$. The span of the resource i_1 is the intersection of the span of i with the mountain M_p. Likewise, the span of the resource i_3 is the intersection of the span of i with the mountain M_q. The capacities and the costs of i_1, i_2 and i_3 are declared to be the same as that of i. We include i_1, i_2, i_3 in $\mathcal{B}$. The input set of jobs and the partiality parameter k, in $\mathcal{B}$ are identical to that of $\mathcal{A}$. This completes the reduction.

It is easy to see that the first two properties are satisfied by $\mathcal{B}$. Let us now consider third property . Given any solution S for the instance $\mathcal{A}$, we can construct a solution S' for $\mathcal{B}$ as follows. For each copy of resource i picked in S, include a single copy of i_1, i_2 and i_3 in S'. Clearly, the cost of the solution S' is at most thrice that of the cost of S. Regarding the fourth property, given a solution S to $\mathcal{B}$, we can construct a solution S' to $\mathcal{A}$ as follows. Consider any resource i in $\mathcal{A}$ and let i_1, i_2 and i_3 be the resources in $\mathcal{B}$ associated with i. Let f_1, f_2, f_3 be the number of copies of i_1, i_2, i_3 picked by solution S. Let $f = \max\{f_1, f_2, f_3\}$. Include f copies of the resource i in the solution S'. It is easy to see that S' is a feasible solution to $\mathcal{A}$ and that the cost of S' is at most the cost of S.

5.6.2 Second Step

In this step we reduce the problem instance $\mathcal{B}$ to an LSPC instance $\mathcal{C}$, with the following properties: (1) the cost of the optimum solution for the instance $\mathcal{C}$ is at most 8 times the optimal cost for the instance $\mathcal{B}$; (2) Given a feasible solution to $\mathcal{C}$, we can construct a feasible solution to $\mathcal{B}$ preserving the cost.

Reduction

In the instance $\mathcal{C}$, we retain only the peak edges of the various mountains in the instance $\mathcal{B}$ so that the number of edges in $\mathcal{C}$ is the same as the number of mountains r in $\mathcal{B}$. Let the mountain ranges in $\mathcal{B}$ ordered from left to right be $M_1, \ldots, M_r$, with e_p being the peak edge of M_p. For any peak edge e in the instance $\mathcal{B}$, let d_e be the number of jobs in $\mathcal{B}$ that contain the edge e; we assign demand d_e to the edge e in the instance $\mathcal{C}$. For any wide resource i in $\mathcal{B}$, fully spanning mountains $M_p, M_{p+1}, \cdots, M_q$, create a long resource i' in $\mathcal{C}$ with the span $e_p, e_{p+1}, \ldots, e_q$. The cost and capacity of i' are the same as that of i.

The narrow resources in the instance $\mathcal{B}$ are used to construct the short resources in the instance $\mathcal{C}$ as follows. Consider any specific mountain M in the instance $\mathcal{B}$ along with the collection of narrow resources R that are contained in the span of M, and let e be the peak edge of M. Let $\mathcal{A}_{SM}$ be the algorithm implied in Theorem 5.3 for the single mountain M. For any integer κ ($1 \leq \kappa \leq d_e$), we add a short resource $i_s^{e,\kappa}$ with capacity κ. The cost C of this resource is determined as follows. We apply $\mathcal{A}_{SM}$ on M, with κ as the partiality parameter, and the set of narrow resources R as the only resources. Then, Theorem 5.3 gives us a solution of cost C consisting of a multiset R' of some resources in R, that covers κ of the jobs in the mountain M. The cost of the short resource $i_s^{e,\kappa}$ will be C. We will call the (multi)set of narrow resources $R' \subseteq R$ in the instance $\mathcal{B}$ as *associated* with the short resource $i_s^{e,\kappa}$. This completes the description of the instance $\mathcal{C}$ of the LSPC problem.

Validity of the reduction

We will now argue the validity of the reduction. Let us consider the first property: the cost of the optimum solution to the instance $\mathcal{C}$ has cost at most 8 times the cost of the optimum solution to the instance $\mathcal{B}$. The following lemma is useful for this purpose.

