Class-constrained Resource Allocation Problems

Tamar Tamir

Class-constrained Resource Allocation Problems

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Tamar Tamir

Submitted to the Senate of
the Technion — Israel Institute of Technology

Tamuz 5761 Haifa July 2001

The research thesis was done under the supervision of Dr. Hadas Shachnai in the department of Computer Science

In this work I present some approximation algorithms for hard problems. At least for one problem, of choosing a thesis advisor, I present an optimal solution. Thank you Hadas!

I thank Hagit Attiya. Many of my research skills were developed under her guidance.
In the summer of 1997, I worked in Hewlett-Packard, Palo-Alto. This was a very enriching experience. I thank Abraham Lempel for initiating it, and Oded Shmueli for making it also a very pleasant experience, for his advice and follow-up interest in my activities.

Finally, I would like to thank many residents of the Taub building for their invaluable contribution to my education and for their part in making my study period so enjoyable.

The generous financial help of the Technion, VATAT, and Gutwirth fund is gratefully acknowledged.

Contents

List of Figures

Abstract

We consider variants of classic packing and scheduling problems. These variants are motivated from problems arising in storage management for multimedia systems, in production planning, and in scheduling parallelizable jobs.

In our packing problems, sets of items of different types need to be placed in multiple bins. The items may be of different sizes and values. Each bin has a limited capacity, and a bound on the number of different types of items it can hold. In the class-constrained multiple knapsack problem, we wish to maximize the total value of the packed items. In the class-constrained fair placement problem, our goal is to place the same (large) portion from each set. In the class-constrained bin-packing problem, we seek to minimize the number of (identical) bins, needed for packing all the items.

A scheduling problem with similar constraints is obtained from classic scheduling, when we characterize each job by its length and two additional parameters, which specify the maximal number of machines that can share its execution (allotment constraint), and the maximal number of machines that can process the job simultaneously (parallelism constraint). Our objective is to minimize the overall completion time of the schedule (makespan). Previous work in this area dealt only with cases in which these parameters are unbounded, or get the value one.

Since all of the above problems are NP-hard, we focus on developing and analyzing approximation algorithms. We present hardness results, approximation algorithms, and characterize classes of instances for which optimal solutions can be found efficiently. For the above packing problems, we consider also the on-line version, in which the set of items to be packed is unknown in advance, and the decision regarding the placement of an item needs to be taken upon arrival.

Notation and Abbreviations

PTAS	—	Polynomial time approximation scheme
FPTAS	—	Fully polynomial time approximation scheme
MK	—	The multiple knapsack problem
BP	—	The bin packing problem
CCMK	—	The class-constrained multiple knapsack problem
CCFP	—	The class-constrained fair placement problem
CCBP	—	The class-constrained bin-packing problem
GCCP	—	Generalized class-constrained packing
I	—	A set of items
n	—	Number of items in I
M	—	Number of distinct classes in the input
N	—	Number of knapsacks in a packing instance
$s(u)$	—	Size of an item $u \in I$
$p(u)$	—	Value of an item $u \in I$
K_j	—	The jth knapsack
v_j	—	Size (volume) of K_j
c_j	—	Number of compartments in K_j
V	—	Total volume of the N knapsacks
σ	—	An input sequence (in online problems)
SPAC	—	Scheduling with parallelism and machine allotment constraints
SAC	—	Scheduling with machine allotment constraints
m	—	Number of machines in a scheduling instance
M_i	—	The ith machine
u_i	—	Rate of M_i
J_j	—	The jth job
t_j	—	Length of J_j
ρ_j	—	Parallelism parameter of J_j
a_j	—	Allotment parameter of J_j
w	—	The makespan

Chapter 1

Introduction

1.1 Class-constrained Packing

1.1.1 Problem Statement

In the well-known *bin packing (BP)* and *multiple knapsack (MK)* problems, a set of items of different sizes and values has to be packed into bins of limited capacities; a packing is legal if the total size of the items placed in a bin does not exceed its capacity. We consider the following *class-constrained* variants of these problems. Suppose that each item has a size, a value, and *a class (color)*; each bin has limited capacity, and a limited number of compartments. Items of different colors cannot be placed in the same compartment. Thus, the number of compartments in each bin bounds the number of *distinct colors of items* it can accommodate. A packing is legal if it satisfies the traditional capacity constraint, as well as the *class constraint*.

Formally, the input to our packing problems is a set of items, I, of size $|I| = n$. Each item $u \in I$ has a size $s(u) \in Z^+$, and a value $p(u) \in Z^+$. Each item in I is colored with one of M distinct colors. Thus, $I = I_1 \cup I_2 \cdots \cup I_M$, where any item $u \in I_i$ is colored with color i. The items need to be placed in bins, where each bin $j, j = 1, 2, \ldots$, has volume v_j and c_j compartments.

The output of our packing problems is a *placement*, which specifies for each bin j which items from each class are placed in j (and accordingly, the classes to which j allocates compartments). A placement is legal if for all $j \geq 1$, bin j contains items of at most c_j colors, and the overall size of the items placed in j does not exceed v_j. We study two optimization problems:

The class-constrained multiple knapsack problem (CCMK), in which there are N bins (to which we refer as knapsacks). A placement determines a subset $I' = I'_1 \cup I'_2 \cdots \cup I'_M$ of I, such that $I'_i \subseteq I_i$ is the subset of packed items of color i. Our goal is to find a legal placement which maximizes the total value of the packed items, given by $\sum_{i=1}^{M} \sum_{u \in I'_i} p(u)$;

The class-constrained bin-packing problem (CCBP), in which the bins are identical, each having size 1 and c compartments, and the items have sizes $s(u) \le 1$. Our goal is to find a legal placement of *all* the items in a minimal number of bins.

We also consider a generalized version of class-constrained packing (GCCP), where each item, u, has a size $s(u)$, a value $p(u)$, and a set $C(u) \in \{1, \ldots, M\}$ of colors, such that it is legal to color u in any color in $C(u)$. Thus, if some bin allocates compartments to the set C_1 of colors, then any item $u \in I$ added to this bin needs to satisfy $C(u) \cap C_1 \ne \emptyset$. (In the above CCMK and CCBP problems, we assume that $\forall u \in I, |C(u)| = 1$). In another generalization of class-constrained packing, the color of each item depends on the knapsack in which it is placed. That is, each item $u \in I$ has a size $s(u)$, a value $p(u)$ and an N-vector, $\vec{c}_u$, where $\vec{c}_u(j)$ is the color of u when it is placed in the knapsack K_j, $1 \le j \le N$.

Some of our results are specific for the case where all the items have the same (unit) size and the same value, that is, for all $u \in I$, $s(u) = p(u) = 1$. This case is of special interest, since these instances of the problem are generated in the application of storage management for continuous-media data (see Section 1.1.2). For this case we study an additional optimization problem:

The class-constrained fair placement problem (CCFP), where the objective is to maximize the value of $0 < c \le 1$ such that, $\forall 1 \le i \le M$, $|I'_i| \ge c \cdot |I_i|$.

For the case of unit-sized items we assume that $\sum_{i=1}^{M} |I_i| = \sum_{j=1}^{N} v_j$, that is, the total number of items in I is equal to the total sum of the knapsack volumes. We look for *a perfect placement*, in which both the CCMK and the CCFP problems are solved optimally. Indeed, such a placement yields the maximal utilization of the knapsack capacities, i.e., the total occupied volume is $V = \sum_{j=1}^{N} v_j$, and maximal fairness, i.e. $c = 1$.

The assumption $|I| = V$ simplifies the presentation of our results; moreover, any input for the storage management problem, which motivated our study of this problem, satisfies this assumption. It is important to note, however, that our results hold for general instances, i.e., for any relation between $|I|$ and V. We elaborate on that in Section 2.5.

In Chapter 4 we study the *online* versions of CCMK and CCBP, where in each step we get a single item, $a \in I$, having size 1 and color $i \in \{1, \ldots, M\}$; we need to pack a before we know any of the subsequent items. Formally, I is given as a sequence $\sigma = a_1, a_2, \ldots$ of length n, such that $\forall k, a_k \in \{1, \ldots, M\}$. For each k, the algorithm can base its decision regarding the packing of a_k solely on the knowledge of $a_1, \ldots a_{k-1}$. The decisions of the algorithm are irrevocable, that is, packed items cannot be repacked at later times, and rejected items (in the knapsack problem) cannot be packed later.

We study also a generalization of online class-constrained packing, in which items may leave the system after a while. Formally, the input sequence may consist of two types of events: (i) arrival of an item a of color $i \in \{1, \ldots, M\}$, and (ii) departure of an item that was packed earlier. The resulting *temporary packing problems* are denoted by CCBP^t and CCMK^t. Generally, a departure event is associated with specific *item* (of specific color, placed in specific bin). We consider also the special case in which a departure event is associated with a *color*, so we may choose which item of that color to remove. In the CCBP^t problem, our objective is to minimize the overall number of bins used throughout the execution of the algorithm. In CCMK^t our goal is to maximize the total number of items packed by the algorithm.

In some cases, we will be interested in the performance of an online packing algorithm on a specific set of inputs. In particular, we will be interested in the sets $S_{c,v}(k,h)$, where for given c, v, the set $S_{c,v}(k,h)$ consists of all the input sequences, σ, in which the number of colors, M_σ, satisfies $kc < M_\sigma \leq (k+1)c$, and the length of σ satisfies $hv < |\sigma| \leq (h+1)v$. Our refined analysis for sets of inputs will yield tighter performance bound, which depend on v, c, $|\sigma|$ and M_σ.

1.1.2 Applications

The above packing problems capture an important set of resource-allocation problems, in which utilities (e.g., processors, storage devices) need to allocate to clients some resource (e.g., processing time, data streams). Each utility has a limited amount of resources. In addition, there is a bound on the number of different types of clients that each utility can service/support. These class-constraints reflect restrictions that often arise in real systems (e.g., machines with limited number of configurations, disks with limited storage capacity). In particular, our research applies to the following problems.

Storage management in multimedia systems: In a multimedia-on-demand (MOD) system (see, e.g.,[56, 23]), a large database of M video program files is kept on a centralized server. Each program file, i, is associated with a popularity parameter, given by $p_i \in [0,1]$, where $\sum_{i=1}^{M} p_i = 1$. The files are stored on N shared disks. Each of the disks is characterized by (i) its storage capacity, that is, the number of files that can reside on it, and (ii) its load capacity, given by the number of data streams that can be read simultaneously from that disk. Assuming that $\{p_1, \ldots, p_M\}$ are known, we can predict the expected load generated by each of the programs at any time. We need to allocate to each file disk space and some fraction of the load capacity, such that the load generated due to access requests to that file is satisfied.

The above storage management problem can be formulated as a special case of the CCMK problem, in which $s(u) = p(u) = 1$ for all $u \in I$: a disk j, with load capacity ℓ_j and storage capacity c_j, is represented by a knapsack K_j, with capacity ℓ_j and c_j compartments, for all $j = 1, ..., N$. The ith file, $1 \leq i \leq M$, is represented by a set I_i, the size of which is proportional to the file popularity. Thus, $n = |I| = \sum_{j=1}^{N} \ell_j$ and $|I_i| = p_i|I|$ (For simplicity, we assume that $p_i|I|$ is an integer; otherwise we can use a standard rounding technique [33]). A solution for the CCMK problem induces a legal assignment of the files to the disks: K_j allocates a compartment to items of color i, *iff* a copy of the ith file is stored on disk j, and the number of items of color i that are packed in K_j is equal to the total load that file i can generate on disk j.

When the requests for resources arrive online, we get instances of online CCBP and online CCMK. The *temporary* CCBP and CCMK problems reflect the nature of user requests, to hold the resource for a limited amount of time (which is often unknown in advance). In Section 4.3.2 we discuss a model where we allow a departure of any item from a *specified color*. This model represents the flexibility of real systems, in providing service from *any* idle device which is in a suitable configuration.

Consider, for example, the transmission of movies, residing on a centralized server, to a set of clients, on demand. Upon request for a movie, any non-overloaded disk, which holds the movie file, can provide the service. In particular, suppose that some movie, f_1, is played from the disks d_1, d_2, to service the requests of the users u_1, u_2; the disk d_2 is overloaded while d_1 is underloaded. When the system completes the transmission from d_1, it may continue servicing u_2 from d_1, thus reducing the load on d_2. In terms of online class-constrained bin-packing, we may choose the bin from which we omit an item, as long as we take an item of the specified color (i.e., use a copy of f_1).

Production planning: Our class-constrained packing problems capture also a production planning problem which arises in computer systems and in many other areas. Consider a set of machines, the jth machine has a limited capacity, v_j, of some physical resource (e.g., storage space, quantity of production materials). In addition, hardware specifications allow machine j to produce items of only c_j different types. For example, suppose that the N machines are printers, each capable of producing 60 pages per minute, and can be loaded with four different types of pages – of specific colors and sizes. The system receives orders for products of M distinct types. Each order u is associated with a demand for $s(u)$ units of the physical resource, a profit, $p(u)$, and its type $i \in \{1, \ldots, M\}$. In our example, a printer job would be the file 'f_1' that has to be output on ten green-A4 pages.

We need to determine how the production work should be distributed among the machines (that is, we need to determine the color and size of the papers that will be loaded on the trays of each printer, and which printer, out of those that has a tray with green-A4 pages will print the file 'f_1'). When the goal is to obtain maximal profit from a given set of machines, we have an instance of the CCMK. When we seek the minimal number of machines required for the completion of all orders, we have an instance of the CCBP. When each order can be processed under a few possible configurations (in our example: 'f_1' can be printed also on blue pages), we get an instance of GCCP.

1.1.3 Related Work

Packing Problems: There is a wide literature on the bin packing (BP) and the multiple knapsack (MK) problems (see, e.g., [35, 8, 32, 16, 10] and detailed surveys in [47, 28, 46]). Since these problems are NP-hard, most of the research work in this area focused on finding approximation algorithms. The special case of MK where $N = 1$, known as the classic *0-1 knapsack problem*, admits a fully polynomial time approximation scheme *(FPTAS)*. That is, for any $\varepsilon > 0$, a $(1-\varepsilon)$-approximation for the optimal solution can be found in $O(n/\varepsilon^2)$, where n is the number of items [34, 22]. In contrast, MK was shown to be NP-hard in the strong sense [19], therefore it is unlikely to have an FPTAS, unless $P = NP$. It was unknown until recently, whether MK possessed a polynomial time approximation scheme *(PTAS)*, whose running time is polynomial in n, but can be *exponential* in $\frac{1}{\varepsilon}$. Chekuri and Khanna [10] resolved this question. They presented an elaborated PTAS for MK, and showed that with slight generalizations this problem becomes APX-hard. Independently, Kellerer developed in [42] a PTAS for the special case of the MK, where all bins are identical.

It is well known (see, e.g., [49]) that bin packing does not belong to the class of NP-hard problems that possess a PTAS. However, there exists an *asymptotic* PTAS (*APTAS*), which uses $(1+\varepsilon)OPT(I) + k$ bins for some fixed k. Vega and Lueker presented an APTAS, with $k = 1$ [54]. Alternatively, a *dual* PTAS, which uses $OPT(I)$ bins of size $(1+\varepsilon)$ was given by Hochbaum and Shmoys [29]. Such a dual PTAS can also be derived from the recent work of Epstein and Sgall [15] on multiprocessor scheduling, since BP is dual to the minimum makespan problem.

Other related work deal with *multi-dimensional packing problems* (also known as *vector packing*): Frieze and Clarke [18] gave a PTAS for the case $N = 1$; Chekuri and Khanna presented in [9] a PTAS for the case where $N \geq 1$ and the knapsacks have d equal-sized dimensions. Note that for the special case in which I contains *a single* item in each color, the CCMK reduces to a special case of 2-dimensional packing: the knapsack K_j has size $v_j \times c_j$, and each item $u \in I$ has size $s(u) \times 1$. This *cardinality-constrained packing problem*, in which the *number* of items packed in each knapsack is constrained, was studied in [44] and recently in [43]. For $N = 1$, an FPTAS for the cardinality-constrained 0-1 knapsack problem is given in [7]. Indeed, an instance of our class-constrained packing problem may contain many items in each color. In terms of 2-dimensional packing, it means that the jth knapsack is of size $v_j \times c_j$; however, the size of an item $u \in I_i$ is $s(u) \times 1$ if u is the first item of I_i in K_j, and $s(u) \times 0$ if K_j already holds items from I_i. Thus, we have not been able to adapt any of the approximation techniques developed for multi-dimensional packing for solving our class-constrained packing problems.

In Section 2.2, we present an algorithm for CCMK with unit-sized items. When all the knapsacks have the same v/c ratio, this algorithm always packs at least a $1 - 1/(1 + c_{min})$-fraction of all the items. Following our work [50], Golubchik et al. [24] derived a tighter analysis of our algorithm, and showed that it actually packs at least a $1 - 1/(1 + \sqrt{c})^2$-fraction of $|I|$, and that this is optimal for *any* algorithm for this problem. They also gave a PTAS for CCMK with unit sizes and values, and showed that this special case of the CCMK is strongly NP-hard, even if all the knapsacks are identical. We strengthen these hardness results in Section 5.1 (The results are given in terms of scheduling problems, but they hold also for packing).

Online bin-packing has been studied since the early 1970's, starting with the works of Garey et al. [21] and Johnson [36]. The performance of *first-fit* (FF) and *best-fit* (BF) was analyzed in

[38], where it was shown that $r_{FF}, r_{BF} \leq 1.7$. The first lower bound for any online bin packing algorithm was given by Yao in [57]. He showed that $r_A \geq 1.5$, for any algorithm $\mathcal{A}$. The current best lower bound, 1.540, is due to van Vliet [53]. Detailed surveys on online packing can be found in [5, 17]. The online *cardinality-constrained* bin-packing problem is studied in [3].

Multimedia on Demand Systems: Multimedia on demand systems were studied intensively in the recent years. However, the assignment problem received only little attention in this context. Specifically, most of the previous work discuss the problem of load balancing on the disks, in which the goal is to have the total broadcast load on the system distributed evenly among the N disks.

The first solution proposed for the load balancing problem was disk striping [11]. In this solution the file data is distributed over multiple disks; Thus, the heavy load caused by a popular program is shared among these disks. In [56] dynamic algorithms were suggested for balancing the load in the system. The paper also addresses the problem of determining the number of copies of each file that should be kept in the system; the goal is to have the total storage capacity allocated to f_i reflect its popularity. We show below that this criterion can yield poor results when used for solving our optimization problems (see in Section 2.1.3). In [45] the load on the disks is balanced dynamically by replications and deletions of video program files. The replication policy is based on the (changing) popularities of the files.

Better performance of the MOD system can be achieved also by data sharing techniques. i.e., data retrieved for one request can be used to service additional requests. Several approaches have been explored in previous work. Data sharing techniques can be categorized as follows: (*i*) *batching* [12, 55], that is, delaying requests with the hope that more requests for the same data will arrive during the batching interval; the entire group is then serviced using a single data stream. (*ii*) *buffering* [39], that is, closing the temporal 'gaps' between successive requests through the use of buffer space, and (*iii*) *adaptive piggybacking* [1, 25] i.e., adjusting consumption (broadcast) rates of requests in progress, for the same data, until their corresponding data streams can be merged into one. Each of these techniques can be applied independently of the resource allocation scheme used in the system, and therefore, they are not directly relevant to this work. However, the use of data sharing techniques together with our static assignment algorithm can improve the overall performance of the system.

1.1.4 Main Results for Packing Problems

In Chapter 2 we consider instances in which all the items have the same (unit) size and the same value.

- We show that the CCMK and the CCFP problems are NP-hard.
- We show that for some instances a *perfect* placement (in which both the CCMK and the CCFP problems are solved optimally) always exists and can be found in polynomial time.

Three simple conditions for the existence of a perfect placement are given. For each condition, we show how the CCMK and the CCFP can be optimally solved, when this condition is satisfied.

- When the conditions are not met, we derive approximate solutions. The approximation ratio depends on the "uniformity" of the knapsacks. Specifically, given $r > 0$ and $\alpha \geq 1$ such that $\forall j,\ r \leq \frac{v_j}{c_j} \leq \alpha \cdot r$, we give an algorithm which achieves $\frac{1}{\alpha}$-approximation for both the CCMK and the CCFP.

- We show that if the number of compartments in each knapsack is at least b, for some $b \geq 1$, i.e., $C_j \geq b$, $\forall 1 \leq j \leq N$, then the CCMK can be approximated to within a factor $b/(b+1)$.

In Chapter 3, we study the general case, in which different items may have different sizes.

- We give a dual *polynomial time approximation scheme (PTAS)* for CCBP, which packs any instance, I, with a fixed number of distinct colors, into at most $OPT(I)$ bins whose size is at most $1 + \varepsilon$.

- We derive a PTAS for CCMK, for any instance in which M, the number of distinct colors, is a fixed constant. The running time of our PTAS depends on the number, t, of bin types in the instance. Specifically, we distinguish between the case where t is some fixed constant and the general case, where t can be as large as N. In both cases, the profit is guaranteed to be at least $(1 - \varepsilon)OPT(I)$.

- We show that the 0-1 class-constrained knapsack problem (CCKP) admits a fully polynomial time approximation scheme (FPTAS). For the case where all items have the same value, we give an *optimal* polynomial time algorithm. Our FPTAS is based on a two-level dynamic programming scheme. Our results for the CCKP are suitable also for instances in which the number of distinct colors depends on the input size.

- We show that the generalizations of class-constrained packing are APX-hard. This holds for the generalization in which each item is associated with a *set* $C(u)$ of colors, such that it is legal to color u in any color in $C(u)$, and also for the generalization in which the color of an item depends on the knapsack in which it is placed. In both cases, the GCCP problem is APX-hard, already for instances where all the items have the same size and the same value.

