This thesis was submitted in partial fulfillment of the requirements for the doctor of philosophy degree (Ph.D.) at Tel-Aviv University

Submitted by
Oded Regev

This work was supervised by Prof. Yossi Azar.

January 2001
Acknowledgments

I am indebted to Yossi Azar, my thesis advisor, without whom this thesis would not exist. His great insights guided me through the nights I spent solving the problems presented in this thesis. Returning home late, I always found my parents, Gadi and Zehavit, and my brother, Amit. I thank them and all my family for supporting me and helping me through the times when things seemed a little too difficult.
Credits

The results in this thesis were obtained jointly with my advisor, Yossi Azar. I am grateful to my co-authors for allowing me to include in this thesis results obtained in joint work with them. These include the results in Chapter 2 which are a joint work with Jiří Sgall and Gerhard J. Woeginger and appear in [16]. The results in Chapter 3 were obtained jointly with Leah Epstein and appear as part of [4]. The results in Chapter 4 appear in [7] and were obtained jointly with Baruch Awerbuch and Stefano Leonardi. The results in Chapter 5 are joint work with Baruch Awerbuch and appear in [9]. Finally, the results in Chapter 6 can be found in [15].
Abstract

In this thesis, we consider scheduling and load balancing problems. In scheduling problems we are given a set of jobs to be assigned to free time slots on one of several processors. In these problems the time axis is the only axis that exists. In load balancing problems on the other hand, each job arrives with its active time period and we have to choose a machine to which we assign it. So in these problems, in addition to the time axis, there exists a load axis which we wish to balance. The goal in both problems can be either to minimize a cost function or to maximize a benefit function. Providing optimal solutions for these problems is usually intractable, hence we are considering approximated solutions for both off-line and on-line aspects of the problems.

A short summary of problems and results in this thesis follows. The first two problems are load balancing problems, the next two are scheduling problems and the last one is a routing problem.

Temporary Task Assignment on Identical Machines

We consider the off-line temporary task assignment problem. Jobs, in addition to their weight, have an arrival and a departure time. The goal is to assign the jobs such that the maximum load over both machines and time is minimized. We show that no polynomial time algorithm can achieve an approximation ratio below 1.5 for this problem. However, for the case where the number of machines is fixed, we present a PTAS.

Online Load Balancing with Unrelated Machines

Here we consider on-line load balancing of temporary tasks on unrelated machines. We prove an inapproximability result for the problem and show that a trivial algorithm almost achieves the best competitive ratio possible. In the special case of the related-restricted machines model we show tight results on the competitive ratio for a whole range of speeds. Our results apply to randomized algorithms as well.

Multiprocessor Scheduling without Migration

For the off-line problem of scheduling jobs in a multiprocessor setting in order to minimize the flow time, the $SRPT$ algorithm is known to perform within a logarithmic factor of the optimal schedule. This algorithm both preempts jobs and migrates jobs between machines. Unlike preemption, migration is not known to be necessary for achieving these
low approximation ratios. We show how one can achieve the same approximation ratio without migrating jobs. This result also applies to the on-line setting where the algorithm achieves the best competitive ratio possible.

**Benefit Maximization for Online Scheduling**

In this on-line scheduling problem jobs arrive over time. Our goal is to maximize the total benefit gained from the scheduling of the jobs. A common model is to give each job its own deadline and to take into account only jobs completed by their deadline. The relatively high competitive ratios encountered in this model motivate the search for more reasonable measures of benefit. We consider a model where the benefit gained from a job is a function of its processing time: the longer a job is delayed the lower the benefit gained. A constant competitive algorithm is shown for this model.

**The Unsplittable Flow Problem**

The Unsplittable Flow Problem (UFP) is a routing problem where we are given a capacitated graph and a set of connection requests with individual demands and profits. The objective is to route a subset of the requests in order to maximize the total profit of the routed requests. The routing must obey edge capacities and use single flow paths. We present algorithms for several variants of the problem. We identify the three main cases of the problem and either improve or match the previously known approximation ratios for all three. However, unlike previous algorithms, all of our algorithms are both strongly polynomial and combinatorial. While the results above apply to the off-line setting, we also present several results for the on-line setting.
Contents

1 Introduction ........................................... 11
   1.1 Load Balancing Problems ............................. 12
   1.2 Scheduling Problems ................................. 14
   1.3 Routing Problems ..................................... 16
   1.4 Quality of Approximation ............................. 17
   1.5 Organization of the Thesis ............................ 17

2 Temporary Task Assignment on Identical Machines .......... 19
   2.1 Introduction ......................................... 19
   2.2 Preliminaries ........................................ 20
   2.3 The Polynomial Time Approximation Scheme .......... 20
   2.4 Analysis .............................................. 23
   2.5 The Unrestricted Number of Machines Case .......... 25
   2.6 Discussion ............................................ 28

3 Online Load Balancing with Unrelated Machines .......... 29
   3.1 Introduction ......................................... 29
   3.2 Inapproximability of the Unrelated Machines Model 30
   3.3 Tight Results for the Related-Restricted Machines Model 32
   3.4 Discussion ............................................ 35
4 Multiprocessor Scheduling without Migration 37
   4.1 Introduction ............................................. 37
   4.2 Preliminaries ........................................... 38
   4.3 The Algorithm ........................................... 38
   4.4 Analysis .................................................. 39
   4.5 Tightness of the Analysis .............................. 49
   4.6 Discussion .............................................. 50

5 Benefit Maximization for Online Scheduling 51
   5.1 Introduction ............................................. 51
   5.2 Preliminaries ........................................... 51
   5.3 The Algorithm ........................................... 52
   5.4 The Analysis ............................................. 53
   5.5 Multiprocessor Scheduling .............................. 56
   5.6 Discussion .............................................. 58

6 The Unsplittable Flow Problem 59
   6.1 Introduction ............................................. 59
   6.2 Preliminaries ........................................... 60
   6.3 Algorithms for \textit{UFP} ................................. 61
      6.3.1 Algorithm for Classical \textit{UFP} .................... 61
      6.3.2 Strongly Polynomial Algorithm ..................... 65
      6.3.3 Algorithm for Extended \textit{UFP} .................... 66
   6.4 Algorithms for \textit{K}-bounded \textit{UFP} .................. 66
      6.4.1 Algorithms for Bounded Demands .................... 67
      6.4.2 A Combined Algorithm ............................... 69
   6.5 Lower Bounds ........................................... 70
   6.6 Online Applications .................................... 72
CONTENTS

6.6.1 Online Algorithms .............................................. 72
6.6.2 Online Lower Bound ............................................. 73
6.7 Discussion ......................................................... 74
Chapter 1

Introduction

In this thesis we consider resource allocation problems. In these problems a set of requests has to be assigned to a set of resources, be it machines, network links, or time periods. The assignment is measured by using a cost function which we try to minimize or by using a benefit function which we try to maximize. An exact solution is a desirable result but, despite the deceptive simplicity of the problems, achieving this optimal solution is often impossible, be it due to limited time or incomplete information. We therefore resort to algorithms whose solution is only an approximation of the optimal solution. Finding such algorithms whose solutions are good approximations of the optimal solution is the main focus of this thesis.

Interest in scheduling and load balancing problems can be traced back to the fifties. In a paper from 1966, Graham studied some of the first algorithms for scheduling problems. The problem considered by Graham is perhaps the simplest of all and since then more and more important cases have been identified. In this thesis we present some recent results in this field.

Most of these problems exist in two flavors: the off-line setting and the on-line setting. In the off-line setting the whole input is given in advance and the algorithm has to find an approximate solution. In the on-line setting the input is given in small parts, usually one request after the other. An on-line algorithm has to handle each request with no knowledge about future requests. Off-line algorithms are usually more powerful and are limited only by their running time. The main limitation of on-line algorithms, however, is their uncertainty of future events and not their running time. Nevertheless, the on-line algorithms presented in this thesis are all polynomial time algorithms. In this thesis we consider both the off-line setting and the on-line setting of each problem.

For us to be able to compare different approximation algorithms a measure of the quality of approximation has to be defined. This measure is known as the approximation ratio for off-line algorithms and the competitive ratio for on-line algorithms. In both cases we basically compare the value of a certain goal function achieved by a given algorithm to that of an optimal solution. The maximum ratio ever achieved by an algorithm is known as its
competitive ratio for on-line algorithms or as its approximation ratio for off-line algorithms.

The two main resource allocation problems found in this thesis are the load balancing problem and the scheduling problem. Both problems are concerned with allocating jobs to a set of machines. In the load balancing problem each job has an arrival time, a departure time and a weight and an algorithm should decide to which machine each job is assigned. The load on a machine at a certain time is the sum of weights of jobs assigned to it. A load balancing algorithm tries to minimize the cost, defined as the maximum machine load. We also consider a routing problem, that is, a generalization of a load balancing problem where instead of machines, jobs are assigned to paths in a network.

In scheduling problems, on the other hand, no notion of load exists since each machine can process at most one job at a time. Instead, each job has an arrival time and a processing time and the algorithm has to find for each job a time period in which to run it and a free machine to process it during that time period. The time period should not be before the job’s arrival time and its length should match the job’s processing time. Several goal functions exist but the simplest one is perhaps the last completion time of a job.

In what follows we describe in more detail several aspects of the problems considered so far. We begin with describing load balancing problems and scheduling problems. We continue with describing a routing problem in which network load has to be balanced instead of machine load. The exact definition regarding the quality of approximation in both off-line and on-line algorithms is then given. The chapter is completed by a summary of the results in this thesis.

1.1 Load Balancing Problems

In the load balancing problem a set of jobs has to be assigned to a set of machines. Each job is given with a load vector indicating its weight on each machine, an arrival time and a departure time. Assigning a job to a machine increases that machine’s load between the arrival and the departure time by the appropriate coordinate in the load vector. Different goals exist, but the most common one is minimizing maximum load ever achieved by a machine in an assignment and a low maximum load is usually a good indicator of a uniform and balanced assignment. Notice that online algorithms learn about events as they happen and, in particular, the departure of jobs in known only at their departure time.

More formally, we are given a set $J$ of $n$ jobs to be assigned to $m$ machines. Each job $j$, $1 \leq j \leq n$, is associated with a load vector $(w_j(1), \ldots, w_j(m))$, an arrival time $a_j$ and a departure time $d_j$. If job $j$ is assigned to machine $i$, it increases the load of machine $i$ by its weight on that machine, $w_j(i)$ from time $a_j$ to time $d_j$. The cost function which we try minimize is the maximum load, defined as $\max_{i \in J} \max_{t} \sum_{j \in J, a_j \leq t < d_j} w_j(i)$ is the load on machine $i$ at time $t$ and $J_i$ denotes the set of jobs assigned to machine $i$. Many papers considered load balancing problems in the past and especially in the last decade (e.g., [2, 6, 10, 12, 18, 23, 27, 35, 41, 52, 60]).
1.1. LOAD BALANCING PROBLEMS

The load balancing problem described above is known as load balancing of temporary tasks with unrelated machines. An important special case is the permanent jobs case where all the jobs have the same departure time. Being a special case of the temporary tasks model, it is often easier to approximate.

In either the permanent or the temporary tasks case one often examines a more restricted machine model in which the load vector has to be of a certain type. In addition to the one described above, three such models exist in all of which the load vector is replaced by a value $w_j$ indicating a job’s weight. In the restricted machines model, each job is equipped with an additional set of admissible machines to which its assignment is allowed. Alternatively, this model can be thought of as load vectors which are allowed to contain only two values, infinity and the job’s weight. Another natural model is the related machines model in which each machine has its own speed and the load caused by assigning a job to a machine is the job’s weight divided by the machine’s speed. The oldest and simplest of all models is the identical machines model in which assigning a job to a machine increases its load by the job’s weight. This model is a special case of all the previous machine models considered.

We begin a survey of results for the on-line model with the model of permanent identical machines which was one of the first to be considered. Graham presented in [36] an on-line algorithm called “List Scheduling” that achieves a competitive ratio of $2 - \frac{1}{m}$, where $m$ is the number of machines. For cases of only two or three machines, this algorithm turns out to be optimal [29]. Improved algorithms for the general case appeared in [2, 26, 32, 18] and the current best upper bound is around $1.92$ [30]. A series of lower bounds appeared [2, 19, 29] with the best lower bound being around $1.853$ [34]. A constant competitive ratio is also achievable for the related machines model [5, 23]. For the restricted machines model the exact competitive ratio is $\Theta(\log m)$, both for deterministic and randomized algorithms [14]. A deterministic algorithm for the unrelated machines model exists with the same $O(\log m)$ competitive ratio [5] and is therefore tight.

For load balancing of temporary tasks on identical machines, Graham’s algorithm turns out to be optimal [12]. A constant competitive algorithm exists for the related machines case with temporary tasks as well [13]. The competitive ratio in the restricted temporary tasks is $\Theta(\sqrt{m})$ [11, 53]. The last model left to consider is the load balancing of temporary tasks on unrelated machines. This is the only online load balancing model whose asymptotic competitive ratio was not known. In this thesis we present a lower bound of $\Omega(m / \log m)$ that almost matches an $O(m)$ upper bound. This essentially settles an open problem presented in [24].

In the off-line setting, load balancing of permanent tasks was shown to have a polynomial time approximation scheme (that is, a polynomial time $(1 + \epsilon)$-approximation algorithm for any fixed $\epsilon > 0$) for any fixed number of machines in [57]. This was later improved to an arbitrary number of machines by Hochbaum and Shmoys [39]. For load balancing of permanent jobs on unrelated machines, Lenstra et al. [50] showed a polynomial time approximation scheme for a fixed number of machines. On the other hand, they showed that if the number of machines is part of the input then no algorithm with an approximation ratio better than $\frac{3}{2}$ can exist unless $P = NP$. 
For off-line load balancing of temporary tasks, the “List Scheduling” algorithm was the best algorithm known. In this thesis we present a polynomial time approximation scheme for the case in which the number of machines is fixed. For the case in which the number of machines is given as part of the input, we show that no algorithm can achieve a better approximation ratio than $\frac{3}{2}$ unless $P = NP$. A recent result considers the case of temporary tasks with unrelated machines [5]. That paper presents a polynomial time approximation scheme for a constant number of machines and a lower bound of 2 for the general case.

1.2 Scheduling Problems

The second problem we consider is the scheduling problem. Here, each job has an arrival time and a processing time and should be assigned to one of the machines. Only one job can be assigned to a machine at any given time but jobs can be postponed and started after their arrival time. Scheduling problems are considered in the uniprocessor case where there is only one machine or in the multiprocessor case where more than one machine exists. The uniprocessor case turns out to be an important case in many scheduling problems. In load balancing on the other hand, problems become interesting only when two or more machines are involved. On-line algorithms learn about a job and its processing time only at its arrival time. An on-line algorithm has no information about a job before its arrival time. For a survey of on-line scheduling see [58].

Since a machine is fully occupied with processing just one job we may allow jobs to be preempted. Without preemption, an algorithm has to schedule jobs by using a continuous time period on a single machine. With preemption, an algorithm might stop processing a job and continue later from where it stopped. We say that an algorithm does not use migration when preempted jobs run on the same machine on which their processing began. Thus, the use of preemption is allowed but is limited to a single machine. This property is regarded as an important one since migrating a job between machines in a multiprocessor setting is often far from trivial and results in a considerable overhead.

Schedules are measured by one of several different ways. A schedule might be measured by the last completion time or the average completion time. These two measures received considerable attention in the past. Both give a rough measure of the quality of the schedule as a whole but do not represent the quality of service as seen by the individual jobs. A better way to measure the quality of a schedule is to consider the average flow time. The flow time of a job is defined as the difference between its completion time and its arrival time. The average flow is a better measure since it depends on the arrival times as well as the completion times. It is also one of the most difficult to approximate. A different model based on flow times is that of maximizing the benefit given by jobs. Here, each job has a benefit function that depends on its flow time. The longer its processing takes, the lower its benefit is. In this thesis we consider both the minimization and the maximization flow problems described above. One important difference to notice between the two models is that in the maximization problem the effect of a job with an exceedingly long flow time is usually just a slight decrease in the total benefit. In the minimization problem, on the
other hand, the long flow time greatly increases the average flow time.

In the non-preemptive case it is impossible to achieve any “reasonable” approximation for the average flow time. Specifically, even in the uniprocessor case one cannot achieve an approximation ratio of $O(n^{1/4-\varepsilon})$ unless $NP = P$ where $n$ is the number of jobs [43]. For the multiprocessor case it is impossible to achieve $O(n^{1/4-\varepsilon})$ approximation ratio unless $NP = P$ [51]. Thus, preemptions really seem to be essential. Minimizing the flow time on one machine with preemption can be done optimally in polynomial time using the natural algorithm shortest remaining processing time (SRPT) [17]. For more than one machine, the preemptive problem becomes $NP$-hard [28]. Leonardi and Raz [51] showed that SRPT achieves a logarithmic approximation ratio for the multiprocessor case by proving a tight bound of $O(\log(\min\{n/m, P\}))$ on $m > 1$ machines with $n$ jobs, where $P$ denotes the ratio between the processing time of the longest and the shortest jobs. In this thesis, we present an alternative algorithm whose approximation ratio is almost identical to that of SRPT but, unlike SRPT, does not use migration. While these algorithms are the best off-line algorithms known, in the on-line setting they are actually optimal, that is, their competitive ratio cannot be improved.

For the easier cost model of minimizing the average completion time, on-line algorithms with constant competitive ratios exist, e.g. [38]. By using off-line algorithms, one can achieve a PTAS [1]. The cost model of minimizing the total completion time is equivalent since the ratio between the two is constant for any given instance. These models will not be addressed in this thesis. Instead, we will consider the flow time model that better captures the quality of a schedule.

The prominent benefit model considered before this thesis was the firm deadline model. In this model each job has its own constant benefit and its own threshold. The benefit function is this constant for flow times of at most the threshold and zero otherwise. The optimal on-line deterministic competitive ratio for the uniprocessor case is $\Theta(\Phi)$, where $\Phi$ is the ratio between the maximum and minimum benefit density, defined as the benefit divided by the processing time [20, 21, 48]. For the special case where $\Phi = 1$, there is a 4-competitive algorithm. The optimal randomized competitive ratio for the uniprocessor case is $O(\log(\min\{\Phi, \Delta\}))$, where $\Delta$ is the ratio between the processing time of the longest and the shortest job [40]. For the multiprocessor case, Koren and Shashua [49] showed that when the number of machines is very large, a $O(\log \Phi)$ competitive algorithm is possible. That result is shown to be optimal and moreover, their algorithm does not use migration. It can be seen that as a result of the firm deadline, this real-time scheduling model is hard to approximate. Moreover, in many real cases, jobs that are delayed by some small constant should reduce our estimate of the overall system performance, but only by some small factor. In the firm deadline model, jobs that are delayed beyond their deadline cease to contribute to the total benefit. Thus, the property we are looking for is the possibility of delaying jobs without drastically harming the overall system performance. In this thesis we study the effects of different benefit functions on the competitive ratio.

A scheduling problem is formally described as a sequence of $n$ jobs to be assigned to one or more machines. Each job $j$ has a release time $r_j$ and a processing time $w_j$. The
on-line scheduler learns about a job only at its release time. The off-line scheduler knows the whole sequence of jobs in advance. Both schedulers may schedule the job at any time after its release time. The system allows preemption, that is, the scheduler may stop a job and later continue running it. Migration however, is not allowed and a preempted job must later continue running on the same machine. Note that a machine can process only one job at a given time. If job $j$ is completed at time $c_j$ then we define its flow time as $f_j = c_j - r_j$ (which is at least $w_j$).

1.3 Routing Problems

Closely related to load balancing problems is the routing problem where an algorithm tries to balance the load on network links instead of machines. In this problem, known as UFP or unsplittable flow problem, each network link, or edge in the network graph, has a limited capacity that limits the amount of load that can be routed through it. The requests are given as source-sink pairs and each request has its own benefit and demand. Requests should be routed through the network using a single flow path such that the total demand routed through an edge is at most its capacity. Since routing all requests is usually impossible, we try to maximize the total benefit from routed requests.

In this thesis we consider the three variants of the UFP problem. Classical UFP is the case where demands are lower than the minimum edge capacity. The extended UFP case is when some demands might be higher than some edge capacities and represents the most general UFP model. A more restricted case than the classical case is that of bounded UFP (or $K$-bounded UFP) where demands are at most $\frac{1}{K}$ of the minimum edge capacity. The well-known problem of maximum edge disjoint path, denoted EDP, is the special case where all demands, profits and capacities are equal to 1 and is one of the first to be considered (see [37]).

The EDP (and hence the UFP) is one of Karp’s original NP-complete problems [42]. An $O(\sqrt{m})$ approximation algorithm is known for EDP [44] where $m$ denotes the number of edges in the network graph (for additional positive results see [55, 56]). Most of the results for UFP deal with the classical case where the maximal demand is at most the minimal capacity. The most popular approach seems to be LP rounding [22, 47, 59] with the best approximation ratio being $O(\sqrt{m})$ [22]. A matching lower bound of $\Omega(m^{1/2-\varepsilon})$ for any fixed $\varepsilon > 0$ is shown in [37] for directed graphs. In this thesis we provide a strongly polynomial $O(\sqrt{m})$ approximation algorithm.

Due to the complexity of the extended UFP, not many results addressed it. The first to attack the problem is a recent attempt by Guruswami et al. [37]. We improve in this thesis the previous upper bound and, more importantly, show an almost matching hardness result. This hardness result separates the extended UFP and the classical UFP.

We now consider the bounded UFP case where the maximum demand is at most $\frac{1}{K}$ the minimum edge capacity (denoted $K$-bounded UFP). It is a special case of classical
1.4. Quality of Approximation

Since most of the algorithms we present in this thesis are approximation algorithms some notion of the quality of an approximation has to be defined. Let $C_{OPT}(I)$ denote the cost of an optimal solution for the instance $I$ and $C_A(I)$ denote the cost of the approximate solution given by an algorithm $A$ for $I$. The exact cost function depends on the problem considered. The following ratio is known as the approximation ratio of an off-line algorithm $A$ or as the competitive ratio of an on-line algorithm $A$:

$$\max_I \frac{C_A(I)}{C_{OPT}(I)}.$$

For maximization problem, we denote by $B_{OPT}(I)$ the benefit of an optimal solution and by $B_A(I)$ the benefit of an approximation algorithm $A$. The definitions of approximation ratio and competitive ratio are then defined as

$$\max_I \frac{B_{OPT}(I)}{B_A(I)}.$$

We should also note that one often encounters a slightly different definition where an extra additive constant is taken into account. That is however unneeded since all the problems considered in this thesis have arbitrary units and are scalable. When we consider randomized algorithms, the cost (or benefit) of an algorithm is taken as its expected cost (or expected benefit).