Lemma 5.12. *Let J be a subset of jobs and R be multiset of resources in the instance $\mathcal{B}$ such that R covers J (note that R contains only narrow or wide resources and J forms a mountain range). Let R_1 and R_2 be the narrow and the wide resources in R respectively. Let R_2' be a multiset constructed by picking twice the number of copies of each resource in R_2. Then, J can be partitioned into two*

sets J_1 and J_2 such that J_1 is solely covered by the resources in R_1 and J_2 is solely covered by the resources in R'_2.

Proof. For now, we assume that the mountain range comprises of a single mountain. Let $P_R(\cdot)$, $P_{R_1}(\cdot)$, $P_{R_2}(\cdot)$ and $P_{R'_2}(\cdot)$ denote the profile of the resources in R, R_1, R_2 and R'_2 respectively. Note that $P_{R_2}(\cdot)$ is a uniform bandwidth profile having uniform height, say h. This is because these correspond to wide resources, which span all of this mountain. Let J_L be the first h jobs among all the jobs in J sorted in ascending ordered by their left end-points. Similarly, let J_R be the first h jobs among all the jobs in J sorted in descending order by their right end-points. Intuitively, J_L and J_R correspond to the h left-most and the h right-most jobs in the mountain.

Let $J_2 = J_L \cup J_R$ and $J_1 = J \setminus J_2$. Let $P_J(\cdot)$, $P_{J_1}(\cdot)$ and $P_{J_2}(\cdot)$ denote the profiles of the jobs in J, J_1 and J_2 respectively.

Note that the profile $P_{R'_2}(t)$ has height $2h$ throughout the span of the mountain whereas the profile $P_{J_2}(\cdot)$ has height at most $2h$ at any edge. Thus R'_2 covers J_2.

We will now show that R_1 covers J_1. Note that $P_{J_1}(e) = P_J(e) - P_{J_2}(e)$ for any edge e. We partition the edges into two parts: $E_0 = \{e : P_{J_1}(e) = 0\}$ and $E_{>0} = \{e : P_{J_1}(e) > 0\}$. For the edges in E_0, there are no jobs remaining in J_1 for R_1 to cover. For the edges in $\mathcal{E}_{>0}$, we note that $P_{J_1}(e) \leq P_J(e) - h$ (because J_2 comprises of the left-most h and right-most h jobs of the mountain). Also note that the profile $P_{R_1}(e) = P_R(e) - P_{R_2}(e) = P_R(e) - h$. Since, R covers J, this implies that R_1 is sufficient to cover J_1.

The proof can easily be extended to a mountain range as the mountains within a mountain range are disjoint. □

We are now ready to show that our reduction is valid. Let $\mathrm{OPT}(\mathcal{B})$ and $\mathrm{OPT}(\mathcal{C})$ be the cost of an optimal solution for the instances $\mathcal{B}$ and $\mathcal{C}$ respectively.

Lemma 5.13. $\mathrm{OPT}(\mathcal{C}) \leq 8 \cdot \mathrm{OPT}(\mathcal{B})$. *Further, given a feasible solution for $\mathcal{C}$, one can convert it to a feasible solution for $\mathcal{B}$ without increasing the cost.*

Proof. Let $\mathrm{OPT} = (R, J)$ denote the optimal solution for the problem instance $\mathcal{B}$, where J is the set of jobs picked by the solution and R is the set of resources covering J (we have $|J| = k$). Let