In Chapter 4, we study the online CCBP and CCMK problems. Most of the results are for the case where all the items have the same size and the same value.

- We present a general lower bound on the competitive ratio of deterministic algorithms. specifically, we show that *any* deterministic algorithm, $\mathcal{A}_d$, has competitive ratio $r_d \geq 2$. We also derive an exact lower bound for any set $S_{c,v}(k, h)$ of instances. For any deterministic algorithm, $\mathcal{A}_d$, and for any $1 < c < v$, $k \geq 0$, and $h \geq k - 1 + \max\{\lceil \frac{k}{c-1} \rceil, \lceil \frac{kc+1}{v} \rceil\}$, the competitive ratio of $\mathcal{A}_d$ on instances from $S_{c,v}(k,h)$ is $r_d(S_{c,v}(k,h)) \geq 1 + \frac{k+1-\lceil \frac{kc+1}{v} \rceil}{h+1}$. We show that a variant of the *first-fit* algorithm, adapted to class-constrained packing, achieves the above lower bounds. Another greedy algorithm, based on partitioning the items into color-sets, is shown to be efficient as well.

- We show that some *any-fit* algorithms, as well as the greedy *next-fit*, have competitive ratio v/c. This should be contrasted with the good performance of these algorithms in the classical BP problem (where they achieve approximation ratio of at most 2)[37, 38].

- For the CCBP^t problem, we show a v/c lower bound on the competitive ratio of any deterministic algorithm, and that all any-fit algorithms achieve this bound.

- It may seem that online algorithms perform better when departure events are associated with a *color*, rather than *specific item*. Indeed, in this case each departure event allows some re-packing of the items. However, we show a lower bound of $\min(v/c, c-1)$ on the competitive ratio of all any-fit algorithms. This implies the same ratio of v/c, obtained for departures of specific items, when $v < c^2 - c$. Also, the color-sets algorithm, which is optimal when no departures are allowed, is shown to achieve the worst possible ratio of v.

- We show that under the traditional performance measures of competitive ratio, there is no advantage in knowing a-priori the *number of items* to be packed, and/or the *number of distinct colors* in the input sequence. This holds for all the problems that we consider.

- For the online CCMK we show a lower bound of c/v on the competitive ratio of any deterministic algorithm, and that any greedy algorithm, which never rejects an item that can be packed, achieves this bound, regardless of the way it packs the items. We also show that there is no competitive algorithm for CCMK^t.

1.2 Scheduling with Machine Allotment and Parallelism Constraints

1.2.1 Problem Statement

A continuing trend in modern computer systems is to distribute computation among several physical processors. This enables to speed up the execution of heavy applications. Ideally, the work required by such applications could be shared by *any* number of processors. However, setup and communication costs and the maximal level of parallelism within each application, bound the number of machines to which it can be *allotted*, and the number of machines by which it can be processed *simultaneously*.

The resulting scheduling problem can be stated as follows. Suppose that n jobs need to be scheduled on m machines; each machine, M_i, $1 \le i \le m$, runs at specific rate, u_i; each job, J_j, $1 \le j \le n$, is associated with a processing time, t_j, an *allotment parameter*, a_j, and a *parallelism parameter*, ρ_j. Thus, the execution of J_j ban be shared by at most a_j machines, and at most ρ_j machines can process J_j *simultaneously*. Our objective is to schedule the jobs on the machines, such that the allotment and parallelism constraints are satisfied, and the overall completion time of all jobs (or the *makespan*) is minimized. We call this problem *scheduling with parallelism and machine allotment constraints (SPAC)*. The *multiprocessor scheduling problem*(also known as *non-preemptive scheduling*), where each job can run on a *single* machine (namely, $\rho_j = a_j = 1,\ \forall\ 1 \le j \le n$), is a special case of our problem; thus, our problem is strongly NP-hard. We note that in SPAC we allow preemptions while processing a job, J_j, on some machine. However, when $\forall j,\ \rho_j = a_j = 1$, such preemptions are redundant.

Note that the allotment parameter, a_j, bounds also the number of machines that can process J_j simultaneously. Thus, w.l.o.g., we assume that $\forall j,\ \rho_j \le a_j$. We study also the special case of SPAC, in which parallelism constraints do not affect the schedule. That is, $\forall j,\ \rho_j \ge a_j$. We refer to this case as the problem of *scheduling with machine allotment constraints (SAC)*.

Denote by $w_{OPT}(I)$ the length of an optimal schedule of an instance I. Note that $w^* = \sum_j t_j / \sum_i u_i$ is a lower bound for $w_{OPT}(I)$. When each job can be allotted (possibly, to run in parallel) to *any* number of machines, this lower bound is obtained by a simple greedy algorithm, based on McNaughton's rule [48]: it starts by scheduling the first job on the first machine; then it proceeds to the next job (whenever the current job, J_j, was allocated t_j processing units), or to the next machine (when the current machine runs for w^* time units). The resulting schedule incurs at most $n + m - 1$ allotments of jobs to machines. However, we cannot predict how the machines will share the execution of the jobs. thus, even if $\sum_j a_j \ge n + m - 1$, we may not be able to obtain the lower bound.

The next example shows how allotment/parallelism constraints come into play in finding a schedule which minimizes the makespan.

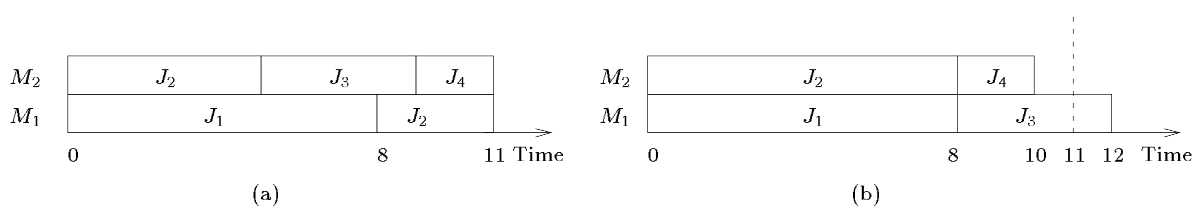

Figure 1.1: Scheduling with parallelism and machine allotment constraints

Example 1.2.1 *Consider a system with two identical machines M_1, M_2, whose rates are $u_1 = u_2 = 1$, and four jobs with $t_1 = t_2 = 8, t_3 = 4, t_4 = 2$, and $\rho_j = 1,\ \forall\ 1 \le j \le 4$. Figure*

1.1(a) presents the schedule produced by a greedy algorithm. The makespan of this schedule is $\sum_j t_j / \sum_i u_i = \frac{22}{2} = 11$. *Assume now that* $a_1 = a_2 = a_3 = 1$, *and* $a_4 = 2$. *Note that the overall number of allotments allowed in the system is equal to the number of allotments incurred by the greedy algorithm; however, since only the execution of* J_4 *can be shared by two machines, the best possible schedule has the length* 12 *(Figure 1.1(b)).*

1.2.2 Motivation

As mentioned earlier, the SPAC problem has important application in scheduling on multiprocessors, and in distributed computing. In distributed systems, load balancing and computation speedup are achieved by partitioning large applications to run on several machines; this is done through process migration (see, e.g., [52]). Therefore, the resources required by each job, J_j, are classified as either (i) *machine dependent* (e.g., peripheral devices), which can be allocated to J_j once it is scheduled to run on *specific* machine, or (ii) *machine independent* (e.g., shared data), which can migrate with segments of the job from one machine to another, along its execution. The limited amount of machine-dependent resources sets a bound, a_j, on the number of machines that can share the execution of J_j (i.e., the number of machines to which J_j can be allotted throughout the schedule); the limited amount of machine-independent resources sets a bound, ρ_j, on the number of machines that can run J_j in *parallel*.

Another application is production planning. Production processes [13] typically involve the usage of consumable resources (i.e., special materials) which cannot migrate from one machine to another, and mobile resources (e.g., human supervision), which allow flexibility in the choice of machines. The maximal amount of consumable resources determines the allotment parameter of a production process; the available amount of mobile resources determines its parallelism parameter.

1.2.3 Related Work

The problem of scheduling a set of jobs on parallel machines with the objective of minimizing the makespan has been studied extensively (Comprehensive surveys appear, e.g., in [28, 6]). The non-preemptive scheduling problem is known to be strongly NP-hard [20], and admits a PTAS: the papers of Hochbaum and Shmoys and of Alon et al. ([29, 2]) give PTASs for identical machines; PTASs for uniform machines were developed by Hochbaum and Shmoys and by Epstein and Sgall ([30, 15]).

For non-parallelizable jobs it was shown by McNaughton [48] that a simple greedy algorithm achieves the minimal makespan for preemptive scheduling, on *identical* machines. The paper of Horvath, Lam and Sethi [31] presented the first optimal algorithm for uniform machines. Another optimal algorithm for uniform machines, that minimizes also the total number of preemptions in the schedule, is presented in the paper [26] of Gonsalez and Sahni. Previous works on parallelizable jobs (see, e.g., [4, 14]) assume an *even* partition of the processing of a job J_j among machines that run J_j in parallel.

1.2.4 Main Results for Scheduling Problems

Unless specified otherwise, all of our results hold for *uniform* machines. Note that although in the SPAC problem we allow preemptions while processing a job on some machine, the algorithms presented in this work do not use such preemptions (while the hardness results cover also schedules in which such preemptions are allowed).

- We study the complexity of the SPAC problem. In particular, we show that SPAC is strongly NP-hard, already for instances with no parallelism constraints (i.e., the SAC problem) and *weak* allotment constraints. Specifically, SAC is strongly NP-hard in the following cases.

  1. On identical machines, where each job can be allotted to at least c machines, for any fixed $c > 1$.
  2. On identical machines, where the *total* number of allotments is unbounded.
  3. For identical jobs, where the total number of allotments is at least $\frac{3}{2}m$.

  These hardness results strengthen the hardness result in [24], which holds for identical jobs with $\sum_j a_j = m$.

- We present a $\max_j(1+1/\rho_j)$-approximation algorithm for the SPAC problem. Our algorithm proceeds in two steps: (i) finding a non-feasible schedule of optimal length, where a job J_j may be processed by $\rho_j + 1$ processors. (ii) transforming this schedule into a feasible one. The running time of the algorithm is $O(\max(n \log n, m \log m))$. For identical machines we modify this algorithm to obtain a $\max_j(1 + 1/(2\rho_j - 1))$-approximation ratio.

- We give a PTAS for the SAC problem. Our PTAS is suitable for instances in which the maximal allotment parameter of any job is some fixed constant.

- We explore the relation between the solvability of the classic preemptive scheduling problem, and the amount of machine allotments/parallelism allowed in the schedule. The classic preemptive scheduling problem belongs to the set of instances in which $\rho_j < a_j$, $\forall j$. We call such instances *parallel-dominated*. We show that the SPAC problem is strongly NP-hard for these instances, even when the jobs are identical and $\forall j,\ a_j = \rho_j + 1$, and solvable by an $O(m \log m)$ algorithm, when the jobs are identical and $\forall j,\ a_j > \rho_j + 1$. This implies, for example, that the preemptive scheduling problem (in which $\rho_j = 1$, for all $1 \leq j \leq n$) becomes strongly NP-hard, when the execution of each job can be shared by at most *two* machines. We also show an interesting distinction between the solvability of our problem in the uniform and the identical machines environments. In particular, SPAC can be solved optimally in polynomial time for *parallel dominated* instances when the machines are identical; however, as mentioned above, on uniform machines the problem is strongly NP-hard.

Chapter 2

Class-constrained Multiple Knapsack with Unit-sized Items

In this chapter we study the CCMK and the CCFP problems on instances in which all the items have the same (unit) size and the same value. In the *class-constrained multiple knapsack problem (CCMK),* we wish to maximize the total number of packed items; in the *class-constrained fair placement problem (CCFP),* our goal is to place the same (large) portion from each set. We look for a *perfect placement*, in which both problems are solved optimally. We first show that the two problems are NP-hard; we then consider some special cases, where a perfect placement exists and can be found in polynomial time. For other cases, we give approximate solutions. Finally, we give a nearly optimal solution for the CCMK. Our results for the CCMK and the CCFP are shown to provide efficient solutions for two fundamental problems in multimedia storage sub-systems.

2.1 Definitions and Notation

Given N knapsacks with the volumes $v_1, \ldots, v_N$, the *packing potential* of the knapsacks, denoted by V, is the total number of items that can be placed in the knapsacks, i.e., $V = \sum_{j=1}^{N} v_j$. For a universe I of unit size items, partitioned to the sets $I_1, \ldots, I_M$, a solution to the CCMK or the CCFP can be represented as two $M \times N$ matrices:

1. The *indicator matrix*, R, a $\{0,1\}$-matrix, $R_{i,j} = 1$ iff a compartment of K_j was allocated to items of type I_i.

2. The *quantity matrix* Q, $Q_{i,j} \in \{0, 1, \ldots, v_j\}$, $Q_{i,j}$ is the number of items of I_i that are placed in K_j.

A legal placement has to satisfy the following conditions:

- $R_{i,j} = 0 \Rightarrow Q_{i,j} = 0$. This condition reflects the fact that items of I_i can be placed in K_j only if a compartment of K_j was allocated to items of type i.

- For each knapsack K_j, $\sum_i Q_{i,j} \leq v_j$, that is, the total number of items placed in K_j does not exceed its capacity.

- For each knapsack K_j, $\sum_i R_{i,j} \leq c_j$, that is, the number of different types of items placed in K_j does not exceed the number of compartments in K_j.

The matrices R and Q determine a subset of items $I' = I'_1 \cup \ldots \cup I'_M$, which is placed into the knapsacks.

Definition 2.1.1 *Given a solution for the CCMK (CCFP), the* packed quantity of I_i, *denoted by* Q_i, *is the total number of items packed from* I_i. *Thus,* $Q_i = |I'_i| = \sum_{j=1}^{N} Q_{i,j}$.

2.1.1 Utilization of a Placement

Our first measure for the quality of a placement is utilization:

Definition 2.1.2 *The* utilization of a placement *is* $\sum_{i=1}^{M} Q_i$.

The maximal possible utilization of a placement is V, meaning that all the packing potential of the knapsacks is exploited. Since $\sum_{i=1}^{M} |I_i| = V$, it also means that all the $|I_i|$ items of the set I_i are packed. Other placements may utilize only part of the overall packing potential:

Definition 2.1.3 *A placement is* γ-utilized *if its utilization equals* $\gamma \cdot V$, *for some* $\gamma \in [0, 1]$.

Our main questions here are: "Can the maximal possible utilization be found in polynomial time?", "Can we find an efficient approximation?"

The CCMK aims at maximizing the utilization of the packing potential: in Section 2.3 we present a *dual approximation algorithm* for the CCMK. In a dual approximation algorithm we approximate the *feasibility* of a solution for a given problem, rather than its optimality; traditional approximation algorithms seek feasible solutions that are suboptimal, where the performance of the algorithm is measured by the degree of suboptimality allowed. In dual approximation algorithm the objective is to find an infeasible solution that is superoptimal; the performance of the algorithm is measured by the degree of infeasibility allowed. The dual approximation algorithm we present in Section 2.3 is superoptimal for the CCMK. Our algorithm allows a small degree of infeasibility, that is, at most one compartment is added to each of the knapsacks.

2.1.2 Fairness of a Placement

Our second criterion for measuring the quality of a placement is fairness:

Definition 2.1.4 *A placement is* γ-fair *if for every set* I_i, $Q_i \geq \gamma \cdot |I_i|$, *for some* $\gamma \in [0, 1]$.

An optimal placement is 1-fair. In a 1-fair placement, for each i, exactly $|I_i|$ items from the set I_i are packed. Since $\sum_{i=1}^{M} |I_i| = V$, it also means that the packing potential of the knapsacks is fully exploited.

Several questions arise when looking for a fair placement: "Does a 1-fair placement exist for any instance of the problem?" "Can we find it efficiently?" "When a 1-fair placement does not exist, can we find (or approximate) an optimal placement efficiently?"

2.1.3 Combining Utilization and Fairness

The Perfect Placement Problem

We first explore the relation between the CCMK and the CCFP.

Definition 2.1.5 *A* perfect placement *is a placement in which all the items of all the sets are packed, and all the knapsacks are full.*

Clearly, any perfect placement is 1-fair and 1-utilized.

We now show that for some instances a perfect placement does not exist. Consider a simple system consisting of two knapsacks, with $c_1 = c_2 = 1$ and $v_1 = v_2 = 10$, and two sets of items: $|I_1| = 15$ and $|I_2| = 5$. The only legal placements are:

1. Each set is packed into a different knapsack. Ten items of I_1 and five items of I_2 are packed.
2. Both compartments are allocated to I_1, or both compartments are allocated to I_2. Clearly, these placements are 0-fair.

Note, that by increasing c_1 to 2 we obtain an instance, for which a perfect placement exists: now we can place items of I_1 into both knapsacks and choose $Q_{1,1} = Q_{2,1} = 5$ and $Q_{1,2} = 10$.

When a perfect placement does not exist, we would like to find the best possible one. However, the two goals of utilization and fairness may conflict. Consider an instance with two knapsacks: $v_1 = 20$, $c_1 = 2$; $v_2 = 10$, $c_2 = 1$; and three sets of items: $|I_1| = 14$, $|I_2| = 14$, $|I_3| = 2$.

A placement which achieves the maximal utilization is presented in Figure 2.1(a): 28 items are packed, i.e., this placement is $\frac{28}{30}$-utilized. However, it is 0-fair - no item of I_3 is packed. Figure 2.1(b) presents the best possible placement with respect to fairness. It is $\frac{10}{14}$-fair and $\frac{26}{30}$-utilized. Generally, any γ-fair placement is at least γ-utilized.

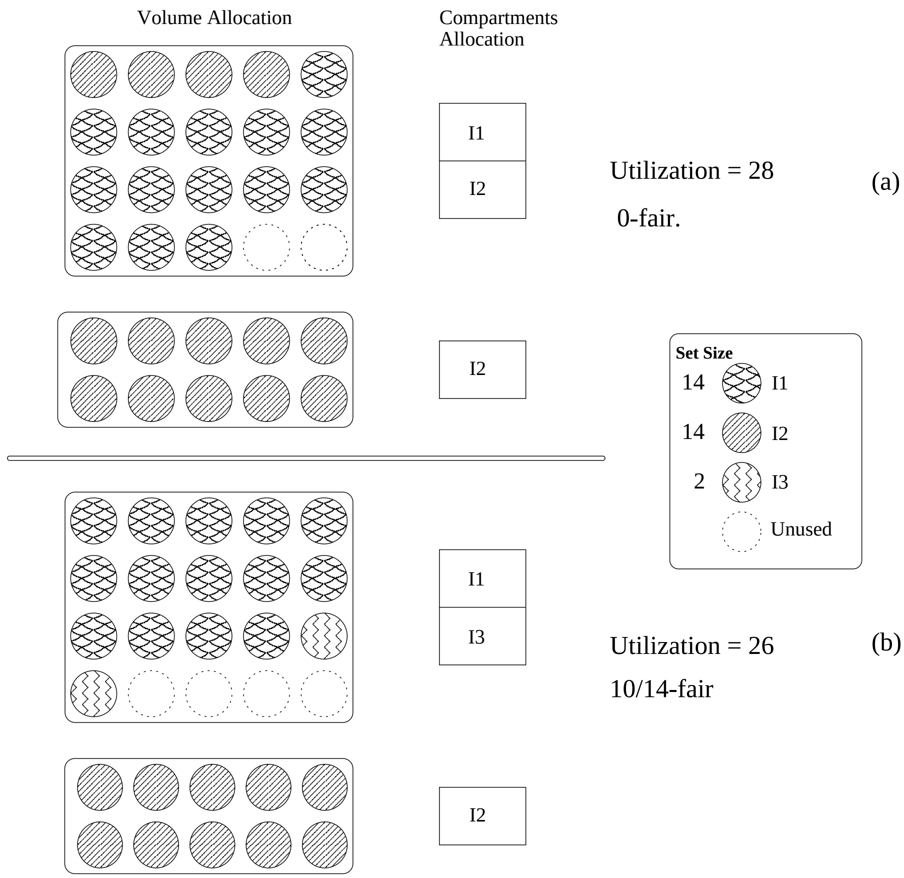

Figure 2.1: Maximal utilization versus maximal fairness

Simple Algorithms

In this section we show that two simple greedy algorithms are not suitable for the CCMK and the CCFP. The first algorithm (presented in [56], in the context of MOD systems) can be used for the CCFP: the algorithm attempts to guarantee "fairness" by allocating to each set of items a number of compartments, that is proportional to its size. Specifically, using an apportionment procedure, the algorithm first determines the number of compartments, R_i, that will be allocated to I_i, $1 \leq i \leq M$; then it selects a subset of R_i knapsacks that will store items from I_i. Finally, the volumes of the knapsacks are split among the sets, so as to achieve maximal fairness.

To realize that this algorithm is not suitable for any of our knapsack problems, consider an instance which consists of three sets, $|I_1| = 6$, $|I_2| = |I_3| = 3$, and eight knapsacks, with $c_j = 1$ for $1 \leq j \leq 8$. The volume of the first knapsack is $v_1 = 5$, and the volumes of the remaining knapsacks are $v_2 = v_3 = \ldots = v_8 = 1$; thus the packing potential of the knapsacks is $V = 12$. The total number of compartments is eight, therefore, by the 'number of compartments' criterion, since half of the items belong to I_1, four compartments should be allocated to this set, while I_2 and I_3 should be placed into two knapsacks each. Clearly, no fair placement exists under these conditions: the set whose items are stored in the first knapsack is allocated extra volume, while the other two sets are "discriminated".