Approximation ratios of off-line algorithms might be polynomial, logarithmic or even constant. As it turns out, for some off-line problems, achieving approximation ratios arbitrarily close to 1 is possible. Then we say that we have a PTAS, or a polynomial time approximation scheme, a scheme that provides $1+\epsilon$ approximation algorithms for any $\epsilon > 0$. When the running time of the algorithm is polynomial in both the input size and $1/\epsilon$ we say that the scheme is an FPTAS, or a fully polynomial time approximation scheme.

1.5 Organization of the Thesis

In the following two chapters, we consider two load balancing problems. In Chapter 2 we consider the off-line temporary task assignment problem. In this model, jobs both arrive
and depart. The goal is to assign the jobs such that the maximum load over both machines and time is minimized. We show that no algorithm can achieve an approximation ratio below 1.5 for this problem. However, for the case where the number of machines is fixed, we present a PTAS. These results appear in [16].

In Chapter 3 we consider on-line load balancing of temporary tasks on unrelated machines. We prove an inapproximability result for the problem and show that a trivial algorithm is almost optimal. In the special case of related-restricted machines model we show tight results on the competitive ratio for a whole range of speeds. We apply our results to randomized algorithms as well. The above results appear as part of [4].

In the next two chapters we address preemptive scheduling problems. In Chapter 4, the off-line problem of scheduling jobs in a multiprocessor setting in order to minimize the average flow time is considered. The only algorithm known for this problem, the SRPT algorithm, is known to perform within a logarithmic factor of the optimal schedule. This algorithm both preempts jobs and migrates jobs between machines. Our algorithm, with its $O(\min\{\log P, \log n\})$ approximation ratio, is almost as good as the SRPT algorithm but does not use migration. Our result also applies to the on-line setting where the algorithm achieves the best competitive ratio possible. The results in this chapter appear in [7].

In Chapter 5 we consider an on-line benefit model where the goal is to maximize the total benefit gained from jobs. A common model is to give each job its own deadline and to take into account only jobs completed by their deadline. In Section 1.2 we have seen that in this model, competitive ratios are relatively high. This motivated us to consider a model where the benefit gained from a job is a function of its processing time: the longer a job is processed the lower the benefit gained. A constant competitive algorithm is shown for this model, both for the uniprocessor and the multiprocessor setting. The results in this chapter can also be found in [9].

The last chapter deals with the Unsplittable Flow Problem (UFP). We identify the three main cases of the problem and either improve or match the previously known approximation ratios for all three. However, unlike previous algorithms, all our algorithms are both strongly polynomial and combinatorial. Specifically, for the classical UFP, we provide a strongly polynomial $O(\sqrt{m})$ approximation algorithm. For the hardest of all three variants, the extended UFP, we improve the previous upper bound and, more importantly, we show an almost matching hardness result. This hardness result separates the extended UFP and the classical UFP. For the bounded UFP we improve previous algorithms and show an $O(K \cdot n^{1/K})$ approximation algorithm. While the results above apply to the off-line setting, we also present several results for the on-line setting. The results in this chapter appear in [15].
Chapter 2

Temporary Task Assignment on Identical Machines

2.1 Introduction

In this chapter we consider the off-line problem of load balancing of temporary tasks on \( m \) identical machines. Recall that in this model each job has an arrival time, departure time and some weight. Each job should be assigned to one machine. The load on a machine at a certain time is the sum of the weights of jobs assigned to it at that time. The goal is to minimize the maximum load over machines and time.

This load balancing problem naturally arises in many applications involving allocation of resources. As a simple concrete example, consider the case where each machine represents a communication channel with bounded bandwidth. The problem is to assign a set of requests for bandwidth, each with a specific time interval, to the channels. The utilization of a channel at a specific time \( t \) is the total bandwidth of the requests, whose time interval contains \( t \), which are assigned to this channel.

In this chapter we show that when the number of machines is fixed a \( PTAS \) exists. However, for the case in which the number of machines is given as part of the input, we show that no algorithm can achieve a better approximation ratio than \( \frac{3}{2} \) unless \( P = NP \). Graham’s algorithm is the best approximation algorithm for this case with an approximation ratio of \( 2 - \frac{1}{m} \). Also, recall that the on-line problem is already solved, since Graham’s algorithm is optimal [12].

Note that similar phenomena occur at other scheduling problems. For example, in the introduction of this thesis, we considered the load balancing problem on unrelated machines. For a fixed number of machines Lenstra et al. [50] showed a \( PTAS \) and if the number of machines is part of the input they showed that no algorithm with an approximation ratio better than \( \frac{3}{2} \) can exist unless \( P = NP \).
Our algorithm works in four phases: the rounding phase, the combining phase, the solving phase and the converting phase. The rounding phase actually consists of two subphases. In the first subphase the jobs’ active time is extended: some jobs arrive earlier, others depart later. In the second subphase, the active time is again extended but each job is extended in the opposite direction to which it was extended in the first subphase. In the combining phase, we combine several jobs with the same arrival and departure time and unite them into jobs with higher weights. Solving the resulting assignment problem in the solving phase is easier and its solution can be converted into a solution for the original problem in the converting phase.

The novelty of our algorithm is in the rounding phase. Standard rounding techniques are usually performed on the weights. If one applies similar techniques to the time the resulting algorithm’s running time is not polynomial. Thus, we designed a new rounding technique in order to overcome this problem.

2.2 Preliminaries

In this chapter, we denote by \( \sigma = \sigma_1, ..., \sigma_{2n} \) the sequence of events where each event is an arrival or a departure of a job. Without loss of generality, we assume that at each time only one job arrives or departs. Therefore, we can view \( \sigma \) as a sequence of distinct times, the time \( \sigma_i \) is the moment after the \( i^{th} \) event happened. In addition, \( \sigma_0 \) denotes the moment at the beginning, before the arrival of any job. We denote the arrival time of job \( j \) by \( a_j \) and its departure time by \( d_j \). We say that a job is active at time \( \tau \) if \( a_j \leq \tau < d_j \). Let \( Q_j = \{ j \mid a_j \leq \sigma_i < d_j \} \) be the active jobs at time \( \sigma_i \). For a given algorithm \( A \) let \( A_j \) be the machine on which job \( j \) is assigned. Let

\[
l_k^A(\tau) = \sum_{\{j \mid A_j = k, j \in Q_k\}} w_j
\]

be the load on machine \( k \) at time \( \sigma_i \), which is the sum of weights of all jobs assigned to \( k \) and active at this time. The cost of an algorithm \( A \) is the maximum load ever achieved by any of the machines, i.e., \( C_A = \max_i, k l_k^A(\tau) \).

2.3 The Polynomial Time Approximation Scheme

Assume without loss of generality that the optimal maximum load is in the range \((1, 2]\). That is possible since Graham’s algorithm can approximate the optimal solution up to a factor of 2, and thus, we can scale all the jobs’ weights by \( 2/l \) where \( l \) denotes the value of Graham’s solution. This does not increase the running time of the scheme, since Graham’s algorithm runs in linear time (for fixed \( m \)).

Let \( \epsilon > 0 \) be a constant, depending on the required precision (we will determine it later). We fix three constants, \( \alpha = \epsilon/\log n \), \( \beta = \alpha \epsilon / m = \epsilon^2 / (m \log n) \), and \( \gamma = \beta \epsilon / m = \epsilon^3 / (m^2 \log n) \).
2.3. The Polynomial Time Approximation Scheme

\[
[\log n] \left\{ \begin{array}{ccccccc}
J_4 & c_4 & J_5 & \vdots & J_6 & c_6 & J_7 & c_7 \\
J_2 & c_2 & J_3 & c_3 & J_7 & c_7 & J_1 & c_1 \\
\end{array} \right. 
\]

Figure 2.1: Partitioning \( J - R \) into \( \{ J_i \} \)

In order to describe the rounding phase with its two subphases we begin with defining the partitions based on which the rounding will be performed. The set \( R \) contains all jobs with weight at least \( \gamma \). We begin by defining a partition \( \{ J_i \} \) of the set of jobs \( J - R \). We set \( M_1 = J - R \) and define sets \( J_i \) and \( M_i \) iteratively as follows. Let \( M_i \) be a set of jobs and consider the sequence of times \( \sigma \) in which jobs of \( M_i \) arrive and depart. The number of such times is \( 2r \) for some \( r \), let \( c_i \) be any time between the \( r \)-th and the \( r + 1 \)-st elements in that set. The set \( J_i \) contains the jobs in \( M_i \) that are active at time \( c_i \). The set \( M_{2i} \) contains the jobs in \( M_i \) that depart before \( \alpha \) at \( c_i \) and the set \( M_{2i+1} \) contains the jobs in \( M_i \) that arrive after \( c_i \). We stop when all unexpanded \( M_i \)'s are empty. The important property of that partition is that the set of jobs that are active at any particular time is partitioned into at most \( \lceil \log n \rceil \) different sets \( J_i \).

\[
\alpha \ldots \alpha + \gamma \left\{ \begin{array}{c}
S_i^1 \\
S_i^2 \\
\vdots \\
S_i^{\alpha + \gamma} \\
\end{array} \right. \]

\[
\alpha \ldots \alpha + \gamma \left\{ \begin{array}{c}
T_i^1 \\
T_i^2 \\
\vdots \\
T_i^{\alpha + \gamma} \\
\end{array} \right. 
\]

Figure 2.2: Partitioning \( J_i \) into \( \{ S_i^j, T_i^j \} \)

We continue by further partitioning the set \( J_i \). We order the jobs according to their arrival time. We denote the smallest prefix of the jobs whose total weight is at least \( \alpha \) by \( S_i^1 \). We order the same set of jobs according to their departure time. We take the smallest suffix whose weight is at least \( \alpha \) and denote that set by \( T_i^1 \). Note that there might be jobs that are both in \( S_i^1 \) and \( T_i^1 \). We remove the jobs in \( S_i^1 \cup T_i^1 \) from \( J_i \), repeat the process with the jobs left in \( J_i \) and similarly define \( S_i^2, T_i^2, \ldots, S_i^{\alpha + \gamma}, T_i^{\alpha + \gamma} \). Each set \( S_i \) and \( T_i \) has total weight between \( \alpha \) and \( \alpha + \gamma \), except for the last pair which may have smaller weight than \( \alpha \). However, if the last pair has smaller weight than \( \alpha \) then it satisfies \( S_i^{\beta_i} = T_i^{\beta_i} \). We denote by \( s_i^j \) the arrival time of the first job in \( S_i^j \) and by \( t_i^j \) the departure time of the last job in \( T_i^j \). Note that \( s_i^1 \leq s_i^2 \leq \ldots \leq s_i^{k_i} \leq c_i \leq t_i^{k_i} \leq \ldots \leq t_i^2 \leq t_i^1 \).

The first subphase of the rounding phase creates a new set of jobs \( J' \) which contains the same jobs as in \( J \) with slightly longer active times. We change the arrival time of all the jobs in \( S_i^j \) for \( j = 1, \ldots, k_i \) to \( s_i^j \). Also, we change the departure time of all the jobs in \( T_i^j \) to
The jobs in $R$ are left unchanged. We denote the sets resulting from the first subphase by $J^i$, $J'^i$, $S'^i$, $T'^i$.

![Diagram](image1)

Figure 2.3: The set $J'$ (after the first subphase)

The second subphase of the rounding phase further extends the active time of the jobs resulting from the first subphase. We take one of the sets $J_i^j$ and the partition to $S'^i \cup T'^i$, $S'^i \cup T'^2$, \ldots, $S'^i \cup T'^k$, we defined earlier. For every $j \leq k$, we order the jobs in $S'^i$ according to an increasing order of departure times. We take the smallest prefix of this ordering whose total weight is at least $\beta$. We extend the departure time of all the jobs in that prefix to the departure time of the last job in that prefix. The process is repeated until there are no more jobs in $S'^i$. The last prefix may have a weight of less than $\beta$. Similarly, extend the arrival times of jobs in $T'^i$. Note that if the weight of the last pair is smaller than $\alpha$ then $S'^i = T'^i$ and these jobs are left unchanged since they already have identical arrival and departure times from the first phase. We denote the sets resulting from the second subphase by $J''$, $J''_i$, $S''_i$, $T''_i$.

![Diagram](image2)

Figure 2.4: The set $J''_i$ (after the rounding phase)

The combining phase of the algorithm involves the weight of the jobs. Let $J''_{st}$ be the set of jobs in $J''$ that arrive at $s$ and depart at $t$. Assume the total weight of jobs whose weight is at most $\gamma$ in $J''_{st}$ is $x$. The combining phase replaces these jobs by $\lceil x / \gamma \rceil$ jobs of weight $\gamma$. We denote the resulting sets by $J''_{st}$. The set $J''$ is created by replacing every $J''_{st}$ with its corresponding $J''_{st}$, that is, $J'' = \bigcup_{s,t} J''_{st}$.

The solving phase of the algorithm solves the modified decision problem of $J''$ by building a layered graph. Note that $J''$ contains both the modified jobs from $J - R$ and the original jobs from $R$. Every time $i$, $i = 0, \ldots, 2n$, in which jobs arrive or depart (including the initial state with no job) has its own set of vertices called a layer. Each layer holds a vertex for every possible assignment of the current active jobs to machines; furthermore, we label each node by the maximum load of a machine in that configuration. The first and
last layers contain a single vertex, as there are no jobs at that point. These vertices are
called a source and a sink.

Two vertices of adjacent layers \( \sigma_{i-1} \) and \( \sigma_i \), \( i = 1, \ldots, 2n \), are connected by an edge if
the transition from one assignment of the active jobs to the other is consistent with the
arrival and departure of jobs at time \( \sigma_i \). More precisely, the vertices are connected if and
only if every job active both before and after \( \sigma_i \) is assigned to the same machine in the
assignments of both vertices. At each event, jobs either arrive or depart but not both (due
to the assumption at the beginning that all the original events are distinct; during rounding
we do not mix arrival and departure events). If \( \sigma_i \) is an arrival, the in-degree of all vertices
in the layer \( \sigma_i \) is 1, since the new configuration determines the old one. Similarly if \( \sigma_i \) is a
departure, the out-degree of all vertices in the layer \( \sigma_{i-1} \) is 1. In both cases, the number of
edges between two layers is linear in the number of vertices on these layers. It follows that
the total number of edges is linear in the number of vertices.

We define a value of a path from the source to sink as the maximal value of its nodes.
Now we can simply find a path with smallest value by any shortest path algorithm in linear
time (since the graph is layered).

In the converting phase the algorithm converts the assignment found for \( J'' \) into an
assignment for \( J \). Assume the number of jobs of weight \( \gamma \) in \( J''_M \) that are assigned to a
certain machine \( i \) is \( r_i \). Remove these jobs and assign all the jobs smaller than \( \gamma \) in \( J''_M \) such
that a total weight of at most \( (r_i + 1)\gamma \) is assigned to each machine. This is possible since the
replacement involves jobs whose weight is at most \( \gamma \) and from volume consideration there
is always at least one machine with a load of at most \( r_i \gamma \) of these jobs. The assignment for
\( J'' \) is also an assignment for \( J' \) and \( J \).

2.4 Analysis

Lemma 2.4.1 Given a solution for the original problem \( J \) whose maximum load is \( \lambda \), the
same solution applied to \( J' \) has a maximum load of at most \( \lambda + \epsilon + \epsilon^3/4 \). Also, given a
solution for \( J' \) whose maximum load is \( \lambda \), the same solution applied to \( J \) has a maximum
load of at most \( \lambda \).

Proof: The second claim is obvious since the jobs in \( J \) are shorter than the corresponding
jobs in \( J' \). As for the first claim, every time \( \tau \) is contained in at most \( \lceil \log n \rceil \) sets \( J_i \).
Consider the added load at \( \tau \) from jobs in a certain set \( J_i \). If \( \tau < s_i^j \) or \( \tau \geq t_i^j \) then the
same load is caused by \( J'_i \) and \( J_i \). Assume \( \tau < c_i \) and define \( s_i^{j+1} = c_i \), the other case is
symmetrical. Then for some \( j \), \( s_i^j \leq \tau < s_i^{j+1} \) and the added load at \( \tau \) is at most the total
load of \( S_i^j \) which is at most \( \alpha + \gamma \). Summing on all sets \( J_i \), we conclude that the maximal
load has increased by at most \( (\alpha + \gamma)\lceil \log n \rceil = \epsilon + \epsilon^3/m^2 \). ■

Lemma 2.4.2 Given a solution for \( J' \) whose maximum load is \( \lambda \), the same solution applied
to \( J'' \) has a maximum load of at most \( \lambda(1 + \epsilon) \). Also, given a solution for \( J'' \) whose maximum load is \( \lambda \), the same solution applied to \( J' \) has a maximum load of at most \( \lambda \).

**Proof:** The second claim is obvious since the jobs in \( J' \) are shorter than the corresponding jobs in \( J'' \). As for the first claim, given a time \( \tau \) and a pair of sets \( S_i^j, T_i^j \) from \( J' \) we examine the increase in load at \( \tau \). If \( \tau < s_i^j \) or \( \tau \geq t_i^j \) the load is not affected by the transformation because no job in \( T_i^j \cup S_i^j \) arrives before \( s_i^j \) or departs after \( t_i^j \). Assume that \( \tau < c_i \), the other case is symmetrical. So \( \tau \) is affected by the decrease in arrival time of jobs in \( T_i^j \). It is clear that the way we extend the jobs in \( T_i^j \) increases the load at \( \tau \) by at most \( \beta \). Also, since \( \tau \geq s_i^j \), we know that the load caused by \( S_i^j \) is at least \( \alpha \) if \( j < k_i \). Thus, an extra load of at most \( \beta \) is created by every pair \( S_i^j, T_i^j \) for \( 1 \leq j < k_i \) only if the pair contributes at least \( \alpha \) to the load. If the last pair \( S_i^j, T_i^j \) has weight smaller than \( \alpha \), it does not contribute, as it is not changed from \( J' \) to \( J'' \); otherwise the analysis is the same as for \( j < k_i \). Since the total load on all machines at any time is at most \( \lambda m \), the increase in load of any machine and therefore in maximum load is at most \( \beta \cdot \frac{\lambda m}{\alpha} = \epsilon \lambda \).

**Lemma 2.4.3** Given a solution for \( J' \) whose maximum load is \( \lambda \), the modified problem \( J'' \) has a maximum load of \( \lambda(1 + \epsilon + \epsilon^2) \). Also, given a solution for \( J'' \) whose maximum load is \( \lambda \), the solution given by the converting phase for the problem \( J'' \) has a maximum load of at most \( \lambda(1 + \epsilon + \epsilon^2) \).

**Proof:** Consider a solution for \( J'' \) whose maximum load is \( \lambda \). If the load of jobs smaller than \( \gamma \) in a certain \( J''_{st} \) on a certain machine \( i \) is \( x \), we replace it by at most \([x/\gamma]\) jobs of weight \( \gamma \) so that this is an assignment to \( J'' \). The increase in load on every machine is at most \( \gamma \) times the number of sets \( J''_{st} \) that contain jobs which are assigned to that machine. As for the other direction, consider a solution whose maximum load is \( \lambda \) to \( J'' \). The increase in load on every machine by the replacement described in the algorithm is also at most \( \gamma \) times the number of sets \( J''_{st} \) that contain jobs which are assigned to that machine.

It remains to estimate the number of sets \( J''_{st} \) that can coexist at a certain time. Most of these sets have weight at least \( \beta \); their number is at most \( \lambda m / \beta \), since the total load at any time is at most \( \lambda m \). For each set \( S_i^j \) and \( T_i^j \), \( j < k_i \), we have at most one set \( J''_{st} \) with weight less than \( \beta \), since the weight of \( S_i^j \) and \( T_i^j \) is at least \( \alpha \), there are at most \( \lambda m / \alpha \) such sets (if \( S_i^j \) and \( T_i^j \) are not disjoint, the small sets \( J''_{st} \) in both of them have the same \( s \) and \( t \), thus we do not need to multiply by \( 2 \)). Last, there may be one set \( J''_{st} \) smaller than \( \beta \) in each \( S_i^k, T_i^k \); there are only \([\log n]\) of such sets. Therefore, the increase in maximum load is at most \( \gamma (\lambda m / \beta + \lambda m / \alpha + [\log n]) = \epsilon \lambda + \epsilon^2 (\lambda m / \alpha + \epsilon \lambda / m) \leq \lambda (\epsilon + \epsilon^2) \).

**Theorem 2.4.1** The algorithm described in the last section is a PTAS running in time \( O(n^{\alpha-3} m^{\alpha} \log m) \), where \( c \) is some absolute constant.

**Proof:** We are given some \( \epsilon' > 0 \) and want to find a solution with maximum load at most \( \lambda(1 + \epsilon') \). If \( \epsilon' \geq 1 \), we use Graham’s List Scheduling. Otherwise we use the algorithm
2.5. THE UNRESTRICTED NUMBER OF MACHINES CASE

described above for $\epsilon = \epsilon' / 6$. For an instance with optimal solution with maximum load $\lambda$, the algorithm yields a solution with maximum load at most $\lambda(1 + \epsilon + \epsilon^2 / 4)(1 + \epsilon)(1 + \epsilon^2)^2 < \lambda(1 + \epsilon')$.

Every layer in the graph stores all the possible assignments of jobs to machines. Since the smallest job is of weight $\gamma$, the maximum number of active jobs at a certain time is $\lambda m / \gamma$. So, the maximum number of edges in the graph and the running time of the algorithm is $O(n m^\lambda m / \gamma) \leq O(n m \epsilon^2 m^3 |\log n|) = O(n^{1 + 2\epsilon - \lambda m^3 |\log n|})$. This yields the result, with the constant $c$ bounded by $c < 2 \cdot 6^3 + 1 = 433$.

2.5 The Unrestricted Number of Machines Case

In this section we show that in case the number of machines is given as part of the input, the problem cannot be approximated up to a factor of 4/3 in polynomial time unless $P = NP$. Then we continue with improving this result by showing that the problem cannot be approximated better than 3/2.

**Theorem 2.5.1** For every $\rho < 1/3$, there does not exist a polynomial $\rho$-approximation algorithm for the temporary tasks assignment problem unless $P = NP$.