R_1 and R_2 be the set of narrow and wide resources in R. Apply Lemma 5.12 for the solution (R, J) and obtain a partition of J into J_1 and J_2 along with R_1 (covering J_1) and R'_2 (covering J_2). Let $\mathcal{M} = M_1, M_2, \ldots, M_r$ be the input mountain range in the instance $\mathcal{B}$ with peak edges $e_1, e_2, \ldots, e_r$, respectively. Consider any mountain M_q. Let k_q be the number of jobs picked in J from the mountain M_q. Let $R_{1,q}$ be the set of (narrow) resources from R_1 contained within the span of M_q. Thus, the set of resources $R_{1,q}$ cover the set of jobs in $M_q \cap J_1$ and let $k'_q = |M_q \cap J_1|$. Corresponding to the value k'_q, we would have included a short resource, say $i_q^{e_1,k'_q}$ in the instance $\mathcal{C}$, where e_1 is the peak edge of M_q; cost of i_q is at most 8 times the cost of $R_{1,q}$ (as guaranteed by Theorem 5.3). The set of long resources in R'_2 cover at least $k_q - k'_q$ jobs within the mountain M_q.

Construct a solution to the instance $\mathcal{C}$ by including $i_1, i_2, \ldots, i_q$; and for each copy of a wide resource i in R'_2, include a copy of its corresponding long resource. Notice that this is a feasible solution to the instance $\mathcal{C}$. The cost of the short resources $\{i_1, i_2, \ldots, i_q\}$ is at most 8 times the cost of R_1 and the cost of the long resources is the same as that of R'_2, which is at most twice that of R_2. Cost of OPT is the sum of costs of R_1 and R_2. Hence, cost of the constructed solution is at most 8 times the cost of OPT.

We now prove the second property: let S be a given a solution to the instance $\mathcal{C}$ of the LSPC problem of cost c; the solution also provides a coverage profile, k_e for each edge e (such that $\sum_e k_e = k$). We produce a feasible solution $S' = (R', J')$ to the instance $\mathcal{B}$ with the same cost c. For each long resource picked by S, we retain the corresponding wide resource in R' (maintaining the number of copies). Consider any edge e in the LSPC instance and let M be the corresponding mountain in the instance $\mathcal{B}$. The solution S contains at most one short resource i_s^{e,k'_e} containing e of capacity $k'_e = w_{i_s}$. Consider the multiset of short resources R' in the instance $\mathcal{B}$ associated with the resource i_s^{e,k'_e}. The multiset R' covers a set of k'_e jobs contained in the mountain M. Include all these k'_e jobs in J'. Choose any other $k_e - k'_e$ jobs contained in M and add these to J'; notice that the wide resources retained in R' can cover these jobs. This way we get a solution S' for the instance $\mathcal{B}$. Cost of the solution S' is at most the cost of S. □

Proof of Theorem 5.2: By composing the reductions given in the two steps, we get a reduction from the PartialResAll problem on a single mountain range to the LSPC problem. The first

step and the second step incur a loss in approximation of 3 and 8, respectively. Thereby, the combined reduction incurs a loss of 24. Theorem 5.4 provides a 16-approximation algorithm for the LSPC problem. Combining the reduction and the above algorithm, we get an algorithm for the PartialResAll for a single mountain range with an approximation ratio of $16 \times 24 = 384$.

Note that the running time of the algorithm depends on $\max_{e \in E} d_e$. We can assume that d_e is polynomially bounded for all $e \in E$, because initially all demands are 1 and so resources must have polynomially bounded capacity. Hence, the algorithm runs in polynomial time.

5.7 Overall Algorithm

Now that we have completed the description of the algorithm, we give an overall review of the algorithm.

1. Use the decomposition Lemma 5.5 to partition the input jobs into a set of mountain ranges.
2. We obtain a constant factor approximation algorithm where the input jobs form a mountain.
3. We then extend this result to a mountain range by reducing the problem to the LSPC problem.
4. We extend this to several mountain ranges by using dynamic programming.

5.8 The PrizeCollectingResAll problem

In this section, we consider the PrizeCollectingResAll problem. We prove the following:

Theorem 5.14. *There is a 4-factor approximation algorithm for the* PrizeCollectingResAll *problem.*

The proof proceeds by exhibiting a reduction from the PrizeCollectingResAll problem to the following full cover problem.