In the CCFP our goal is to find placements in which Q_i, the packed quantity of I_i, reflects its size. Note that in the above example, if the number of compartments allocated to each set of items is not determined prior to the placement, then a *perfect* placement exists: for example, we can place the items of I_1 into K_1 and K_2, items of I_2 into K_3, K_4, and K_5; and items of I_3 into K_6, K_7 and K_8. Indeed, items of I_1 are placed into only two knapsacks, but since one of these knapsacks is large (K_1), all the items of I_1 can be packed.

The second algorithm is based on the *Longest Remaining Time First (LRTF)* algorithm [27]. LRTF provides a $(\frac{4}{3} - \frac{1}{n})$-approximation for the makespan problem, where n is the number of machines. When LRTF is adapted to the CCFP, we place items from the largest remaining set into the knapsack with the largest remaining volume.

Note that this algorithm can yield poor, non-fair placements. Consider an instance with two knapsacks: $v_1 = L + 1, c_1 = L$ and $v_2 = L, c_2 = 1$, for some $L > 1$; suppose that there are $L + 1$ sets, $|I_1| = L + 1$, and in the L remaining sets $|I_i| = 1$. The optimal placement packs the L items of the small sets into K_1 and L items of I_1 into K_2. This is an $\frac{L}{L+1}$-fair and $\frac{2L}{2L+1}$-utilized placement. The LRTF algorithm first chooses $Q_{1,1} = L + 1$ and one more item from another set is placed in K_2. Then no available compartments are left in K_2. The resulting placement is 0-fair and $\frac{L+2}{2L+1}$-utilized.

Alternatively, consider a variant of the LRTF, which sorts the knapsacks by the ratio $\frac{v_j}{c_j}$. Then items from the largest remaining set are placed into the knapsack in which the (updated) ratio $\frac{v_j}{c_j}$ is maximal. Note that this algorithm also yields inefficient solutions for the CCFP. Consider, e.g., two knapsacks with $v_1 = 7, c_1 = 1$ and $v_2 = 12, c_2 = 2$; and three sets with $|I_1| = 10, |I_2| = 7$, and $|I_3| = 2$. In the optimal placement I_2 is placed in K_1, while I_1 and I_3 are placed in K_2. The above algorithm initially packs seven items of I_1 into K_1, and cannot complete the packing of all the items. In particular, it is 0-fair for I_3. Using slightly different rule, which places items from the set with the largest remaining 'unpacked fraction', into the knapsack with the largest (updated) $\frac{v_j}{c_j}$ ratio, may increase fairness, with a corresponding decrease in utilization. A more complicated algorithm, which combines sorting by the ratios v_j/c_j with other ideas, can be useful in obtaining approximation algorithms for the CCMK (We elaborate on that in Section 2.3).

2.1.4 Hardness of the Perfect Placement Problem

The next question we consider is whether we can detect efficiently if a perfect placement exists.

Theorem 2.1.1 *Given N knapsacks with volumes v_j, and c_j compartments in knapsack j, and sets of items $I_1, \ldots, I_M$, it is NP-hard to determine if a perfect placement exists for this instance.*

Proof: We show a reduction from the *partition problem*, which is known to be NP-hard [20]. The partition problem consists of a finite set A, and size $s(a)$ for each $a \in A$. The problem is to determine if there exists a subset A' of A such that $\sum_{i \in A'} s(i) = \sum_{i \in A \setminus A'} s(i)$.

Given an instance for partition, consider the placement problem consisting of two sets $|I_1| = |I_2|$, and $|A|$ knapsacks with $c_j = 1$ and $v_j = s(a_j), \forall 1 \leq j \leq |A|$. For this problem, every perfect placement induces a desired partition and vice-versa. ■

Any perfect placement is both 1-fair and 1-utilized. Therefore, the above reduction is suitable for the CCMK and the CCFP; we conclude that each of these problems is NP-hard.

2.2 Finding a Perfect Placement

2.2.1 Simple Conditions for the Existence of a Perfect Placement

In this section we present simple conditions for the existence of a perfect placement. The first one considers inputs in which the 'number of compartments' constraint can be ignored. Formally, if for all the knapsacks, $c_j \geq v_j$, then a perfect placement exists, and can be found in polynomial time (Observe that if $c_j \geq v_j$ for all $1 \leq j \leq N$, then we can greedily place the sets into the knapsacks, until all the items are packed).

The next simple condition considers the *sizes* of the different sets.

Theorem 2.2.1 *Let $0 < \varepsilon \leq 1$ be the maximal number such that for all $1 \leq i \leq M$, $|I_i| \geq \frac{\varepsilon \cdot |I|}{M}$. If, for all the knapsacks, $c_j \geq \frac{v_j \cdot M}{\varepsilon \cdot V} + 1$, then there exists a perfect placement which uses at most $M + N - 1$ different compartments.*

Proof: Consider the simple greedy algorithm, which packs the sets of items by filling the knapsacks one after the other. Specifically, the knapsack K_j is filled until it contains v_j items, we then continue to pack into K_{j+1}, and so on. Since $\sum_i |I_i| = |I| = V = \sum_j v_j$, the algorithm terminates when the last knapsack, K_N is filled with the last item of I_M. Thus, all the packing potential of the knapsacks is exploited and all the items are packed. Intuitively, the conditions indicate that even the smallest sets are large enough to fill any knapsack. Formally, $|I_i| \geq \frac{\varepsilon \cdot |I|}{M}$ implies that any subset of $c_j - 1$ sets includes at least $(c_j - 1)|I| \cdot \frac{\varepsilon}{M} = (c_j - 1)V \cdot \frac{\varepsilon}{M}$ items. Since, $c_j \geq \frac{v_j \cdot M}{\varepsilon \cdot V} + 1$, the subset's total size is at least v_j. The additional compartment is needed for small number of items of the first set placed in K_j - for this set, the size is not predicted, since some of its items were already placed in previous knapsacks.

If all the items of one set are placed in one knapsack, we use only one compartment for this set. If it is placed in k different knapsacks, we use k compartments for this set. Since the algorithm makes at most $N-1$ splits, the total number of compartments used by this algorithm is $M+N-1$. ■

2.2.2 Uniform Capacity Ratio

In this section we present an alternative condition for the existence of perfect placement. We require that the volume and the number of compartments in a knapsack be correlated - knapsacks with high volume should have many compartments and vice-versa.

Definition 2.2.1 *For any* $1 \leq j \leq N$*, the* capacity ratio *of* K_j *is* $\frac{v_j}{c_j}$*. The set of knapsacks* $K_1, \ldots, K_N$ *has a* uniform capacity ratio *if there exists a constant* $r > 0$*, such that* $\frac{v_j}{c_j} = r$*,* $\forall 1 \leq j \leq N$*.*

Intuitively, the capacity ratio of a knapsack K_j gives the average number of items contained in each compartment when the volume of K_j is totally utilized. We show that, for a set of knapsacks with uniform capacity ratio, if $\sum_{j=1}^{N} c_j \geq M + N - 1$, then a perfect placement exists and can be found efficiently. This holds for *any* distribution on the sizes of the sets $I_1, \ldots, I_M$.

Theorem 2.2.2 *If the capacity ratio is uniform and* $\sum_{j=1}^{N} c_j \geq M + N - 1$*, then a perfect placement exists and can be found in* $O(\max(N \log N, M \log M))$ *steps.*

Proof: We present a polynomial time algorithm which terminates with a perfect placement. The algorithm, denoted by $\mathcal{A}_u$, proceeds by placing exactly v_j items of at most c_j different sets into each of the knapsacks. We assume that the sets are given in a non-decreasing order of their sizes, i.e., $|I_1| \leq |I_2| \leq, \ldots, \leq |I_M|$.

In each step we keep the remainders of the sets in a sorted list, denoted by L. The list, $L[1], \ldots, L[m]$, $1 \leq m \leq M$, is updated during the algorithm, that is, we remove from L sets that were fully packed, and we move to their updated place, sets that were only partially packed. L is the *volume request list*, that is, each entry of L represents the request for volume, which is the number of unpacked items, of some set I_i. The knapsacks are given in a non-decreasing order of the number of compartments they have, i.e., $c_1 \leq c_2 \leq, \ldots, \leq c_N$.

The main idea in the algorithm is to keep the average size of the requests in L large enough. Specifically, for each $1 \leq k \leq N$, we will show that when the knapsack K_k is filled, either the average request size is at least r, or a trivial greedy algorithm can be applied to pack the remaining requests in the remaining knapsacks.

The algorithm uses two knapsack-filling procedures: some of the knapsacks are filled using the *greedy-filling* procedure; the other knapsacks will be filled using the *moving-window* procedure. We now describe the two procedures.

Greedy-Filling:
The greedy-filling procedure fills a knapsack with items, starting from the smallest set, $L[1]$, and continuing until the knapsack is *saturated*, that is, until it contains exactly v_j items. The last

Greedy-Filling(j)
$i \leftarrow 1$
repeat
  Allocate one compartment of K_j for items of the set I_k corresponding to $L[i]$.
  Place items of I_k in K_j: $Q_{k,j} = \min\{L[i], v_j\}$
  $v_j \leftarrow v_j - Q_{k,j}$
  $i \leftarrow i + 1$
until $v_j = 0$
Remove from L the sets that were fully packed into K_j.
Update the size of the new smallest set that was only partially packed into K_j.
Remove K_j from the knapsack list.

Figure 2.2: The *greedy-filling* procedure

set may split, that is, only part of its items will be packed into K_j. A formal description of the greedy-filling procedure is given in Figure 2.2.

Moving-Window:
The moving-window procedure fills a knapsack, K_j, with the first sequence of c_j sets whose total size is at least v_j. We search the list L using a moving window of size c_j. Initially, the window covers the set of the smallest c_j sets. In every iteration we replace the smallest set in the window by the next set in L, until the number of items that are contained in the window is large enough to saturate K_j. If the subset of the c_j largest sets includes less than v_j items, then we cannot saturate K_j. However, we show below that this never happens. A formal description of the moving-window procedure is given in Figure 2.3. Note that the window advances until it covers c_j sets, such that $v_j > L[i-1] + \ldots + L[i+c_j-2]$ and $v_j \leq L[i] + \ldots + L[i+c_j-1]$ (see Figure 2.4). At this stage we can clearly place in K_j all the items in the sets $L[i], \ldots, L[i+c_j-2]$, and saturate K_j by adding some of the items in the set $L[i+c_j-1]$.

The Algorithm $\mathcal{A}_u$: The algorithm proceeds in iterations, in the jth iteration we fill K_j by the following rules:

1. If there are less than c_j sets, or if the subset of the c_j smallest sets contains more than v_j items, then fill K_j using the greedy procedure.

2. If the subset of the c_j smallest sets contains at most v_j items, then fill K_j using the moving-window procedure.

We now show that the algorithm terminates with a perfect placement. We distinguish between two phases in the execution of $\mathcal{A}_u$:

Moving-Window(j)
$i \leftarrow 1$
$a = L[1] + \ldots + L[c_j]$.
while $(a < v_j$ and $|L| \geq c_j + i)$
    $a \leftarrow a - L[i] + L[c_j + i]$
    $i \leftarrow i + 1$
Fill K_j:
    Allocate compartments of K_j to all the sets corresponding to the requests
        $L[i], \ldots, L[i + c_j - 1]$.
    For each k, $i \leq k \leq i + c_j - 2$
        Place in K_j $L[k]$ items of the corresponding set.
    $Q_{k',j} = v_j - (L[i] + \ldots + L[i + c_j - 2])$.
    Place in K_j $Q_{k',j}$ items of the set $I_{k'}$ corresponding to $L[i + c_j - 1]$.
Update the list L:
    Remove from L the sets $L[i], \ldots, L[i + c_j - 2]$.
    If the set $I_{k'}$ corresponding to $L[i + c_j - 1]$ is not completely packed,
        Move $L[i + c_j - 1]$ to its appropriate new place in the sorted list L.
Remove K_j from the knapsack list.

Figure 2.3: The *moving-window* procedure

1. In the first phase, each knapsack K_j that is filled contains $c_j - 1$ or c_j sets of items. This phase includes executions of the moving-window procedure and some executions of the greedy procedure.

2. The second phase starts when, for the first time, the number of sets placed in K_j, for some $1 \leq j \leq N$, is smaller than c_j. This can clearly happen only when the greedy procedure is used.

Note that for some inputs, one of the phases may not occur. For each of the two phases, we show that any knapsack K_j that is filled during this phase will contain exactly v_j items of at most c_j different sets.

We use the following notation:

- C^k is the total number of compartments that are available after k knapsacks are filled.
- M^k is the number of sets that are not fully packed after k knapsacks are filled.
- N^k is the number of empty knapsacks after k knapsacks are filled ($N^k = N - k$).

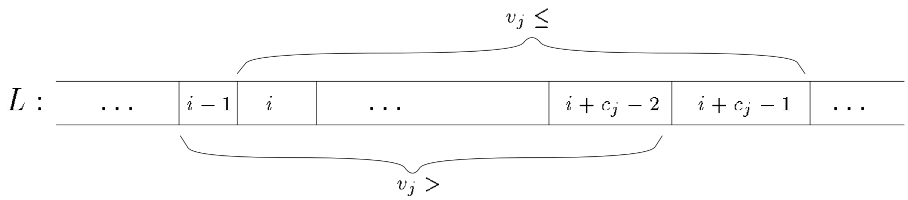

Figure 2.4: The window scanning the set-list L

- r is the capacity ratio, that is, $\forall j, \frac{v_j}{c_j} = r$.

We now consider the first phase.

Lemma 2.2.3 *Each of the knapsacks that are filled during the first phase of $\mathcal{A}_u$ will contain exactly v_j items that belong to at most c_j different sets.*

Proof: We show the following invariant for the ratio between the number of remaining sets, knapsacks, and compartments, during the first phase. Let K_b be the first knapsack filled in the second phase: b can be any number between 1 and N.

Claim 2.2.4 *For every $0 \leq k < b$, $C^k \geq M^k + N^k - 1$.*

Proof: The proof is by induction on k, the number of knapsacks that were already filled.
Base: $k = 0$. It is given that $\sum_{j=1}^{N} c_j \geq M + N - 1$. Thus, using our notation, $C^0 \geq M^0 + N^0 - 1$.
Induction Step: Consider the kth iteration, in which K_k is filled.

1. After this iteration K_k is no longer available, therefore $C^k = C^{k-1} - c_k$.
2. By the definition of the first phase, we fully pack in this iteration $c_k - 1$ or c_k sets. Therefore, $M^k = M^{k-1} - (c_k - 1)$ or $M^k = M^{k-1} - c_k$. Hence, $M^k \leq M^{k-1} - (c_k - 1)$, that is, $M^{k-1} \geq M^k + c_k - 1$
3. Clearly, $N^k = N^{k-1} - 1$.

By the induction hypothesis, $C^{k-1} \geq M^{k-1} + N^{k-1} - 1$, therefore

$$\begin{aligned} C^k &= C^{k-1} - c_k \\ &\geq M^{k-1} + N^{k-1} - 1 - c_k \\ &\geq M^k + c_k - 1 + N^k + 1 - 1 - c_k \\ &= M^k + N^k - 1 \end{aligned}$$

■

Claim 2.2.5 *For every $1 \leq k < b$, the average set size when we fill K_k is at least r.*

Proof: Using Claim 2.2.4, after $k-1$ knapsacks are filled, the average set size is

$$\frac{\text{number of remaining items}}{\text{number of remaining sets}} \geq \frac{\text{remaining packing potential}}{\text{number of remaining sets}}$$

$$= \frac{r \cdot C^{k-1}}{M^{k-1}} \geq \frac{r \cdot (M^{k-1} + N^{k-1} - 1)}{M^{k-1}} \geq r.$$

Recall that at the beginning $\sum_{i=1}^{M} |I_i| = \sum_{j=1}^{N} v_j$. Therefore, at any phase, the number of non-packed items is at least the remaining packing potential. Also, $N^{k-1} \geq 1$ since at least K_k is still empty after $k-1$ knapsacks are filled. ∎

We conclude that, for every $k < b$, the largest c_k sets after $(k-1)$ iterations contain at least $r \cdot c_k$ items, which is equal to v_k. This means that if the moving-window procedure is applied, the window never reaches the end of L (the largest sets) without saturating K_k. Knapsacks that are filled by the greedy procedure are clearly saturated, since the greedy procedure is applied when even the smallest c_k sets are large enough to saturate K_k. ∎

We now analyze the second phase of the algorithm. Note that we reach the second phase when the smallest $c_b - 1$ sets together contain more than v_b items.

Lemma 2.2.6 *Each of the knapsacks that are filled during the second phase of $\mathcal{A}_u$ will contain exactly v_j items that belong to at most c_j different sets.*

Proof: Let L_b be the set list at the end of the first phase. We first show that for every $j \geq b$, K_j can be saturated by any subset of $c_j - 1$ sets from L_b.

Claim 2.2.7 *At the beginning of the second phase, for every $j \geq b$, any subset of $c_j - 1$ sets contains more than v_j items.*

Proof: By definition, K_b is the first knapsack filled in the second phase. Therefore less than $c_b - 1$ sets are packed into K_b. This can happen only if the subset of the $c_b - 1$ smallest sets contains more than $v_b = r \cdot c_b$ items. Therefore the average size of the smallest $c_b - 1$ sets is larger than $\frac{r \cdot c_b}{c_b - 1}$. Consider a knapsack $K_j, j > b$. The knapsacks are sorted such that $c_j \geq c_b$, and the sets are sorted in a non-decreasing order of their sizes, therefore the average size of the smallest $c_j - 1$ sets is larger than $\frac{r \cdot c_b}{c_b - 1}$. In addition, since $c_j \geq c_b$,

$$\frac{r \cdot c_b}{c_b - 1} \geq \frac{r \cdot c_j}{c_j - 1}.$$

This means that the total number of items in the smallest $c_j - 1$ sets is larger than

$$(c_j - 1) \cdot \frac{r \cdot c_j}{c_j - 1} = r \cdot c_j = v_j.$$

Clearly, if the smallest $c_j - 1$ sets in L_b contain more than v_j items, then any $c_j - 1$ sets in L_b contain more than v_j items. ∎

We now use Claim 2.2.7 to show that any knapsack filled after K_b contains at most $c_j - 1$ different sets, and is saturated by additional items packed from one set of L_b.

By the greedy-filling procedure, K_b is filled until v_b items are packed. The last set packed into K_b may include some more unpacked items. The fraction left from the split set is now the smallest set in L_b (since even before the split it was part of the smallest available set). In other words, the fraction is $L[1]$. We now turn to fill K_{b+1}. Consider the smallest c_{b+1} sets. These sets are composed of $L[1]$ and additional $c_{b+1} - 1$ sets. By Claim 2.2.7 the additional $c_{b+1} - 1$ sets include more than v_{b+1} items, therefore (no matter what the size of the fraction $L[1]$), the smallest c_{b+1} sets include together more than v_{b+1} items, and the greedy-filling procedure is used: v_{b+1} items of at most c_{b+1} sets are placed in K_{b+1}. Again, the last set may be only partially packed. The same argument holds until we reach a knapsack K_l, such that less than c_l sets are left. Since $\sum_{i=1}^{M} |I_i| = \sum_{j=1}^{N} v_j$ at the beginning, and since all the knapsacks before K_l were saturated, the total number of non-packed items equals the total left volume. Hence, we can fill K_l using the greedy procedure. This argument holds for all the remaining knapsacks, until K_N is saturated and no sets are left. ∎

Combining Lemmas 2.2.3 and 2.2.6 we conclude that each of the knapsacks K_j, $1 \le j \le N$, is filled by exactly v_j items of at most c_j different sets. Thus, the algorithm terminates with a perfect placement.

We now turn to compute the running time of $\mathcal{A}_u$. The algorithm $\mathcal{A}_u$ can be implemented in $O(\max(M \log M,\ N \log N))$ steps: $O(M \log M) + O(N \log N)$ steps are required for sorting. Given that the knapsacks and the sets are sorted, the total time for filling the knapsacks is $O(M + N) + O(N \log M)$. The first phase of the algorithm, in which we fill the knapsacks by scanning the list L with the moving-window, can be implemented in $O(M) + O(N \log M)$ steps. The idea is that there is no need to scan the list L from the beginning for each knapsack. Note that the each knapsack $K_j, j > 1$ is filled by items from exactly c_j sets. Thus, the location in L of these c_j consequent sets must encircle the largest set with at most $\frac{v_j}{c_j}$ items. Since L is sorted, we can find this set (e.g., using skip-lists) in $O(\log M)$ steps. We can now find in $O(c_j)$ steps the set of c_j sets that will fill K_j. Finally, after we fill K_j, we have to reposition in L the c_jth set in the window according to its remaining size. Since L is sorted, this can be implemented in $O(\log M)$ steps.

Let P_1 denote the set of knapsacks filled in the first phase. Recall that when the list L includes less than c_j sets (when K_j is the next knapsack to be filled), we move to the second phase. Thus, $\sum_{K_j \in P_1} c_j \le M$, and therefore, the total time for locating the windows and fill all the knapsacks in P_1 is $O(N \log M) + O(M)$. During the second phase of the algorithm, we fill the knapsacks greedily, in $O(M + N)$ steps. ∎

2.2.3 Approximating a Perfect Placement

When the knapsacks do not have uniform capacity ratio, the degree of non-uniformity is measured by the minimal α such that, for some $r > 0$,

$$\forall j,\ r \leq \frac{v_j}{c_j} \leq \alpha \cdot r. \tag{2.1}$$

(When $\alpha = 1$ we have uniform capacity ratio, in any other case $\alpha > 1$). In this section we show how a perfect placement can be approximated. The approximation ratio is proportional to α.