**Proof**: We show a reduction from the exact cover by 3-sets ($X3C$) which is known to be NP-complete [33, 42]. In that problem, we are given a set of $3n$ elements, $A = \{a_1, a_2, ..., a_{3n}\}$, and a family $F = \{T_1, ..., T_m\}$ of $m$ triples, $F \subseteq A * A * A$. Our goal is to find a covering in $F$, i.e. a subfamily $F'$ for which $|F'| = n$ and $\bigcup_{T_i \in F'} T_i = A$.

Given an instance for the $X3C$ problem we construct an instance for our problem. The number of machines is $m$, the number of triples in the original problem. There are three phases in time. First, there are times $1, ..., m$, each corresponding to one triple. Then, times $m + 1, ..., m + 3n$ each corresponding to an element of $A$. And finally, the two times $m + 3n + 1, m + 3n + 2$.

There are four types of jobs. The first type are $m$ jobs of weight 3 starting at time 0. Job $r$, $1 \leq r \leq m$ ends at time $r$. To any appearance of $a_j$ in a triple $T_i$ corresponds a job of the second type of weight 1 that starts at $i$ and ends at $m + j$ and another job of the third type of weight 1 that starts at time $m + j$. Among all the jobs that start at time $m + j$, one ends at $m + 3n + 2$ while the rest end at $m + 3n + 1$. The fourth type of jobs are $m - n$ jobs of weight 3 that start at $m + 3n + 1$ and end at $m + 3n + 2$.

We show that there is an assignment with maximum load at most 3 if and only if there is an exact cover by 3-sets. Suppose there is a cover. We assign a job of the first type that ends at time $i$ to machine $i$. We assign the three jobs of the second type corresponding to $T_i$ to machine $i$. At time $m + j$, some jobs of type two depart and the same number of jobs of type three arrives. One of these jobs is longer than the others since it ends at time $m + 3n + 2$. We assign that longer job to machine $i$ where $T_i$ is the triple in the covering
Figure 2.5: An assignment for the problem corresponding to \( m = 4, n = 2, F = \{(1,2,3), (1,4,5), (4,5,6), (2,3,4)\} \)

that contains \( j \). At time \( m + 3n + 1 \) many jobs depart. We are left with \( 3n \) jobs, three jobs on each of the \( n \) machines corresponding to the 3-sets chosen in the cover. Therefore, we can assign the \( m - n \) jobs of the fourth type on the remaining machines.

Now, assume that there is an assignment whose maximum load is at most 3. One important property of our assignment problem is that at any time \( \tau \), \( 0 \leq \tau < m + 3n + 2 \) the total load remains at \( 3m \) so the load on each machine has to be 3. We look at the assignment at time \( m + 3n + 1 \). Many jobs of type three depart and only the long ones stay. The number of these jobs is \( 3n \) and their weight is 1. Since \( m - n \) jobs of weight 3 arrive at time \( m + 3n + 1 \), the \( 3n \) jobs must be assigned to \( n \) machines. We take the triples corresponding to the \( n \) machines to be our covering. Assume by contradiction that this is not a covering. Therefore, there are two 3-sets that contain the same element, say \( a_j \). At time \( m + j \) only one long job arrives. The machine to which a shorter job was assigned remains with a load of 3 until time \( m + 3n + 1 \) and then the short job departs and its load decreases to at most 2. This is a contradiction since at time \( m + 3n + 1 \) there are \( n \) machines each with 3 long jobs.

We improve the above result in the following theorem:

**Theorem 2.5.2** For every \( \rho < \frac{3}{2} \), there does not exist a polynomial time \( \rho \)-approximation algorithm for the temporary tasks assignment problem unless \( P = NP \).

**Proof:** The proof is by reduction from edge-coloring of cubic graphs (cubic means that all vertices have degree three): A feasible edge 3-coloring of a simple cubic graph \( G = (V,E) \) is a coloring of \( E \) with the colors 1, 2 and 3, such that for every vertex the incident edges receive three distinct colors. Deciding whether a given cubic graph \( G = (V,E) \) possesses a feasible edge 3-coloring is NP-complete [33]. Since \( G \) is cubic, \( |V| = 2q \) and \( |E| = 3q \) holds for some positive integer \( q \). Moreover, since in a feasible edge 3-coloring of \( G \) every color class forms a perfect matching, every color will occur exactly \( q \) times.
2.5. THE UNRESTRICTED NUMBER OF MACHINES CASE

Given a graph $G$ we describe an instance of the load balancing problem. Let $e_1, \ldots, e_{3q}$ be an arbitrary enumeration of the edges in $E$, and let $v_1, \ldots, v_{2q}$ be an arbitrary enumeration of the vertices in $V$. We construct an instance of the temporary task assignment problem with $m = 3q$ machines and $n = 18q$ jobs.

- For every edge $e_i$ ($i = 1, \ldots, 3q$), there is a corresponding job of weight 2 starting at time 0 and ending at time $i$.

- Let $v_j$ ($j = 1, \ldots, 2q$) be a vertex, and let $e_x, e_y$ and $e_z$ be the three edges incident to $v_j$. Then there are six jobs $J_{j,x}, J_{j,y}, J_{j,z}$ and $K_{j,1}, K_{j,2}, K_{j,3}$ that correspond to $v_j$ and that all have weight 1. The jobs $J_{j,x}, J_{j,y}, J_{j,z}$ start at times $x, y, z$, respectively, and all end at time $3q + j$. The jobs $K_{j,1}, K_{j,2}, K_{j,3}$ start at time $3q + j$ and end at times $5q + 1, 5q + 2, \text{ and } 5q + 3$, respectively.

- Finally, there are $3q$ dummy jobs that all have weight 2. The dummy jobs are divided into three classes of $q$ jobs. The $q$ dummy jobs in class $c$ ($c = 1, 2, 3$) start at time $5q + c$ and end at time $5q + 4$.

This completes the description of the assignment instance. We claim that this assignment instance possesses an assignment with maximum load 2 if and only if the graph $G$ possesses a feasible edge 3-coloring.

Proof of the “if” part. Suppose that the graph $G$ possesses a feasible edge 3-coloring. Let $e_i = [v_j, v_k]$ be an edge in $E$ that receives color $c \in \{1, 2, 3\}$ in this coloring. The following jobs are processed on machine $i$: From time 0 to time $i$, process the job that corresponds to edge $e_i$. From time $i$ to time $3q + j$ process job $J_{j,i}$, and from time $3q + j$ to time $5q + c$, process job $K_{j,c}$. Analogously, from time $i$ to time $3q + k$ process job $J_{k,i}$, and from time $3q + k$ to time $5q + c$, process job $K_{k,c}$. Finally, from time $5q + c$ to time $5q + 4$ process a dummy job from class $c$. It is easy to see that in this assignment all jobs are processed, and at any moment in time every machine has load at most 2.
Proof of the “only if” part. Now assume that there is an assignment with maximum load 2. Note that at any moment \( \tau \) in time, \( 0 \leq \tau \leq 5q + 4 \), the total available load is exactly \( 6q = 2m \). Hence, in an assignment with maximum load 2 every machine must be constantly busy. Without loss of generality we assume that for \( 1 \leq i \leq 3q \), machine \( i \) processes the job corresponding to edge \( e_i \). Moreover, at time \( 5q + 4 \) machine \( i \) completes one of the dummy jobs. We color edge \( e_i \) by the class \( c \) of this dummy job.

We claim that in the resulting coloring, every vertex is incident to three differently colored edges: Suppose otherwise. Then there exist two edges \( e_x = [v_j, v_k] \) and \( e_y = [v_j, v_k] \) that receive the same color \( c \). Consider machine \( x \) at time \( x \) in the assignment. The job corresponding to \( e_x \) ends at time \( x \), and the only available jobs are \( J_{j,x} \) and \( J_{k,x} \). Since machine \( x \) is busy all the time, it must process these two jobs. Consider machine \( x \) at time \( 3q + j \) in the assignment. The processing of job \( J_{j,x} \) ends, and the machine must process some job from time \( 3q + j \) to time \( 5q + c \); the only possible job for this is job \( K_{j,c} \). By similar arguments we get that job \( K_{j,c} \) is simultaneously processed on machine \( y \), a contradiction. Hence, the constructed edge-coloring indeed is a feasible edge 3-coloring. 

2.6 Discussion

Both of the lower bounds shown above are proved directly by a reduction from \( NP \)-complete problems. Although the gap between the 1.5 lower bound and the 2 upper bound is low, one should still consider how can the gap be closed. One interesting approach is to improve the lower bound by using more sophisticated techniques such as PCP reductions.

The approximation scheme described in this chapter applies to load balancing over identical machines. Several extensions of the identical machines model exist, of which one of the most powerful is the unrelated machines model. Recall that in this model each job machine pair has its own associated load. In a recent paper [3], an approximation scheme for load balancing over unrelated machines was shown.
Chapter 3

Online Load Balancing with Unrelated Machines

3.1 Introduction

The eight basic on-line load balancing problems described in the introduction of this thesis are the identical, related, restricted and unrelated machines models with either permanent or temporary tasks. Asymptotically tight bounds were given to all these problems except one: the assignment of temporary tasks to unrelated machines remained open. In this chapter, we present an inapproximability result by employing a cyclic load transfer method. We ignore the off-line version of the problem since it was recently solved in [3].

We also investigate a model that is a combination of the restricted and the related machines model. It is more powerful than both of them but is still not as powerful as the unrelated machines model. In this model, named the related-restricted machines model, each job has its own weight and an admissible machine set. In addition each machine has a speed. A job can be assigned to one of its admissible machines where the added load is its weight divided by the machine’s speed. As it turns out, tight results can be achieved for the case where two different machine speeds exist. This model seems like a realistic one and is currently the most powerful model for which an approximability result exists.

The results in [11] (and later in [53]) show a lower bound of $\Omega(\sqrt{m})$ on the competitive ratio that any on-line algorithm (deterministic or randomized) may have for restricted assignment of temporary tasks. This $\Omega(\sqrt{m})$ lower bound for restricted assignment dates back to 1992 but was still the best lower bound for unrelated assignment. The lower bound for the restricted assignment was shown to be tight in [13] by a matching $O(\sqrt{m})$ upper bound (the “Robin-Hood” algorithm). For the unrelated assignment, on the other hand, no on-line algorithm exists apart from a trivial upper bound. Our results show that this trivial $O(m)$ competitive algorithm is almost optimal hence proving the inapproximability of this model. Specifically, by using a cyclic load transfer method we achieve an $\Omega(m/\log m)$ lower
bound. The open problem mentioned in [24] regarding the existence of a better approximation algorithm is thus answered negatively. We extend the inapproximability result to randomized algorithms as well.

3.2 Inapproximability of the Unrelated Machines Model

In this section we present the inapproximability result for online load balancing of temporary tasks on unrelated machines. Namely, we show a lower bound of $\Omega(m/\log m)$ almost matching an upper bound of $O(m)$. It can be seen that the simple algorithm assigning each job to its fastest machine is an $O(m)$ competitive algorithm.

We proceed with the inapproximability result:

**Theorem 3.2.1** Any online algorithm for the load balancing of temporary tasks on unrelated machines is $\Omega(m/\log m)$ competitive.

**Proof:** Let $k$ be the largest integer power of 2 such that $k \leq m/\log m$. Assume by contradiction that there is an online algorithm whose competitive ratio is below $k/2$. We describe a sequence of jobs given by an adversary such that there exists an optimal assignment whose maximum load is at most 1. The sequence ends as soon as there is a machine whose load in the online assignment is at least $k/2$.

The lower bound uses $l = \log k$ sets of $k$ machines each. The sets are denoted by $M_1, M_2, \ldots, M_l$. In addition, a machine denoted by $m_0^1$ is used. Also, denote by $m_i^j$ the $j$'th machine in the set $M_i$, $1 \leq i \leq l$, $1 \leq j \leq k$. Note that the total number of machines used, $l \cdot k + 1$, does not exceed $m$ (when $m > 2$).

The adversary proceeds in phases. Before the start of phase $t$, we define a set of $l + 1$ machines which we call active. One machine in each $M_i$ is active and we denote its index by $a_i(t)$. The machine $m_0^0$ is always active and we use the notation $a_0(t) = 1$. The load of $m_i^{a_i(t)}$, $i = 0, \ldots, l$, in the online assignment is denoted by $b_i(t)$. We begin with setting $a_i(0) = 1$ for $i = 1, \ldots, l$.

A phase is composed of an arrival of one job and the departure of a set of jobs. The job presented by the adversary has infinite weight on non-active machines. Its weight on $m_i^{a_i(t)}$ is $2^i/k$ for $i = 0, \ldots, l$. Assume the online algorithm assigns it to machine $m_i^{a_i(t)}$. In case the new load, $b_i(t + 1)$, is $k/2$ or more the sequence stops. Otherwise, the phase is completed with the departure of the jobs assigned by the online algorithm to $m_i^{a_{i+1}(t)}$, $\ldots$, $m_i^{a_l(t)}$ (no job leaves when $i = l$). The set of active machines for the next phase is set as follows: $a_j(t + 1) = 1$ for any $i + 2 \leq j \leq l$ and unless $i = l$ we also set $a_{i+1}(t + 1) = 1 + \lfloor 2b_i(t) \rfloor$. All other active machines stay the same. Note that since $b_i(t) < k/2$ the above definition of $a_i(t+1)$ is valid, that is, $a_i(t+1) \leq k$. Also note that by the above construction, non-active machines are always empty.
If we consider the vector of loads of the online assignment, \((b_0(t), b_1(t), ..., b_l(t))\), we note that the vector increases lexicographically after each phase. That is, at least one of the coordinates increases while all previous coordinates do not change. The increase is by at least \(1/k\). Since the adversary sequence is completed once one of the coordinates exceeds \(k/2\), the sequence is completed after a finite number of phases, or specifically, at most \(O(k^{2^k}) = O(m^{2\log m})\) phases.

In what follows we complete the proof by showing an optimal assignment where the maximum load does not exceed 1 during the whole sequence. In case the job arriving at phase \(t\) is assigned by the online algorithm to \(m_i^{a_i(t)}\), \(i = 0, ..., l - 1\), then the optimal algorithm assigns it to machine \(m_i^{a_i(t)+1}\). Otherwise, the job arriving at phase \(t\) is assigned by the online algorithm to \(m_i^{a_i(t)}\) and the optimal algorithm assigns it to machine \(m_0^1\).

First, jobs assigned by the optimal assignment to \(m_0^1\) are assigned by the online algorithm to an active machine in \(M_i\). Since that machine’s load is not more than \(k/2\) and all other machines in \(M_i\) are empty, the incurred load on \(m_0^1\) is at most \(1/2\). Now consider jobs assigned to \(m_{i+1}^j\), \(i = 0, ..., l - 1\), by the optimal assignment. These are assigned by the online algorithm to the active machine in \(M_i\). Moreover, when they were assigned, the load on the active machine in \(M_i\) was at least \((j - 1)/2\) and less than \(j/2\) because \(a_{i+1}\), which was equal to \(j\), is defined as twice this load plus one rounded down. Therefore, their total load on a machine in \(M_i\) is at most \(1/2\) and their total load on a machine in \(M_{i+1}\) is at most 1.

Concluding, the above sequence was shown to have an optimal assignment of maximum load 1. Moreover, as long as the online maximum load is below \(k/2\) the online load vector was shown to increase lexicographically. This contradicts our assumption that an online algorithm with competitive ratio below \(k/2\) exists and completes the proof.

The following lemmas demonstrate a general technique for converting deterministic lower bounds into randomized ones.

**Lemma 3.2.1** Let \(R\) be a randomized on-line algorithm for load balancing of temporary tasks on unrelated machines which achieves a competitive ratio of \(c\). Then there exists a deterministic on-line algorithm \(D\), allowed to split jobs between different machines, which achieves a competitive ratio \(\leq c\) against an optimal algorithm which is not allowed to split jobs.

**Proof:** Algorithm \(R\) determines the probability \(p_j(i)\) that a job \(j\) is assigned to machine \(m_i\), \(\sum_i p_j(i) = 1\). We denote the expectation of the maximum load of \(R\) by \(l^R\). For any input sequence, \(l^R \leq c \cdot l^{OPT}\).

We define \(D\) as follows. For an arriving job \(j\), \(D\) splits the job between the machines according to the probabilities set by \(R\). So \(D\) assigns a load of \(p_j(i) \cdot w_j\) to each machine \(m_i\). We denote the maximum load of \(D\) by \(l^D\).
We now prove that for any input sequence $l_D \leq l_R$. Note that at any given time $t$, the load of $D$ on machine $m_i$ is $l_i^D(t) = E(l_i^R(t))$. Hence, $l_D(t) = \max_i(l_i^D(t)) = \max_i(E(l_i^R(t)))$. Using the fact that the maximum of expectations is at most the expectation of maxima, $l_D(t) \leq E(\max_i(l_i^R(t))) = l_R(t)$. The proof is completed by noting that $l_D = \max_t(l_D(t)) \leq \max_t(l_R(t)) = l_R \leq c \cdot l^{OPT}$.

**Lemma 3.2.2** Let $c$ be a lower bound on the competitive ratio of any deterministic algorithm for unrelated assignment of temporary tasks. If $c$ can be proven with an adversarial strategy of jobs in which each job has an admissible set of at most $k$ machines, then there is a lower bound of $\frac{c}{k}$ on the competitive ratio of any randomized on-line algorithm for the same problem.

**Proof:** We will prove that there is a lower bound of $\frac{c}{k}$ on the competitive ratio of any deterministic on-line algorithm $D$ which is allowed to split jobs (compared to an optimum which is not allowed to split them). According to the previous lemma, this implies a lower bound of $\frac{c}{k}$ for randomized on-line algorithms as well. Consider the sequence used for the deterministic lower bound. Since each job in that sequence can only be assigned to at most $k$ machines, $D$ must assign at least $\frac{1}{k}$ of the job’s weight to one machine. Now, consider a deterministic online algorithm $D’$ which simulates $D$ and assigns each job to just one machine: the machine to which $D$ assigned the largest part of the job. This is an online algorithm that does not split jobs and therefore our adversarial strategy creates a load of at least $l_D’ \geq c \cdot l^{OPT}$. Since $D$ assigns at least a $\frac{1}{k}$-fraction of each job to the same machine as $D’$, its load at the end of the same sequence must be at least $l_D \geq \frac{1}{k} \cdot c \cdot l^{OPT}$ which gives the required competitive ratio. ■

**Theorem 3.2.2** Any online randomized algorithm for the load balancing of temporary tasks on unrelated machines is $\Omega(m/(\log m)^2)$ competitive.

**Proof:** The construction in Theorem 3.2.1 uses admissible sets of at most $\log m$ machines. The theorem then follows as a corollary of Lemma 3.2.2. ■

**3.3 Tight Results for the Related-Restricted Machines Model**

The result in the previous Section shows that approximating the unrelated machines model is almost infeasible. As an alternative to the unrelated machines model we consider the so-called related-restricted machines model. Here, each machine has its own speed and each job has a weight and a set of admissible machines. However, note that the lower bound presented in the last section still applies here so approximating is still infeasible. We show that by limiting the number of different machine speeds to a constant number, we can approximate the problem better. Specifically, in the case where only two different machine speeds are involved we obtain the following lower bound. Later, we will present a matching upper bound.
3.3. TIGHT RESULTS FOR THE RELATED-RESTRICTED MACHINES MODEL

Theorem 3.3.1 Any online algorithm for the load balancing of temporary tasks in the related-restricted machines model with speeds \( \{1, s\} \) is \( \Omega \left( \min \{ m/s, \sqrt{ms} \} \right) \) competitive. The same holds for randomized algorithms as well.

Proof: A lower bound of \( \Omega(\sqrt{m}) \) already exists for the special case of the restricted assignment model and therefore we can limit ourselves to showing a lower bound of \( \Omega \left( \min \{ m/s, \sqrt{ms} \} \right) \) for \( s \leq \sqrt{m} \). Let \( m' = m/3 \) and \( b = \min \{ m'/s, \sqrt{ms} \} \). Assume by contradiction that there is an online algorithm whose competitive ratio is below \( b \). We describe a sequence of jobs given by an adversary such that there exists an optimal assignment whose maximum load is at most 1. The sequence ends as soon as there is a machine whose load in the online assignment is at least \( b \).

The lower bound uses three sets of machines, \( M_0, M_1, \) and \( M_2 \). Both \( M_1 \) and \( M_2 \) contain \( m' \) machines whereas \( M_0 \) contains \( m'/b \) machines. The \( j \)'th machine in \( M_i \) is denoted \( m_i^j \). The idea behind the lower bound is to force the online algorithm to assign jobs to machines in \( M_0 \) while an optimal algorithm can assign them to machines in \( M_1 \).

The lower bound proceeds in phases and stops as soon as the load on a machine reaches \( b \). At the beginning of a phase \( t \) we choose three active machines, one in each \( M_i \). Their indices are denoted by \( a_0(t), a_1(t) \) and \( a_2(t) \). They are initially set to \( a_0(0) = a_1(0) = a_2(0) = 1 \). One job is presented in each phase. Its weight on \( m_0^{a_0(t)} \) or on \( m_1^{a_1(t)} \) is \( 1/s \) where the weight on \( m_2^{a_2(t)} \) is 1. In case the online algorithm assigned the job to \( m_0^{a_0(t)} \), all jobs assigned by the online algorithm to \( M_1 \) or \( M_2 \) leave. If the job is assigned to \( m_1^{a_1(t)} \) all jobs assigned to \( M_2 \) leave. Otherwise, the job was assigned to \( m_2^{a_2(t)} \) and no jobs leave. For the next phase, we set \( a_0(t+1) = 1 + \lfloor b_2(t)/s \rfloor, a_1(t+1) = 1 + \lfloor b_0(t) \rfloor \) and \( a_2(t+1) = 1 + \lfloor b_1(t) \rfloor \) where \( b_i(t) \) denotes the total on-line load on machines in \( M_i \). This is possible first because in \( M_0 \) there are \( m'/b \) machines each of which has a load of less than \( b \) so that \( a_1(t+1) = 1 + \lfloor b_0(t) \rfloor \leq b \cdot m'/b = m' \).

In addition, note that at any given time only one machine in \( M_1 \) and one machine in \( M_2 \) are not empty in the online assignment. Therefore, \( a_2(t+1) = 1 + \lfloor s b_1(t) \rfloor \leq s \cdot b \leq m' \) and \( a_0(t+1) = 1 + \lfloor b_2(t)/s \rfloor \leq b/s \leq m'/b \).