Problem Definition: We are given a demand profile which specifies an integral demand d_e for each edge e. The input resources are of two types, called S-type (short for single) and M-type (short

for multiple). A resource i has a capacity w_i, and cost c_i. A valid solution consists of a multiset of resources such that it includes at most 1 copy of any S-type resource; however arbitrarily many copies of any M-type resource may be picked. A feasible solution S is a valid solution such that for any edge e, the total capacity of the resources in S containing e is at least the demand d_e of the edge e. The objective is to find a feasible solution having minimum cost. We call this problem the Single Multiple Full Cover (SMFC) problem.

The full cover problem, (0-1)-ResAll is considered in [14]. The (0-1)-ResAll problem specifies demands for edges, and a feasible solution consist of a set of resources such that the demand of every edge is fulfilled by the cumulative capacity of the resources containing that edge. The main qualification is that in this problem setting, any resource may be picked up *at most once*. In [14], it is shown that this problem admits a 4-factor approximation algorithm. The SMFC problem easily reduces to the (0-1)-ResAll problem: *S-type* resources may be picked up at most once, and keep copies of the *M-type* resources so that it suffices to select any one of the copies. Thus the algorithm and the performance guarantee claimed in [14] also implies the following:

Theorem 5.15. *There is a* 4*-factor approximation to the* SMFC *problem.*

We proceed to exhibit our reduction from the PrizeCollectingResAll problem to the SMFC problem. Given an instance $\mathcal{I}$ of the PrizeCollectingResAll problem, we will construct an instance $\mathcal{O}$ of the SMFC problem, such that any optimal solution OPT($\mathcal{I}$) can be converted (at no extra cost) into an optimal solution OPT($\mathcal{O}$) for the instance $\mathcal{O}$. Consider any job j in the instance $\mathcal{I}$; we will create a S-type resource $r(j)$ in the instance $\mathcal{O}$ corresponding to j. The resource $r(j)$ will have the same length, left and right end-points as those of the job j, and will have a cost p_j (the penalty associated with job j). The resources in instance $\mathcal{I}$ will be labeled as M-type resources in the instance $\mathcal{O}$. The other parameters, such as demands of edges, are inherited by $\mathcal{O}$ from the instance $\mathcal{I}$.

We show that any feasible solution $S_{\mathcal{I}}$ to the PrizeCollectingResAll problem corresponds to a feasible solution $S_{\mathcal{O}}$ (of the same cost) for the SMFC problem. Let $\mathcal{J}'$ denote the set of jobs that are not covered by the solution $S_{\mathcal{I}}$ (thus, the solution pays the penalty for each of the jobs in $\mathcal{J}'$).

The multiset of resources in $S_{\mathcal{O}}$ consists of the (M-type) resources that exist in the solution $S_{\mathcal{I}}$, and the S-type resources $r(j)$ in $\mathcal{O}$ corresponding to every job j in $\mathcal{J}'$. Any job j that is actually covered by the set of resources in $S_{\mathcal{I}}$ is also covered in the solution $S_{\mathcal{O}}$, and the resources utilized to cover the job are the same. A job j that is not covered by the resources in $S_{\mathcal{I}}$ pays a penalty p_j in the solution $S_{\mathcal{I}}$; however this job j in $\mathcal{O}$ can be covered by the S-type resource $r(j)$ in the solution $S_{\mathcal{O}}$. Thus, the solution $S_{\mathcal{O}}$ is a feasible solution to the instance $\mathcal{O}$, and has cost equal to the cost of the solution $S_{\mathcal{I}}$.

In the reverse direction, suppose we are given a solution $S_{\mathcal{O}}$ to the instance $\mathcal{O}$. We will convert the solution into a *standard* form, i.e. a solution in which if a S-type resource $r(j)$ (for some job j) is included, then this resource is used to cover job j. Suppose job j is covered by some other resources in the solution $S_{\mathcal{O}}$, while resource $r(j)$ covers some other jobs (call this set J'). We can clearly *exchange* the resources between job j and the set of jobs J' so that job j is covered by resource $r(j)$. So we may assume that the solution $S_{\mathcal{O}}$ is in standard form. But now, given a standard form solution $S_{\mathcal{O}}$, we can easily construct a feasible solution $S_{\mathcal{I}}$ for the PrizeCollectingResAll instance $\mathcal{I}$: if a job j in $S_{\mathcal{O}}$ is covered by the S-type resource $r(j)$, then in $S_{\mathcal{I}}$, this job will not be covered (and a penalty p_j will be accrued); all jobs j in $S_{\mathcal{O}}$ that are covered by M-type resources will be covered by the corresponding resources in $S_{\mathcal{I}}$.