Theorem 2.2.8 *Let $\alpha \geq 1$ be the minimal number satisfying (2.1). If $\sum_{j=1}^{N} c_j \geq M + N - 1$, then a $1/\alpha$-utilized placement can be found in polynomial time.*

Proof: Consider the following instance, A', of the placement problem: for each knapsack K_j, $1 \leq j \leq N$, the volume of K_j in A' is $v'_j = \lceil r \cdot c_j \rceil$. Let $V' = \sum_{j=1}^{N} v'_j$. For each set, I_i, the set size of I_i in A', denoted by $|I'_i|$ is determined by solving the following MAX-MIN problem:

$$\text{Maximize} \quad \min \frac{|I'_i|}{|I_i|}, \quad \text{such that } \sum_{i=1}^{M} |I'_i| = V', \quad |I'_i| \text{ integer.} \tag{2.2}$$

First, note that any legal placement of A' induces a legal placement for the original instance. This follows from the next claim:

Claim 2.2.9 *For each $1 \leq j \leq N$, $v'_j \leq v_j$.*

Proof: For each $1 \leq j \leq N$, v_j is an integer. Therefore, for the knapsack achieving the minimal capacity ratio, $v_j = r \cdot c_j = \lceil r \cdot c_j \rceil = v'_j$. For each knapsack K_j such that $\frac{v_j}{c_j} > r$, there exists some $\varepsilon > 0$ such that $v_j = (r + \varepsilon) \cdot c_j = \lceil (r + \varepsilon) \cdot c_j \rceil \geq \lceil r \cdot c_j \rceil = v'_j$. ∎

Next, we show that a perfect placement exists, and can be found efficiently for A'. Note that the knapsacks in A' do not necessarily have uniform capacity ratio, however, the non-uniformity is small enough to show that the algorithm $\mathcal{A}_u$ presented in Section 2.2.2 is suitable for A'. We follow the proof of Theorem 2.2.2 to show that $\mathcal{A}_u$ fills each knapsack with exactly v'_j items of at most c_j sets. It is easy to verify that the Claims 2.2.4 and 2.2.5 hold for A'. Let K_k be a knapsack filled during the first phase of $\mathcal{A}_u$. By Claim 2.2.5, the average set size when we fill K_k is at least r. Indeed, v'_j may be larger than $r \cdot c_j$, however, since the number of items included in the window is an integer and since $v'_j = \lceil r \cdot c_j \rceil$ is the smallest integer not smaller than $r \cdot c_j$, we conclude that whenever the moving-window procedure is applied, the window does not reach the end of L without saturating K_k. Similarly, for the second phase of $\mathcal{A}_u$, Claim 2.2.7 holds for A' and we conclude that all the knapsacks are saturated. By Claim 2.2.9, a legal placement in A' is a legal placement for the original instance. Since $V' \geq \frac{1}{\alpha} V$, a perfect placement in A' induces a placement that is $\frac{1}{\alpha}$-utilized for the original instance. ∎

Remark 2.2.1 *For solving the CCMK, any choice of set sizes that satisfies* $\forall i, |I'_i| \leq |I_i|$ *and* $\sum_{i=1}^{M} |I'_i| = V'$ *can be applied here. The solution of (2.2) also solves efficiently the CCFP: for each set* I_i, $|I'_i| \geq \frac{1}{\alpha}|I_i| - 1$.

In order to achieve a good approximation we would like to have α as close as possible to 1. In Section 2.4 we show that if compartments can be moved among the knapsacks, then $\alpha = \frac{c_k}{c_k - 1}$, where K_k is the knapsack having the maximal capacity ratio. This model is relevant to MOD storage sub-systems, in which storage resources can be moved among the disks.

2.3 Approximation Algorithm for the CCMK

In this section we present a dual approximation algorithm for the CCMK. Specifically, we show that it is sufficient to add one compartment to each knapsack, in order to eliminate the gap between the performance of an optimal, probably exponential time algorithm, and a polynomial time algorithm. Recall that a *dual approximation algorithm* finds an infeasible solution that is superoptimal. Its performance is measured by the degree of infeasibility allowed. The proposed algorithm is allowed to place items of $c_j + 1$ (instead of c_j) different sets into K_j, $1 \leq j \leq N$.

Theorem 2.3.1 *Given an instance* A *of the CCMK, by adding a single compartment to each knapsack, we can find in polynomial time a placement, whose utilization is at least the maximal possible utilization for* A.

Proof: Let A be an instance of the CCMK. Denote by A^+ the instance generated from A by adding one compartment to each knapsack. We present a polynomial time algorithm which finds a legal placement in A^+. The algorithm, denoted by $\mathcal{A}_r$, proceeds by filling K_j with at most v_j items of at most $c_j + 1$ different sets, $\forall 1 \leq j \leq N$. The total number of items packed from A^+ by $\mathcal{A}_r$ is at least the total number of items packed from A by an optimal algorithm.

As in the uniform-ratio case (Section 2.2.2), we assume that the sets of items are given in non-decreasing order of their sizes, i.e., the sets satisfy $|I_1| \leq |I_2| \leq, \ldots, \leq |I_M|$. We keep the remaining sets in a sorted list, denoted by L, which is updated during the algorithm. For simplicity, we use $L[i]$ to denote both the set in position i in L and the size of this set. $\mathcal{A}_r$ uses the two knapsack-filling procedures: *greedy-filling* and *moving-window* introduced in Section 2.2.2. However, the moving-window procedure is slightly changed: now the window covers $c_j + 1$ sets (instead of c_j). Also, if the subset of the $c_j + 1$ largest sets includes less than v_j items, then we cannot saturate K_j, and we do the best we can: we fill K_j with these $c_j + 1$ sets (in the uniform-ratio case this never happens). The knapsacks are given in a non-increasing order by their *capacity ratio*, i.e., $\frac{v_1}{c_1} \geq \frac{v_2}{c_2} \geq, \ldots, \geq \frac{v_N}{c_N}$.

<u>**The Algorithm $\mathcal{A}_r$:**</u> At first, the algorithm uses the moving-window procedure to fill the knapsacks one after the other, that is, in the jth iteration we fill K_j. Note that the moving-window

procedure places in K_j items of exactly c_j+1 sets. Sometimes, it is not possible to pack from that amount of sets (see below). When this happens for the first time we turn to fill all the remaining knapsacks using the greedy-filling procedure.

The following rules are used when $\mathcal{A}_r$ fills K_j.

- If there are less than $c_j + 1$ sets in L or if the subset of the c_j smallest sets in L contains at least v_j items, fill sequentially K_j and all the remaining knapsacks using the greedy-filling procedure.
- Otherwise (the subset of the c_j smallest sets in L contains less than v_j items), fill K_j with items from $c_j + 1$ different sets, using the moving-window procedure.

Optimality: Denote by $G = \{K_{g_1}, K_{g_2}, \ldots, K_{g_n}\}$ the set of knapsacks that are not saturated by $\mathcal{A}_r$, and denote by $w_1, w_2, \ldots, w_n$ the resulting waste of volume in each non-saturated knapsack, i.e., the volume of K_{g_i} is not fully exploited and only $v_{g_i} - w_i$ items are placed in K_{g_i}. We show that there is no legal placement of A in which the total utilization exceeds $\sum_{j=1}^{N} v_j - (w_1+w_2+\ldots+w_n)$. Let K_b be the first knapsack that is filled greedily. We distinguish between two phases of $\mathcal{A}_r$:

1. The knapsacks $K_1, \ldots, K_{b-1}$ are filled (using the moving-window procedure).
2. The knapsacks $K_b, \ldots, K_n$ are filled (using the greedy-filling procedure).

For the first phase we show that the total waste of volume for A^+ is at most the total waste of volume in an optimal placement of A. For the second phase we show that there is no waste, and all the knapsacks filled during this phase are saturated.

For simplicity, assume that whenever the moving-window procedure is executed, the list L is scanned from left to right, that is, the smallest set, $L[1]$, is the leftmost set and the largest set is the rightmost in L. During execution of the moving-window procedure, the window moves from left to right. There are two possible scenarios:

1. **Saturating:** in which K_j is saturated, meaning that there exists some i, which is the index in L of the smallest set in the window, such that

   (i) $(i > 1$ and $v_j > L[i-1] + \ldots + L[i+c_j-1])$ or $(i = 1$ and $v_j > L[1] + \ldots + L[c_j])$.

   (ii) $v_j \le L[i] + \ldots + L[i+c_j]$.

   At this phase, we can clearly place in K_j the c_j sets $L[i], \ldots, L[i+c_j-1]$, and saturate K_j by adding some of the items of $L[i+c_j]$. The sets $L[i], \ldots, L[i+c_j-1]$ are removed from L, and the fraction left from the set $L[i+c_j]$ is moved to its updated place in the list L. Note that since some of the items of this set are packed, the position of that fraction in L is left to its original position.

2. **Non-saturating:** in which the window reaches the rightmost position in L, but the subset of the $c_j + 1$ largest sets covered by the window contains less than v_j items. All items in the sets $L[i], \ldots, L[i + c_j]$ are packed, and these sets are removed from L.

In both cases we can consider the removed sequence of sets as a *hole* in L. In a saturating execution, K_j creates a hole of c_j sets in L, and one additional set (the fraction left from the last set) is moved left of the hole. In a non-saturating execution, K_j creates a hole of $c_j + 1$ sets in L.

We examine the sequence of holes created in L during the first phase of the algorithm. We first show that the last non-saturated knapsack, K_{g_n}, creates at the right end of L, a hole which is the union of the holes in L created by the knapsacks in G and other holes created by other knapsacks.

Claim 2.3.2 *There exists a set $S \subseteq \{K_1, \ldots, K_{g_n}\}$, such that $G \subseteq S$ and the hole created by K_{g_n} unites the holes created by the knapsacks in S, into a single hole positioned at the right end of L.*

Proof: Consider the hole in L that contains the $c_{g_n} + 1$ sets packed in K_{g_n}, and all the holes that are united when these $c_{g_n} + 1$ sets are removed from L. Let H denote this united hole. Since each knapsack K_j, $j < g_n$ creates a hole of c_j *consecutive* sets in L, then for each such knapsack, either *all* or *none* of these c_j sets is contained in H (we ignore the $(c_j + 1)$th set: K_j may be filled with only some of the items in this set, in which case the set is not removed from L).

Let S be the set of all the knapsacks, K_j, that contribute c_j or $c_j + 1$ sets to H. By definition, the hole H unites the holes created by the jobs in S into a single hole. In addition, any non-saturated knapsack, $K_{g_j} \in G$, was filled before K_{g_n} and the hole it created includes the rightmost available set in L (at the time it was filled). Thus, the hole created by K_{g_j} is right to the hole created by K_{g_n}. Finally, since K_{g_n} is not saturated, H must include the largest $c_{g_n} + 1$ available sets, and in particular the rightmost one, thus, it unites all the holes created by previous knapsacks from G. We conclude that $G \subseteq S$ and that H is positioned at the right end of L. ∎

Lemma 2.3.3 *Let K_{b-1} be the last knapsack filled by $\mathcal{A}_r$ during the first phase, then the total number of items placed in $K_1, \ldots, K_{b-1}$ is at least the number of items placed into $K_1, \ldots, K_{b-1}$ in an optimal algorithm for the instance A.*

Proof: By Claim 2.3.2, the knapsacks in S are filled with at least all the items of the $\sum_{K_j \in S} c_j$ largest sets. Also, since $G \subseteq S$, knapsacks that are not in S are saturated. Thus, it is sufficient to show that there is no placement of the instance A, in which the total number of items packed in the knapsacks in S is larger than the total number of items in the $\sum_{K_j \in S} c_j$ largest sets.

Clearly, in any legal placement of the instance A, for any j, items from at most c_j different sets are placed in K_j. Moreover, for any set S of knapsacks, items from at most $\sum_{K_j \in S} c_j$ different sets are placed in the knapsacks of S. Thus, the total number of items in the $\sum_{K_j \in S} c_j$ largest sets is an upper bound for the total number of items packed in the knapsacks in S. ∎

For the first phase we have shown that the total volume wasted in knapsacks filled during this phase does not exceed the waste of volume in these knapsacks under an optimal placement for A. We complete the proof of optimality by showing that all the knapsacks which are filled during second phase of $\mathcal{A}_r$ are saturated.

Lemma 2.3.4 *Each of the knapsacks that are filled during the second phase of $\mathcal{A}_r$ contains exactly v_j items, which belong to at most $c_j + 1$ different sets.*

Proof: The second phase consists of successive executions of the greedy-filling procedure. $\mathcal{A}_r$ reaches the second phase if, for the next knapsack, K_b, the list L contains at most c_b sets, or if the subset of the c_b smallest sets contains at least v_b items.

In the analysis of the second phase, we have to consider two issues: that all the knapsacks are saturated, and that at most $c_j + 1$ sets participate in the filling of K_j.

In order to realize that the knapsacks are saturated, note that at the beginning of the algorithm $\sum_{i=1}^{M} |I_i| = \sum_{j=1}^{N} v_j$, that is, the total number of items is equal to the total available volume. Since during the first phase, no knapsack is filled by more than v_j items, the sum of the sizes of the remaining sets always exceeds the total remaining volume. This assures that the greedy procedure will never terminate because it run out of items.

Clearly, in the greedy filling of K_b, at most c_b sets are used. The last set from which items are packed into K_b may split. The fraction that is left is now the smallest set (since even before the split it was part of the smallest available set). In other words, the fraction is $L[1]$ (note that if the subset of the c_b smallest sets in L contains exactly v_b items, then we have no fraction at all. In this case, the size of the new $L[1]$ is at least the size of each of the sets that are removed from L). We now turn to fill greedily the remaining knapsacks. In order to show that for any $j > b$, at most $c_j + 1$ sets participate in the filling of K_j, we first prove the following claim:

Claim 2.3.5 *After K_b is filled greedily by the smallest sets, then for any $j > b$, the list L includes at most c_j sets, or the subset of the c_j sets $L[2], \ldots, L[c_j + 1]$ contains at least v_j items.*

Proof: Assume that x sets participate in the greedy filling of K_b. Clearly, $x \leq c_b$. Thus, the largest set among these x sets must contain at least $\frac{v_b}{x} \geq \frac{v_b}{c_b} \geq \frac{v_j}{c_j}$ items. When the list L is updated, the smallest $x - 1$ sets are removed, and the fraction of the largest (xth) set turn to be $L[1]$. Since the sets in L are sorted by non-decreasing order of their sizes, each of the sets $L[2], \ldots, L[c_j + 1]$ (assuming $|R| > c_j$), contains at least $\frac{v_j}{c_j}$ items, meaning that the subset of these c_j sets contains at least v_j items. ∎

Consider a knapsack $K_j, j > b$. If, after the filling of K_b, the list L includes at most c_j sets, then, clearly, K_j will be filled by at most c_j sets. Otherwise, note that if the c_j sets $L[2], \ldots, L[c_j+1]$ are large enough to fill K_j, then any c_j sets, which do not include $L[1]$, are large enough for K_j. We show that the greedy algorithm never uses more than $c_j + 1$ sets to fill K_j:

After K_b is filled, we turn to fill K_{b+1}. Consider the subset of the $c_{b+1} + 1$ smallest sets. It consists of $L[1]$ and additional c_{b+1} sets. By the above discussion, the subset of the additional c_{b+1} sets includes at least v_{b+1} items, therefore, no matter what the size of the fraction $L[1]$, the smallest $c_{b+1} + 1$ sets include together at least v_{b+1} items, and we can pack into K_{b+1} exactly v_{b+1} items from at most $c_{b+1} + 1$ sets. Again, the last set may split. The same argument holds for all the remaining knapsacks. That is, every knapsack K_j will contain exactly v_j items, that belong to at most $c_j + 1$ sets. ■

By combining Lemmas 2.3.3 and 2.3.4 we conclude that the total amount of wasted volume for the instance A^+ does not exceed the total amount of wasted volume under an optimal placement of A. In particular, if there exists a perfect placement of A, then our algorithm finds a perfect placement of A^+.

The running time of $\mathcal{A}_r$ is $O(M+N)\log(M+N)$. It can be achieved when the list L is kept as a skip-list, as detailed in the proof of Theorem 2.2.2. ■

The dual approximation scheme yields the following approximation algorithm for the CCMK:

1. Find an optimal solution, assuming $c_j = c_j + 1, \forall 1 \leq j \leq N$.
   Let R', Q' be the resulting indicator and quantity matrices.

2. Let $R = R', Q = Q'$.

3. For each knapsack $K_j, j = 1, \ldots, N$: if $\sum_i Q_{i,j} = c_j + 1$ (i.e., $c_j + 1$ compartments are used), let I_s be the set from which the quantity placed in K_j is minimal, that is, $Q_{s,j} = \min_{1 \leq i \leq M} Q_{i,j} > 0$, then $Q_{s,j} = 0$.

In other words, we turn the non-feasible placement into a feasible one, by using only the c_j fullest compartments in each knapsack. Note that this way we omit from the subset of packed items at most $\frac{v_j}{c_j+1}$ items.

Corollary 2.3.6 *Given an instance A of the CCMK, let U_{opt} be the utilization obtained by an optimal placement of the items, then we can find in $O(M+N)\log(M+N)$ steps a placement which achieves utilization $U = U_{opt} - \varepsilon$ for $\varepsilon = \sum_{j=1}^{N} \frac{v_j}{c_j+1}$.*

In particular, if the number of compartments in each knapsack is at least b, for some $b \geq 1$, that is, $c_j \geq b$ for all $1 \leq j \leq N$, then the above approximation algorithm achieves utilization $(1-\alpha)U_{opt}$ for $\alpha = \frac{1}{b+1}$.

2.4 Application to MOD Systems

In this section we show how our results for the CCMK and the CCFP apply to storage management in multimedia systems. Consider a system having N disks: the storage capacity of disk j is c_j, and its load capacity is v_j, $1 \leq j \leq N$. The database associated with the MOD system consists of M video program files $\{f_1, \ldots, f_M\}$, with the corresponding popularities $\{p_1, \ldots, p_M\}$. The popularity parameter of f_i reflects the portion of the total load generated due to access requests to f_i. Knowing these popularities and the total load capacity of the system, we can determine the average load generated by each of the files.

As mentioned in Section 1.1.2, the problem of assigning files to disks can be formulated as an instance of our packing problems, with the disks represented by knapsacks, and the files – by sets of items. The popularities of the files determine the set sizes, such that $\sum_{i=1}^{M} |I_i| = \sum_{j=1}^{N} v_j$. When our objective is to maximize utilization, we need to solve the CCMK; when the goal is to maximize fairness, we need to solve the CCFP. A solution for any of our two variants of the knapsack problems will induce a legal static assignment. In terms of the matrices R and Q:

- $R_{i,j} = 1$ iff a copy of the file i is stored on the disk j.
- $Q_{i,j} \in \{0, 1, \ldots, v_j\}$ is the total load that file i can create on the disk j.

Thus, our results in Section 2.2 yield efficient algorithms for finding a *perfect* assignment of files to the disks, in which the load capacity of the system is totally utilized, and the requests to each of the files can be satisfied.

2.4.1 Approximating Uniform Capacity Ratio

We now consider a slightly different model, in which the storage sub-system consists of N *disk arrays*, $D_1, \ldots, D_N$: D_j has a fixed load capacity, v_j, and, in addition, there is a limit, C, on the *total* number of storage units that can be allocated to the disk arrays. We would like to find an allocation of the storage units to the disk arrays. That is, for any $1 \leq j \leq N$ we need to determine c_j, the storage capacity of D_j, such that $\sum_{j=1}^{N} c_j = C$. This model reflects the situation in which several disk arrays are used for storing the files. The storage capacity of a disk array is the sum of the storage capacities of the individual disks. Thus, the storage capacity of a disk array (with a fixed load capacity) can vary, depending on the storage capacities of the disks composing the array. We show below that in such a system, the overall storage capacity can be distributed among the disk arrays, so as to achieve a nearly uniform capacity ratio. This enables us to find an almost perfect assignment of the files to the disks.

Let $V = \sum_{j=1}^{N} v_j$ be the total load capacity of the system. Indeed, it is easy to determine the storage capacity of each disk array, so as to obtain a 'uniform capacity ratio': in particular, we can choose $c_j = C \cdot v_j/V$. This yields a uniform capacity ratio with $r = V/C$. However, since we require that each disk array holds an integral number of files, we need to round the c_j's in a way that minimizes the violation of uniformity. Formally, we need to solve the following integer programming problem:

$$\text{Minimize} \quad \alpha = \frac{\max_{1\le j\le N} \frac{v_j}{c_j}}{\min_{1\le j\le N} \frac{v_j}{c_j}}, \quad \text{such that } \sum_{j=1}^{N} c_j = C, \quad c_j \text{ integer.} \tag{2.3}$$

This problem can be optimally solved in $O(N \log N + N \log C)$ steps, by using, e.g., the algorithm 'SOLVE-FAIR' (see Chapter 6 of [33]). The solution provides an allocation of the storage units to the disks such that for some $r > 0$,

$$\forall j,\ r \le \frac{v_j}{c_j} \le \alpha \cdot r, \tag{2.4}$$

and α is minimized. By Theorem 2.2.8 we have:

Corollary 2.4.1 *If storage units can be distributed among the disks, a $1/\alpha$-utilized assignment can be found in $O(M \log M + N \log N + N \log C)$ steps, where α is the value of the optimal solution of (2.3).*

To observe that α is small (that is, close to 1), note that in any optimal solution of (2.3), the capacity ratio of D_j satisfies

$$\lfloor \frac{V}{C} \rfloor \le \frac{v_j}{c_j} \le \lceil \frac{V}{C} \rceil, \quad \forall 1 \le j \le N\ .$$

This immediately yields a bound of 2 on α. More accurately, $\alpha \le \lceil \frac{V}{C} \rceil / \lfloor \frac{V}{C} \rfloor$.