Note that the vector \((b_0(t), b_1(t), b_2(t))\) lexicographically increases after each phase by at least \( 1/s \). Moreover, the adversary described above proceeds as long as the maximum load is below \( b \). Therefore, after a finite number of phases the load on one of the machines is going to reach \( b \).

The proof is completed by showing an optimal assignment where the maximum load does not exceed 1. In case the job arriving at phase \( t \) is assigned by the online algorithm to machine \( m_i^{a_i(t)} \), then the optimal algorithm assigns it to machine \( m_j^{a_j(t)} \) where \( j = i + 1 \mod 3 \).

Now consider jobs assigned to \( m_i^j \) by the optimal assignment. These are assigned by the online algorithm to a machine in \( M_0 \). Moreover, when they were assigned, the total online load on machines in \( M_0 \) was at least \( j - 1 \) and less than \( j \). Since jobs assigned by the online algorithm to \( M_0 \) never leave, the load on \( m_i^j \) is at most 1. Similarly, consider jobs assigned to \( m_2^j \) by the optimal assignment. These are assigned by the online algorithm to an active
machine in $M_1$. When they were assigned, the total online load on the active machine in $M_1$ was at least $(j - 1)/s$ and less than $j/s$. Since jobs assigned by the online algorithm to $M_1$ leave all together and never leave individually, the load on $m_j^1$ is at most $s \cdot 1/s = 1$. A similar argument holds for jobs assigned to $M_0$ by the optimal assignment.

Concluding, the above adversarial sequence was shown to have an optimal assignment of maximum load 1 and as long as the online maximum load is below $b$, an online load vector was shown to increase lexicographically. This contradicts our assumption that a $b$-competitive online algorithm exists and completes the proof. By Lemma 3.2.2, the result also holds for the randomized case since there are exactly three machines in any admissible set.

In the rest of this section, we describe two online algorithms whose combination achieves a competitive ratio matching the lower bound stated above. In both algorithms we assume that $OPT = 1$. Nevertheless, by using standard doubling techniques we can overcome that assumption while increasing the competitive ratio by a factor of at most 4. Both algorithms are based on a modified Robin Hood algorithm [13] with some threshold $b$. A machine is said to be poor at a certain time if its load it less than $b$ and is said to be rich otherwise. For a machine rich at time $t$ we define its windfall time as $t_0$ if it is rich from time $t_0$ to $t$ but is poor at time $t_0 - 1$. As its name implies, the Robin Hood algorithm tries to assign jobs to the poor machines or in case none exists, to the a rich machine with the most recent windfall time.

**Algorithm 1** In case an incoming job has at least one admissible fast machine, remove all slow machines from its admissible machines. Then use the Robin Hood algorithm with a threshold $b = \max \{ \sqrt{m}, m/s \}$.

**Claim 3.3.1** The above algorithm is $O(\max \{ \sqrt{m}, m/s \})$ competitive.

**Proof:** First notice that jobs assigned to slow machines are not allowed to be assigned to any fast machine. Therefore, the contribution of a job to the total load of an optimal assignment is at least its contribution to the total load of the online assignment. Hence, the total load on all machines at any given time is at most that of an optimal assignment which is at most $m$. This implies that the number of rich machines at any given time is at most $m/b$.

By the definition of the algorithm, any job assigned to a slow rich machine $i$ has only slow rich machines in its admissible set. Moreover, the windfall time of these machines is before the windfall time of $i$, that is, they are all already rich at the windfall time of $i$. Therefore, in an optimal assignment, all jobs assigned to $i$ after its windfall time are assigned to at most $m/b$ slow machines. Their total load on $i$ is at most $m/b$. Adding the job made $i$ rich, we get that the total load on $i$ at any given time is at most $b + 1 + m/b = O(b)$.

Now consider a fast rich machine. Unlike slow machines, some of the jobs assigned to fast machines might result from the algorithm’s removal of slow admissible machines. Thus
jobs assigned to a fast rich machine $i$ after its windfall time are assigned in an optimal assignment to one of at most $m/b$ fast machines or to any one of the slow machines. The total load on machine $i$ is therefore at most $b + 1 + m/b + m/s = O(b)$ and the required competitive ratio is achieved.

**Algorithm 2** Use the Robin Hood algorithm with a threshold $b = \sqrt{ms}$.

**Claim 3.3.2** The above algorithm is $O(\sqrt{ms})$ competitive.

**Proof:** The total load on the fast machines at any given time is at most that of an optimal assignment which is at most $m$. The total load on the slow machines at any given time is at most $sm$ since these jobs might be assigned to fast machines by an optimal assignment. This implies that the number of fast rich machines at any given time is at most $m/b$ and that the number of slow rich machines is at most $sm/b$.

Consider a job assigned to a rich machine $i$. All other machines in its admissible set are rich and their windfall time is before the windfall time of $i$, that is, they are all already rich at the windfall time of $i$. Therefore, all jobs assigned to $i$ after its windfall time are assigned in an optimal assignment to at most $sm/b$ slow machines and to at most $m/b$ fast machines. Therefore, if $i$ is slow, their total load on $i$ is at most $2sm/b$ and if $i$ is fast, their total load on $i$ is at most $2m/b$. Adding the job that made $i$ rich, we get that the total load on $i$ at any given time is at most $b + s + 2sm/b = O(\sqrt{ms})$.

**Algorithm 3** If $s > m^{1/3}$ use the first algorithm; otherwise, use the second algorithm.

Theorem 3.3.2 The above algorithm is $O(\min\{\max\{m/s, \sqrt{m}\}, \sqrt{ms}\})$-competitive for load balancing of temporary tasks in the related-restricted machines model with speeds $\{1, s\}$.

### 3.4 Discussion

The main result presented in this chapter is almost tight. The gap of $O(\log m)$, as small as it is, still exists and closing this gap is a problem of some interest. A problem that is of comparable importance is to determine the competitive ratio of the related-restricted machines model with three or more speeds. We conjecture that upper bounds better than the trivial $O(m)$ upper bound should exist for a small constant number of machine speeds.
CHAPTER 3. ONLINE LOAD BALANCING WITH UNRELATED MACHINES
Chapter 4

Multiprocessor Scheduling without Migration

4.1 Introduction

This chapter considers the problem of scheduling jobs in a multiprocessor setting in order to minimize the average flow time or the total flow time (the two are equivalent since their ratio is the number of jobs). In the introduction of this thesis we have seen that without preemption good approximation ratios cannot be achieved. The question we address in this chapter is whether migration is also a necessary component for a good scheduling algorithm, in either the online or the offline setting.

The SRPT algorithm is the best approximation algorithm known for both the offline and the online settings. However, it can be seen that it uses migration. Without migration, no offline or online approximations were known. In this chapter we introduce a new algorithm that does not use migration and whose approximation ratio is as good as SRPT. More specifically, our algorithm guarantees, on all input instances, a small performance gap in comparison to the optimal offline schedule that allows both preemption and migration. Recall that $n$ is the number of jobs, $m$ is the number of machines, and denote by $P$ the ratio between the processing time of the longest and the shortest jobs. Then our algorithm performs within an $O(\min\{\log P, \log n\})$ factor of the optimal preemptive algorithm that allows migration. The algorithm can be easily implemented in polynomial time in the size of the input instance.

Our algorithm is also on-line. We note that in the proof of the $\Omega(\log P), \Omega(\log(n/m))$ lower bounds of [51] for on-line algorithms, the optimal algorithm does not use migration. Hence, the (randomized) lower bound holds also for a non-migratory algorithm. This implies that our algorithm is optimal with respect to the parameter $P$, i.e., no on-line algorithm can achieve a better bound both when the off-line algorithm is or is not allowed to migrate jobs (the first claim is obviously stronger), while there is still a small gap between the $O(\log n)$
upper bound and the $\Omega(\log(n/m))$ lower bound. We also show an $\Omega(\log n)$ competitive lower bound for our algorithm.

Our algorithm and its analysis draw on some ideas from the SRPT algorithm\[51\]. However, unlike SRPT, our algorithm may continue to run a job on a machine even when a shorter job is waiting to be processed. This seems essential in the non-migratory setting since being too eager to run a shorter job may result in an unbalanced commitment to machines. A non-migratory algorithm has to trade off between the commitment of a job to a machine and the decrease in the flow time yielded by running a shorter job. Our algorithm runs the job with the shortest remaining processing time among all the jobs that were already assigned to that machine, and a new job is assigned to a machine if its processing time is considerably shorter than the job that is currently running.

4.2 Preliminaries

We use the notations described in the introduction for the on-line preemptive scheduling model. In this chapter, our model allows preemption but does not allow migration, i.e., a job that is running can be preempted but must later continue its execution on the same machine on which its execution began. The scheduling algorithm decides which of the jobs should be executed at each time. Recall that a machine can process at most one job in any given time and that a job cannot be processed before its release time. We measure the total flow time, that is, $\sum_{j \in J} f_j$ and denote the total flow time of algorithm $A$ by $F^A$. The goal of the scheduling algorithm is to minimize the total flow time for each given instance of the problem. In the off-line version of the problem all the jobs are known in advance. In the on-line version of the problem each job is introduced at its release time and the algorithm bases its decision only upon the jobs that were already released.

4.3 The Algorithm

A job is called alive at time $t$ for a given schedule if it has already been released but has not been completed yet. Our algorithm classifies the jobs that are alive into classes according to their remaining processing times. A job $j$ whose remaining processing time is in $[2^k, 2^{k+1})$ is in class $k$ for an integer $-\infty < k < \infty$. Notice that a given job changes its class during its execution. The algorithm holds a pool of jobs that are alive and have not been processed at all. In addition, the algorithm holds a stack of jobs for each of the machines. The stack of machine $i$ holds jobs that are alive and have already been processed by machine $i$. The algorithm works as follows:

- Each machine processes the job at the top of its stack.

- When a new job arrives the algorithm looks for a machine that is idle or currently processing a job of a higher class than the new job. In case it finds one, the new job
in pushed onto the machine's stack and its processing begins. Otherwise, the job is inserted into the pool.

- When a job is completed on some machine it is popped from its stack. The algorithm compares the class of the job at the top of the stack with the minimum class of a job in the pool. If the minimum is in the pool then a job that achieves the minimum is pushed onto the stack (and removed from the pool).

Clearly, when a job is assigned to a machine it will be processed only on that machine and thus the algorithm does not use migration. In fact, the algorithm bases its decisions only on the jobs that were released up to the current time and hence is an on-line algorithm. Note that it may seem that the algorithm has to keep track of all the infinite number of classes through which a job evolves. However, the algorithm recalculates the classes of jobs only at arrival or completion of a job.

### 4.4 Analysis

We denote by $A$ our scheduling algorithm and by $OPT$ the optimal off-line algorithm that minimizes the flow time for any given instance. For our analysis we can even assume that $OPT$ may migrate jobs between machines. Whenever we talk about time $t$ we mean the moment after the events of time $t$ happened. For a given scheduling algorithm $S$ we define $V^S(t)$ to be the volume of a schedule at a certain time $t$. This volume is the sum of all the remaining processing times of jobs that are alive. In addition, we define $\delta^S(t)$ to be the number of jobs that are alive. $\Delta V(t)$ is defined to be the volume difference between our algorithm and the optimal algorithm, i.e., $V^A(t) - V^{OPT}(t)$. We also define $\Delta \delta(t) = \delta^A(t) - \delta^{OPT}(t)$ the alive jobs difference at time $t$ between $A$ and $OPT$. For a generic function $f$ ($V$, $\Delta V$, $\delta$ or $\Delta \delta$) we use $f_{h \leq k}(t)$ to denote the value of $f$ at time $t$ when restricted to jobs of classes between $h$ and $k$. Similarly, the notation $f_{=k}(t)$ will represent the value of function $f$ at time $t$ when restricted to jobs of class precisely $k$.

Let $\gamma^S(t)$ be the number of non-idle machines at time $t$. Notice that because our algorithm does not migrate jobs, there are situations in which $\gamma^A(t) < m$ and $\gamma^A(t) \geq m$. We denote by $T$ the set of times in which $\gamma^A(t) = m$, that is, the set of times in which none of the machines is idle. Denote by $P_{\text{min}}$ the processing time of the shortest job and by $P_{\text{max}}$ the processing time of the longest job. Note that $P = P_{\text{max}} / P_{\text{min}}$. Denote by $k_{\text{min}} = \lfloor \log P_{\text{min}} \rfloor$ and $k_{\text{max}} = \lceil \log P_{\text{max}} \rceil$ the classes of the shortest and longest jobs upon their arrival.

We start by observing the simple fact that the flow time is the integral over time of the number of jobs that are alive (for example, see [51]):

**Fact 4.4.1** For any scheduler $S$,

$$F^S = \int_0^T \delta^S(t)dt.$$
Next we note that the algorithm preserves the following property of the stacks:

**Lemma 4.4.1** In each stack the jobs are ordered in a strictly increasing class order and there is at most one job whose class is at most $k_{\text{min}}$.

*Proof:* At time $t=0$ the lemma is true since all the stacks are empty. The lemma is proved by induction on time. The classes of jobs in the stacks change in one of three cases. The first is when the class of the currently processed job decreases. Since the currently processed job is the job with the lowest class, the lemma remains true. The second case is when a new job arrives. In case it enters the pool there is no change in any stack. Otherwise it is pushed onto a stack whose top is of a higher class which preserves the first part of the lemma. Since the class of the new job is at least $k_{\text{min}}$, the second part of the lemma remains true as well. The third case is when a job is completed on some machine. If no job is pushed onto the stack of that machine the lemma remains easily true. If a new job is pushed onto the stack then the lemma remains true in much the same way as in the case of the arrival of a new job.

**Corollary 4.4.1** There are at most $2 + \log P$ jobs in each stack.

*Proof:* The number of classes of jobs in each stack is at most $k_{\text{max}} - k_{\text{min}} + 1 \leq 2 + \log P$.

We look at the state of the schedule at a certain time $t$. First let’s look at $t \notin T$:

**Lemma 4.4.2** For $t \notin T$, $\delta^A(t) \leq \gamma^A(t)(2 + \log P)$.

*Proof:* By definition of $T$, at time $t$ at least one machine is idle. This implies that the pool is empty. Moreover, all the stacks of the idle machines are obviously empty. So, all the jobs that are alive are in the stacks of the non-idle machines. The number of non-idle machines is $\gamma^A(t)$ and the number of jobs in each stack is at most $2 + \log P$ according to Corollary 4.4.1.

Now, assume that $t \in T$ and let $t < t$ be the earliest time for which $[t, t) \subset T$. We denote the last time in $[t, t)$ in which a job of class more than $k$ was processed by $t_k$. In case such jobs were not processed at all in the time interval $[t, t)$ we set $t_k = t$. So, $t \leq t_k \leq t_{k_{\text{max}}} \leq \ldots \leq t_{k_{\text{min}}} \leq t$.

**Lemma 4.4.3** For $t \in T$, $\Delta V_{\leq k}(t) \leq \Delta V_{\leq k}(t_k)$.

*Proof:* Notice that in the time interval $[t_k, t)$, algorithm $A$ is constantly processing on all the machines jobs whose class is at most $k$. The off-line algorithm may process jobs of
higher classes. Moreover, that can cause jobs of class more than \( k \) to actually lower their classes to \( k \) and below therefore adding even more to \( V_{\leq k}^{OPT}(t) \). Finally, the release of jobs of class \( \leq k \) in the interval \([t_k,t]\) is not affecting \( \Delta V_{\leq k}(t) \). Therefore, the difference in volume between the two algorithms cannot increase.

**Lemma 4.4.4** For \( t \in \mathcal{T}, \Delta V_{\leq k}(t_k) \leq m2^{k+2} \).

**Proof:** First we claim that at any moment \( t_k - \epsilon \), for any \( \epsilon > 0 \) small enough, the pool does not contain jobs whose class is at most \( k \). In case \( t_k = \hat{t} \), at any moment just before \( t_k \) there is at least one idle machine which means the pool is empty. Otherwise, \( t_k > \hat{t} \) and by definition we know that a job of class more than \( k \) is processed just before \( t_k \). Therefore, the pool does not contain any job whose class is at most \( k \).

At time \( t_k \) jobs of class at most \( k \) might arrive and fill the pool. However, these jobs increase both \( V_{\leq k}^{OPT}(t_k) \) and \( V_{\leq k}^A(t_k) \) by the same amount, so jobs that arrive exactly at \( t_k \) do not change \( \Delta V_{\leq k}(t_k) \) and can be ignored.

Since the jobs in the pool at time \( t_k \) can be ignored, we are left with the jobs in the stacks. Using Lemma 4.4.1, \( \Delta V_{\leq k}(t_k) \leq m(2^{k+1} + 2^k + 2^{k-1} + \ldots) \leq m2^{k+2} \).

**Lemma 4.4.5** For \( t \in \mathcal{T}, \Delta V_{\leq k}(t) \leq m2^{k+2} \).

**Proof:** Combining Lemma 4.4.3 and 4.4.4, we obtain \( \Delta V_{\leq k}(t) \leq \Delta V_{\leq k}(t_k) \leq m2^{k+2} \).
\[
\begin{align*}
\leq & \quad \frac{\Delta V_{<k_2}(t)}{2^{k_2}} + \sum_{i=k_1}^{k_2-1} \frac{\Delta V_{<i}(t)}{2^{i+1}} \\
& - \frac{\Delta V_{<k_1-1}(t)}{2^{k_1}} + 2\delta_{OPT}^{\geq k_1, \leq k_2}(t) \\
& \leq 4m + \sum_{i=k_1}^{k_2-1} 2m + \delta_{OPT}^{\leq k_1-1}(t) + 2\delta_{OPT}^{\geq k_1, \leq k_2}(t) \\
& \leq 2m(k_2 - k_1 + 2) + 2\delta_{OPT}^{\leq k_2}(t).
\end{align*}
\]

The first inequality follows since \(2^i\) is the minimum processing time of a job of class \(i\). The second inequality follows since the processing time of a job of class \(i\) is less than \(2^{i+1}\). The fourth inequality is derived by applying Lemma 4.4.5, observing that \(\Delta V_{<k_1-1}(t) \geq -V_{<k_1-1}^{OPT}(t)\) and that \(2^{k_1}\) is the maximum processing time of a job of class at most \(k_1 - 1\). The claim of the lemma then follows.

The following corollary of Lemma 4.4.6 is used in the proof of the \(O(\log P)\) approximation ratio of Theorem 4.4.1.

**Corollary 4.4.2** For \(t \in \mathcal{T}\), \(\delta^A(t) \leq 2m(4 + \log P) + 2\delta_{OPT}(t)\).

**Proof:** We express

\[
\delta^A(t) = \delta_{\leq k_{max}, \geq k_{min}}^A(t) + \delta_{<k_{min}}^A(t) \\
\leq 2m(k_{max} - k_{min} + 2) + 2\delta_{OPT}(t) + m \\
\leq 2m(4 + \log P) + 2\delta_{OPT}(t)
\]

The second inequality follows from the claim of Lemma 4.4.6 when \(k_2 = k_{max}\) and \(k_1 = k_{min}\), and from the claim of Lemma 4.4.1 stating that the stack of each machine contains at most one job of class less than \(k_{min}\). The third inequality is obtained since \(k_{max} - k_{min} + 5/2 \leq \log P + 4\).

**Theorem 4.4.1** \(F^A \leq 2(5 + \log P) \cdot F^{OPT}\), that is, algorithm \(A\) has a \(2(5 + \log P)\) approximation factor even compared to the optimal off-line algorithm that is allowed to migrate jobs.

**Proof:**

\[
F^A = \int_{t} \delta^A(t)dt \\
= \int_{t \in \mathcal{T}} \delta^A(t)dt + \int_{t \notin \mathcal{T}} \delta^A(t)dt
\]
\[ \leq \int_{t \notin T} \gamma^A(t)(2 + \log P)dt \]
\[ + \int_{t \in T} (2m(4 + \log P) + 2\delta^{OPT}(t))dt \]
\[ \leq (2 + \log P) \int_{t \notin T} \gamma^A(t)dt \]
\[ + 2(4 + \log P) \int_{t \in T} mdt + 2 \int_{t \in T} \delta^{OPT}(t)dt \]
\[ \leq (8 + 2 \log P) \int_{t} \gamma^A(t)dt + 2 \int_{t} \delta^{OPT}(t)dt \]
\[ \leq 2(5 + \log P) \cdot F^{OPT} \]

The first equality is from the definition of \( F^A \). The second is obtained by looking at the time in which none of the machines is idle and the time in which at least one machine is idle separately. The third inequality uses Lemma 4.1.2 and Corollary 4.1.2. The fifth inequality is true since \( \gamma^A(t) = m \) when \( t \in T \). Finally, \( \int_{t} \gamma^A(t)dt \) is the total time spent processing jobs by the machines which is exactly \( \sum_{j \in J} w_j \). That sum is upper bounded by the flow time of \( OPT \) since each job’s flow time must be at least its processing time.

\[ \blacksquare \]

We now turn to prove the \( O(\log n) \) approximation ratio of the algorithm. A different argument is required to prove this second bound. The main idea behind the proof of the \( O(\log P) \) approximation ratio was to bound for any time \( t \in T \), the alive jobs difference between \( A \) and \( OPT \) by \( O(m \log P) \). A similar approach does not allow us to prove the \( O(\log n) \) approximation ratio: It is possible to construct instances where the the alive jobs difference is \( \Omega(n) \).

Leonardi and Raz \textsuperscript{[51]} proved the \( O(\log(n/m)) \) approximation ratio for SRPT when migration is allowed arguing that the worst case ratio between the SRPT flow time and the optimal flow time can be raised only if a “big” alive jobs difference is kept for a “long” time period. This observation holds also for our non-migratory algorithm. This is formally stated in Lemma 4.4.7 for any time \( t \in T \) when no machine is idle and in Lemma 4.4.8. These lemmas prove that the minimum remaining processing time of a set of unfinished jobs is exponentially decreasing with the size of the alive jobs difference between \( A \) and \( OPT \). Thus, either new jobs are released at a rate exponential in the size of the alive jobs difference, or the ongoing processed jobs are consumed and the alive jobs difference is reduced.