This completes the reduction, and the proof of Theorem 5.14.

Chapter 6

Conclusion and Open Problems

In this thesis, we presented several algorithms for solving the Round-UFP, Max-UFP and Bag-UFP problems on paths and trees. We saw that some special cases of the Round-UFP problem can have much better algorithms. We also showed how an algorithm for Round-UFP can be used to solve the Max-UFP and Bag-UFP problems. The idea of convex decomposition of fractional LP solutions is useful for this. We gave improved constant factor approximation algorithms for all these problems under the *no bottleneck assumption*. We also studied the Online Interval Coloring problem and gave a constant factor competitive algorithm. Finally, we studied the PartialResAll and the PrizeCollectingResAll problems and gave $O(\log(n+m))$-approximation and 4-approximation algorithms for them. There are several areas where there is a scope for improvements. We discuss some of them below.

For Round-UFP on paths, we gave a 3-approximation algorithm for the case of uniform capacities. This algorithm requires $4r$ colors, where r is the maximum congestion. However, we don't know of any example where the optimum coloring requires more than $2r$ colors. Moreover, our greedy algorithm when directly applied (without partitioning into small and large demands) also requires at most $2r$ colors on all examples that we have tried. It will be good to prove that this (or some other algorithm) requires at most $2r$ colors or prove that there is an example which requires more than $2r$ colors.

For arbitrary capacities and demands with NBA, we believe that the 24-approximation algo-

rithm can be improved significantly. Again there is no example where the optimum coloring requires more than $2r$ colors. To improve the constant factor (24), we may need to consider $\frac{1}{2}$-small and $\frac{1}{2}$-large demands. It may also be the case that if we don't divide the demands into these two classes, a much better approximation is possible. But we need some new techniques for doing this.

Improving the $(2+\epsilon)$-approximation for MAX-UFP with NBA is a formidable challenge. If we follow the small and large demands paradigm, to get a 2-approximation we need to have optimal solutions for both these instances, which is not possible for small demands (as it is NP-hard). So, we have to consider the demands together. Here, some new ideas are required to handle them together, as the existing techniques don't work well for these two classes.

For BAG-UFP, improving the 65-approximation should not be very difficult. Again, considering $\frac{1}{2}$-small and $\frac{1}{2}$-large demands can be useful here. Moreover, we are using the approximation algorithm for *throughout maximization for real-time scheduling* as a black box. If we can directly attack the problem, a much better approximation is possible.

For ROUND-UFP and MAX-UFP on trees, if we can use the tree structure more effectively, instead of breaking it into two paths and thereby losing a factor of 2, a better approximation is possible. A possible approach could be to consider the requests based on the depth of the least common ancestor (LCA) of the source and destination of a request.

For the ONLINE INTERVAL COLORING problem on paths with arbitrary capacities and arbitrary demands with NBA, designing an algorithm with a small constant approximation factor would be a significant challenge. The best lower bound for this problem with uniform capacities and arbitrary demands is $\frac{24}{7} \approx 3.43$ by Epstein et al. [29], improving the lower bound of 3 by Kierstead and Trotter for unit capacities and unit demands. Clearly, there is a big gap between the upper and lower bounds which needs to be closed. For trees, closing the gap between the upper bound of $O(\log n)$ and the lower bound of $\Omega\left(\frac{\log n}{\log \log n}\right)$ is a long-standing open problem.