2.4.2 Achieving Almost Optimal Utilization

In Section 2.3 we presented a dual approximation algorithm for the CCMK. In terms of MOD systems, it means that we can achieve the optimal utilization of a system by adding one storage unit to each disk. Indeed, for a given MOD system, such changes in configuration may be impossible, however, we can use the approximation algorithm derived from algorithm $\mathcal{A}_r$, and the result in Corollary 2.3.6. This implies that if each of the disks can store at least b files, for some $b \ge 1$, that is, $c_j \ge b$ for all $1 \le j \le N$, then the above approximation algorithm achieves utilization $(1-\alpha)U_{opt}$, with $\alpha = \frac{1}{b+1}$.

2.5 The Case where $|I| \ne V$

We now show the validity of our results for general instances of the CCMK and the CCFP problems, in which the total number of items is not necessarily equal to the sum of volumes of the knapsacks, namely, $|I| \neq V$. Note that the definitions of utilization and fairness, as given in Sections 2.1.1 and 2.1.2, do not depend on the ratio between $|I|$ and V. This ratio, however, influences the definition of *perfect placement.* When $|I| \neq V$, the maximal possible utilization is $min(|I|, V)$. When the objective is to maximize fairness, an optimal placement is $min(1, \frac{V}{|I|})$-fair[1]. For a general instance, a placement is *perfect* if it is optimal with respect to utilization as well as fairness. Formally,

Definition 2.5.1 *A* perfect placement *is a placement that is* $\min(1, V/|I|)$*-fair, in which one of the following is satisfies:* (*i*) *all the items are packed, or* (*ii*) *all the knapsacks are full.*

We now argue, that all the results presented in Section 2.2 hold for general inputs:

- if $V > |I|$, then add one *dummy set* of size $V - |I|$ to obtain an instance in which $V = |I|$. After placing the items in the knapsacks, omit the 'dummy' items.
- if $V < |I|$, a perfect placement is $\frac{V}{|I|}$-fair. For each set i, determine $|I_i^*| = \frac{V}{|I|}|I_i|$ (rounded to integers, such that $\sum_i |I_i^*| = V$). The resulting instance fulfills $|I^*| = V$.

All the algorithms presented in Section 2.2 can be applied to the above adjusted instances, to produce perfect placements for the original instances. Special tuning is needed in the proof of Theorem 2.2.1: when $V > |I|$, the added dummy set may not satisfy the condition $|I_i| \geq \frac{\varepsilon \cdot |I|}{M}$. To solve this potential problem, the algorithm has to consider the dummy set first: note that one (arbitrarily small) fraction of a set is allowed in each knapsack, hence, the dummy set can serve as the fraction placed in the first knapsack.

Our results in Section 2.3 also hold for general inputs, namely, the statements of Theorem 2.3.1 and Corollary 2.3.6 remain valid. In fact, the algorithm $\mathcal{A}_r$, used in the proof of Theorem 2.3.1, can be applied for any instance of the CCMK. It is sufficient to show the validity of and Lemmas 2.3.3 and 2.3.4. First note that only the proof of Lemma 2.3.4 assumes that $|I| = V$. We now show how the proof of this lemma can be modified to argue that all the knapsacks filled during the second phase are filled optimally also when $|I| \neq V$. We consider separately two cases:

(i) if $|I| > V$, then as in the case $|I| = V$, the sum of the sizes of the remaining sets always exceeds the total remaining volume.

(ii) if $V > |I|$, then all the knapsacks filled during the second phase are saturated until L is empty, or until all the knapsacks are filled. If L is empty (no requests are left), it means that we packed all the items of I, which is clearly optimal. If we run out of knapsacks, then, as in the case where $|I| \geq V$, it means that all the knapsacks filled in this phase are saturated. Thus, the only waste of volume is the inevitable $w = w_1 + w_2 + \ldots + w_n$ (from the first phase).

[1]Standard rounding techniques [33] can be applied here to determine the exact number of items to be packed from each type.

In both cases the utilization achieved by $\mathcal{A}_r$ is at least the maximal possible utilization for the original instance.

Chapter 3

Class-constrained Packing of Items with Arbitrary Sizes

In this chapter we study the CCMK and the CCBP problems on instances in which items may have arbitrary sizes and values. In the *class-constrained multiple knapsack (CCMK)* problem, our goal is to maximize the total value of packed items, whereas in the *class-constrained bin-packing (CCBP)*, we seek to minimize the number of (identical) bins, needed for packing all the items.

We give a *polynomial time approximation scheme* (PTAS) for CCMK and a dual PTAS for CCBP. We also show that the 0-1 class-constrained knapsack admits a *fully* polynomial time approximation scheme, even when the number of distinct colors of items depends on the input size. Finally, we introduce the *generalized class-constrained packing problem* (*GCCP*), where each item may have more than one color. We show that GCCP is APX-hard, already for the case of a single knapsack, where all items have the same size and the same value.

3.1 A Dual Approximation Scheme for CCBP

In this section we derive a dual-approximation scheme for the CCBP problem. Let $OPT_B(I)$ be the minimal number of bins needed for packing an instance I. For a given $\varepsilon > 0$, we present an algorithm, A_B^ε, which packs I into $OPT_B(I)$ bins; the number of different colors in any bin does not exceed c, and the sum of the sizes of the items in any bin does not exceed $1 + \varepsilon$. The running time of A_B^ε is polynomial in n and exponential in M, $\frac{1}{\varepsilon^2}$. We assume throughout this section that the parameter $M > 1$ is a fixed constant.

We note that even if $M > 1$ is some fixed constant, we cannot adopt the technique commonly used for packing (see, e.g., [29, 41, 54]), where we first consider the large items (of size $\geq \varepsilon$), and then add the small items. As we show below, in the presence of class constraints, one cannot extend even an *optimal* placement of the large items into an *almost* optimal placement of all items. Formally,

Fact 3.1.1 *For any $0 < \varepsilon < \frac{1}{2}$, there exists an instances of CCBP, for which an optimal packing of the large items (i.e., $u \in I$, such that $s(u) \geq \varepsilon$) cannot be extended to a $(1+\varepsilon)$-approximation to the optimal.*

Proof: We assume that $\frac{1}{\varepsilon}$ is an integer. Consider the following instance of the CCBP problem (demonstrated in Figure 3.1, for $\varepsilon = \frac{1}{4}$).

- $I = I_1 \cup I_2 \cup I_3$ (that is, $M = 3$). I_1 consists of two items of size $\frac{1}{2}$ and $\frac{2}{\varepsilon} - 4$ items of size $\frac{\varepsilon}{2}$; I_2 consists of one item of size ε, and I_3 consists of one item of size ε.
- The bins have capacity 1 and two compartments, that is, $c = 2$.

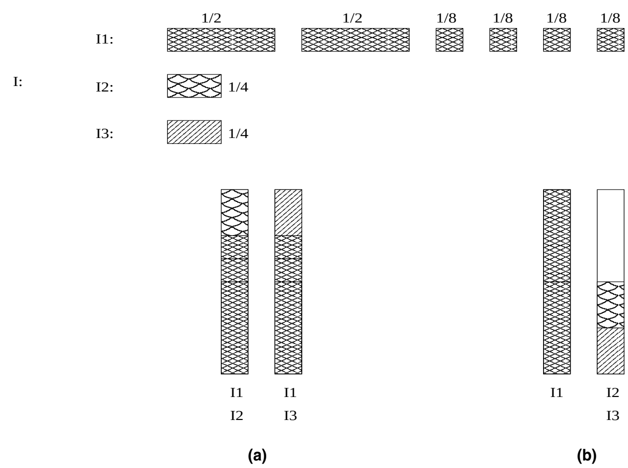

Figure 3.1: Packing large items first

An optimal packing of I uses two bins (see Figure 3.1(a)): in each bin we pack one item of size $\frac{1}{2}$ and $\frac{1}{\varepsilon} - 2$ small items of I_1, and the single item of I_2 or I_3. The size sum of the items in each bin is therefore $\frac{1}{2} + (\frac{1}{\varepsilon} - 2)\frac{\varepsilon}{2} + \varepsilon = 1$. A possible optimal packing of the large items of I (of size $\geq \varepsilon$) is presented in Figure 3.1(b). The first bin is filled with the two large items of I_1 and the second bin includes the two items of I_2 and I_3. Since there is no available compartment for I_1 in the second bin, and no additional place for all the small items of I_1 in the first bin (even if we allow an overflow of ε), we need to 'open' an additional bin for the small items, resulting in a $\frac{3}{2}$ ratio. ■

Our algorithm operates in two stages. In the first stage we eliminate small items, i.e., we transform I into an instance I' which consists of large items, and possibly one small item in each color. In the second stage we pack I'. We show that packing I' is much simpler than packing I, and that we only need to slightly increase the bin capacities, by factor $1+\varepsilon$. We show that a natural extension of known packing algorithms to the class-constrained problem, yields a complicated scheme. The reason is that without elimination, we need to handle the small items of each color *separately.* The elimination technique presented below involves some interaction between small items of different colors.

Various techniques can be used to approximate the optimal solution for I'. We adopt the technique presented by Epstein and Sgall [15]. Alternatively, we could use *interval partition* ([41, 54]).

Note that when there are no class constraints, our elimination technique can be used to convert any instance I into one which contains a *single* small item.

3.1.1 Eliminating Small Items

We now describe our *elimination* technique, and show that the potential 'damage' when solving the problem on the resulting instance, is small. For a given parameter, $\delta > 0$, denote by *small* the subset of items of sizes $s(u) < \delta$. Other items are considered *large.* Our scheme applies the following transformation to a given instance I.

1. Partition I into M sets by the colors of the items.
2. For each color $1 \leq i \leq M$, partition the set of small items of color i into groups: the total size of the items in each group is in the interval $[\delta, 2\delta)$; one group may have total size $< \delta$. This can be done, e.g., by grouping the small items greedily: we start to form a new group, when the total size of the previous one exceeds δ.

The resulting instance, I', consists of *non-grouped-items*, which are the original large items of I, large *grouped-items* of sizes in $[\delta, 2\delta)$, and at most M *small grouped-items* (one in each color).

From the way I' is constructed, we have

Fact 3.1.2 *For each color $1 \leq i \leq M$, the total size of the items of color i in I' is equal to the total size of the items of color i in I.*

Given a packing of I', we can replace each grouped item by the set of small items from which it was formed. In this process, neither the total size of the items nor the set of colors contained in each bin is changed. Hence,

Fact 3.1.3 *Any legal packing of I' into m bins induces a legal packing of I into m bins.*

Now, we need to bound the potential damage from solving the problem on I' (rather than the original instance I). We show that any legal packing of I induces a packing of I' into a bit larger bins. Note that in the transformation from I to I' we cannot use a simple replacement, since small items who belong to the same group in I' may be packed in different bins. Let $S(A)$ denote the total size of a set A of items.

Lemma 3.1.4 *Given a packing of I in m bins of size 1, we can pack I' into m bins of size $1+2\delta$.*

Proof: First, replace all the large items of I with their corresponding non-grouped-item in I'. The remaining items are small, and need to be replaced by grouped items. For these items (only) we show that such a replacement exists. In fact, we prove a stronger claim. For a bin B, let S_B denote the set of small items packed in B. For a set $C_1 \subseteq \{1, ..., M\}$ of colors, let $S_B^{C_1}$ denote the subset of S_B of items with colors in C_1. Denote by $C(I, B)$ the set of colors of small items of I placed in B, and by $C(I', B)$ the set of colors of the grouped items of I' placed in B. We show that the small items of I can be replaced by all the grouped items of I', such that

(i) for any bin B and a set of colors C_1, the items in $S_B^{C_1}$ are replaced by grouped items in I' whose colors are in C_1, and whose total size is at most $S(S_B^{C_1}) + 2\delta$;

(ii) for any bin B, $C(I', B) \subseteq C(I, B)$.

The proof is by induction on m, the number of bins used for packing I.
Base: $m = 1$, that is, all the items of I are placed in one bin, B. By Fact 3.1.2, for each color $1 \leq i \leq M$, the total size of the items of color i in I' is equal to the total size of the items of color i in I. Hence, we can replace all the items of I by all the items of I'.
Induction Step: Assume that the claim holds for any instance with $m' < m$ bins. Given an instance with m bins, let B_1, B_2 be two arbitrary bins, which contain small items from I of total size $S(S_{B_1})+S(S_{B_2})$. Consider an instance with $m-1$ bins, derived from our instance by replacing B_1 and B_2 by one new bin B_{new}, which contains all the small items of B_1 and B_2. By the induction hypothesis, we can replace the small items of I by the grouped items of I', such that (i) and (ii) hold. The solution for the new instance induces a solution for the original instance: bins other than B_1 and B_2 are filled as in the new instance. The grouped items that are placed in B_{new} split between B_1 and B_2 as follows.

1. Let $C_1 = C(I, B_1) \setminus C(I, B_2)$; place all the C_1-colored grouped-items from B_{new} in B_1.
2. Let $C_2 = C(I, B_2) \setminus C(I, B_1)$; place all the C_2-colored grouped-items from B_{new} in B_2.
3. Let $C_3 = C(I, B_1) \cap C(I, B_2)$; place arbitrary C_3-colored grouped-items from B_{new} in B_1, until it first reaches the capacity $S(S_{B_1})$; place the remaining C_3-colored grouped-items from B_{new} in B_2.

By the induction hypothesis, B_{new} contains C_1-colored grouped-items of size at most $S(S_{B_1}^{C_1}) + 2\delta$, thus, the overflow in B_1 after step 1 is at most 2δ. Similarly, the overflow in B_2 after step 2 is at most 2δ. In addition, the total overflow in B_1 after step 3 is at most 2δ, since the size of each grouped-item is at most 2δ and we stop filling B_1 as soon as its contents exceed $S(S_{B_1})$. Finally, the overflow in B_2 after step 3 is at most 2δ, since by the induction hypothesis the total size of the items in B_{new} is at most $S(S_{B_1}) + S(S_{B_2}) + 2\delta$. ■

In particular, for an optimal packing of I, we have:

Corollary 3.1.5 *Let $OPT_B(I, v)$ be the minimal number of bins of size v needed for packing an instance I, then $OPT_B(I', 1 + 2\delta) \leq OPT_B(I, 1)$.*

Note that our proof induces a replacement of the small items of I by grouped items, such that the *total* overflow (when summing over all bins) is at most 2δ. However, we cannot predict how this overflow will split among the bins.

3.1.2 Packing the Grouped Instance

The PTAS of Epstein and Sgall

Epstein and Sgall [15] presented PTASs for multiprocessor scheduling problems on m uniform machines. When each job is represented by an item and each processor is represented by a bin, their general approximation schema can be modified to yield a dual PTAS for bin packing. Specifically, for a given parameter $\delta > 0$, the dual PTAS packs any instance I into the optimal number of bins, with the size of each bin increased at most by factor $1 + \delta$ (This can be done using binary search on m, the minimal number of bins, as in [29]).

The PTAS in [15] is based on partitioning the set of jobs, I, into $O(\frac{1}{\delta^2})$ subsets. The jobs in each subset have the same length (the original lengths are rounded). The short jobs are replaced by a few jobs of short, but non-negligible length. This partition can be represented as a *configuration vector* of length $O(\frac{1}{\delta^2})$. Any vector whose entries are at most the entries of this vector is a configuration vector of some set $S \subseteq I$ of jobs. The algorithm constructs a graph whose vertices are a source, a target and one vertex for each possible configuration vector. There is an edge connecting the two configurations n' and n'' if a schedule of n' on i machines can be extended to a schedule of n'' on $i+1$ machines. The weight of an edge (n', n'') reflects the price of moving from the configuration n' to the configuration n'', that is, the time needed for the $(i+1)$th machine to process the jobs in $n'' - n'$.

Having the weights defined in such a way, the problem of finding a good schedule is reduced to the problem of finding a good path in the graph. We seek a path of length m from the source (whose configuration represents an empty set of jobs), to the target (whose configuration represents the set of all jobs). The evaluation of a path depends on our objectives. For example, when minimizing the makespan, a good path is a one that minimizes the maximal weight of some participating edge; when minimizing the sum of the completion times, a good path is a one that minimizes the total weight of the participating edges. The time complexity of the algorithm depends on the size of the graph, which is dominated by the total number of configurations. For a given δ, the graph has size $O(n^f)$ where $f = \frac{81}{\delta^2}$.

When this scheme is used for bin-packing, each configuration represents a set of items. An edge connecting the two configurations n' and n'' implies that a placement of n' in i bins can be extended to placement of n'' in $i+1$ bins. The weight of the edge is the total size of items in $n'' - n'$ - that are supposed to be placed in the $(i+1)$th bin. For a given value of m, if there is a packing of I into m bins of size 1, then there is a path of length m from the source to the target with all edges having weight at most $1+\delta$ (recall that we use binary search to find the minimal value of m).

The Algorithm A_{ES}:

A natural extension of Epstein-Sgall's algorithm for the CCBP problem is to refine the configurations to describe the number of items from each size set and *from each color*. Such extension results in a graph of size $O(n^f)$ where $f = M(2M+1)^4/\delta^2$. The value of f increases from $81/\delta^2$ to $M(2M+1)^4/\delta^2$ since the small items of each color are handled separately. In addition, the total size of small items of each color is rounded to the nearest multiple of $\frac{\delta}{9}$. Thus, if rounded items of c colors are packed into the same bin, we may get in this bin an overflow of $c\frac{\delta}{9}$. Given that there is only one small item in each color, as in I', we can use the *actual* sizes of the small items. This decreases significantly the possible number of configurations and prevents any potential overflow due to rounding.

Hence, given an instance I', consisting of large items and at most M small items, each of different color, we derive a simplified version of the PTAS in [15]. In this version, denoted by A_{ES}:

1. Each set of items $S \subseteq I'$ is represented by a unique configuration of length $O(\frac{M}{\delta^2})$, which indicates how many items in S belong to each color and to each size class. Note that there is no need to round the total size of the small items: we only need to indicate which small items are included in the set (a binary vector of length M). In terms of [15], this means that we do not need to accumulate the size of small items in each configuration, and no precision is lost because of small items.

2. To guarantee that each bin contains items of at most c colors, we connect in the configuration graph only vertices representing configurations which differ by at most c colors. In other words, there is an edge connecting the two configurations n' and n'' only if $n'' - n'$ is positive in entries that belong to at most c colors.

We summarize in the next Lemma:

Lemma 3.1.6 *Let m be the minimal number of bins of size v needed for packing I', then, for a given $\delta > 0$, A_{ES} finds a packing of I' into m bins of size $v(1+\delta)$. The running time of A_{ES} is $O(n^f)$ where $f = \frac{M}{\delta^2}$.*

3.1.3 The Algorithm A_B^ε

Let $\delta = \frac{\varepsilon}{4}$; $v = 1 + 2\delta$. The algorithm A_B^ε proceeds as follows.

1. Construct I' from I, using the algorithm in Section 3.1.1.
2. Use A_{ES} for packing I' into bins of size $v(1+\delta)$. Let m be the number of bins used.
3. Ungroup the grouped items to obtain a packing of I in m bins of size $v(1+\delta)$.

Theorem 3.1.7 *The algorithm A_B^ε uses at most $OPT_B(I)$ bins. The sum of the sizes of the items in any bin does not exceed $1+\varepsilon$. The running time of A_B^ε is $O(n^f)$ where $f = \frac{16M}{\varepsilon^2}$.*

Proof: We first show that the sum of the sizes of the items in any bin does not exceed $1+\varepsilon$: Combining Fact 3.1.3 and Lemma 3.1.6, we conclude that each of the m bins is filled up to capacity $v(1+\delta)$. Clearly, for any $\varepsilon \leq 2$, $\delta \leq \frac{1}{2}$ (otherwise, we can choose a smaller value for δ), thus, $v(1+\delta) \equiv (1+2\delta)(1+\delta) \leq 1 + 4\delta = 1 + \varepsilon$.

Next, we show that the number of bins used by A_B^ε is at most $OPT_B(I,1)$: By Lemma 3.1.6, I' is packed by A_{ES} in m bins of size $v(1+\delta)$, such that $m \leq OPT_B(I', v)$. Then, we use these m bins to pack I. From Corollary 3.1.5, $OPT_B(I',v) \leq OPT_B(I,1)$. Therefore $m \leq OPT_B(I,1)$. Finally, setting $\delta = \frac{\varepsilon}{4}$, the running time of A_B^ε follows directly from the running time of A_{ES}. ■

We note that our assumption that M is fixed is needed only in the second step, when packing I'. Hence, this assumption can be relaxed when our technique, for eliminating small items, is used for other purposes.

3.2 Approximation Schemes for CCMK

3.2.1 The PTAS of Chekuri and Khanna

Chekuri and Khanna developed in [10] a PTAS for MK. Let $P(U)$ denote the value of a set $U \subseteq I$, and let $OPT(I)$ be the value of an optimal solution when packing I. The PTAS in [10] is based on two steps: (i) *Guessing items:* identify a set of items $U \subseteq I$ such that $P(U) \geq (1-\varepsilon)OPT(I)$

and U has a feasible packing. *(ii) Packing items:* given such U, find a feasible packing of $U' \subseteq U$, such that $P(U') \geq (1-\varepsilon)P(U)$.

For sake of completeness, we summarize the implementation of the guessing items step as given in [10]. For a guessed value P satisfying $(1-\varepsilon)OPT(I) \leq P \leq OPT(I)$, discard all the items $u \in I$ with $p(u) < \varepsilon P/n$. Then, divide all the profits by $\varepsilon P/n$, and round each down to the nearest power of $(1+\varepsilon)$. The resulting instance, I', has $h \leq \lceil \log_{1+\varepsilon} \frac{n}{\varepsilon} \rceil = O(\frac{\ln n}{\varepsilon})$ distinct profits. Thus, we assume that the items of I are partitioned into h sets $I^1, \ldots, I^h$ with items in each set having the same profit. Let $\mathcal{D}$ be a subset of items which gives a solution with profit P, and let $\mathcal{D}^j$ be the subset of items in $\mathcal{D}$ in the jth profit class.