We need to introduce more notation. Recall that \( T \) is defined to be the set of times in which \( \gamma^A(t) = m \). We denote by \( T = \int_{t \in T} dt \) the size of set \( T \). For any \( t \in T \), define by \( \delta^{A,P}(t) \) the number of jobs in the pool of algorithm \( A \), i.e., not assigned to a machine, at time \( t \), and by \( \Delta \delta^P(t) = \delta^{A,P}(t) - 2\delta^{OPT}(t) \) the difference between the number of jobs in the pool of algorithm \( A \) and twice the number of jobs not finished by the optimal algorithm. For any machine \( l \), time \( t \), define by \( \delta^{A,l}(t) \) the number of jobs assigned to machine \( l \) at
time $t$ in the schedule of algorithm $A$. Moreover, define by $\mathcal{T} = \{ t | \delta^A_I(t) > 0 \}$, the set of times when machine $l$ is assigned with at least one job, and by $\mathcal{T}^t = \int_{t \in \mathcal{T}} dt$ the size of set $\mathcal{T}^t$.

**Lemma 4.4.7** For any time $t \in \mathcal{T}$, if $\Delta \delta^P(t) \geq 2m$, for $i \geq 3$, then the pool of algorithm $A$ contains at least $2m$ jobs of remaining processing time at most $\frac{V^A(t)}{m^2}$.

**Proof:** Let $k_{\text{high}}$ be the maximum integer such that $\delta^A_{\geq k_{\text{high}}}(t) \geq 2m$ and let $k_{\text{low}}$ be the maximum integer such that $\delta^A_{< k_{\text{low}}}(t) < 2m$. Note that both numbers are well defined and $k_{\text{low}} \leq k_{\text{high}}$ since there are at least $6m > 2m$ jobs in the pool. Then,

$$2m \leq \delta^A_{\geq k_{\text{high}}}(t) \leq \delta^A_{k_{\text{high}}}(t) \leq \frac{V^A(t)}{2^{k_{\text{high}}}}$$

thus yielding $2^{k_{\text{high}}} \leq \frac{V^A(t)}{2m}$. In particular, the last inequality follows since $2^{k_{\text{high}}}$ is the minimum processing time of a job of class $k_{\text{high}}$.

By the definition of $k_{\text{high}}$, we have:

$$\Delta \delta^P_{\leq k_{\text{high}}}(t) = \delta^A_{\leq k_{\text{high}}}(t) - 2\delta^P_{\leq k_{\text{high}}}(t)$$

$$= \delta^A_{k_{\text{high}}}(t) - \delta^A_{> k_{\text{high}}}(t) - 2(\delta^P_{\leq k_{\text{high}}}(t) - \delta^P_{> k_{\text{high}}}(t))$$

$$= \Delta \delta^P(t) - \delta^A_{> k_{\text{high}}}(t) + 2\delta^P_{> k_{\text{high}}}(t)$$

$$\geq 2m(i - 1).$$

where the last inequality follows since $\delta^A_{> k_{\text{high}}}(t) < 2m$.

From Lemma 4.4.6, we get:

$$\Delta \delta^A_{\leq k_{\text{high}}}(t) = \delta^A_{\leq k_{\text{high}} - k_{\text{low}}}(t) + \delta^A_{< k_{\text{low}}}(t) - 2\delta^P_{\leq k_{\text{high}}}(t)$$

$$\leq \delta^A_{\leq k_{\text{high}} - k_{\text{low}}}(t) + \delta^A_{< k_{\text{low}}}(t) - 2\delta^P_{\leq k_{\text{high}}}(t)$$

$$\leq \delta^A_{< k_{\text{low}}}(t) + 2m(k_{\text{high}} - k_{\text{low}} + 2),$$

thus yielding $\delta^A_{< k_{\text{low}}}(t) \geq 2m(i + k_{\text{low}} - k_{\text{high}} - 3)$. Therefore, we get that $2m > 2m(i + k_{\text{low}} - k_{\text{high}} - 3)$ and thus $k_{\text{low}} \leq k_{\text{high}} - i + 3$. It follows that there exist at least $2m$ jobs of class at most $k_{\text{low}} \leq k_{\text{high}} - i + 3$ in the pool of the algorithm. The remaining processing time of these $2m$ jobs is bounded by $2^{k_{\text{high}} - i + 4} \leq \frac{V^A(t)}{m^2}$, thus proving the claim. ■
Lemma 4.4.8 For any machine \( l \), time \( t \in T^l \), if \( \delta^{A_l}(t) \geq i \) for \( i \geq 1 \) then there exists a job with remaining processing time at most \( \frac{T^l}{2^{i+1}} \) assigned to machine \( l \) at time \( t \).

Proof: For any time \( t \in T^l \), there is at most one job assigned to machine \( l \) for every specific class. Assume machine \( l \) is assigned with a job of highest class \( k \), obviously satisfying \( T^l \geq 2^k \). If \( \delta^{A_l}(t) \geq i \) then there is a job of class at most \( k - i + 1 \) assigned to machine \( l \), with processing time at most \( 2^{k-i+2} \leq \frac{T^l}{2^{i+1}} \).

We partition the set of time instants \( T \) when no machine is idle into a collection of disjoint intervals \( I_k = [t_k, r_k) \), \( k = 1, \ldots, s \), and associate an integer \( i_k \) with each interval, such that for any time \( t \in I_k \), \( 2m(i_k - 1) < \Delta \delta^P(t) < 2m(i_k + 2) \).

Each maximal interval of times \([t_b, t_e]\) contained in \( T \) is dealt with separately. Assume we already dealt with all times in \( T \) which are smaller than \( t_b \), and we have created \( k - 1 \) intervals. We then define \( t_k = t_b \). Given \( t_k \) we choose \( i_k = \left\lfloor \frac{\Delta \delta^P(t_k)}{2m} \right\rfloor \). Given an interval’s \( t_k \) and \( i_k \), we define \( r_k = \min \{ t_e, t_t > t_k, \Delta \delta^P(t) \geq 2m(i_k + 2) \text{ or } \Delta \delta^P(t) \leq 2m(i_k - 1) \} \), that is, \( r_k \) is the first time \( \Delta \delta^P(t) \) reaches the value \( 2m(i_k + 2) \), or the value \( 2m(i_k - 1) \). As long as \( r_k < t_e \), we continue with the next interval beginning at \( t_{k+1} = r_k \).

Observation 1 When an interval \( k \) begins, \( 2mi_k \leq \Delta \delta^P(t_k) < 2m(i_k + 1) \). When it ends, either \( \Delta \delta^P(r_k) \leq 2m(i_k - 1) \), \( \Delta \delta^P(r_k) \geq 2m(i_k + 2) \) or \( \Delta \delta^P(r_k) \leq 0 \).

Proof: The first part is clear by the way we choose \( i_k \). The second part is clear when \( r_k \) is not equal to some \( t_e \), that is, \( t_k \in T \). Otherwise, \( r_k \notin T \) and therefore \( \delta^{A,P}(r_k) = 0 \) and \( \Delta \delta^P(r_k) \leq 0 \).

Denote by \( x_k = r_k - t_k \) the size of interval \( I_k \), and define \( T_i = \{ \cup I_k | i_k = i \} \), \( i \geq 1 \), as the union of the intervals \( I_k \) with \( i_k = i \). We indicate by \( T_i = \int_{t \in T_i} dt \) the size of set \( T_i \). We also denote by \( D = \max \{ T, \max_{t \in T} \{ V^A(t)/m \} \} \). The following lemma relates the number of jobs, \( n \), and the values of \( T_i \).

Lemma 4.4.9 The following lower bound holds for the number of jobs:

\[
n \geq \frac{m}{8D} \sum_{i \geq 4} T_i 2^{i-5}.
\]

Proof: Consider an interval \( I_k \), with a corresponding \( i_k \geq 4 \). According to Observation 1, when the interval starts \( \Delta \delta^P(t_k) \) is between \( 2mi_k \) and \( 2m(i_k + 1) \). This interval ends when \( \Delta \delta^P(r_k) \) goes above \( 2m(i_k + 2) \) or below \( 2m(i_k - 1) \) (it might also reach 0 but \( 0 < 2m(i_k - 1) \)). In the first case we have the evidence of at least \( m \) jobs finished by \( OPT \) (recall that \( \Delta \delta^P(t) = \delta^{A,P}(t) - 2\delta^{OPT}(t) \)). In the second case we have the evidence of at least \( 2m \) jobs
that either leave the pool to be assigned to a machine by algorithm $A$ or arrive to both $A$ and $OPT$. In both cases we charge $n_k \geq m$ jobs to interval $I_k$. We can then conclude with a first lower bound $n_k \geq m$ on the number of jobs charged to any interval $I_k \in \mathcal{T}_i, i_k \geq 4$.

Next, we give a second lower bound, based on Lemma 4.4.7, stating that during an interval $I_k = [t_k, r_k)$ there exist in the pool $2m$ jobs with remaining processing time at most $\frac{D}{2^{k-\tau}}$, since $\Delta \delta^E(t) > 2m(i_k - 1)$ for any $t \in [t_k, r_k)$. This implies that all the $m$ machines are processing jobs with remaining processing time at most $\frac{D}{2^{k-\tau}}$. We look at any subinterval of $I_k$ of length $\frac{D}{2^{k-\tau}}$. For each machine, during this subinterval, either a job is finished by the algorithm or a job is preempted by a job of lower class. Therefore, we can charge at least $m$ jobs that are either released or finished with any subinterval of size $\frac{D}{2^{k-\tau}}$ of $I_k$. A second lower bound on the number of jobs charged to any interval is then given by $n_k \geq m \lceil \frac{x_k 2^{2k-5}}{D} \rceil$.

Observe now that each job is charged at most 4 times, when it is released, when it is assigned to a machine by $A$, when it is finished by $A$ and when it is finished by $OPT$. Then,

$$\frac{m}{2D} \sum_{i \geq 4} T_i 2^{i-5} \leq \sum_{k \mid i_k \geq 4} m \max \{1, \left\lceil \frac{x_k 2^{i_k-5}}{D} \right\rceil \} \leq 4n,$$

where the first inequality is obtained by summing over $I_k$’s instead of over $T_i$’s and the simple fact that $\alpha \leq \max \{1, \lceil 2\alpha \rceil \}$ and the second inequality follows from the lower bounds we have shown on the charged jobs. The lemma easily follows.

We next bound the number of jobs that are assigned to a machine $l$ during the time instants of $\mathcal{T}^l$. We partition the set of time instants $\mathcal{T}^l$ into a set of disjoint intervals $I_k^l = [t_k^l, r_k^l)$, $k = 1, \ldots , s^l$, and associate an integer $d_k^l$ with each interval, such that for any time $t \in I_k^l$, $\delta^A(t) = i_k^l$. Consider a maximal interval of times $[t^l_0, t^l_e)$ contained in $\mathcal{T}^l$. Assume $t^l_0 = t^l_0$. Given $t^l_0$ and $d_k^l$, define $r_k^l = \min \{t^l_e, t \not\in [t^l_0, t^l_e), \delta^A(t) \neq d_k^l \}$. In case $r_k^l < t^l_0$, we set $t^l_{k+1} = r_k^l$. Denote by $x_k^l = r_k^l - t_k^l$ the size of interval $I_k^l$. Define by $\mathcal{T}^l_i = \{ \cup I_k^l \mid I_k^l = i \}$, $i \geq 1$, the union of the intervals $I_k^l$ when the number of jobs assigned to machine $l$ is exactly $i$, and by $T_i^l = \int_{t \in \mathcal{T}^l_i} dt$ the size of set $\mathcal{T}^l_i$.

**Lemma 4.4.10** The following lower bound holds for the number of jobs assigned to machine $l$:

$$n^l \geq \frac{1}{4T^l} \sum_{i \geq 1} T_i^l 2^{i-2}.$$

**Proof:** We will proceed charging at least one job with every interval $I_k^l$. Every job will be charged at most twice, when it is assigned to a machine by $A$ and when it is finished by $A$. 
4.4. ANALYSIS

Consider the generic interval $I_k^l$ with the corresponding $\delta^l_k$. The interval starts when $\delta^A_l(t_k)$ reaches $i^l_k$, from above or from below. The interval ends when $\delta^A_l(r_k)$ reaches $i^l_k + 1$ or $i^l_k - 1$. In the first case we have the evidence of one job that is assigned to a machine by $A$ and in the second case of one job that is finished by $A$. In both cases we charge one job to interval $I_k^l$.

Next we give a second lower bound, based on Lemma 4.4.8. Lemma 4.4.8 states that during an interval $I_k^l = [t_k^l, r_k^l]$, machine $l$ is constantly assigned with a job of remaining processing time at most $\frac{\alpha^l}{2^l_j}$. We look at any subinterval of $I_k^l$ of length $\frac{T^l}{2^l_j}$. During this subinterval, either a job is finished by the algorithm or a job is preempted by a job of a lower class. In any case, a job that is assigned or finished during any subinterval of size $\frac{T^l}{2^l_j}$ is charged. A second lower bound on the number of jobs charged to any interval is then given by $n_k \geq \left\lfloor \frac{x^l_j 2^l_j - 2}{T^l} \right\rfloor$.

Observe now that each job is considered at most twice, when it is assigned to machine $l$ and when it is finished by $A$. Then, from the following inequalities:

$$\frac{1}{2T^l} \sum_{i \geq 1} T^l_i 2^{-i} \leq \sum_{k \mid k^l_i \geq 1} \max \{1, \left\lfloor \frac{x^l_j 2^l_j - 2}{T^l} \right\rfloor \} \leq 2n^l,$$

the claim follows.

Before completing the proof, we still need a simple mathematical lemma:

**Lemma 4.4.11** Given a sequence $a_1, a_2, \ldots$ of non-negative numbers such that $\sum_{i \geq 1} a_i \leq A$ and $\sum_{i \geq 1} 2^i a_i \leq B$ then $\sum_{i \geq 1} i a_i \leq \log(4B/A)A$.

**Proof:** Define a second sequence, $b_i = \sum_{j \geq i} a_j$ for $i \geq 1$. Then it is known that $A \geq b_1 \geq b_2 \geq \ldots$. Also, it is known that $\sum_{i \geq 1} 2^i (b_i - b_{i+1}) \leq B$. This implies that $\sum_{i \geq 1} 2^i b_i \leq 2B$.

The sum we are trying to upper bound is $\sum_{i \geq 1} b_i$. It is upper bounded by assigning $b_i = A$ for $1 \leq i \leq k$ and $b_i = 0$ for $i > k$ where $k$ is large enough such that $\sum_{i \geq 1} 2^i b_i \geq 2B$. A choice of $k = \lceil \log(2B/A) \rceil$ is adequate and the sum is upper bounded by $kA$ from which the result follows.

**Theorem 4.4.2** $F^A = O(\log n)F^{OPT}$, that is, algorithm $A$ has an $O(\log n)$ approximation ratio even compared with the optimal off-line algorithm that is allowed to migrate jobs.

**Proof:**

$$F^A = \int_0^\infty \delta^A(t)dt$$
\[ F \leq \sum_{l=1}^{m} \int_{T_i} \delta^{A} (t) \, dt + \int_{t \in T} \delta^{A,P} (t) \, dt \]
\[ = \sum_{l=1}^{m} \int_{T_i} \delta^{A} (t) \, dt \]
\[ + \int_{t \in T} (2\delta^{OPT} (t) + \Delta \delta^{P} (t)) \, dt \]
\[ \leq \sum_{l=1}^{m} \int_{T_i} \delta^{A} (t) \, dt + 2\delta^{OPT} \]
\[ + \sum_{i} \int_{t \in T_i} 2m (i + 2) \, dt \]
\[ \leq \sum_{l=1}^{m} \sum_{i \geq 1} i \cdot T_i^l + 2\delta^{OPT} + 2m \sum_{i} (i + 2) \cdot T_i \]
\[ = \sum_{l=1}^{m} \sum_{i \geq 1} i \cdot T_i^l + 2\delta^{OPT} + 2m \sum_{i} (i - 3) \cdot T_i \]
\[ + 2m \sum_{i} 5 \cdot T_i \]
\[ \leq \sum_{l=1}^{m} \sum_{i \geq 1} i \cdot T_i^l + 2\delta^{OPT} + 2m \sum_{i \geq 1} (i - 3) \cdot T_i \]
\[ + 10\delta^{OPT} \]
\[ \leq \sum_{l=1}^{m} \sum_{i \geq 1} i \cdot T_i^l + 12\delta^{OPT} + 2m \sum_{i \geq 1} i \cdot T_i+3 \]

The second equality is obtained by separately considering for any machine \( l \) the contribution to the flow time due to the jobs assigned to machine \( l \), and the contribution to the flow due to jobs in the pool. The fourth inequality is obtained by partitioning the pool \( \mathcal{T} \) into the \( T_i \)'s, such that at any time \( t \in T_i \), \( \Delta \delta^{P} (t) < 2m (i + 2) \). The seventh inequality is obtained by observing that \( m \cdot \sum \delta^{P} (t) \leq \sum j \cdot w_j \leq \delta^{OPT} \), since all machines are busy processing jobs at any time \( t \in \mathcal{T} \).

We upper bound \( \sum_{i \geq 1} i \cdot T_i+3 \) under the constraint \( \sum_{i \geq 1} T_i+3 \leq T \leq D \), and the constraint on \( n \) given by Lemma 4.4.9. By choosing \( a_i = T_i+3 \) we see that \( \sum_{i \geq 1} a_i \leq D \) and \( \sum_{i \geq 1} 2^i a_i \leq 32 \frac{D}{m} \). These two bounds together with Lemma 4.4.11 result in the upper bound \( \sum_{i \geq 1} i \cdot a_i \leq (7 + \log \frac{n}{m})D \). A similar argument is used for the other sum. We choose \( a_i = T_i^l \) instead. First note that \( \sum_{i \geq 1} a_i \leq T^l \). Then, according to Lemma 4.4.10, \( \sum_{i \geq 1} 2^i a_i \leq 16n^l T^l \). The upper bound by Lemma 4.4.11 is \( \sum_{i \geq 1} i \cdot a_i \leq (6 + \log n^l)T^l \).

We finally express the total flow time of algorithm \( A \) as:

\[ F^A \leq \sum_{l=1}^{m} \sum_{i \geq 1} i \cdot T_i^l + 12\delta^{OPT} + 2m \sum_{i \geq 1} i \cdot T_i+3 \]
4.5. TIGHTNESS OF THE ANALYSIS

\[
\sum_{l=1}^{m} O(T^l \log n^l) + 12F^{OPT} + O(mD \log \frac{n}{m}) \\
= O(\log n) \sum_{l=1}^{m} T^l + 12F^{OPT} + O(\log \frac{n}{m})F^{OPT} \\
= O(\log n)F^{OPT}.
\]

The third equality follows since \(mD \leq \max\{mT, \max_{t \in T} \{V^A(t)\}\} \leq \sum_{j \in J} w_j \leq F^{OPT}\). The fourth equality follows since \(\sum_{l=1}^{m} T^l = \sum_{j} w_j \leq F^{OPT}\). \(\blacksquare\)

4.5 Tightness of the Analysis

In this section we present an \(\Omega(\log n)\) lower bound on the competitive ratio of the algorithm analyzed in the previous section. The number of machines is an arbitrary \(m\) but note that the lower bound only uses the first two machines. While presenting the lower bound we assume an order between jobs with the same release time. The algorithm deals with jobs released at the same time following the specified order. Moreover, we assume that the algorithm can schedule a job on a machine processing a job of a higher class even if there is an idle machine. These assumptions agree with the definition of the algorithm.

Let \(P\) be a power of 2. The maximum processing time of a job is \(2P\) and the minimum is 1. At each time \(r_i = P(2 - \sum_{j=i}^{\frac{1}{2}\log P - 1} \frac{1}{2^{j+1}})\), for \(i = 0, \ldots, \frac{1}{2}\log P - 1\), three jobs are released; one job of processing time \(\frac{P}{2^n}\) followed by two jobs of processing time \(\frac{P}{2^{n+1}}\). Finally, two jobs of size 1 are released every time unit between time \(2P\) and \(4P - 1\).

Let us consider the behaviour of the algorithm and of the optimal solution on this instance. At time \(r_0\) the released job of processing time \(P\) is assigned to machine 1, a job of processing time \(P/2\) is assigned to machine 2 while the second job of processing time \(P/2\) preempts the job on execution on machine 1. The two jobs of processing time \(P/2\) are finished at time \(r_0 + P/2\) where the job of size \(P\) is again executed on machine 1. In general, at time \(r_i\), \(i = 1, \ldots, \frac{1}{2}\log P\), \(i\) jobs are in the queue of machine 1, the smallest of remaining processing time \(\frac{P}{2^n}\), while no job is on execution on machine 2. The job on execution on machine 1 is preempted by the job of size \(\frac{P}{2^n}\) that is released at time \(r_i\). Immediately afterwards it is preempted by a job of size \(\frac{P}{2^{n+1}}\) that is released at time \(r_i\) while the other job of size \(\frac{P}{2^n}\) released at time \(r_i\) is scheduled on machine 2.

At time \(2P\), \(\frac{1}{2}\log P\) jobs are in the queue of machine 1, the smallest of size 2, while no job is in the pool or assigned to the other machines. The two jobs of size 1 released every time unit between time \(2P\) and \(4P - 1\) are scheduled on machines 1 and 2. Observe that \(\frac{1}{2}\log P\) jobs are waiting on queue 1 between time \(2P\) and \(4P\) thus leading to a flow time of \(\Omega(P \log P)\) for the algorithm on this instance of \(n = \frac{3}{2}\log P + 4P\) jobs.

On the other hand, an optimal solution schedules the job of size \(\frac{P}{2^n}\) released at time \(r_i\)
for \( i = 0, \ldots, \frac{1}{2} \log P - 1 \) on machine 1 and the two jobs of size \( \frac{P}{2^{n+1}} \) released at time \( r_i \) on machine 2. Observe that in this schedule machines 1 and 2 are idle at time \( r_i \). Thus, the jobs of size 1 are scheduled at their release times on machines 1 and 2 and are completed before the next two jobs of size 1 are released. The flow time of the optimal solution is then \( O(P) \).

It follows that the ratio between the flow time of the algorithm and the flow time of the optimal solution is \( \Omega(\log P) \) that is also \( \Omega(\log n) \).