A far more challenging task is to design good approximation algorithms for these problems without NBA. For MAX-UFP on paths, a breakthrough was achieved when a $(7+\epsilon)$-approximation was given by [12]. To do this, they had to introduce new techniques, one of which is a novel geometric dynamic programming algorithm for the maximum weight independent set of rectangles problem.

Since the congestion bound r is very bad without NBA, for ROUND-UFP we need significantly new ideas. A combination of the congestion bound r and clique bound ω may do the job. We may also require a completely new and better lower bound.

For the PARTIALRESALL problem, the main goal is to either come up with a constant factor approximation algorithm, or to show that none exists by establishing a matching lower bound. One way to design the former is to design a constant factor approximation algorithm for the PRIZECOLLECTINGRESALL problem having the *Lagrangian Multiplier Preserving* property. Note that by using the Jain-Vazirani framework, we can immediately obtain a constant factor approximation algorithm for the PARTIALRESALL problem. It is also not clear whether the factors $O(\log n)$ and 4 for the PARTIALRESALL and PRIZECOLLECTINGRESALL problems respectively are the best possible.

Here are some future directions and open questions for these problems.

- Is there a 2-approximation algorithm for ROUND-UFP with uniform capacities?
- Can we improve the approximation factor of ROUND-UFP, MAX-UFP and BAG-UFP problems on paths and trees?
- What is the approximability of these problems without the *no-bottleneck assumption*? For MAX-UFP on paths, a $(7+\epsilon)$-approximation is known.
- Is there a better constant factor competitive algorithm for the ONLINE INTERVAL COLORING problem on paths?
- For the ONLINE INTERVAL COLORING problem on trees, is it possible to close the gap between the upper bound of $O(\log n)$ and the lower bound of $\Omega\left(\frac{\log n}{\log \log n}\right)$?
- Is there a constant factor approximation algorithm for the PARTIALRESALL problem?
- Is there a constant factor approximation algorithm for the PRIZECOLLECTINGRESALL problem having the *Lagrangian Multiplier Preserving* property?
- What is the hardness of approximation of these problems?

Bibliography

[1] Udo Adamy and Thomas Erlebach. Online coloring of intervals with bandwidth. In Klaus Jansen and Roberto Solis-Oba, editors, *WAOA*, volume 2909 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.

[2] Matthew Andrews, Julia Chuzhoy, Sanjeev Khanna, and Lisa Zhang. Hardness of the undirected edge-disjoint paths problem with congestion. In *IEEE Symposium on Foundations of Computer Science*, pages 226–244, 2005.

[3] Yossi Azar, Amos Fiat, Meital Levy, and N. S. Narayanaswamy. An improved algorithm for online coloring of intervals with bandwidth. *Theor. Comput. Sci.*, 363(1):18–27, 2006.

[4] Yossi Azar and Oded Regev. Combinatorial algorithms for the unsplittable flow problem. *Algorithmica*, 44(1):49–66, 2006.

[5] Nikhil Bansal, Zachary Friggstad, Rohit Khandekar, and Mohammad R. Salavatipour. A logarithmic approximation for unsplittable flow on line graphs. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 702–709, 2009.

[6] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. *Journal of the ACM*, 48(5):1069–1090, 2001.

[7] Amotz Bar-Noy, Sudipto Guha, Joseph Naor, and Baruch Schieber. Approximating the throughput of multiple machines under real-time scheduling. In *ACM Symposium on Theory of Computing*, pages 622–631, 1999.

[8] R. Bar-Yehuda. Using homogeneous weights for approximating the partial cover problem. *J. Algorithms*, 39(2):137–144, 2001.

[9] Yair Bartal and Stefano Leonardi. On-line routing in all-optical networks. *Theor. Comput. Sci.*, 221(1-2):19–39, 1999.

[10] Piotr Berman and Bhaskar DasGupta. Improvements in throughout maximization for real-time scheduling. In *ACM Symposium on Theory of Computing*, pages 680–687, 2000.