We proceed to guess an h-vector, $\vec{k}$, of integers, $1 \leq k_j \leq h/\varepsilon^2$; k_j reflects the contribution of $\mathcal{D}^j$ to the overall profit in some optimal packing, in multiples of $\frac{\varepsilon^2 P}{h}$, i.e.,

$$k_j \frac{\varepsilon^2 P}{h} \leq P(\mathcal{D}^j) \leq (k_j+1)\frac{\varepsilon^2 P}{h}.$$

We seek a vector $\vec{k} = (k_1, \ldots, k_h)$ satisfying $\sum_{j=1}^{h} k_j \leq \frac{h}{\varepsilon^2}$. As shown in [10], the number of such vectors is $O(n^{O(1/\varepsilon^3)})$, thus, $\vec{k}$ can be guessed in polynomial time. Given the contribution of each profit set I^j in an optimal packing, we select for the set U the items of I^j with smallest *sizes* whose total profit accumulates to the required contribution. These small items can clearly replace any set of items with the same contribution from $\mathcal{D}^j$. Thus, U is feasible.

3.2.2 Our PTAS for CCMK - An Overview

In the following we adopt the guessing approach in [10], for obtaining a PTAS for CCMK. For some $t \geq 1$, assume that there are t types of bins, such that bins of type j have the same capacity, v_j, and the same number of compartments, c_j. Let $c = \max_j c_j$ be the maximal number of compartments in any type of bins. We denote by B_j the set of bins of type j; let $m_j = |B_j|$ be the number of bins in B_j. Our schema proceeds in three steps:

1. Guess a subset $U \subseteq I$ of items that will be packed in the bins, such that $P(U) \geq (1-\varepsilon)OPT(I)$, and U has a feasible class-constrained packing in the knapsacks.

2. Guess a partition of the bins to at most
$$r = \sum_{\ell=1}^{c} \binom{M}{\ell} = O(M^c) \tag{3.1}$$
color sets, i.e., associate with each subset of bins the set of colors that can be packed in these bins. We call a subset of bins of the same type and color set a *block*; thus, the resulting number of blocks is $T \leq t \cdot r$.

3. Find a feasible packing of $U' \subseteq U$ in the bin blocks, such that $P(U') \geq (1-\varepsilon)P(U)$.

As in [10], the overall scheme is more involved since we iterate between the guessing and packing steps. This is due to the fact that $OPT(I)$ is not known in advance. However, the maximal number of guessing-packing iterations is bounded by the maximal number of guesses for a value P, such that

$$(1-\varepsilon)OPT(I) \leq P \leq OPT(I). \tag{3.2}$$

To bound this number, we show a γ-approximation for the problem for some constant $0 < \gamma \leq 1$.

Claim 3.2.1 *Let $c_m = \min_j c_j$ be the minimal number of compartments in any type of bins, then the CCMK problem has a polynomial time $\frac{c_m}{2M}$-approximation algorithm.*

Proof: The following algorithm, $\mathcal{A}$, is a $\frac{c_m}{2M}$-approximation algorithm for CCMK:

1. Ignore the colors and find a $\frac{1}{2}$-approximate solution for MK for the input I (such an algorithm is given is [51]).
2. For each knapsack, K_j, find the c_j most valuable colors in the resulting packing and remove all the items which are not in these colors.

Let z_1 be the optimal profit for the non class-constrained problem, and let z_2 be the profit obtained in step 1 of $\mathcal{A}$. Let z^* be the optimal profit for the class constrained problem, and let z be the profit obtained by $\mathcal{A}$.

Note that since we pack in each knapsack items of at most c_j colors, $\mathcal{A}$ produces a valid packing. In addition,

$$z \geq \frac{c_m}{M} z_2 \geq \frac{c_m}{2M} z_1 \geq \frac{c_m}{2M} z^*.$$

Thus, $\mathcal{A}$ is a $\frac{c_m}{2M}$-approximation algorithm for the CCMK problem. ∎

Having a $\frac{c_m}{2M}$-approximation for $OPT(I)$, we can find a P satisfying (3.2) using binary search over the resulting range.

Corollary 3.2.2 *The number of guessing-packing iterations for finding P which satisfies (3.2) is at most $\lceil \log(\frac{2M}{\varepsilon}) \rceil$.*

In Sections 3.2.3, 3.2.5, and 3.2.6 we describe in detail the implementation of each of the steps in our PTAS. The implementation of the second and the third steps varies, depending on t, the number of bin types. As a result, the time complexity of our PTAS depends also on that parameter.

3.2.3 Guessing Items

We now show that the first step, of guessing items, can be generalized to the class-constrained version. This generalization is straightforward: after reducing (to h) the distinct number of profits in I (this is detailed in Section 3.2.1), we further partition each profit set I^j into M color sets $I_1^j, \ldots, I_M^j$, with the items in each set having the same profit and the same color. For a subset of item, $\mathcal{D}$, whose packing gives the profit P, denote by $\mathcal{D}_i^j$ the subset of items in $\mathcal{D}$ in the jth profit class and in color i. We now guess an $h \times M$-matrix, $K = \{k_{i,j}\}$, of integers, $1 \leq k_{i,j} \leq hM/\varepsilon^2$; $k_{i,j}$ reflects the contribution of $\mathcal{D}_i^j$ to the overall profit in some optimal packing, in multiples of $\frac{\varepsilon^2 P}{Mh}$, i.e.,

$$k_{i,j}\frac{\varepsilon^2 P}{Mh} \leq P(\mathcal{D}_i^j) \leq (k_{i,j}+1)\frac{\varepsilon^2 P}{Mh}. \tag{3.3}$$

We seek a matrix $K = \{k_{i,j}\}$ satisfying $\sum_{j=1}^{h}\sum_{i=1}^{M} k_{i,j} \leq \frac{Mh}{\varepsilon^2}$. Given the contribution of each profit-color set I_i^j in an optimal packing, we select for U the items of I_i^j with smallest *sizes* whose total profit accumulates to the required contribution.

In order to show that the above algorithm finds in polynomial time a set $U \subseteq I$ as required in the guessing step, we need to show the following claims:

Claim 3.2.3 *Solving the problem for the rounded profit instance I' rather than I, can decrease the optimal profit by a factor of at most* $(1-\varepsilon)$.

Proof: The proof follows from the fact that for any subset U of I, U is transformed into a set $U' \subseteq I'$ whose profit satisfies

$$(1-\varepsilon)P(U) \leq \frac{P(U)}{1+\varepsilon} \leq \frac{P(U')\varepsilon P}{n} \leq P(U).$$

In particular, this holds for an optimal subset of packed items. ∎

Claim 3.2.4 *There exists an $h \times M$-matrix, K, of integers, $1 \leq k_{i,j} \leq hM/\varepsilon^2$, such that the set U induced by K has a feasible class-constrained packing and $P(U) \geq (1-\varepsilon^2)P$.*

Proof: Let $\mathcal{D}$ be the subset of items as defined above. Recall that U is the subset of items induced by the matrix K; then, from the right inequality in (3.3) we get that

$$P = P(\mathcal{D}) = \sum_{j=1}^{h}\sum_{i=1}^{M} P(\mathcal{D}_i^j) \leq \sum_{j=1}^{h}\sum_{i=1}^{M} k_{i,j}\frac{\varepsilon^2 P}{Mh} + \varepsilon^2 P = P(U) + \varepsilon^2 P.$$

This yields the claim. ∎

Claim 3.2.5 *The number of possible guesses for the matrix K is* $n^{O(\frac{M}{\varepsilon}\ln(1/\varepsilon))}$.

Proof: Note that the total number matrices K is the number of hM-tuples whose coordinates sum to hM/ε^2, given by $\binom{hM+\frac{hM}{\varepsilon^2}-1}{hM-1}$. Now, for any $n \geq l \geq 0$, $\binom{n}{l} \leq (\frac{\varepsilon n}{l})^l$. Hence, we get that

$$\binom{hM + \frac{hM}{\varepsilon^2} - 1}{hM - 1} \leq (\frac{3e}{\varepsilon^2})^{hM} = e^{h \cdot M \cdot \ln(\frac{3e}{\varepsilon^2})} = n^{O(\frac{M}{\varepsilon}\ln(1/\varepsilon))}.$$

This completes the proof. ∎

Combining Claims 3.2.3, 3.2.4, and 3.2.5, we have

Corollary 3.2.6 *For a given P, a set U such that $P(U) \geq (1-\varepsilon)P$, and U has a feasible class-constrained packing in the knapsacks, can be found in $n^{O(\frac{M}{\varepsilon}\ln(1/\varepsilon))}$ guesses.*

3.2.4 Reducing the Number of Distinct Item Sizes

Using the restricted number of distinct profit values we can also reduce the number of distinct sizes in the set U to $O(\ln n/\varepsilon^2)$. This stage is similar to the corresponding step in [10], and is based on the *shifting* technique introduced in the PTAS for bin-packing in [34]. Let A be a set of a items with identical profits, identical colors and arbitrary sizes. We order the items in A in non-decreasing order by sizes and divide them into $g = (1 + 1/\varepsilon)$ groups $A_1, ..., A_g$ with $A_1, ..., A_{g-1}$ containing $\lfloor a/g \rfloor$ items each and A_g containing the remaining items. We discard the items in A_{g-1} and for each $i < g - 1$ we increase the size of every item in A_i to the size of the smallest item in A_{i+1}. Since A is ordered by size, no item in A_i is larger than the smallest item in A_{i+1} for $1 \leq i < g$. Thus, if A has a feasible packing then the modified instance also has one. We discard at most an ε fraction from the profit (of the items in A_{g-1}) and the modified instance has at most $2/\varepsilon$ distinct sizes ($g - 2$ sizes of the items in $A_1, ..., A_{g-2}$ and at most g distinct sizes of the items in A_g). Applying this to each profit-color class in U, we can now assume that U consists of $s \leq O(M \ln n/\varepsilon^2)$ classes, i.e., $U_1 \cup \cdots \cup U_s$, and the items in U_k are of the same size, the same profit,and the same color.

3.2.5 Instances with a Fixed Number of Bin Types

Assume that $t \geq 1$ is a *fixed* constant. In the second step we partition the bins into blocks. We need to determine $m_{j,l}$, the number of bins of type j associated with the lth color set. Thus, the number of guesses is $O(n^T)$ (since we need to determine the size of T blocks, and each block can consist of at most n bins).

The third step of our scheme is implemented as follows. Let $n_k = |U_k|$ be the number of items in the class U_k, $1 \leq k \leq s$. We first partition U_k to $1 \leq T' \leq T$ subsets, where T' is the number of distinct blocks in which the items of U_k are placed. Clearly, the items of U_k can only be placed in blocks that include the color of the items in U_k in their color set. Denote by $n_{k,b}$ the number of items in the class U_k that are placed in the block b. For $i = 1, \ldots, 2T/\varepsilon$, let $\sigma_i^k = \lceil \frac{i\varepsilon n_k}{2T} \rceil$.

Claim 3.2.7 *By considering only partitions of U_k in which $\forall b, \exists i,\ n_{k,b} = \sigma_i^k$, we can decrease the optimal profit gained by items of U_k at most by factor $(1-\varepsilon)$.*

Proof: First note that if $\frac{\varepsilon n_k}{2T} \leq 1$ then clearly all possible integer values for $n_{k,b}$ are considered. If $\frac{\varepsilon n_k}{2T} > 1$ then $\forall 1 \leq i \leq 2T/\varepsilon$,

$$\begin{aligned}\sigma_i^k - \sigma_{i-1}^k &= \lceil \frac{i\varepsilon n_k}{2T} \rceil - \lceil \frac{(i-1)\varepsilon n_k}{2T} \rceil \leq \frac{i\varepsilon n_k}{2T} + 1 - \frac{(i-1)\varepsilon n_k}{2T} \\ &= 1 + \frac{\varepsilon n_k}{2T} \leq \frac{2\varepsilon n_k}{2T} = \frac{\varepsilon n_k}{T}. \end{aligned} \tag{3.4}$$

Consider any partition of U_k into the blocks. For any block b, if there is no i such that $n_{k,b} = \sigma_i^k$ then we can remove the less-profitable items until we reach some σ_i^k. By (3.4) we remove at most $\frac{\varepsilon n_k}{T}$ items. Thus, we remove a total of at most εn_k items - with an overall loss of at most a factor of ε from the total profit of U_k. ■

Therefore, we can assume that $n_{k,b}$ is one of $2T/\varepsilon$ values. We can now describe the partition of U_k as a list of pairs (b, i), where $1 \leq b \leq T$ and $1 \leq i \leq 2T/\varepsilon$. The number of such subsets of pairs for U_k is at most $2^{O(T^2/\varepsilon)}$, and as we need to consider all the classes, $U_1, \ldots, U_s$, the overall number of guesses is $2^{s \cdot T^2/\varepsilon} = 2^{O((M \ln n/\varepsilon^2)\cdot(T^2/\varepsilon))} = n^{O(MT^2/\varepsilon^3)}$.

By the *guessing items* step of our scheme, the set U has a feasible packing. In particular, there exists a partition of the U_k's among the T blocks which corresponds to a feasible packing. We show that we can pack $U' \subseteq U$ such that $P(U') \geq (1-\varepsilon)P(U)$. In fact, for each block, b, we have guessed a set of items that has a feasible packing in b. Since the bins in each block are identical, we can use, for each block separately, any known PTAS for bin packing (see, e.g., [41] whose time complexity is $O(n/\varepsilon^2)$) and take for each block the $m_{j,l}$ most profitable bins. Since such a PTAS packs *all* the guessed items in at most $(1+\varepsilon)m_{j,l}$ bins, we discard at most ε of the profit. If $m_{j,l} \leq \lceil \frac{1}{\varepsilon} \rceil$ then we can use a PTAS for a constant number of bins (given in [18]). Thus, for a given partition of the U_k's to the blocks, we can find a packing with profit at least $(1-\varepsilon)P(U)$ in $O(Tn/\varepsilon^2)$ steps.

We summarize in the next result

Theorem 3.2.8 *The CCMK with a fixed number of distinct colors, $M \geq 1$, and a fixed number of bin types, $t \geq 1$, admits a PTAS, whose running time is $O(\frac{T}{\varepsilon^2} n^{O(\frac{M}{\varepsilon}\ln(1/\varepsilon) + \frac{MT^2}{\varepsilon^3})})$.*

Proof: The proof follows from a combination of (i) number of guesses of the value of P; (ii) the number of guesses of the matrix K; (iii) the number of guesses of the partition of U to bin blocks, and (iv) the complexity of the PTAS for bin packing, for each guess.

Thus, we get that the overall running time is at most

$$\lceil \log(\frac{2M}{\varepsilon}) \rceil \cdot O(n^{O(\frac{M}{\varepsilon}\ln(1/\varepsilon))} \cdot n^{O(MT^2/\varepsilon^3)} \cdot \frac{Tn}{\varepsilon^2})$$

which yields the statement of the theorem. ■

3.2.6 Instances with Arbitrary Number of Bin Types

Suppose that the number of bin types is $t = O(\log n)$. Recall that $m_j = |B_j|$ is the number of bins of the jth type. We need to determine for all $1 \le j \le t,\ 1 \le l \le r$ the number $m_{j,l}$ of bins of type j associated with the lth color set. A naive implementation of this step of our scheme results in $O(n^{O(lgn)})$ guesses. Thus, we limit the possible partitions of the bins to blocks as follows. For $i = 0, \ldots, 2r/\varepsilon$, let $\beta_i^j = \lceil \frac{i\varepsilon m_j}{2r} \rceil$.

Claim 3.2.9 *By considering only partitions of m_j in which $\forall l, \exists i,\ m_{j,l} = \beta_i^j$, we can decrease the optimal profit gained by items packed in the bins of B_j by a factor of at most $(1 - \varepsilon)$.*

Proof: First note that if $\frac{\varepsilon m_j}{2r} \le 1$ then clearly all possible integer values for $m_{j,l}$ are considered. If $\frac{\varepsilon m_j}{2r} > 1$ then, as in the proof of Claim 3.2.7, $\forall 1 \le i \le 2r/\varepsilon$,

$$\beta_i^j - \beta_{i-1}^j \le \frac{\varepsilon m_j}{r}. \tag{3.5}$$

Consider any partition of m_j among the r color sets. For any color set l, if there is no i such that $m_{j,l} = \beta_i^j$ then we can add bins of type j until we reach some β_k^j. From (3.5) we add at most $\frac{\varepsilon m_j}{r}$ bins. Thus, summing over all color sets, we add a total of at most εm_j bins. Once the placement is completed we can pick for each j the m_j most profitable bins with an overall loss of at most a factor of ε from the total profit. ∎

Hence, we can consider only partitions in which the number of bins allocated to the lth color set is a multiple of $\varepsilon m_j/2r$, and the total number of bins of type j is at most $(1+\varepsilon)m_j$. Since $m_{j,l}$ is one of $2r/\varepsilon$ values, we can describe the partition of m_j as a list of pairs (l, i), where $1 \le l \le r$ and $1 \le i \le \frac{2r}{\varepsilon}$. We get that the number of guesses is $2^{O(r^2/\varepsilon)}$. The overall number of guesses, when taking all bin types, is $2^{O(\frac{r^2}{\varepsilon}\log n)} = n^{O(r^2/\varepsilon)}$.

In the third step we adapt the packing steps in [10], with the bins partitioned to blocks, as defined above.

Indeed, in the general case, t can be as large as N. The packing steps in [10] assume that there are $O(\log(n/\varepsilon))$ subsets of bins such that the capacities of bins from the same subset are in the range $[v, v(1+\varepsilon))$ for some v. In the following we show that the set of bins can be partitioned in a way that fits this assumption.

Lemma 3.2.10 *Any set of bins with t bin types can be transformed into a set with $O(\log(n/\varepsilon))$ subsets of bin types, such that the bins in each subset have the same number of compartments and their capacities are in the range $[v, v(1+\varepsilon))$ for some v, and for any $U \subseteq I$ that has a feasible packing in the original set, there exists $U'' \subseteq U$ that has a feasible packing in the transformed set, and $P(U'') \ge (1 - \varepsilon)P(U)$.*

Proof: Let v_{max} denote the maximal capacity of a bin. Consider the set of bins whose capacities are in the range $[\varepsilon v_{max}/n^{\frac{3M}{\varepsilon}}, v_{max}]$. We partition this set of bins into subsets, $A^0, A^1, \ldots$, such that A^j consists of the bins whose capacities are in the range

$$[\frac{\varepsilon v_{max}}{n^{\frac{3M}{\varepsilon}}}(1+\varepsilon)^j,\ \frac{\varepsilon v_{max}}{n^{\frac{3M}{\varepsilon}}}(1+\varepsilon)^{j+1}).$$

Distinguishing between bins by number of compartments, we partition each A^j into at most c subsets, $A^j_1, \ldots, A^j_c$. We now have at most $c\log(n^{\frac{3M}{\varepsilon}}/\varepsilon)$ different subsets if bins.

Now, for any $U \subseteq I$ that is packed in the original set of bins, we can add $w \le M$ bins, one for each color $1 \le i \le M$, and pack, for each color, i, all the items in color i, previously packed in bins of sizes smaller than $\varepsilon v_{max}/n^{\frac{3M}{\varepsilon}}$, in a *single* bin with a single compartment. The total capacity of the w added bins is at most $\varepsilon v_{max}/n^{(\frac{3M}{\varepsilon}-1)} \le \varepsilon v_{max}/n^{\frac{2M}{\varepsilon}}$. Therefore, we can transform the given instance to an instance, where all the bins, except for $w \le M$, are of sizes in the range $[\varepsilon v_{max}/n^{\frac{3M}{\varepsilon}}, v_{max}]$, arranged in the sets $\{A^j_i\}$ defined above.

Since the set U is unknown in advance, we have to guess $w \le M$ and the partition of the extra capacity of the small bins (which is at most $\varepsilon v_{max}/n^{\frac{2M}{\varepsilon}}$) among the w new bins. The capacity of each of the new bins will be a multiple $1 \le k \le n$ of the minimal size $\varepsilon v_{max}/n^{\frac{3M}{\varepsilon}}$. Thus, the overall number of guesses is $n^w \le n^{O(M)}$.

Finally, after the items of U are packed in the transformed set of bins, we need to eliminate the extra w bins. Clearly, we can remove any set of w bins with larger capacities and at least one compartment. Consider the set of bins of $T'' = \log(n^{\frac{2M}{\varepsilon}}/\varepsilon)$ types with largest capacities. The number of bins in the set is at least T'', and

$$\begin{aligned} T'' + w &\le \frac{2M}{\varepsilon}\log n + \log\frac{1}{\varepsilon} + M \\ &\le (1+\varepsilon)\frac{2M}{\varepsilon}\log n + (1+\varepsilon)\log\frac{1}{\varepsilon} \\ &\le (1+\varepsilon)T''. \end{aligned}$$

That is, we increase the number of bins in this set at most by factor $(1+\varepsilon)$. Since the smallest bin size in this set is $\varepsilon v_{max}/n^{\frac{2M}{\varepsilon}}$, we can select the T'' most profitable bins. The set of items, U'', packed in the bins that were not omitted from the instance has profit at least $(1-\varepsilon)P(U)$. ■

We can now proceed to apply the above PTAS to the resulting set of bins. Thus,

Theorem 3.2.11 *For any instance I of the CCMK, in which $M \ge 1$ is fixed, there is a PTAS which obtains a total profit of $(1-\varepsilon)OPT(I)$ in time $O(n^{O(\frac{M}{\varepsilon}\ln(1/\varepsilon)+r^2/\varepsilon+\ln(1/\varepsilon)/\varepsilon^8)})$, where r is defined in (3.1).*

Proof: We get the overall complexity of our PTAS by combining the following steps: (i) guessing a value of P; (ii) guessing the matrix K; (iii) guessing the partition of U to bin blocks;

(iv) guessing the partition of the extra capacity among the new bins, and (v) packing the items [10]. Thus, we have that the overall running time is at most

$$\lceil \log(\frac{2M}{\varepsilon}) \rceil \cdot O(n^{O(\frac{M}{\varepsilon}\ln(1/\varepsilon)+r^2/\varepsilon+M+\ln(1/\varepsilon)/\varepsilon^8)})$$

which yields the desired result. ■

3.3 The Class-Constrained 0-1 Knapsack Problem

In this section we consider the *class-constrained 0-1 knapsack problem (CCKP)*, in which we need to place a subset U of I in a single knapsack of size $v \geq \max\{s(u)|u \in I\}$. The objective is to pack items of maximal value from I in the knapsack, such that the sum of the sizes of the packed items does not exceed v, and the number of different colors in the knapsack does not exceed c.