### 4.6 Discussion

In our algorithm jobs are kept in a pool after their release time until they are assigned to a machine. An interesting open problem is to devise an efficient algorithm that assigns jobs to machines at the time of release. Another challenging open problem is to devise a constant off-line approximation algorithm for optimizing total flow time even if both preemption and migration are allowed.
Chapter 5

Benefit Maximization for Online Scheduling

5.1 Introduction

In this chapter we consider the benefit model for on-line preemptive scheduling. In this model jobs arrive to the on-line scheduler at their release time. Each job arrives with its own processing time and its benefit function. Recall that the flow time of a job is the time that passes from its release to its completion. The benefit function specifies the benefit gained for any given flow time: the longer the processing of a job takes, the lower its benefit is. A scheduler’s goal is to maximize the total gained benefit. We present a constant competitive ratio algorithm for that model in the uniprocessor case for benefit functions that do not decrease too rapidly. We also extend the algorithm to the multiprocessor case while maintaining constant competitiveness. The multiprocessor algorithm does not use migration, i.e., preempted jobs continue their execution on the same processor on which they were originally processed.

5.2 Preliminaries

We use the notation described in the introduction for the on-line preemptive scheduling model. In this chapter, each job $j$, in addition to its arrival and processing time, has an arbitrary monotone non-increasing non-negative benefit density function $B_j(t)$ for $t \geq w_j$, and the benefit gained is $w_jB_j(f_j)$, where $f_j$ is its flow time. Note that the benefit density function may be different for each job. The goal of the scheduler is to schedule the jobs so as to maximize the total benefit, i.e., $\sum_j w_jB_j(f_j)$. Note that the benefit density function of different jobs can be uncorrelated and the ratio between their values can be arbitrarily
large. We do, however, restrict each $B_j(t)$ to satisfy

$$\frac{B_j(t)}{B_j(t + w_j)} \leq C$$

for some fixed constant $C$. That is, if we delay a job by its processing time then we lose only a constant factor in its benefit.

## 5.3 The Algorithm

The basic idea of the algorithm is to schedule a job whose current benefit density is as high as possible. The problem with such an algorithm is that it may preempt jobs in order to gain a small improvement in the benefit density and hence delay a large number of jobs. To overcome this problem we schedule a new job only if its benefit density is significantly higher than that of the current job. In addition, we prefer partially processed jobs to non-processed jobs of similar benefit density. The algorithm combines the above ideas and is formally described below.

We begin by defining three 'storage' locations for jobs. The first is the pool where new jobs arrive and stay until their processing begins. Once the scheduler decides a job should begin running, the job is removed from the pool and pushed onto the stack where its processing begins. Two different possibilities exist at the end of a job's life cycle. The first is a job that is completed and can be popped from the stack. The second is a job that after remaining too long in the stack got thrown into the garbage collection. The garbage collection holds jobs whose processing we prefer to defer. The actual processing will only occur when the system reaches an idle state. However, in the analysis we assume that a job thrown into the garbage collection is never executed and we gain nothing from it.

The job at the top of the stack is the job that is currently running. The other jobs in the stack are preempted jobs. For each job $j$, denote by $s_j$ the time it enters the stack. We define its breakpoint as the time $s_j + 2w_j$. If a job is still running when it reaches its breakpoint, it is thrown into the garbage collection. We also define priorities for each job in the pool and in the stack. The priority of job $j$ at time $t$ is denoted by $d_j(t)$. For $t \leq s_j$, it is $B_j(t + w_j - r_j)$ and for time $t > s_j$, it is $d_j = B_j(s_j + w_j - r_j)$. In other words, the priority of a job in the pool is its benefit density if it would have run to completion starting at the current time $t$. Once it enters the stack its priority becomes fixed, i.e. remains the priority at time $s_j$.

We describe Algorithm ALG1 as an event-driven algorithm. The algorithm takes action at time $t$ when a new job is released, when the currently running job is completed or when the currently running job reaches its breakpoint. If some events happen at the same time we handle the completion of jobs first.

- A new job $l$ arrives. If $d_l(t) > d_k$, where $k$ is the job at the top of the stack or if the stack is empty, push job $l$ into the stack and run it. Otherwise, just add job $l$ to the
pool.

- The job at the top of the stack is completed or reaches its breakpoint. Then, pop jobs from the top of the stack and insert them into the garbage collection as long as their breakpoints have been reached. Unless the stack is empty, let $k$ be the index of the new job at the top of the stack. Continue running job $k$ only if $d_j(t) \leq 4d_k$ for all $j$ in the pool. Otherwise, get the job from the pool with maximum $d_j(t)$, push it into the stack, and run it.

- Whenever the machine is idle (i.e., no jobs in the stack or in the pool) run any uncompleted job from the garbage collection until a new job arrives.

We note several facts about this algorithm:

**Observation 2** Every job enters the stack at some point in time. Then, by time $s_j + 2w_j$, it is either completed or reaches its breakpoint and gets thrown into the garbage collection.

**Observation 3** The priority of a job is monotone non-increasing over time. Once the job enters a stack, its priority remains fixed until it is completed or thrown away. At anytime the priority of each job in a stack is at least 4 times higher than the priority of the job below it.

**Observation 4** Whenever the pool is not empty, the machine is not idle, that is, the stack is not empty. Moreover, the priority of jobs in the pool is always at most 4 times higher than the priority of the currently running job.

### 5.4 The Analysis

We begin by fixing an input sequence and hence the behavior of the optimal algorithm and the on-line algorithm. We denote by $f_j^{OPT}$ the flow time of job $j$ by the optimal algorithm. As for the on-line algorithm, we only consider the benefit of jobs which were not thrown into the garbage collection. Denote the set of these jobs by $A$. So, for $j \in A$, let $f_j^{ON}$ be the flow time of job $j$ by the on-line algorithm. By definition,

$$V^{OPT} = \sum_j w_j B_j(f_j^{OPT})$$

and

$$V^{ON} \geq \sum_{j \in A} w_j B_j(f_j^{ON}).$$

We also define the pseudo-benefit of a job $j$ by $w_j \hat{d}_j$. That is, each job donates a benefit of $w_j \hat{d}_j$ as if it runs to completion without interruption from the moment it enters the stack.
Define the pseudo-benefit of the online algorithm as
\[ V^{PSEUDO} = \sum_j w_j \hat{d}_j. \]

For \( 0 \leq t < w_j \), we define \( B_j(t) = B_j(w_j) \). In addition, we partition the set of jobs \( J \) into two sets, \( J_1 \) and \( J_2 \). The first is the set of jobs which are still processed by the optimal scheduler at time \( s_j \), when they enter the stack. The second is the set of jobs which have been completed by the optimal scheduler before they enter the stack.

**Lemma 5.4.1** For the set \( J_1 \), \( \sum_{j \in J_1} w_j B_j(f_j^{OPT}) \leq C \cdot V^{PSEUDO} \).

**Proof:** We note the following:
\[ w_j B_j(f_j^{OPT}) \leq C \cdot w_j B_j(f_j^{OPT} + w_j) \leq C \cdot w_j B_j(s_j - r_j + w_j) = C \cdot w_j \hat{d}_j \]
where the first inequality is by our assumptions on \( B_j \) and the second is by our definition of \( J_1 \). Summing over jobs in \( J_1 \), we have
\[ \sum_{j \in J_1} w_j B_j(f_j^{OPT}) \leq C \sum_{j \in J_1} w_j \hat{d}_j \leq C \cdot V^{PSEUDO}. \]

**Lemma 5.4.2** For the set \( J_2 \), \( \sum_{j \in J_2} w_j B_j(f_j^{OPT}) \leq 4C \cdot V^{PSEUDO} \).

**Proof:** For each \( j \in J_2 \), we define its ‘optimal processing times’ as
\[ \tau_j = \{ t | \text{job} \ j \ \text{is processed by OPT at time} \ t \}. \]

\[
\sum_{j \in J_2} w_j B_j(f_j^{OPT}) = \sum_{j \in J_2} \int_{t \in \tau_j} B_j(f_j^{OPT}) dt \\
\leq \sum_{j \in J_2} \int_{t \in \tau_j} B_j(t - r_j) dt \\
\leq C \cdot \sum_{j \in J_2} \int_{t \in \tau_j} d_j(t) dt.
\]

According to the definition of \( J_2 \), during the processing of job \( j \in J_2 \) by the optimal algorithm, the on-line algorithm still keeps the job in its pool. By Observation 4 we know that the job’s priority is not too high; it is at most 4 times the priority of the currently running job and, specifically, at time \( t \in \tau_j \), its priority is at most 4 times the priority of
the job at the top of the stack in the on-line algorithm. Denote that job by $j(t)$. So,

$$C \cdot \sum_{j \in J} \int_{t \in \tau_j} d_j(t) dt \leq 4C \cdot \sum_{j \in J} \int_{t \in \tau_j} \hat{d}_j(t) dt$$

$$\leq 4C \cdot \int_{t \in \tau_j} \hat{d}_j(t) dt$$

$$\leq 4C \cdot \int_{t} \hat{d}_j(t) dt$$

$$\leq 4C \cdot \sum_{j \in J} w_j \hat{d}_j = 4C \cdot V^{PSEUDO}.$$

\[\Box\]

**Corollary 5.4.1** $V^{OPT} \leq 5C V^{PSEUDO}$.

**Proof:** Combining the two lemmas we get,

$$V^{OPT} = \sum_{j \in J} w_j B_j(f_j^{OPT}) + \sum_{j \in J} w_j B_j(f_j^{OPT})$$

$$\leq C \cdot V^{PSEUDO} + 4C \cdot V^{PSEUDO}$$

$$= 5C V^{PSEUDO}.$$

\[\Box\]

**Lemma 5.4.3** $V^{PSEUDO} \leq 2C \cdot V^{ON}$

**Proof:** We show a way to divide a benefit of $C \cdot V^{ON}$ between all the jobs such that the ratio between the gain allocated to each job and its pseudo-gain is at most 2.

We begin by ordering the jobs so that jobs are preempted only by jobs appearing earlier in the order. This is done by looking at the preemption graph: each node represents a job and the directed edge $(j, k)$ indicates that job $j$ preempts job $k$ at some time in the on-line algorithm. This graph is acyclic since the edge $(j, k)$ exists only if $\hat{d}_j > \hat{d}_k$. We use a topological order of this graph in our construction. Jobs can only be preempted by jobs appearing earlier in this order.

We begin by assigning a benefit of $w_j \hat{d}_j$ to any job $j$ in $A$, the set of jobs not thrown into the garbage collection. At the end of the process the benefit allocated to each job, not necessarily in $A$, will be at least $\frac{1}{2} w_j \hat{d}_j$.

According to the order defined above, we consider one job at a time. Assume we arrive at job $j$. When $j \in A$, it already has a benefit of $w_j \hat{d}_j$ assigned to it. Otherwise, job $j$ gets thrown into the garbage collection. This job enters the stack at time $s_j$ and leaves it at time $s_j + 2w_j$. During that time the scheduler actually processes the job for less than $w_j$
time. So, job \( j \) is preempted for more than \( w_j \) time. For any job \( k \) running while job \( j \) is preempted, we denote by \( U_{k,j} \) the set of times when job \( j \) is preempted by job \( k \). Then, we move a benefit of \( |U_{k,j}| \cdot \hat{d}_j \) from \( k \) to \( j \). Therefore, once we finish with job \( j \), its allocated benefit is at least \( w_j \hat{d}_j \).

How much benefit is allocated to each job \( j \) at the end of the process? We have seen that before moving on to the next job, the benefit allocated to job \( j \) is at least \( w_j \hat{d}_j \) (whether or not \( j \in A \)). When job \( j \) enters the stack at time \( s_j \) it preempts several jobs; these jobs appear later in the order. Since jobs are added and removed only from the top of the stack, as long as job \( j \) is in the stack, the set of jobs preempted by it remains unchanged. Each job \( k \) of this set gets a benefit of at most \( w_j \hat{d}_k \) from \( j \). However, since all of these jobs exist together with \( j \) in the stack at time \( s_j \), the sum of their priorities is at most \( \frac{1}{2} \hat{d}_j \) (according to Observation 3). So, after moving all the required benefit, job \( j \) is left with at least \( \frac{1}{2} w_j \hat{d}_j \), as needed.

In order to complete the proof,

\[
V^{\text{PSEUDO}} = \sum_j w_j \hat{d}_j = 2 \sum_j \frac{1}{2} w_j \hat{d}_j \\
\leq 2 \sum_{j \in A} w_j \hat{d}_j \\
\leq 2C \sum_{j \in A} w_j B_j (s_j - r_j + 2w_j) \\
\leq 2C \sum_{j \in A} w_j B_j (f_j^{ON}) \\
\leq 2C \cdot V^{ON}.
\]

\[\blacksquare\]

**Theorem 5.4.1** Algorithm ALG1 is \( 10C^2 \) competitive.

**Proof:** By combining the previous lemmas, we conclude that

\[
V^{ON} \geq \frac{V^{\text{PSEUDO}}}{2C} \geq \frac{V^{OPT}}{10C^2}.
\]

\[\blacksquare\]

### 5.5 Multiprocessor Scheduling

We extend Algorithm ALG1 to the multiprocessor model. In this model, the algorithm holds \( m \) stacks, one for each machine, as well as \( m \) garbage collections. Jobs not completed
by their deadline get thrown into the corresponding garbage collection. Their processing
can continue later when the machine is idle. As before, we assume we get no benefit from
these jobs. The multiprocessor Algorithm ALG2 is as follows:

- A new job $l$ arrives. If there is a machine such that $d_l(t) > 4d_k$ where $k$ is the job
at the top of its stack or its stack is empty, push job $l$ into that stack and run it.
Otherwise, just add job $l$ to the pool.

- The job at the top of a stack is completed or reaches its breakpoint. Then, pop jobs
from the top of that stack as long as their breakpoints have been reached. Unless the
stack is empty, let $k$ be the index of the new job at the top of the stack. Continue
running job $k$ only if $d_j(t) \leq 4d_k$ for all $j$ in the pool. Otherwise, get the job from
the pool with maximum $d'_j(t)$, push it into that stack, and run it.

- Whenever a machine is idle (i.e., no jobs in its stack or in the pool) run any uncom-
pleted job from its garbage collection until a new job arrives.

We define $J_1$ and $J_2$ in exactly the same way as in the uniprocessor case.

**Lemma 5.5.1** For the set $J_1$, $\sum_{j \in J_1} w_j B_j(f_j^{OPT}) \leq C \cdot V^{PSEUDO}$.

**Proof:** Since the proof of Lemma 5.4.1 used the definition of $J_1$ separately for each job, it
remains true in the multiprocessor case as well. \qed

The following lemma extends Lemma 5.4.2 to the multiprocessor case:

**Lemma 5.5.2** For the set $J_2$, $\sum_{j \in J_2} w_j B_j(f_j^{OPT}) \leq 4C \cdot V^{PSEUDO}$.

**Proof:** For each $j \in J_2$, we define its ‘optimal processing times’ by machine $i$ as

$$\tau_{j,i} = \{t | job j is processed by OPT on machine i at time t\}.$$

$$\sum_{j \in J_2} w_j B_j(f_j^{OPT}) = \sum_{j \in J_2} \sum_{1 \leq i \leq m} \int_{t \in \tau_{j,i}} B_j(f_j^{OPT}) dt$$

$$\leq \sum_{j \in J_2} \sum_{1 \leq i \leq m} \int_{t \in \tau_{j,i}} B_j(t - \tau_j) dt$$

$$\leq C \cdot \sum_{j \in J_2} \sum_{1 \leq i \leq m} \int_{t \in \tau_{j,i}} d_j(t) dt.$$

According to the definition of $J_2$, during the processing of job $j \in J_2$ by the optimal
algorithm, the on-line algorithm still keeps the job in its pool. By Observation 4 we know
that the job’s priority is not too high; it is at most 4 times the priority of the currently
running jobs and, specifically, at time $t$ for machine $i$ such that $t \in \tau_{j,i}$, its priority is at
most 4 times the priority of the job at the top of stack $i$ in the on-line algorithm. Denote
that job by $j(t,i)$. So,

$$C \cdot \sum_{j \in J_2} \sum_{1 \leq i \leq m} \int_{t \in \tau_{j,i}} d_j(t) dt \leq 4C \cdot \sum_{j \in J_2} \sum_{1 \leq i \leq m} \int_{t \in \tau_{j,i}} \hat{d}_j(t,i) dt \leq 4C \cdot \sum_{1 \leq i \leq m} \int_{t \in \tau_{i,i}} \hat{d}_j(t,i) dt \leq 4C \cdot \sum_{1 \leq i \leq m} \int_{t} \hat{d}_j(t,i) dt \leq 4C \cdot \sum_{j \in J} w_j \hat{d}_j = 4C \cdot V^{PSEUDO}.$$ 

\[\Box\]

**Lemma 5.5.3** $V^{PSEUDO} \leq 2C \cdot V^{ON}$.

**Proof:** By using Lemma 5.4.3 separately on each machine we obtain the same result for the
multiprocessor case. \[\Box\]

Combining all the results together we get

**Theorem 5.5.1** Algorithm ALG2 for the multiprocessor case is $10C^2$ competitive.

### 5.6 Discussion

Although a constant competitive algorithm is a significant improvement, some kind of lower
bound is still missing. The lower bound should demonstrate the relation between the
difficulty of the problem and the ratio $C$. Moreover, a more careful analysis of the algorithms
might provide a better competitive ratio.
Chapter 6

The Unsplittable Flow Problem

6.1 Introduction

In this chapter we consider the three variants of the unsplittable flow problem (UFP). We are given a directed or undirected graph $G = (V, E)$, $|V| = n$, $|E| = m$, a capacity function $u$ on its edges and a set of $l$ terminal pairs of vertices $(s_j, t_j)$ with a demand $d_j$ and profit $r_j$. A feasible solution is a subset $S$ of the terminal pairs and a single flow path for each such pair such that the capacity constraints are fully met. The objective is to maximize the total profit of the satisfied terminal pairs.

Both before and after the $O(\sqrt{m})$ approximation algorithm for classical UFP in [22], there were attempts to achieve the same approximation ratio using combinatorial methods. Up to now however, these were found only for restricted versions of the problem [47, 37] and were not optimal. In this chapter we present a combinatorial algorithm that not only achieves the $O(\sqrt{m})$ result for classical UFP but is also the first strongly polynomial algorithm for that problem.

Recall that the extended UFP is the case where both demands and capacities are arbitrary (specifically, some demands might be higher than some capacities). Due to its complexity, not many results addressed it. We improve the best approximation ratio through a strongly polynomial algorithm. By proving a lower bound for the extended UFP over directed graphs we infer that this case is really harder than the classical UFP. Specifically, for large demands we show that unless $P = NP$ it is impossible to approximate extended UFP better than $O(m^{1-\epsilon})$ for any $\epsilon > 0$.

We now consider the $K$-bounded UFP case, that is, the case where the maximum demand is at most $\frac{1}{K}$ the minimum edge capacity. Recall that for $K < \log n$, previous algorithms achieved an approximation ratio of $O(Kn^{\frac{1}{K}-1})$. We improve the result to a strongly polynomial $O(Kn^{\frac{1}{K}})$ approximation algorithm which, as a special case, is a $O(\sqrt{m})$ approximation algorithm for the half disjoint case. Since this ratio is better than the
\( \Omega(m^{1/2} - \epsilon) \) lower bound for classical UFP, we achieve a separation between classical UFP and \( K \)-bounded UFP for \( K \geq 3 \) (this does not imply a separation for \( K = 2 \) since the lower bound is shown using instances where \( m = \Theta(n) \)). The improvement is achieved by splitting the requests into a low demand set and a high demand set. The sets are treated separately by algorithms similar to those of [8] where in the case of high demands the algorithm has to be slightly modified. We would like to note that in our approximation ratios involving \( n \), we can replace \( n \) with \( D \) where \( D \) is an upper bound on the longest path ever used (which is obviously at most \( n \)).

As a by-product of our methods, we provide online algorithms for UFP. Here, the network is known but requests arrive one by one and a decision has to be made without knowing which requests follow. We show on-line algorithms whose competitive ratio is somewhat worse than that of the off-line algorithms. We also show that one of our algorithms is optimal in the on-line setting by slightly improving a lower bound of [8].

We conclude this introduction with a short summary of the main results in this chapter. We denote by \( d_{\text{max}} \) the maximum demand and by \( u_{\text{min}} \) the minimum edge capacity.

- Classical UFP (\( d_{\text{max}} \leq u_{\text{min}} \)) - Strongly polynomial \( O(\sqrt{m}) \) approximation algorithm.
- Extended UFP (arbitrary \( d_{\text{max}}, u_{\text{min}} \)) - Strongly polynomial \( O(\sqrt{m} \log(2 + \frac{d_{\text{max}}}{u_{\text{min}}})) \) approximation algorithm; A lower bound of \( \Omega(m^{1/2} - \epsilon) \) and of \( \Omega(m^{d_{\text{max}}/u_{\text{min}}} / \sqrt{\log(2 + \frac{d_{\text{max}}}{u_{\text{min}}})}) \) for directed graphs.
- Bounded UFP (\( d_{\text{max}} \leq \frac{1}{K} u_{\text{min}} \)) - Strongly polynomial \( O(Kn^{d_{\text{max}}/u_{\text{min}}}) \) approximation algorithm.

### 6.2 Preliminaries

Let \( G = (V, E) \), \( |V| = n \), \( |E| = m \), be a (possibly directed) graph and a capacity function \( u : E \to \mathbb{R}^+ \). An input request is a quadruple \( (s_j, t_j, d_j, r_j) \) where \( \{s_j, t_j\} \) is the source-sink terminal pair, \( d_j \) is the demand and \( r_j \) is the profit. The input is a set of the above quadruples for \( j \in T = \{1, \ldots, l\} \). Let \( D \) be a bound on the length of any routing path; note that \( D \) is at most \( n \).

We denote by \( u_{\text{min}} \) (\( u_{\text{max}} \)) the minimum (maximum) edge capacity in the graph. Similarly, we define \( d_{\text{min}}, d_{\text{max}}, r_{\text{min}} \) and \( r_{\text{max}} \) to be the minimum/maximum demand/profit among all input requests. We define two functions on sets of requests, \( S \subseteq T \):

\[
    r(S) = \sum_{j \in S} r_j \quad d(S) = \sum_{j \in S} d_j
\]

A feasible solution is a subset \( \mathcal{P} \subseteq T \) and a route \( P_j \) from \( s_j \) to \( t_j \) for each \( j \in \mathcal{P} \) subject to the capacity constraints, i.e., the total demand routed through an edge is bounded by
the its capacity. Some of our algorithms order the requests so we will usually denote by $L_j(e)$ the relative load of edge $e$ after routing request $j$, that is, the sum of demands routed through $e$ divided by $u(e)$. Without loss of generality, we assume that any single request can be routed. That is possible since we can just ignore unrouteable requests. Note that this is not the $d_{\max} \leq u_{\min}$ assumption made in classical UFP.