[11] R. Bhatia, J. Chuzhoy, A. Freund, and J. Naor. Algorithmic aspects of bandwidth trading. *ACM Transactions on Algorithms*, 3(1), 2007.

[12] Paul Bonsma, Jens Schulz, and Andreas Wiese. A constant factor approximation algorithm for unsplittable flow on paths. In *IEEE Symposium on Foundations of Computer Science*, pages 47–56, 2011.

[13] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA, 1998.

[14] V. Chakaravarthy, A. Kumar, S. Roy, and Y. Sabharwal. Resource allocation for covering time varying demands. In *European Symposium on Algorithms*, 2011.

[15] Venkatesan T. Chakaravarthy, Anamitra R. Choudhury, and Yogish Sabharwal. A near-linear time constant factor algorithm for unsplittable flow problem on line with bag constraints. In *Foundations of Software Technology and Theoretical Computer Science*, pages 181–191, 2010.

[16] Venkatesan T. Chakaravarthy, Arindam Pal, Sambuddha Roy, and Yogish Sabharwal. Scheduling resources for executing a partial set of jobs. In *Foundations of Software Technology and Theoretical Computer Science*, 2012.

[17] Venkatesan T. Chakaravarthy, Vinayaka Pandit, Yogish Sabharwal, and Deva P. Seetharam. Varying bandwidth resource allocation problem with bag constraints. In *IPDPS*, pages 1–10, 2010.

[18] Amit Chakrabarti, Chandra Chekuri, Anupam Gupta, and Amit Kumar. Approximation algorithms for the unsplittable flow problem. *Algorithmica*, 47(1):53–78, 2007.

[19] D. Chakrabarty, E. Grant, and J. Könemann. On column-restricted and priority covering integer programs. In *IPCO*, pages 355–368, 2010.

[20] Chandra Chekuri, Alina Ene, and Nitish Korula. Unsplittable flow in paths and trees and column-restricted packing integer programs. In *APPROX-RANDOM*, pages 42–55, 2009.

[21] Chandra Chekuri, Marcelo Mydlarz, and F. Bruce Shepherd. Multicommodity demand flow in a tree and packing integer programs. *ACM Transactions on Algorithms*, 3(3), 2007.

[22] Marek Chrobak and Maciej Slusarek. On some packing problem related to dynamic storage allocation. *ITA*, 22(4):487–499, 1988.

[23] Julia Chuzhoy and Paolo Codenotti. Resource minimization job scheduling. In *APPROX-RANDOM*, pages 70–83, 2009.

[24] Julia Chuzhoy, Sudipto Guha, Sanjeev Khanna, and Joseph Naor. Machine minimization for scheduling jobs with interval constraints. In *IEEE Symposium on Foundations of Computer Science*, pages 81–90, 2004.

[25] Julia Chuzhoy and Joseph Naor. New hardness results for congestion minimization and machine scheduling. *J. ACM*, 53(5):707–721, 2006.

[26] Julia Chuzhoy, Rafail Ostrovsky, and Yuval Rabani. Approximation algorithms for the job interval selection problem and related scheduling problems. *Math. Oper. Res.*, 31(4):730–738, 2006.

[27] Khaled Elbassioni, Naveen Garg, Divya Gupta, Amit Kumar, Vishal Narula, and Arindam Pal. Approximation algorithms for unsplittable flow problems on paths and trees. In *Foundations of Software Technology and Theoretical Computer Science*, 2012.

[28] Leah Epstein, Thomas Erlebach, and Asaf Levin. Online capacitated interval coloring. *SIAM J. Discrete Math.*, 23(2):822–841, 2009.

[29] Leah Epstein and Meital Levy. Online interval coloring with packing constraints. *Theor. Comput. Sci.*, 407(1-3):203–212, 2008.

[30] T. Erlebach. Approximation algorithms and complexity results for path problems in trees of rings. *Mathematical Foundations of Computer Science 2001*, pages 351–362, 2001.

[31] Thomas Erlebach and Frits C. R. Spieksma. Interval selection: Applications, algorithms, and lower bounds. *J. Algorithms*, 46(1):27–53, 2003.