Throughout this section we assume that the numbers c and M are given as part of the input (Otherwise, using an FPTAS for the classic 0-1 knapsack problem, we can examine all the $\binom{M}{c}$ possible subsets of c colors). We handle separately instances from three categories:

1. Instances with *uniform* value, in which all the items have the same value (regardless of their size or color). In the context of production planning (see in Section 1.1.2), such instances reflect systems in which the revenue is proportional to the *number* of items produced (regardless of their types and demands for resources)
2. Instances with *color-based* values, in which the items in each color have the same value (and arbitrary sizes), i.e., for $1 \leq i \leq M$, the value of any item in color i is p_i. This class reflects systems in which the revenue depends on the product type, regardless of its demand for resources
3. Instances with *arbitrary* values, independent of the item sizes and colors.

For uniform-value instances we present a polynomial time optimal algorithm. Note that the problem becomes NP-hard when we have to pack the items into two (or more) knapsacks. Formally,

Theorem 3.3.1 *The CCMK problem with $N > 1$ knapsacks is NP-hard for uniform-value instances.*

Proof: We show a reduction from the *Partition problem*, which is known to be NP-hard [20]. The partition problem consists of a finite set A of $2k$ items with integral sizes $a_1, a_2, \ldots, a_{2k}$. The problem is to determine, if there exists a subset A' of A such that $|A'| = k$ and $\sum_{j \in A'} a_j = \sum_{j \in A \setminus A'} a_j$.

Given an instance for partition, $A = \{a_1, a_2, \ldots, a_{2k}\}$, let $\sum_{j=1}^{2k} a_j = 2S$ (if the sum is odd then clearly no partition exists). We construct an input for the CCMK with two knapsacks with $v = S$ and $c = k$. The items to be packed are of $M = 2k$ colors. There are $2k$ items, one in each color: the ith item has size a_i and value $p_i = 1,\ 1 \leq i \leq M$. It is easy to verify that there is a packing with value $2k$ if and only if there exists a desired partition. ■

When dealing with uniform-value instances, packing a subset of items of *maximal value* is equivalent to packing the *maximal number* of items. Intuitively, we would like to pack small items first: indeed, when $c \geq M$ the color constraints can be ignored, and the resulting problem can be solved optimally, by a greedy algorithm which adds to the knapsack the smallest available item, until no item can be added. However, when $c < M$, the greedy approach ceases to be the optimal. In fact, we show that this simple greedy algorithms, $\mathcal{A}_g$, can perform poorly, even for instances with uniform value. Formally, $\mathcal{A}_g$ proceeds as follows.

1. Sort the items by sizes (small items first).
2. Repeat as long as possible: if the knapsack contains items of less than c colors, add to the knapsack the smallest available item, otherwise, add the smallest available item from one of the c colors that are included in the knapsack.

Claim 3.3.2 *$\mathcal{A}_g$ has an unbounded approximation ratio.*

Proof: For some $\alpha > 1$, consider an instance consisting of items of $M = 2c$ different colors: c colors with one item of size $\varepsilon < \frac{v}{\alpha c}$ and additional item of size v; and c colors with α items of size $\frac{v}{\alpha c}$ each. The optimum packs $c\alpha$ items of the last c colors, while the algorithm $\mathcal{A}_g$ packs the c smallest items of the first c colors, resulting in approximation ratio of α. ■

For non-uniform values, we present an FPTAS for the CCKP. Our FPTAS has different time complexity when applied to instances with color-based values, and to instances with arbitrary values. Note that for instances with non-uniform values the problem is NP-hard. Indeed, when $c = M = n$, i.e., there is one item in each color and no color-constraints, we get an instance of the classic 0-1 knapsack problem. Note that the above also implies that the problem is NP-hard for instances with color-based values.

As in the FPTASs for the non class-constrained problem [34], and for the cardinality-constrained problem [7], we combine scaling of the profits with dynamic programming (DP). However, in the class-constrained problem we need two levels of DP recursions, as described below.

3.3.1 An Optimal Solution Using Dynamic Programming

Assume that we have an upper bound, $\hat{P}$, on the optimal value of the solution to our problem. Given such an upper bound, we can formulate an algorithm, based on dynamic programming, to compute the optimal solution. The time complexity of the algorithm is polynomial in $\hat{P}, n$ and M.

For each color $i \in \{1, \ldots, M\}$, let n_i denote the number of items of color i in the instance I. Thus, $n = \sum_{i=1}^{M} n_i$. Let $f_i(a, \ell)$ denote the smallest size sum of items with total value a of ℓ distinct colors out of the colors $1, \ldots, i$, where $i = 1, \ldots, M$; $a = 0, \ldots, \hat{P}$; $\ell = 1, \ldots, c$.

Dynamic Programming for Instances with Color-Based values

When packing instances with color-based values, in any optimal solution, we can replace the packed items of color i by the smallest items of color i, for all $1 \leq i \leq M$. Thus, we consider below only solutions in which we pack the smallest items of each color. For each $1 \leq j \leq n_i$, let $s_i(j)$ denote the size sum of the j smallest items of color i. Recall that p_i is the value of the items in color i.

The dynamic programming recursion is defined as follows. We initialize

$$f_0(a, \ell) = \begin{cases} 0 & a = 0 \text{ and } \ell = 0 \\ +\infty & \text{otherwise} \end{cases}$$

and $f_i(a, 0) = +\infty \quad \forall 1 \leq i \leq M, \ \forall a > 0$. Then, for $i = 1, \ldots, M$, the entries of f_i can be computed from those of f_{i-1} by using for $\ell \in \{1, ...c\}$, $a \in \{0, ..., \hat{P}\}$ the formula

$$f_i(a, \ell) = \min \begin{cases} f_{i-1}(a, \ell) \\ \min\limits_{1 \leq j \leq n_i, \ a \leq jp_i} (f_{i-1}(a - jp_i, \ell - 1) + s_i(j)) \end{cases}$$

The first line considers the case in which color i is not included in the packing. The second line considers the cases in which color i is included and j items of color i are packed.

The value of an optimal solution for the problem is given by

$$\max_{a=0,\ldots,\hat{P};\ \ell=0,\ldots,c} \{a : f_M(a, \ell) \leq v\}.$$

The time complexity of the algorithm is $O(\sum_{i=1}^{M} n_i \hat{P} c) = O(n\hat{P}c)$.

For the special case of uniform-value instances, we can assume w.l.o.g., that $\forall u \in I, p(u) = 1$. Since we pack at most n items, we can bound the maximal profit by $\hat{P} = n$, and we get an optimal $O(n^2 c)$ algorithm.

Dynamic Programming for Arbitrary Values

When the items have arbitrary values, the algorithm consists of two stages. In the first stage we calculate for each color i the value $h_{i,k}(a)$, which denotes the smallest size sum of items with total value a, out of the first k items of color i (the items in each color are given in arbitrary order). $h_{i,k}(a)$ is calculated for $k = 1, \ldots, n_i$, and $a = 0, \ldots, \hat{P}$.

For each color $i = 1, \ldots, M$, the DP algorithm calculates h_i as follows. We initialize $h_{i,0}(0) = 0$ and $h_{i,0}(a) = +\infty$ for $a > 0$. Denote by p_k^i, s_k^i the value and size of the kth item of color i, respectively.

For $k = 1, \ldots, n_i$, the entries of $h_{i,k}$ can be computed from those of $h_{i,k-1}$ by using the formula

$$h_{i,k}(a) = \begin{cases} h_{i,k-1}(a) & \text{if } a < p_k^i \\ \min(h_{i,k-1}(a), h_{i,k-1}(a - p_k^i) + s_k^i) & \text{otherwise} \end{cases}$$

The time complexity of computing the table h_i is $O(\hat{P} n_i)$, which gives a total of $O(\hat{P} n)$ for the calculation of all the tables h_i, $\forall 1 \leq i \leq M$.

In the second stage of the algorithm, we calculate the tables f_i, $\forall 1 \leq i \leq M$. The table f_i can be calculated using the tables h_i, $\forall 1 \leq i \leq M$. We initialize

$$f_0(a, \ell) = \begin{cases} 0 & a = 0 \text{ and } \ell = 0 \\ +\infty & \text{otherwise} \end{cases}$$

and $f_i(a, 0) = +\infty \quad \forall 1 \leq i \leq M$, $\forall a > 0$. Then, for $i = 1, \ldots, M$, the entries of f_i can be computed from those of f_{i-1} by using for $\ell \in \{1, ...c\}$, $a \in \{0, ..., \hat{P}\}$ the formula

$$f_i(a, \ell) = \min \begin{cases} f_{i-1}(a, \ell) \\ \min\limits_{1 \leq a' \leq a} (f_{i-1}(a - a', \ell - 1) + h_{i,n_i}(a')) \end{cases}$$

The first line considers the case in which color i is not included in the packing. The second line considers the cases in which color i is included in the packing and contributes a' to the total value.

The value of an optimal solution for the problem is given by

$$\max_{a=0,\ldots,\hat{P};\ \ell=0,\ldots,c} \{a : f_M(a, \ell) \leq v\}.$$

The time complexity of the recursion is $O(Mc\hat{P}^2)$. Adding the time needed to construct the tables h_i, which is $O(\hat{P} n)$, we have a total of $O(Mc\hat{P}^2 + n\hat{P})$.

3.3.2 FPTAS for Non-Uniform Values 0-1 Knapsack Problem

By using the pseudo-polynomial algorithm above we can devise an FPTAS for instances with non-uniform values. First, we need an upper bound on the value, z^*, of an optimal solution. Such an upper bound can be obtained from the $\frac{c}{2M}$-approximation algorithm, $\mathcal{A}$ (given in Section 3.2.2). Let z be the profit obtained by $\mathcal{A}$. Then, clearly, $\frac{2M}{c} z$ is an upper bound on the value of z^*.

We scale the item values by replacing each value p_i by $q_i = \lceil \frac{p_i n}{\varepsilon z} \rceil$, where $1 - \varepsilon$ is the required approximation ratio. Now, we need to scale accordingly the upper bound, $\hat{P}$, on the profit.

Lemma 3.3.3 *$\hat{P} = \frac{2M}{c}\lceil \frac{n}{\varepsilon} \rceil + n$ is an upper bound for the scaled problem.*

Proof: Recall that $\frac{2M}{c}z$ is an upper bound for the non-scaled instance. Since we pack at most n items, the rounding of the q_i's can add at most n to the total scaled value. Thus, if we can achieve profit larger than $\hat{P}$ in the scaled problem, the same items achieve profit larger than $\frac{2M}{c}\lceil \frac{n}{\varepsilon} \rceil \frac{z\varepsilon}{n} \geq \frac{2M}{c}z$ in the non-scaled instance. ∎

Finally, we apply the DP scheme and return the optimal 'scaled' solution as the approximated solution for the non-scaled instance. (In color-based instances, the scaled values are used in the calculation of f; in arbitrary-value instances, the scaled values are used in the calculation of the tables h_i). As in [34] and [7] this scaling technique yields an FPTAS for CCKP.

Theorem 3.3.4 *The above algorithm is an FPTAS for CCKP, whose running time is $O(Mn^2/\varepsilon)$ for instances with color-based values, and $O(\frac{1}{\varepsilon}M^2n^2(1+\frac{M}{c\varepsilon}))$ for instances with arbitrary values.*

Proof: We first show that the returned solution value is at least $(1-\varepsilon)z^*$. Let R^* and R^*_s be an optimal item set for the original and for the scaled problem, respectively. The sizes of the items and the knapsack are not influenced by the scaling, thus, R^*_s is a feasible solution for the original problem. For a set U, let $q(U) = \sum_{j \in U} q(j)$ be the total 'scaled' values of the items in U. R^*_s is optimal for the scaled problem, thus,

$$q(R^*_s) \geq q(R^*). \tag{3.6}$$

By definition, for any value p_i, we have,

$$\frac{z\varepsilon}{n}(q_i - 1) < p_i \leq \frac{z\varepsilon}{n}q_i. \tag{3.7}$$

Using Equations 3.6 and 3.7 we get that

$$p(R^*) - p(R^*_s) \leq \frac{z\varepsilon}{n}(q(R^*) - (q(R^*_s) - |R^*_s|)) \leq \frac{z\varepsilon}{n}|R^*_s| \leq z\varepsilon \leq z^*\varepsilon.$$

The time complexity is obtained by setting $\hat{P} = \frac{2M}{c}\lceil \frac{n}{\varepsilon} \rceil + n$ in the time complexities of the pseudo-polynomial algorithms. For instances with color-based values, the time complexity of the algorithm is $O(nc\hat{P}) = O(Mn^2/\varepsilon)$. For arbitrary values we get $O(Mc\hat{P}^2 + n\hat{P}) = O(\frac{1}{\varepsilon}M^2n^2(1+\frac{M}{c\varepsilon}))$. ∎

For the CCMK problem it is now natural to analyze the greedy algorithm, which packs the knapsacks sequentially by applying the above FPTAS for a single knapsack to the remaining items. Recently, this algorithm was analyzed for the MK [10]. Let Greedy(ε) refer to this algorithm, with error-parameter ε in the FPTAS for a single knapsack. It turns out that the analysis for MK is suitable also for CCMK. The main idea is that for any knapsack, K_j, the set of items that are packed in K_j in some optimal placement and are not packed at all by Greedy, is available for Greedy when it fills K_j. The fact that this set is available and *feasible* is valid also in class-constrained packing, thus, the whole proof presented in [10] is applied here and we get:

Theorem 3.3.5 *Greedy(ε) yields a $(2+\varepsilon)$-approximation for CCMK.*

Since MK is a special case of CCMK, it follows from [10] that the bound is tight; also, the performance of the algorithm cannot be improved by ordering the knapsacks in non-increasing order by their capacities.

3.4 Generalized Class-Constrained Packing

We consider two generalizations of class-constrained packing. For both, we show that the resulting problem is APX-hard, that is, there exists $\varepsilon_1 > 0$ such that it is NP-hard to decide whether an instance has a maximal profit P, or if every legal packing has profit at most $(1-\varepsilon_1)P$.

We use an approximation-preserving reduction from the maximum 3-bounded 3-dimensional matching problem (3DM-3), defined as follows.

Input: A set of triplets $T \subseteq X \times Y \times Z$, where $|X| = |Y| = |Z| = n$; the number of occurrences of any item of $X \cup Y \cup Z$ in T is at most 3. The number of triplets is $|T| \geq n$.

Output: A 3-dimensional matching in T of maximal cardinality, i.e., a subset $T' \subseteq T$, such that any item in X, Y, Z appears at most once in T', and $|T'|$ is maximal.

Kann showed in [40] that 3DM-3 is APX-hard, that is, there exists $\varepsilon_0 > 0$ such that it is NP-hard to decide whether an instance has a matching of size n, or if every matching has size at most $(1-\varepsilon_0)n$.

3.4.1 Class-constrained Packing with Color-sets

In the first generalization we consider, each item $u \in I$ is associated with a set $C(u)$ of colors. Denote by $C(K_j)$ the set of colors to which the knapsack K_j allocates compartments ($|C(K_j)| \leq c_j$). Then, u can be packed in K_j iff $C(u) \cap C(K_j) \neq \emptyset$. We show that the resulting GCCP$_1$ problem is APX-hard. This hardness result holds even for a single knapsack, and for instances, I, in which all the items have the same size and the same value. Moreover, for each item $u \in I$, the cardinality of $C(u)$ is a constant (at most 4).

Theorem 3.4.1 *The GCCP$_1$ problem is APX-hard, even for a single knapsack and instances with uniform value and size.*

Proof: Given an instance of 3DM-3, we construct the following instance I for the generalized class-constrained 0-1 knapsack problem. The knapsack has capacity $3n$, and n compartments. There are $3n$ items: for each item i of X we have an item x_i, and similarly y_j for each $j \in Y$ and z_k for $k \in Z$. All the items $u \in I$ have the same size $s(u) = 1$, and the same value $p(u) = 1$. Let

t be the cardinality of T, and denote by $e_1, \ldots, e_t$ the triplets in T. The color sets of the items are as follows.

$$C(x_i) = \{x_i\} \cup \{e_\ell | i \in e_\ell\}$$
$$C(y_j) = \{y_j\} \cup \{e_\ell | j \in e_\ell\}$$
$$C(z_k) = \{z_k\} \cup \{e_\ell | k \in e_\ell\}$$

Note that the reduction is polynomial, and since the 3DM instance is 3-bounded, $|C(u)| \leq 4$ for each $u \in I$.

Clearly, colors of type x_i, y_j, z_k can contribute at most one to the total profit, while colors of type e_ℓ can contribute at most three. For a given packing, each color of type e_ℓ which contributes three to the profit, induces a triplet in T, and since each item can be packed only once, the set of these triplets form a matching in the 3DM-3 instance.

Let t^* be the size of an optimal matching, and let p^* be the maximal profit from packing I.

Claim 3.4.2 *For any $\varepsilon_1 > 0$, if $p^* = 3n$ then $t^* = n$, and if $p^* \leq (1 - \varepsilon_1)3n$ then $t^* \leq n(1 - \frac{3}{2}\varepsilon_1)$.*

Proof: First, if $p^* = 3n$, then by the above discussion, each of the n compartments must be allocated to a color of type e_ℓ and a matching of size n is induced. Thus, $t^* = n$. Assume that $t^* > n - x$, for some $0 \leq x \leq n$. Then we can pack three items of each of the $n - x$ colors of type e_ℓ participating in the matching, and at least x additional items of other colors - one of each color. Thus, $p^* > 3(n - x) + x = 3n - 2x$. In particular, for $x = \frac{3}{2}n\varepsilon_1$, we get that if $p^* \leq (1 - \varepsilon_1)3n$ then $t^* \leq n(1 - \frac{3}{2}\varepsilon_1)$. ∎

Thus, if for any $\varepsilon_1 > 0$ we could decide whether a given instance, I, has maximal profit P, or if every packing has profit at most $(1 - \varepsilon_1)P$, then, taking $\varepsilon_1 = \frac{2}{3}\varepsilon_0$ we could contradict the APX-hardness of 3DM-3. ∎

Remark 3.4.1 *Note that the $GCCP_1$ instance constructed in our reduction has $M = 3n + t$ distinct colors. The question whether $GCCP_1$ is APX-hard when M is a constant remains open.*

3.4.2 Class-Constrained Packing with Knapsack-Dependent Colors

In this section we consider another generalization of class-constrained packing, in which the color of each item depends on the knapsack in which it is placed. That is, each item $u \in I$ has a size $s(u)$, a profit $p(u)$ and an N-vector, $\vec{c}_u$, where $\vec{c}_u(\ell)$ is the color of u when it is placed in the knapsack K_ℓ, $1 \leq \ell \leq N$. This generalization is motivated from production systems in which the configuration required for processing a task may be different on different machines.

Theorem 3.4.3 *The class constrained packing problem with knapsack-dependent colors is APX-hard, even for instances with uniform value and size, and identical knapsacks.*

Proof: Given an instance of 3DM-3, we construct the following instance I for the class-constrained problem with knapsack dependent colors. Let t be the cardinality of T. There are $N = t$ knapsacks with $v_\ell = 3$, $c_\ell = 1, \forall 1 \leq \ell \leq t$. For each item i of X we have an item x_i, and similarly y_j for each $j \in Y$ and z_k for $k \in Z$. There are $2t$ *additional* items, arranged in pairs w_h^1, w_h^2, $\forall 1 \leq h \leq t$.

All the items $u \in I$ have the same size $s(u) = 1$, and the same value $p(u) = 1$.

Denote by $e_1, \ldots, e_t$ the triplets in T. The color vectors of the items are as follows.

$$\vec{c}_{x_i}(\ell) = \begin{cases} e_\ell & \text{if } i \in e_\ell \\ x_i & \text{otherwise} \end{cases} \qquad \forall 1 \leq \ell \leq N,\ 1 \leq i \leq n$$

$$\vec{c}_{y_j}(\ell) = \begin{cases} e_\ell & \text{if } j \in e_\ell \\ y_j & \text{otherwise} \end{cases} \qquad \forall 1 \leq \ell \leq N,\ 1 \leq j \leq n$$

$$\vec{c}_{z_k}(\ell) = \begin{cases} e_\ell & \text{if } k \in e_\ell \\ z_k & \text{otherwise} \end{cases} \qquad \forall 1 \leq \ell \leq N,\ 1 \leq k \leq n$$

$$\vec{c}_{w_h^1}(\ell) = \vec{c}_{w_h^2}(\ell) = w_h \qquad \forall 1 \leq \ell \leq N,\ 1 \leq h \leq t$$

That is, the ℓth triplet of T has a unique color, e_ℓ, and an item in this triplet is colored with e_ℓ if and only if it is placed in K_ℓ; otherwise, the item is colored with a unique color. In addition, each pair of the additional items has a unique color (independent of the knapsack in which it is placed).