Before describing the various algorithms, we begin with a simple useful lemma:

**Lemma 6.2.1** Given a sequence $\{a_1, \ldots, a_n\}$, a non-increasing non-negative sequence $\{b_1, \ldots, b_n\}$ and two sets $X, Y \subseteq \{1, \ldots, n\}$, let $X^i = X \cap \{1, \ldots, i\}$ and $Y^i = Y \cap \{1, \ldots, i\}$. If for every $1 \leq i \leq n$

$$\sum_{j \in X^i} a_j > \alpha \sum_{j \in Y^i} a_j$$

then

$$\sum_{j \in X} a_j b_j > \alpha \sum_{j \in Y} a_j b_j$$

**Proof:** Denote $b_{n+1} = 0$. Since $b_j - b_{j+1}$ is non-negative,

$$\sum_{j \in X} a_j b_j = \sum_{j \in X} a_j \sum_{i=j}^n (b_i - b_{i+1})$$

$$= \sum_{i=1, \ldots, n} (b_i - b_{i+1}) \sum_{j \in X^i} a_j$$

$$> \alpha \sum_{i=1, \ldots, n} (b_i - b_{i+1}) \sum_{j \in Y^i} a_j$$

$$= \alpha \sum_{j \in Y} a_j \sum_{i=j}^n (b_i - b_{i+1})$$

$$= \alpha \sum_{j \in Y} a_j b_j$$

\[\blacksquare\]

Using this lemma we show that algorithms that maximize the total demand also maximize the total profit if only they consider the requests in a nonincreasing order of profit to demand ratio.

### 6.3 Algorithms for UFP

#### 6.3.1 Algorithm for Classical UFP

In this section we show a simple algorithm for classical UFP (the case in which $d_{\max} \leq u_{\min}$). The algorithm’s approximation ratio is the same as the best currently known algo-
Algorithm PROUTE is an $O(\sqrt{m})$ approximation algorithm for classical UFP.

Proof: First, we look at the running time of the algorithm. The number of iterations done in Routine$_2$ is:

$$\log \frac{\alpha_{\text{max}}}{\alpha_{\text{min}}} = \log(n \frac{r_{\max} u_{\max}}{r_{\min} d_{\min}})$$

which is polynomial. Routine$_1$ looks for a non-overflowing path $P$ with $F(j, P) > \alpha$. The latter condition is equivalent to $\sum_{e \in P} \frac{1}{u(e)} < \frac{r_j}{d_j \alpha}$ and thus a shortest path algorithm can be used.

Consider an optimal solution routing requests in $Q \subseteq T$. For each $j \in Q$ let $Q_j$ be the route chosen for $j$ in the optimal solution. The total profit of either $Q \cap T_1$ or $Q \cap T_2$ is at least $\frac{r(Q)}{2}$. Denote that set by $Q'$ and its index by $i' \in \{1, 2\}$, that is, $Q' = Q \cap T_{i'}$. Now consider the values given to $\alpha$ in Routine$_2$ and let $\alpha' = 2^{i'}$ be the highest such that $r(\{j \in Q' | F(j, Q_j) > \alpha'\}) \geq r(Q)/4$. It is clear that such an $\alpha'$ exists. From now on we
limit ourselves to Routine$_1$($\alpha' , l'$) and show that a good routing is obtained by it. Denote by $\mathcal{P}$ the set of requests routed by Routine$_1$($\alpha' , l'$) and for $j \in \mathcal{P}$ denote by $P_j$ the path chosen for it.

Let $\mathcal{Q}'_{\text{high}} = \{ j \in \mathcal{Q}' | F(j, Q_j) > \alpha' \}$ and $\mathcal{Q}'_{\text{low}} = \{ j \in \mathcal{Q}' | F(j, Q_j) \leq 2\alpha' \}$ be sets of higher and lower 'quality' routes in $\mathcal{Q}'$. Note that the sets are not disjoint and that the total profit in each of them is at least $r(\mathcal{Q}) \alpha'$ by the choice of $\alpha'$. From the definition of $F$,

$$
r(\mathcal{Q}'_{\text{low}}) = \sum_{j \in \mathcal{Q}'_{\text{low}}} F(j, Q_j) \sum_{e \in Q_j} \frac{d_j}{u(e)} \leq 2\alpha' \sum_{j \in \mathcal{Q}'_{\text{low}}} \sum_{e \in Q_j} \frac{d_j}{u(e)} \leq 2\alpha' \sum_{j \in \mathcal{Q}'} \sum_{e \in Q_j} \frac{d_j}{u(e)} = 2\alpha' \sum_{e} \sum_{j \in \mathcal{Q}} \frac{d_j}{u(e)} \leq 2\alpha' \sum_{e} 1 = 2m\alpha'
$$

where the last inequality is true since an optimal solution cannot overflow an edge. Therefore,

$$
r(\mathcal{Q}) \leq 8m\alpha'.
$$

Now let $E_{\text{heavy}} = \{ e \in E | L_4(e) \geq \frac{1}{4} \}$ be a set of the heavy edges after the completion of Routine$_1$($\alpha' , l'$). We consider two cases. The first is when $|E_{\text{heavy}}| \geq \sqrt{m}$. According to the description of the algorithm, $F(j, P_j) > \alpha' \text{ for every } j \in \mathcal{P}$. Therefore,

$$
r(\mathcal{P}) = \sum_{j \in \mathcal{P}} F(j, P_j) \sum_{e \in P_j} \frac{d_j}{u(e)} \geq \alpha' \sum_{j \in \mathcal{P}} \sum_{e \in P_j} \frac{d_j}{u(e)} \geq \alpha' \sum_{j \in \mathcal{P}} \sum_{e \in P_j} \frac{L_4(e)}{u(e)} \geq \frac{1}{4} \sqrt{m} \alpha'
$$

where the last inequality follows from the assumption that more than $\sqrt{m}$ edges are loaded more than fourth their capacity. By combining the two inequalities we get:

$$
\frac{r(\mathcal{Q})}{r(\mathcal{P})} \leq 32 \sqrt{m} = O(\sqrt{m})
$$

which completes the first case.

From now on we consider the second case where $|E_{\text{heavy}}| < \sqrt{m}$. Denote by $R = Q'_{\text{high}} \setminus P$. We compare the profit given by our algorithm to that found in $R$ by using
Lemma 6.2.1. Since $\frac{r_j}{d_j}$ is a non increasing sequence, it is enough to bound the total demand routed in prefixes of the two sets. For that we use the notation $R^k = R \cap \{1, ..., k\}$ and $\mathcal{P}^k = \mathcal{P} \cap \{1, ..., k\}$ for $k = 1, ..., L$. For each request $j \in R^k$ the algorithm cannot find any appropriate path. In particular, the path $Q_j$ is not chosen. Since $j \in \mathcal{Q}'_{high}$, $F(j, Q_j) > \alpha'$ and therefore the reason the path is not chosen is that it overflows one of the edges. Denote that edge by $e_j$ and by $E^k = \{e_j | j \in R^k\}$.

Lemma 6.3.1 $E^k \subseteq E_{heavy}$

Proof: Let $e_j \in E^k$ be an edge with $j \in R^k$, a request corresponding to it. We claim that when the algorithm fails finding a path for $j$, $L_j(e_j) \geq \frac{1}{4}$. For the case $i' = 1$, the claim is obvious since the demand $d_j \leq u_{min}/2$ and in particular, $d_j \leq u(e_j)/2$. Thus, the load of $e_j$ must be higher than $u(e_j)/2$ for the path $Q_j$ to overflow it. For the case $i' = 2$, we know that $u_{min}/2 < d_j \leq u_{min}$. In case $u(e_j) > 2u_{min}$, the only way to overflow it with demands of size at most $d_{max} \leq u_{min}$ is when the edge is loaded at least $u(e_j) - u_{min} \geq u(e_j)/2$. Otherwise, $u(e_j) \leq 2u_{min}$ and since $d_j \leq u_{min} \leq u(e)$ we know that the edge cannot be empty. Since we only route requests from $T_2$ the edge’s load must be at least $u_{min}/2 \geq u(e_j)/4$.

Since each request in $R^k$ is routed through an edge of $E^k$ in the optimal solution, $d(R^k) \leq \sum_{e \in E^k} u(e)$. The highest capacity edge $f \in E^k$ is loaded more than fourth its capacity since it is in $E_{heavy}$ and therefore $d(\mathcal{P}^k) \geq \frac{u(f)}{4}$. By Lemma 6.3.1, $|E^k| \leq |E_{heavy}| < \sqrt{m}$ and hence,

\[
d(R^k) < \sqrt{m} \cdot u(f) \leq 4\sqrt{m} \cdot d(\mathcal{P}^k).
\]

We use Lemma 6.2.1 by combining the inequality above on the ratio of demands and the nonincreasing sequence $\frac{r_j}{d_j}$. This yields

\[
\sum_{j \in R} \frac{r_j}{d_j} d_j \leq 4\sqrt{m} \sum_{j \in \mathcal{P}} \frac{r_j}{d_j} d_j,
\]

or,

\[
r(R) \leq 4\sqrt{m} \cdot r(\mathcal{P}).
\]

Since $\mathcal{Q}'_{high} = R \cup \mathcal{P}$,

\[
r(\mathcal{Q}'_{high}) = r(R) + r(\mathcal{P}) \leq (1 + 4\sqrt{m})r(\mathcal{P}).
\]

Recall that $r(\mathcal{Q}'_{high}) \geq r(\mathcal{Q})/4$ and therefore

\[
\frac{r(\mathcal{Q})}{r(\mathcal{P})} \leq 4 + 16\sqrt{m} = O(\sqrt{m})
\]

\[
\Box
\]
6.3.2 Strongly Polynomial Algorithm

Routine$_1$ is strongly polynomial. Routine$_2$ however calls it $\log \frac{u_{\text{max}}}{d_{\text{min}}}$ times. Therefore, it is polynomial but still not strongly polynomial. We add a preprocessing step whose purpose is to bound the ratio $\frac{u_{\text{max}}}{d_{\text{min}}}$. Recall that $l$ denotes the number of requests.

$$SPROUTE(T):$$
run Routine$_3(T_1)$ and Routine$_3(T_2)$ and choose the better solution

Routine$_3(S)$:
For each edge such that $u(e) > l \cdot d_{\text{max}}$, set $u(e)$ to be $l \cdot d_{\text{max}}$.
Throw away requests whose profit is below $\frac{u_{\text{max}}}{d_{\text{min}}}$. Take the better out of the following two solutions:
Route all requests in $S_{\text{tiny}} = \{ j \in S | d_j \leq \frac{u_{\text{max}}}{d_{\text{min}}} \}$ on any simple path.
Run Routine$_2(S \setminus S_{\text{tiny}})$.

**Theorem 6.3.2** Algorithm SROUTE is a strongly polynomial $O(\sqrt{m})$ approximation algorithm for classical UFP.

**Proof:** Consider an optimal solution routing requests in $Q \subseteq S$. Since the demand of a single request is at most $d_{\text{max}}$, the total demand routed through a given edge is at most $l \cdot d_{\text{max}}$. Therefore, $Q$ is still routable after the first preprocessing phase. The total profit of requests whose profit is lower than $\frac{u_{\text{max}}}{d_{\text{min}}}$ is $r_{\text{max}}$. In case $r(Q) > 2r_{\text{max}}$, removing these requests still leaves the set $Q'$ whose total profit is at least $r(Q) - r_{\text{max}} \geq \frac{r(Q)}{2}$. Otherwise, we take $Q'$ to be the set containing the request of highest profit. Then, $r(Q')$ is $r_{\text{max}} \geq \frac{r(Q)}{2}$.

All in all, after the two preprocessing phases we are left with an UFP instance for which there is a solution $Q'$ whose profit is at least $\frac{r(Q)}{2}$.

Assume that the total profit in $Q' \cap S_{\text{tiny}}$ is at least $\frac{r(Q)}{4}$. Since the requests in $S_{\text{tiny}}$ have a demand of at most $\frac{u_{\text{max}}}{d_{\text{min}}}$ and there are at most $l$ of them, they can all be routed on simple paths and the profit obtained is at least $\frac{r(Q)}{4}$. Otherwise, the profit in $Q' \setminus S_{\text{tiny}}$ is at least $\frac{r(Q)}{4}$ and since algorithm PROUTE is an $O(\sqrt{m})$ approximation algorithm, the profit we obtain is also within $O(\sqrt{m})$ of $r(Q)$.

The preprocessing phases by themselves are obviously strongly polynomial. Recall that the number of iterations performed by Routine$_2$ is $\log(n \frac{u_{\text{max}}}{d_{\text{min}}})$. The ratio of profits is at most $l$ by the second preprocessing phase. The first preprocessing phase limits $u_{\text{max}}$ to $k \cdot d_{\text{max}}$. So, the number of iterations is at most $\log(n \frac{u_{\text{max}}}{d_{\text{min}}})$. In case $S = T_1$, $d_{\text{max}} \leq \frac{u_{\text{max}}}{d_{\text{min}}}$ and $d_{\text{min}} \geq \frac{u_{\text{min}}}{l}$ since tiny requests are removed. For $S = T_2$, $d_{\text{max}} \leq u_{\text{min}}$ and $d_{\text{min}} \geq u_{\text{min}}/2$. We end up with at most $O(\log n + \log l)$ iterations which is strongly polynomial. ■
6.3.3 Algorithm for Extended UFP

In this section we show that the algorithm can be used for the extended case in which demands can be higher than the lowest edge capacity.

Instead of using just two sets in SROUTE, we define a partition of the set of requests $T$ into $2 + \max\{\lceil \log d_{\text{max}}/u_{\text{min}} \rceil, 0\}$ disjoint sets. The first, $T_1$, consists of requests for which $d_j < u_{\text{min}}/2$. The set $T_i$ for $i > 1$ is of requests for which $2^{i-3}u_{\text{min}} < d_j \leq 2^{i-2}u_{\text{min}}$. The algorithm is as follows:

$$ESROUTE(T):$$
for any $1 \leq i \leq 2 + \max\{\lceil \log d_{\text{max}}/u_{\text{min}} \rceil, 0\}$ such that $T_i$ is not empty
run Routine3($T_i$) on the resulting graph
choose the best solution obtained

Theorem 6.3.3 Algorithm $ESROUTE$ is a strongly polynomial $O(\sqrt{m} \log(2 + d_{\text{max}}/u_{\text{min}}))$ approximation algorithm for extended UFP.

Proof: The proof of Theorem 6.3.1 and of Theorem 6.3.2 hold also for the extended case. The only part which has to be proved is Lemma 6.3.1. The following replaces the lemma:

Lemma 6.3.2 $E^k \subseteq E_{\text{heavy}}$

Proof: Let $e_j \in E^k$ be an edge with $j \in R^k$, a request corresponding to it. We claim that when the algorithm fails finding a path for $j$, $L_j(e_j) \geq 1/4$. For the case $k' = 1$, the claim is obvious as before. For the case $k' > 1$, we know that $2^{k'-3}u_{\text{min}} < d_j \leq 2^{k'-2}u_{\text{min}}$. In case $u(e_j) > 2^{k'-1}u_{\text{min}}$, the only way to overflow it with demands of size at most $2^{k'-2}u_{\text{min}}$ is when the edge is loaded at least $u(e_j) - 2^{k'-2}u_{\text{min}} \geq u(e_j)/2$. Otherwise, $u(e_j) \leq 2^{k'-1}u_{\text{min}}$ and since $j$ is routed through this edge in the optimal solution $d_j \leq u(e_j)$. Therefore, the edge cannot be empty. Since we only route requests from $T_i$, the edge’s load must be at least $2^{k'-3}u_{\text{min}} \geq u(e_j)/4$.

The number of iterations $ESROUTE$ performs is at most $l$ since we ignore empty $T_i$’s. For $T_1$, the number of iterations of Routine2 is the same as in SROUTE. For a set $T_i$, $i > 1$, the number of iterations of Routine2 is $\log(n \min \frac{d_{\text{max}}}{u_{\text{min}}})$. As before, the preprocessing of Routine3 reduces this number to $\log(nl^2d_{\text{max}}/u_{\text{min}})$. Since the ratio $\frac{d_{\text{max}}}{u_{\text{min}}}$ is at most 2 in each $T_i$, we conclude that $ESROUTE$ is strongly polynomial.

6.4 Algorithms for $K$-bounded UFP

In the previous section we considered the classical UFP in which $d_{\text{max}} \leq u_{\text{min}}$. We also extended the discussion to extended UFP. In this section we show better algorithms for
6.4. ALGORITHMS FOR K-BOUNDED UFP

K-bounded UFP in which \( d_{max} \leq \frac{K}{K+1} u_{min} \) where \( K \geq 2 \).

6.4.1 Algorithms for Bounded Demands

In this section we present two algorithm for bounded UFP. The first deals with the case in which the demands are in the range \([\frac{u_{min}}{K+1}, \frac{u_{min}}{K}]\). As a special case, it provides an \( O(\sqrt{n}) \) approximation algorithm for the half-disjoint paths problem where edge capacities are all the same and the demands are exactly half the edge capacity. The second is an algorithm for the K-bounded UFP where demands are only bounded by \( \frac{u_{min}}{K} \) from above.

\[
EKROUTE(T):
\]
\[
\mu \leftarrow 2D
\]
sort the requests in \( T \) according to a non-increasing order of \( r_j/d_j \)
foreach \( j \in T \) in the above order
if \( \exists \) a path \( P \) from \( s_j \) to \( t_j \) s.t.
\[
\sum_{e \in P} (\mu^{L_j-1}(e) - 1) < D
\]
then route the request on \( P \) and for \( e \in P \) set \( L_j(e) = L_j-1(e) + \frac{d_j}{u(e)} \)
else reject the request

\[
BKROUTE(T):
\]
\[
\mu \leftarrow (2D)^{1+\frac{1}{K-1}}
\]
sort the requests in \( T \) according to a non-increasing order of \( r_j/d_j \)
foreach \( j \in T \) in the above order
if \( \exists \) a path \( P \) from \( s_j \) to \( t_j \) s.t.
\[
\sum_{e \in P} (\mu^{L_j-1}(e) - 1) < D
\]
then route the request on \( P \) and for \( e \in P \) set \( L_j(e) = L_j-1(e) + \frac{d_j}{u(e)} \)
else reject the request

Note that algorithm \( EKROUTE \) uses a slightly different definition of \( L \). This ‘virtual’ relative load allows it to outperform \( BKROUTE \) on instances where the demands are in the correct range.

**Theorem 6.4.1** Algorithm \( EKROUTE \) is a strongly polynomial \( O(K \cdot D^{\frac{1}{K}}) \) approximation algorithm for UFP with demands in the range \([\frac{u_{min}}{K+1}, \frac{u_{min}}{K}]\). Algorithm \( BKROUTE \) is a strongly polynomial \( O(K \cdot D^{\frac{1}{K-1}}) \) approximation algorithm for K-bounded UFP.

**Proof:** The first thing to note is that the algorithms never overflow an edge. For the first algorithm, the demands are at most \( \frac{u_{min}}{K} \) and the only way to exceed an edge capacity is to route request \( j \) through an edge \( e \) that holds at least \( \frac{K \cdot u(e)}{u_{min}} \) requests. For such an edge,
\[ L_{j-1}(e) \geq 1 \text{ and } \mu^{L_{j-1}(e)} \geq \mu - 1 \geq D. \] For the second algorithm, it is sufficient to show that in case \( L_{j-1}(e) > 1 - \frac{1}{K} \) for some \( e \) then \( \mu^{L_{j-1}(e)} - 1 \geq D; \) that is true since \( \mu^{L_{j-1}(e)} - 1 \geq \left( (2D)^{1+\frac{1}{K}} \right)^{1-\frac{1}{K}} - 1 = 2D - 1 \geq D. \) Therefore, the algorithms never overflow an edge.

Now we lower bound the total demand accepted by our algorithms. We denote by \( \mathcal{Q} \) the set of requests in the optimal solution and by \( \mathcal{P} \) the requests accepted by either of our algorithm. For \( j \in \mathcal{Q} \) denote by \( Q_j \) the path chosen for it in the optimal solution and for \( j \in \mathcal{P} \) let \( P_j \) be the path chosen for it by our algorithm. We consider prefixes of the input so let \( Q^k \) be \( \mathcal{Q} \cap \{1, \ldots, k\} \) and \( P^k \) be \( \mathcal{P} \cap \{1, \ldots, k\} \) for \( k = 1, \ldots, L \). We prove that

\[
d(\mathcal{P}^k) \geq \frac{\sum_{e \in P^k} u(e)(\mu^{L^k(e)} - \mu^{L^k-1(e)})}{6KD\mu^K}.
\]

The proof is by induction on \( k \) and the induction base is trivial since the above expression is zero. Thus, it is sufficient to show that for an accepted request \( j \)

\[
\frac{\sum_{e \in P_j} u(e)(\mu^{L_j(e)} - \mu^{L_{j-1}(e)})}{6KD\mu^K} \leq d_j.
\]

Note that for any \( e \in P_j \), \( L_j(e) - L_{j-1}(e) \leq \frac{1}{K} \) for both algorithms. In addition, for both algorithms \( L_j(e) - L_{j-1}(e) \leq 3 \frac{d_j}{u(e)} \) where the factor 3 is only necessary for \( EKROUTE \) where the virtual load is higher than the actual increase in relative load. The worst case is when \( K = 2, u(e) = (1.5 - \epsilon)u_{\text{min}} \) and \( d_j = (\frac{1}{2} + \epsilon)u_{\text{min}} \); the virtual load increases by \( \frac{1}{2} \) whereas \( \frac{d_j}{u(e)} \) is about \( \frac{3}{5} \). Looking at the exponent,

\[
\mu^{L_j(e)} - \mu^{L_{j-1}(e)} = \mu^{L_{j-1}(e)}(\mu^{L_j(e)} - L_{j-1}(e) - 1) \\
= \mu^{L_{j-1}(e)}(\mu^{\frac{1}{K}}(L_j(e) - L_{j-1}(e)) - 1) \\
\leq \mu^{L_{j-1}(e)}\mu^{\frac{1}{K}K(L_j(e) - L_{j-1}(e))} \\
\leq \mu^{L_{j-1}(e)}\mu^{\frac{1}{K}3K \frac{d_j}{u(e)}}
\]

where the first inequality is due to the simple relation \( x^y - 1 \leq xy \) for \( 0 \leq y \leq 1, 0 \leq x \) and that for \( e \in P_j \), \( L_j(e) - L_{j-1}(e) \leq \frac{1}{K} \). Therefore,

\[
\sum_{e \in P_j} u(e)(\mu^{L_j(e)} - \mu^{L_{j-1}(e)}) \leq \sum_{e \in P_j} \mu^{L_{j-1}(e)}\mu^{\frac{1}{K}3Kd_j} \\
= 3K\mu^{\frac{1}{K}d_j} \sum_{e \in P_j} \mu^{L_{j-1}(e)} \\
= 3K\mu^{\frac{1}{K}d_j}(\sum_{e \in P_j} (\mu^{L_{j-1}(e)} - 1) + |P_j|) \\
\leq 3K\mu^{\frac{1}{K}(D + D)d_j} \\
= 6KD\mu^{\frac{1}{K}d_j}
\]
where the last inequality holds since the algorithm routes the request through \( P_j \) and the length of \( P_j \) is at most \( D \).