[32] R. Gandhi, S. Khuller, and A. Srinivasan. Approximation algorithms for partial covering problems. *J. Algorithms*, 53(1):55–84, 2004.

[33] N. Garg. Saving an ϵ: a 2-approximation for the k-MST problem in graphs. In *ACM Symposium on Theory of Computing*, 2005.

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

[35] U. I. Gupta, D. T. Lee, and J. Y.-T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467, 1982.

[36] Venkatesan Guruswami, Sanjeev Khanna, Rajmohan Rajaraman, F. Bruce Shepherd, and Mihalis Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. *J. Comput. Syst. Sci.*, 67(3):473–496, 2003.

[37] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-hard problems.* PWS Publishing Co., Boston, MA, USA, 1997.

[38] K. Jain and V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and Lagrangian relaxation. *J. ACM*, 48(2):274–296, 2001.

[39] H.A. Kierstead and W.T. Trotter. An extremal problem in recursive combinatorics. *Congressus Numerantium*, 33:143–153, 1981.

[40] Hal A. Kierstead. The linearity of first-fit coloring of interval graphs. *SIAM J. Discrete Math.*, 1(4):526–530, 1988.

[41] Hal A. Kierstead and Jun Qin. Coloring interval graphs with first-fit. *Discrete Mathematics*, 144(1-3):47–57, 1995.

[42] Jon Kleinberg. *Approximation algorithms for disjoint paths problems*. PhD thesis, Department of EECS, MIT, 1996.

[43] Jon M. Kleinberg and Ronitt Rubinfeld. Short paths in expander graphs. In *37th Annual Symposium on Foundations of Computer Science*, pages 86–95, 1996.

[44] Jon M. Kleinberg and Éva Tardos. Disjoint paths in densely embedded graphs. In *36th Annual Symposium on Foundations of Computer Science*, pages 52–61, 1995.

[45] Petr Kolman and Christian Scheideler. Simple on-line algorithms for the maximum disjoint paths problem. *Algorithmica*, 39(3):209–233, 2004.

[46] Petr Kolman and Christian Scheideler. Improved bounds for the unsplittable flow problem. *J. Algorithms*, 61(1):20–44, 2006.

[47] J. Könemann, O. Parekh, and D. Segev. A unified approach to approximating partial covering problems. *Algorithmica*, 59(4), 2011.

[48] Amit Kumar, Arindam Pal, Prashant Sachan, and Saurav Singh. Online algorithms for interval coloring problems. Manuscript, 2012.

[49] N. S. Narayanaswamy. Dynamic storage allocation and on-line colouring interval graphs. In Kyung-Yong Chwa and J. Ian Munro, editors, *COCOON*, volume 3106 of *Lecture Notes in Computer Science*, pages 329–338. Springer, 2004.

[50] Christos Nomikos, Aris Pagourtzis, and Stathis Zachos. Routing and path multicoloring. *Inf. Process. Lett.*, 80(5):249–256, 2001.

[51] Sriram V. Pemmaraju, Rajiv Raman, and Kasturi R. Varadarajan. Max-coloring and online coloring with bandwidths on interval graphs. *ACM Transactions on Algorithms*, 7(3):35, 2011.

[52] Prabhakar Raghavan and Clark D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[53] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[54] P. J Wan and L. Liu. Maximal throughput in wavelength-routed optical networks. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1998.

[55] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 2011.

Biography of the Author

Arindam Pal completed Bachelor of Engineering from Jadavpur University, Kolkata in 2000 and Master of Engineering from Indian Institute of Science, Bangalore in 2002, both in Computer Science and Engineering. He worked as a Software Engineer in Microsoft and Yahoo! from February 2002 to July 2007. From August 2007 to November 2012, he worked for his Ph.D. degree at the Department of Computer Science and Engineering, IIT Delhi. He is currently working as a Research Scientist at TCS Innovation Labs Kolkata. His research areas are approximation algorithms, combinatorial optimization, graph theory and machine learning.