The last step in the proof is to upper bound the total demand accepted by an optimal algorithm. Denote the set of requests rejected by our algorithm and accepted by the optimal one by \( R^k = Q^k \setminus P^k \). For \( j \in R^k \), we know that \( \sum_{e \in Q_j} (\mu^{L_j - 1}(e) - 1) \geq D \) since the request is rejected by our algorithm. Hence,

\[
D \cdot d(R^k) \leq \sum_{j \in R^k} \sum_{e \in Q_j} d_j(\mu^{L_j - 1}(e) - 1) \\
\leq \sum_{j \in R^k} \sum_{e \in Q_j} d_j(\mu^{L_j}(e) - 1) \\
= \sum_{e} \sum_{j \in R^k \cap Q_j} d_j(\mu^{L_j}(e) - 1) \\
= \sum_{e} (\mu^{L_j}(e) - 1) \sum_{j \in R^k \cap Q_j} d_j \\
\leq \sum_{e} (\mu^{L_j}(e) - 1) u(e),
\]

where the last inequality holds since the optimal algorithm cannot overflow an edge.

By combining the two inequalities shown above,

\[
d(Q^k) \leq d(P^k) + d(R^k) \leq d(P^k) + d(P^k) \frac{6KD}{D} \mu^{\frac{1}{K}} = (1 + 6K \mu^{\frac{1}{K}})d(P^k)
\]

The algorithm followed a non-increasing order of \( \frac{P}{d} \) and by Lemma 6.2.1 we obtain the same inequality above for profits. So, the approximation ratio of the algorithm is

\[
1 + 6K \mu^{\frac{1}{K}} = O(K \cdot \mu^{\frac{1}{K}})
\]

which, by assigning the appropriate values of \( \mu \), yields the desired results. \( \square \)

### 6.4.2 A Combined Algorithm

In this section we combine the two algorithms presented in the previous section: the algorithm for demands in the range \( \left[ \frac{u_{\min}}{K + 1}, \frac{u_{\max}}{K + 1} \right] \) and the algorithm for the \( K \)-bounded UFP. The result is an algorithm for the \( K \)-bounded UFP with an approximation ratio of \( O(K \cdot D^{\frac{1}{K}}) \).

We define a partition of the set of requests \( T \) into two sets. The first, \( T_1 \), includes all the requests whose demand is at most \( \frac{1}{K + 1} \). The second, \( T_2 \), includes all the requests whose demand is more than \( \frac{1}{K + 1} \) and at most \( \frac{1}{K} \).

\[
\text{CKROUTE}(T):
\]

- Take the best out of the following two possible solutions:
  - Route \( T_1 \) by using \( BKROUTE \) and reject all requests in \( T_2 \)
  - Route \( T_2 \) by using \( EKROUTE \) and reject all requests in \( T_1 \)
Theorem 6.4.2 Algorithm CKROUTE is a strongly polynomial $O(KD^{\frac{1}{r}})$ approximation algorithm for $K$-bounded UFP.

Proof: Let $Q$ denote an optimal solution in $T$. Since BKROUTE is used with demands bounded by $\frac{1}{K+1}$ its approximation ratio is $O(KD^{\frac{1}{r}})$. The same approximation ratio is given by EKROUTE. Either $T_1$ or $T_2$ have an optimal solution whose profit is at least $\frac{\pi}{2}$ and therefore we obtain the claimed approximation ratio. □

6.5 Lower Bounds

In this section we show that in cases where the demands are much larger than the minimum edge capacity UFP becomes very hard to approximate, namely, $\Omega(m^{1-\epsilon})$ for any $\epsilon > 0$. We also show how different demand values relate to the approximability of the problem. The lower bounds are for directed graphs only.

Theorem 6.5.1 [31] The following problem is NPC:

2DIRPATH:
INPUT: A directed graph $G = (V,E)$ and four nodes $x,y,z,w \in V$
QUESTION: Are there two edge disjoint directed paths, one from $x$ to $y$ and the other from $z$ to $w$ in $G$?

Theorem 6.5.2 For any $\epsilon > 0$, extended UFP cannot be approximated better than $\Omega(m^{1-\epsilon})$.

Proof: For a given instance $A$ of 2DIRPATH with $|A|$ edges and a small constant $\epsilon$, we construct an instance of extended UFP composed of $l$ copies of $A$, $A^1,A^2,...,A^l$ where $l = |A|^{1/\epsilon}$. The instance $A^i$ is composed of edges of capacity $2^{l-i}$. A special node $y^0$ is added to the graph. Two edges are added for each $A^i$, $(y^{i-1},x^i)$ of capacity $2^{l-i} - 1$ and $(y^{i-1},z^i)$ of capacity $2^{l-i}$. All $l$ requests share $y^0$ as a source node. The sink of request $1 \leq i \leq l$ is $w^i$. The demand of request $i$ is $2^{l-i}$ and its profit is 1. The above structure is shown in the following figure for the hypothetical case where $l = 4$. Each diamond indicates a copy of $A$ with $x,y,z,w$ being its left, right, top and bottom corners respectively. The number inside each diamond indicates the capacity of $A$'s edges in this copy.

![Figure 6.1: The UFP instance for the case $l = 4$](image)
We claim that for a given YES instance of 2DIRPATH the maximal profit gained from
the extended UFP instance is \(l\). We route request \(1 \leq i \leq l\) through \([y^0_i, x^1_i, y^2_i, ..., y^{2i-1}_i, z^i, w^i]\). Note that the path from \(x^j\) to \(y^j\) and from \(z^j\) to \(w^j\) is a path in \(A^j\) given by the YES instance.

For a NO instance, we claim that at most one request can be routed. That is because the path chosen for a request \(i\) ends at \(w^j\). So, it must arrive from either \(z^j\) or \(x^j\). The only edge entering \(x^j\) is of capacity \(2^{2j-1} - 1\) so \(z^j\) is the only option. The instance \(A^j\) is a NO instance of capacity \(2^{2j-1}\) through which a request of demand \(2^{2j-1}\) is routed from \(z^j\) to \(w^j\). No other path can therefore be routed through \(A^j\) so requests \(j > i\) are not routable. Since \(i\) is arbitrary, we conclude that at most one request can be routed through the extended UFP instance and its profit is 1.

The gap created is \(l = |A|^{d^2}\) and the number of edges is \(l \cdot (|A| + 2) = O(1^{1+\epsilon})\). Hence, the gap is \(\Omega(m^{1+\epsilon}) = \Omega(m^{1-\epsilon'})\) and since \(\epsilon\) is arbitrary we complete the proof.

**Theorem 6.5.3** For any \(\epsilon > 0\) extended UFP with any ratio \(d_{\text{max}} / u_{\text{min}} \geq 2\) cannot be approximated better than \(\Omega(m^{1+\epsilon'} \sqrt{\log (d_{\text{max}} / u_{\text{min}})})\).

**Proof:** For a given instance \(A\) of 2DIRPATH with \(|A|\) edges and a small constant \(\epsilon\), we construct an instance of extended UFP with the given ratio \(d_{\text{max}} / u_{\text{min}}\). We begin with describing our basic building block from which the UFP instance will be built. Let \(\delta < \frac{1}{k}\) be a small constant. The \(i^{th}\) building block, denoted \(B^i\), contains \(k^2 + \frac{k(k-1)}{2}\) copies of the graph \(A\) with edge capacity \(2^{2i-1} + k\delta\) which will be placed for clarity on a two dimensional grid. The copies are denoted by \(A_{(a,b)}\) where \(1 \leq a \leq k\) and \(1 \leq b \leq 2k - a\) are two integers and \((a,b)\) is its position on the grid. There are \(2k\) input nodes and \(2k\) output nodes. The first \(k\) input nodes \(x^i_{1}, 1 \leq i \leq k\), are located at \((0, j)\) whereas the next \(k\) input nodes \(x^i_{2}, 1 \leq j \leq k\), are at \((0, k + j)\). The output nodes \(y^i_{1}, 1 \leq j \leq k\), are located at \((k + 1, j)\) and \(z^i_{1}, 1 \leq j \leq k\), are at \((k + j + 1, 0)\). The edges are described by their location on the grid and are all of length one on that grid. A copy of \(A\) located at \((a,b)\) is connected through its \(z\) node to an edge above it (the edge from \((a, b + 1)\) to \((a, b)\)) through its \(w\) node to the edge below it, through \(x\) to the left edge and through \(y\) to the right edge. For each \(1 \leq j \leq k\) there exist \(k + 1\) horizontal edges, \((0, j), (1, j)), \((1, j), (2, j)), ..., ((k, j), (k + 1, j))\). The capacity of each of these edges is \(2^{2i-1}\). For \(1 \leq j \leq k\) we add \(k - j + 1\) horizontal edges \(((0, k + 1), (1, k + j)), ((1, k + 1), (2, k + 1))))..., ((k - j, k + 1), (k - j + 1, k + 1))) and \(k + 1\) vertical edges \(((k - j + 1, k + 1), (k - j + 1, k + 1)), \((k - j + 1, k + j - 1), (k - j + 1, k + j - 2))..., ((k - j + 1, 1), (k - j + 1, 0))\). The capacity of each of these edges is \(2^{2i-1} + j\delta\).

The UFP instance consists of the \(l\) blocks \(B^1, B^2, ..., B^l\). There are \(k\) additional nodes denoted \(y^0_i, 1 \leq j \leq k\), which act as input nodes. For \(1 \leq i \leq l\), \(2k\) connecting edges of capacity \(2^l\) are used, \((y^0_{j-1}, x^j_i)\) and \((y^0_{j}, z^j_i)\), \(1 \leq j \leq k\). The request set consists of requests denoted \(r_{(i,j)}\) for \(1 \leq i \leq l, 1 \leq j \leq k\) from \(y^0_i\) to \(w^j_i\) with demand \(2^{2i-1} + j\delta\). All requests are of profit 1.
We claim that for a given YES instance of 2DIRPATH the maximal profit gained from
the extended UFP instance is $l \cdot k$, that is, all requests can be routed. We route request $r_{(i,j)}$ 
through $[y_i^j, x_j^1, y_j^1, x_j^2, y_j^2, ..., y_j^{j-1}, z_j^j, w_j^j]$. Note that the path from $x_j^i$ to $y_j^i$ is a horizontal path in $B^i$ through $k$ copies of $A$ and the path from $z_j^i$ to $w_j^i$ is a path in $B^i$ going through 
the point $(k - j + 1, k + j)$ and passing $2k - 1$ copies of $A$. The load in $B^i$ on the edges of 
the path from $x_j^i$ to $y_j^i$ is $2^{l-i-1} + j \delta + 2^{l-i-2} + j \delta + ... + 1 + j \delta \leq 2^{l-i-1} - 1 + lk \delta \leq 2^{l-i}$.
The load in $B^i$ on the edges of the path from $z_j^i$ to $w_j^i$ is $2^{l-i} + j \delta$. Therefore, no edge is 
overloaded and the total profit gained is as claimed.

For a NO instance, we claim that at most one request can be routed. Assume request 
$r_{(i,j)}$ is routed through the UFP instance. Request $r_{(i,j)}$ is of demand $2^{l-i} + j \delta$ and by 
tracing back the request from its sink $w_j^i$ it can be seen that the capacities of the edges are 
such that the request must be routed through a vertical path from $(k - j + 1, k + j)$ to the 
sink. This immediately implies that any request $r_{(i,j)}$ with $i > i$ cannot be routed since it 
must exit $B^i$ through one of the $y$ nodes and thus cross the path of request $r_{(i,j)}$ in one of 
the $A$ junctions. In addition, a request $r_{(i,j)}$ with $i < j$ cannot be routed since the sink 
$w_j^i$ is located to the right of the vertical path and thus must cross it as well. By using the 
two observations above we conclude that at most one request can be routed through the 
extended UFP instance and its profit is 1.

The ratio $\frac{\text{max}}{\text{min}}$ created is at most $2^l$ and hence we choose $l = \lceil \log(\frac{\text{max}}{\text{min}}) \rceil$. In addition, 
we choose $k = \lceil A^{1/\epsilon} \rceil$. The number of edges used is $m = l(3k^2 + 4k + (k^2 + \frac{k(k-1)}{2}) \cdot |A|) = O(l \cdot k^2 \cdot |A|) = O(l \cdot k^{2+\epsilon})$. Therefore, the gap created is $lk = \Omega(m \frac{\epsilon \ln \sqrt{\log(\frac{\text{max}}{\text{min}})}}{\epsilon})$ and 
by choosing a small $\epsilon$ the proof is completed.

\section{6.6 Online Applications}

\subsection{6.6.1 Online Algorithms}

Somewhat surprisingly, variants of the algorithms considered so far can be used in the 
online setting with slightly worse bounds. For simplicity, we present here an algorithm for 
the throughput $K$-bounded UFP in which $r_j = d_j$ for every $j \in T$.

First note that for throughput $K$-bounded UFP, both EKROUTE and BKROUTE 
can be used as online deterministic algorithms since sorting the requests becomes unnecessary. By splitting $T$ into $T_1$ and $T_2$ as in CKROUTE we can combine the two algorithms:
6.6. ONLINE APPLICATIONS

**ONLINE\textsc{K Route}(T):**

Choose one of the two possible routing methods below with equal probabilities:

- Route $T_1$ by using $\text{EK Route}$ and reject all requests in $T_2$
- Route $T_2$ by using $\text{EK Route}$ and reject all requests in $T_1$

**Theorem 6.6.1** Algorithm $\text{ONLINE\textsc{K Route}}$ is an $O(K \cdot D^{\frac{1}{K}})$ competitive online algorithm for throughput $K$-bounded UFP.

**Proof:** The expected value of the total accepted demand of the algorithm for any given input is the average between the total accepted demands given by the two routing methods. Since each method is $O(K \cdot D^{\frac{1}{K}})$ competitive on its part of the input, the theorem follows.

6.6.2 Online Lower Bound

In this section we show an $\Omega(K \cdot n^{\frac{1}{K}})$ lower bound for deterministic on-line algorithms in the throughput $K$-bounded UFP. This matches the upper bound of $\text{EK Route}$ and slightly improves the previously known lower bound of $\Omega(n^{\frac{1}{K}})$ [8]. The lower bound is proved over a line network of length $n$. For simplicity, we assume that $n = r^K$ for some integer $r$. Otherwise, we can just use the largest $r$ such that $r^K \leq n$ and prove the lower bound over a part of the line. All the requests are of demand $\frac{1}{K}$ and the edge capacities are all 1.

The lower bound can be represented by a subtree of a tree of height $K$ with an outdegree of $n^{\frac{1}{K}}$. Each node in the subtree corresponds to one or more requests over some interval. The root corresponds to requests over the interval $[0, n]$. Node $j$ in level $0 \leq i \leq K$ corresponds to the interval $[r^{K-i} \cdot j, r^{K-i} \cdot (j+1)]$. Note that the segments corresponding to a node’s children are a partition of its own segment.

The lower bound is constructed together with its corresponding subtree by a DFS traversal of the tree. The traversal begins at the tree’s root. At each node, the algorithm is given at most $K$ requests over the interval corresponding to the current node. If the algorithm does not accept any of the $K$ requests, the node’s children are not traversed and the next node in the DFS traversal is visited. Otherwise, once the algorithm accepts a request, we start traversing each of its children recursively.

Note that in case we arrive to a leaf in the tree, the algorithm cannot accept any requests over its interval. That is because its interval is contained in $K$ other accepted intervals; one for each of the node’s ancestors. Therefore, the sequence of requests is well defined.

We need the following simple lemma for trees:

**Lemma 6.6.1** For a tree in which the out-degree of each node is either zero (a leaf) or $\delta > 1$, the number of leaves is at least $\delta - 1$ times larger than the number of internal nodes.
Proof: We use induction on the tree growing from the root up. A tree with just one node (leaf) has the required property. Then, a tree that grows is a replacement of a leaf in an internal node and \( \delta \) leaves. Therefore, the number of leaves has grown by \( \delta - 1 \) while the number of internal nodes has grown by 1. The required ratio is maintained.

\[ \text{Theorem 6.6.2} \quad \text{The competitive ratio of any deterministic on-line algorithm for the } K\text{-bounded UFP is at least } \Omega(K \cdot n^{1/K}). \]

Proof: The algorithm's value from the input described above is the number of internal nodes in the subtree. That is because the algorithm accepts one request from each internal node but no requests from the leaves. A better solution for the same input is to accept all the requests represented by the leaves of the subtree. In that case, the value is \( K \) times the number of leaves since each leaf corresponds to \( K \) intervals. That is an allowed assignment since the intervals corresponding to the leaves do not intersect. By comparing the two solutions and using the previous lemma, we obtain the stated lower bound on the competitive ratio.

6.7 Discussion

Using combinatorial methods we showed algorithms for all three variants of the UFP problem. We improve previous results and provide the best approximations for UFP by using strongly polynomial algorithms. Due to their relative simplicity we believe that further analysis should lead to additional performance guarantees such as non linear bounds. Also, the algorithms might perform better over specific networks. An interesting open problem to consider is finding better results for the half-disjoint case and for the undirected case. No known lower bounds exist for these two cases and the upper bounds are relatively high. We conjecture that a considerably better upper bound exists but there is no compelling evidence to believe that this is indeed the case.
Bibliography


תדמנוןCKERERCHIEF
ע"ש רמון ובצלאל סאקרל
 Beit-ShearLEM-DER TECHNION
על-ידי
עודד רגב

העבודה הוכנה בהדרכתו של פרופ' יוסי עזר

ינואר 2001
תְזוֹמוֹנָה ואיזון עוֹמְסִים

בעבודה זו נתברר שהซอוט קבוצת כסף של משימה כולם
לבקוע את החומרים ואת המעבדים בה הוא זרות. בעיות
איוֹת עוֹמְסִים עומדות על כל מבחר את המעבדים במדור
משימה. בכל אדום מחוזר נרצה לחקות כל האפשר את
מיחור הפתרון ואצל כל האפשר את הרווחה. ברוב
המקרים בה נעסק פתרון אופטימלי של Closure בחלק
להשוואת לכל נכסת פתרון פתרון מקוונים טויבים לכל
האופשר.

הבעיה האבריאנה היא סכוף לפני היא בועית או יוֹי עוֹמְסִים זמניים.
בעוביה זו, הוסף לעומס, מחזורים לכל משימה על ידי דני
הᠦANGUAGE והزواיגה שלחל. מטרותיו של למאז השמה לשימור
כשר יועמסו הגוזים יניב הופעל על מכונה היא קומכ לכל
האופשר. נררא נוא אולגרוט מיטות בועי ביבוס יоко
הופשיט מ-1.5. נור נבר פתרון המכנסות בקוף,
נור ראש נינצ' לא צ.OnItemClickListener קירוב, כלומר, קירוב עד כדי קובע
קטן בתםנון.

בעית ייוֹי עוֹמְסִים סכוף במטפל היא בועית איזוּי
העמסים המקוונים. עַמש כל בָּרגיֵלזיה של העביה ב韓
בעבר מחזב המחבר במקוון ש✂ בלח מַשְיוֹת.
בעבר נ Yayınיה בנא משתמש ופָּמָה יניקת לא יעק
דהי וברמקה מַדְוָה של שֵּחַת מַהְיוֹת בקּוּינני לחשו
 모습ות קירוב מַדְוָה.

בעית התיפור האריאנה אלַי נטיית היא בועית התיפור
לא מדיה אמטעֵך רב עָמשי. בעית התיפור היריזלה
ייזו ייס קירוב לאometimes יניק להשה יעי יי שימוש
ב"עקרון יראית עָמשי ריצת מַיֶּל". לַרַעורת חוכמוה
של שיטה dele, השימוש בדרמה מחוזה סלכ בלחית כ십시
ואו המראה ציצי יניק להשלוף את אומת ייס קירוב לא שימוש
ב当たり前. מואצת ובนโยת שימוים בָּכָרִיטות מקוון.
נمشار ומשווק בעיות שונות לבין أيضاً העומד נושא נשען השפעה נושא להגדיר כל שיתוף את הרווח במשה לבושו. בערב נושא ההלביד לเพื่อ השימה משען סופי שלחורי אובד כל רוח ממשלות. קיורוב טוב לפי הדרת זכר וסתייבר כדברי הגדול האנס. אנו לב(io מגה הלאופיט בرارו המושג מכל משימה ומך בדמן ובו שוכבה הרוח המושג משימוט שועטוב זם רז היוה קפה. המרה הומוש במקורה משימוט מורגנס מודי. האלוגיטיים שטייצ לביעה זי צחליגרימ בועל יש קיורוב קבוש.

לבסוף נחבו ובישות יחוות בורשה וב לכל קיורוב שקיורוב משלי לכל בישה יחוות מזווחה דרישה זום עליוה ללהב בושת התי לאッシュ טסיל יזום לכל בישה קיורוב יאני אילוףראוח התרוג מקיורובות של קיורוב יש קמיו וצלאה נשלות בקורים העיירה של בישה. זי צלא באיהまして המביב החברות של כל האלוגיטים לכל עין מח. יסמי הקיורוב אומט ניש לכל אתח משไหลות המביב יחיו זים ואטור מוטי קורובوحدים הקדומים. מעבר لكل, כל האלוגיטים יאון לצי הנום קומברטוריים ופולוגליונים בוזרא החודק זאתי ביגוד לתחזוקה הקדומים. תלק מתחזקותינו ירובב גמسائل הממקום.