Note that the reduction is polynomial, and since the 3DM instance is 3-bounded, each item has at most 4 distinct colors.

Let t^* be the size of an optimal matching, and let p^* be the maximal profit from packing I.

Claim 3.4.4 *For any* $0 \leq x \leq n$, $p^* = 2t + n - x$ *if and only if* $t^* = n - x$.

Proof: Since each knapsack has a single compartment, it can accommodate three items if and only if they compose a triplet in T. Also, since we can always place in a knapsack one pair of the "additional" items, in any optimal packing each knapsack includes at least two items. Thus, there is a maximal matching of size $n - x$ iff in an optimal packing there are $n - x$ knapsacks with three items in each and $t - (n - x)$ knapsacks with two items in each. That is, $t^* = n - x$ iff $p^* = 3(n - x) + 2(t - (n - x)) = 2t + n - x$. ■

In particular, for $x = \varepsilon_0 n$, we get that

$$p^* \in [2t + n - \varepsilon_0 n, 2t + n] \text{ if and only if } \quad t^* \in [(1 - \varepsilon_0)n, n]. \tag{3.8}$$

Let $\varepsilon_1 = \frac{\varepsilon_0 n}{2t+n}$. Note that $t \leq 3n$, since the number of occurrences of any item of $X \cup Y \cup Z$ in T is at most 3. Thus, ε_1 is a fixed constant.

Assume that for any $\varepsilon > 0$ we could decide whether a given instance, I, has maximal profit $2t + n$, or if every packing has profit at most $(1 - \varepsilon)(2t + n)$. In particular, this holds for ε_1. However, since $(1-\varepsilon_1)(2t+n) = (1 - \frac{\varepsilon_0 n}{2t+n})(2t+n) = 2t+n-\varepsilon_0 n$, using (3.8) we can contradict the APX-hardness of 3DM-3. ∎

Chapter 4

Online Class-Constrained Packing

In this Chapter we study the *online* versions of CCMK and CCBP, with unit-sized item. We are given a set of bins, each having a capacity v and c compartments, and n items of M classes (colors) and the same (unit) size. The items arrive as a sequence, one at a time. In each step we get an item, $a \in I$, having size 1 and color $i \in \{1, \ldots, M\}$; we need to pack a before we know any of the subsequent items. We need to fill the bins with items, subject to capacity constraints, and such that items of different classes are placed in separate compartments. In the *class-constrained bin-packing problem (CCBP)*, our goal is to find a legal placement of all the items in a minimal number of bins; in the *class-constrained multiple knapsack problem (CCMK)*, we wish to maximize the total number of packed items.

We derive tight bounds for the online variants of CCBP and CCMK. For CCBP we present a lower bound of $(1 + \alpha)$ on the competitive ratio of *any* deterministic algorithm, where $\alpha \in (0, 1]$ depends on v, c, M and n. We show that this ratio is achieved by the algorithm *first-fit*. We then consider the *temporary CCBP*, in which items may be packed for a *bounded* time interval (which is *unknown* in advance). We obtain a lower bound on the competitive ratio of any deterministic algorithm, and show that it is achieved by all *any-fit* algorithms. Finally, tight bounds are derived for the online CCMK and *temporary* CCMK problems.

4.1 Preliminaries

For a given (possibly partial) packing, a bin is denoted *full* if it contains v items. Other bins are *non-full*. In addition, a bin is denoted *occupied* if it contains items of c distinct colors, while other bins are *non-occupied*. Thus, a bin can be in one of four states: full and occupied, full and non-occupied, etc. For a bin B, we denote by $C(B)$ the set of colors contained in B. Initially, $\forall B, C(B) = \emptyset$. During the placement of the items, whenever B allocates a compartment to some color i, the color i is added to $C(B)$.

Let $\sigma = a_1, a_2, \ldots$ be a sequence of items to be packed. Clearly, upon arrival, an item a_k of color i has to be placed in some bin B such that: (i) B is not full, and (ii) $i \in C(B)$ or B is not occupied. A bin fulfilling these two requirement is *possible* for a_k. Thus, an online placement algorithm needs to determine in which possible bin a_k will be placed. It could be either a bin which already holds some items, or a new (empty) bin, which is always possible. In CCMK, an item can also be rejected.

We first give a simple lower bound on $N_{opt}(\sigma)$.

Claim 4.1.1 *For any* $v > c > 1, \sigma$, $N_{opt}(\sigma) \geq \max\{\lceil \frac{n}{v} \rceil, \lceil \frac{M_\sigma}{c} \rceil\}$.

Proof: The total size of the items in σ is $|\sigma| = n$. Thus, even if all the bins are full, the capacity bound for each bin implies that at least $\lceil n/v \rceil$ bins are used. Similarly, the total number of compartments needed for all the items of σ is at least M_σ. Thus, even if all the bins are occupied, at least $\lceil M_\sigma / c \rceil$ bins are used. ∎

Note that when $c = 1$ we must pack items of different colors in separate bins, and the greedy algorithm is optimal for CCBP. Hence, throughout the discussion of the CCBP problem, we assume that $c > 1$. An upper bound on $N_{opt}(\sigma)$ follows from the approximation algorithm presented in Section 2.2.2.

Property 4.1.2 *For any* $v > c > 1, \sigma$, $N_{opt}(\sigma) \leq \max\{\lceil \frac{n}{v} \rceil, \lceil \frac{M_\sigma - 1}{c-1} \rceil\}$.

An algorithm is called *any-fit* if it never opens a new bin for an item that can be packed in one of the open bins. Any-fit algorithms may differ in the way they choose the bin in which the item will be placed.

The *first-fit* algorithm always packs an arriving item into the first (lowest indexed) possible bin. That is, we place a new item, $a \in I_i$, in the first non-full bin, B, which is non-occupied, or $i \in C(B)$. If no such bin exists, we open for a a new bin.

Note that since all items have size 1, the non class-constrained versions of our problems, can be solved optimally by a greedy algorithm, which packs each arriving item in the 'currently filled' bin. For the BP problem, this algorithm uses $\lceil n/v \rceil$ bins, which is clearly optimal. For the MK problem this algorithm packs $\min(n, mv)$ items, which is also optimal. Since we are interested in instances in which the value of c imposes a restriction on the packing, we assume that $c < \min(M, v)$.

4.1.1 Performance Measures

Let $\mathcal{A}$ be an online algorithm for CCBP. We denote by $N_A(\sigma)$, $N_{opt}(\sigma)$, the number of bins used by $\mathcal{A}$, an optimal (offline) algorithm, for packing the items in σ. By standard definition (see, e.g., [5]), we say that $\mathcal{A}$ is ρ-competitive, for some $\rho \geq 1$, if for any input sequence, σ, $r_A(\sigma) = N_A(\sigma)/N_{opt}(\sigma) \leq \rho$. Another measure of interest is the *asymptotic worst-case ratio*, given by

$$r_A = \lim_{N_0 \to \infty} \sup (\max_{\sigma}\{N_A(\sigma)/N_0 \mid N_{opt}(\sigma) = N_0\}).$$

In some cases, we will be interested in the performance of a packing algorithm, $\mathcal{A}$, on *a set of inputs*. Formally, for a set S, $r_A(S)$ denotes the competitive ratio of $\mathcal{A}$ on inputs from S, that is, $\forall \sigma \in S,\ N_A(\sigma) \leq r_A(S) N_{opt}(\sigma)$. In particular, we will be interested in $r_A(S_{c,v}(k,h))$, where for given c, v, the set $S_{c,v}(k,h)$ consists of all the input sequences, σ, in which the number of colors, M_σ, satisfies $kc < M_\sigma \leq (k+1)c$, and $hv < |\sigma| \leq (h+1)v$. Our refined analysis for sets of inputs will yield tighter performance bounds which depend on $v, c, |\sigma|$ and M_σ.

4.2 Deterministic Algorithms for CCBP

In this section we consider deterministic algorithms for CCBP. We first present a general lower bound on the competitive ratio of *any* deterministic algorithm. Then, we show that *first-fit* achieves this lower bound. Another greedy algorithm, based on partitioning the items into color-sets, is also shown to be efficient.

The deterministic algorithms that we analyze do not use the values of n, M_σ, while our lower bound holds for any deterministic algorithm, even one that knows n and M_σ in advance. We therefore conclude that a-priori knowledge of these parameters cannot improve the performance of a placement algorithm.

4.2.1 A Lower Bound for any Deterministic Algorithm

Recall that for given c, v, an input sequence, σ, is in $S_{c,v}(k,h)$ *iff* $kc < M_\sigma \leq (k+1)c$, and $hv < |\sigma| \leq (h+1)v$. In other words, $\lceil M_\sigma/c \rceil = k+1$ and $\lceil |\sigma|/v \rceil = h+1$.

Theorem 4.2.1 *For any deterministic algorithm, $\mathcal{A}_d$, and for any $v > c > 1$, $k \geq 0$, and $h \geq k - 1 + \max\{\lceil \frac{k}{c-1} \rceil, \lceil \frac{kc+1}{v} \rceil\}$,*

$$r_d(S_{c,v}(k,h)) \geq 1 + \frac{k+1-\lceil \frac{kc+1}{v} \rceil}{h+1}.$$

Proof: We show that there exists an input sequence, $\sigma \in S_{c,v}(k,h)$, such that $\mathcal{A}_d$ uses at least $k+h+2-\lceil \frac{kc+1}{v} \rceil$ bins to pack σ, while $N_{opt}(\sigma) = h+1$. The sequence σ is constructed online by the adversary according to the way $\mathcal{A}_d$ packs the items. We denote by $B_1, B_2, \ldots$ the sequence of bins used by $\mathcal{A}_d$.

Let $x = h - k + 1 - \lceil \frac{kc+1}{v} \rceil \geq 0$. The length of σ is $n = (k+x)v + kc + 1$, thus $\lceil \frac{n}{v} \rceil = k + x + \lceil \frac{kc+1}{v} \rceil = h + 1$. The number of distinct colors in σ is $kc+1$. Therefore, $\sigma \in S_{c,v}(k,h)$. We assume that this is known to the algorithm in advance; thus, $\mathcal{A}_d$ can use the values h, k in making its deterministic decisions. The sequence σ is constructed as follows.

Step 1. For $i = 1$ to kc:

pack items of color i, until for the first time, an item of color i is placed in one of the bins $B_{k+x+1}, B_{k+x+2}, \ldots$.

Step 2. Pack items of color $kc+1$, until $|\sigma| = (k+x)v + kc + 1$.

Claim 4.2.2 *$\mathcal{A}_d$ uses at least $h + k + 2 - \lceil \frac{kc+1}{v} \rceil = 2k + x + 1$ bins to pack σ.*

Proof: Clearly, if B_{2k+x+1} is used in Step 1, then at least $2k+x+1$ bins are used for the whole sequence. Otherwise, the claim is proved by the following facts (See Figure 4.1):

- After Step 1, each of the bins $B_{k+x+1}, B_{k+x+2}, \ldots, B_{2k+x}$ contains exactly c items of different colors. This holds since the adversary moves to the next color whenever an item is placed in one of these bins, for each of the kc colors $1, 2, \ldots, kc$.
- The items packed in Step 2 cannot be placed in $B_{k+x+1}, B_{k+x+2}, \ldots, B_{2k+x}$ (which are occupied by items of colors different than $kc+1$); therefore, they must go to the bins $B_1, B_2, \ldots, B_{k+x}$ or $B_{2k+x+1}, B_{2k+x+2}, \ldots$.
- Since $|\sigma| = (k+x)v+kc+1$ and exactly kc items are placed in the bins $B_{k+x+1}, B_{k+x+2}, \ldots, B_{2k+x}$, the remaining $(k+x)v+1$ items must be placed in $B_1, B_2, \ldots, B_{k+x}$ or $B_{2k+x+1}, B_{2k+x+2}, \ldots$. Even if we fill each of $B_1, B_2, \ldots, B_{k+x}$, at least one item must go to B_{2k+x+1}.

■

Claim 4.2.3 *For any $h \geq k - 1 + \lceil \frac{k}{c-1} \rceil$, $N_{opt}(\sigma) = h + 1$.*

Proof: The length of σ is $n = (k+x)v + kc + 1$. The number of distinct colors in σ is $M_\sigma = kc+1$. By Property 4.1.2, $N_{opt}(\sigma) \leq \max\{\lceil \frac{n}{v} \rceil, \lceil \frac{M_\sigma - 1}{c-1} \rceil\}$. We have $\lceil \frac{n}{v} \rceil = h+1$, and $\lceil \frac{M_\sigma - 1}{c-1} \rceil = \lceil \frac{kc}{c-1} \rceil$. Given that $h \geq k - 1 + \lceil \frac{k}{c-1} \rceil$, we get $h + 1 \geq \lceil \frac{kc}{c-1} \rceil$; that is, $N_{opt}(\sigma) \leq h+1$. ■

Combining Claims 4.2.2 and 4.2.3 we conclude that for any c, v, k and $h \geq k-1+\max\{\lceil \frac{k}{c-1} \rceil, \lceil \frac{kc+1}{v} \rceil\}$, $r_d(S_{c,v}(k,h)) \geq 1 + \frac{k+1-\lceil \frac{kc+1}{v} \rceil}{h+1}$. ■

Recall that $r_d = \limsup_{N_0 \to \infty} (\max_\sigma \{N_d(\sigma)/N_0 \mid N_{opt}(\sigma) = N_0\})$. For any $N_0 > 1$, let $k = h = N_0 - 1$, and consider an instance with bins having $c \geq k+1$ and $v \geq kc+1$. Let σ be the input sequence constructed by the adversary for $S_{c,v}(k,h)$. Note that $h \geq k - 1 + \max\{\lceil \frac{k}{c-1} \rceil, \lceil \frac{kc+1}{v} \rceil\}$. By Claims 4.2.2 and 4.2.3, $\mathcal{A}_d$ uses at least $k + h + 2 - \lceil \frac{kc+1}{v} \rceil = 2N_0 - 1$ bins to pack σ, while $N_{opt}(\sigma) = h + 1 = N_0$. That is, $r_d(\sigma) \geq \frac{2N_0 - 1}{N_0}$.

Corollary 4.2.4 *For any deterministic algorithm, $\mathcal{A}_d$, $r_d \geq 2$.*

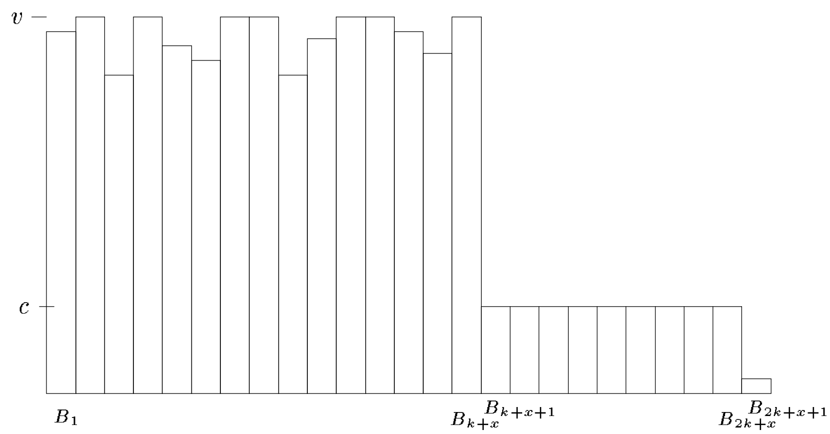

Figure 4.1: The placement produced by $\mathcal{A}_d$ (in proof of Theorem 4.2.1)

4.2.2 First-fit, an Optimal Deterministic Algorithm

In this section we consider the *first-fit* algorithm, $\mathcal{A}_F$. Clearly, the following property holds for $\mathcal{A}_F$:

Property 4.2.5 *At any time during the execution of $\mathcal{A}_F$, each bin, except maybe for the last one, is either full or occupied.*

Note that, given a placement of the items by $\mathcal{A}_F$, for each bin, B, and color, $i \in C(B)$, B allocated a compartment to i only if all the previous bins that contain items of I_i are full. Moreover, as long as B is not full, any $a \in I_i$ can be placed in B. Hence, we get the next result.

Property 4.2.6 *In any placement produced by $\mathcal{A}_F$, each non-full bin, B, holds the last item of each of the colors in $C(B)$.*

We show that the competitive ratio of $\mathcal{A}_F$ for any input sequence, σ, and for any values of v, c is less than 2. Moreover, for any c, v, k, h, the competitive-ratio of $\mathcal{A}_F$ on $S_{c,v}(k,h)$ matches the lower bound given in Theorem 4.2.1.

Theorem 4.2.7 *For any c, v, k, h,*

$$r_F(S_{c,v}(k,h)) \leq 1 + \frac{k+1-\lceil\frac{kc+1}{v}\rceil}{h+1}.$$

Proof: Let $\sigma \in S_{c,v}(k,h)$ be an input sequence for $\mathcal{A}_F$. Assume that there are m_1 full bins and m_2 non-full bins in the placement produced by $\mathcal{A}_F$ for σ. By Claim 4.1.1, $N_{opt}(\sigma) \geq \lceil\frac{n}{v}\rceil = h+1$. Thus, $r_F(\sigma) \leq \frac{m_1+m_2}{h+1}$.

By Property 4.2.5, all the non-full bins, except maybe for the last one, are occupied. Therefore, the length of σ is $n \geq m_1 v + (m_2 - 1)c + 1$ (The last non-full bin contains at least one item). Since $\sigma \in S_{c,v}(k,h)$, $h + 1 = \lceil \frac{n}{v} \rceil \geq m_1 + \lceil \frac{(m_2-1)c+1}{v} \rceil$. The coefficients of m_1 and m_2 in this inequality are 1 and $\frac{c}{v} < 1$; thus, $m_1 + m_2$ is maximized when m_2 gets a maximal value. This value can be bounded as follows.

Claim 4.2.8 $m_2 \leq k + 1$.

Proof: By Properties 4.2.5 and 4.2.6 each non-full bin, except maybe for the last one, contains items of c distinct colors. The last non-full bin contains items of at least one additional color. Thus, in order to have $k+2$ or more non-full bins we need items of at least $(k+1)c+1$ distinct colors; however, $M_\sigma \leq (k+1)c$. ■

Setting $m_2 = k + 1$, we get $m_1 \leq h + 1 - \lceil \frac{kc+1}{v} \rceil$. Thus, $m_1 + m_2 \leq k + 1 + h + 1 - \lceil \frac{kc+1}{v} \rceil$, and $r_F(\sigma) = 1 + \frac{k+1-\lceil \frac{kc+1}{v} \rceil}{h+1}$. ■

For a given c, let $S_c(k)$ denote the set of input sequences such that $\sigma \in S_c(k)$ iff $kc < M_\sigma \leq (k+1)c$ (that is, $\lceil \frac{M_\sigma}{c} \rceil = k + 1$).

Theorem 4.2.9 *For any $k \geq 0$ and $c > 1$, $r_F(S_c(k)) \leq 2 - \frac{1}{k+1} \lceil \frac{kc+1}{v} \rceil$.*

Proof: Let $\sigma \in S_{c,v}(k,h)$ be an input sequence for $\mathcal{A}_F$. If $h \geq k$, then by Theorem 4.2.7,

$$r_F(\sigma) \leq 1 + \frac{k+1-\lceil \frac{kc+1}{v} \rceil}{h+1} \leq 1 + \frac{k+1-\lceil \frac{kc+1}{v} \rceil}{k+1} = 2 - \frac{1}{k+1} \lceil \frac{kc+1}{v} \rceil.$$

If $h < k$, then by Claim 4.1.1, $N_{opt}(\sigma) \geq \max\{h+1, k+1\} = k+1$. Assume that there are m_1 full bins and m_2 non-full bins in the placement produced by $\mathcal{A}_F$ for σ. Thus, $r_F(\sigma) \leq \frac{m_1+m_2}{k+1}$. As in the proof of Theorem 4.2.7, we have that $m_1 + m_2$ gets its maximal value when $m_2 = k + 1$, and $m_1 \leq h + 1 - \lceil \frac{kc+1}{v} \rceil < k + 1 - \lceil \frac{kc+1}{v} \rceil$. Therefore, $m_1 + m_2 < 2(k+1) - \lceil \frac{kc+1}{v} \rceil$. ■

4.2.3 Other Deterministic Algorithms

Recall that an *any-fit* (AF) algorithm never opens a new bin for an item which can be packed in one of the open bins. Different AF algorithms differ in the way they choose the bin in which the item will be placed. Since AF algorithms open a new bin only if none of the open bins are possible, any AF algorithm fulfills Property 4.2.5. In the presence of class-constraints, we distinguish between two levels of AF algorithms.

1. **Weak Any Fit** in which we are allowed to allocate a new *compartment* for an arriving item of color i, even if some possible bin holds items of color i.

2. Strong Any Fit in which we never allocate a new *compartment* for an arriving item of color i if some possible bin holds items of color i.

Theorem 4.2.10 *For any algorithm $\mathcal{A}$ that fulfills Property 4.2.5, and for any instance with bins having volume v and c compartments, $r_A \leq \frac{v}{c}$.*

Proof: We show that for any sequence σ, $N_A(\sigma) \leq \frac{v}{c} N_{opt}(\sigma) + 1$. By Property 4.2.5, when $\mathcal{A}$ terminates, each bin, except maybe for the last one, contains at least c items. Thus, $N_A(\sigma) \leq \frac{n}{c} + 1$. By Claim 4.1.1, $N_{opt}(\sigma) \geq \frac{n}{v}$. Thus, $N_A(\sigma) \leq \frac{v}{c} N_{opt}(\sigma) + 1$, which implies that $r_A \leq \frac{v}{c}$. ■

We show that this bound is tight for some AF algorithm. This distinguishes the class-constrained problem from the classic BP problem, for which all these algorithms have the constant competitive ratio 2 [37, 38].

Theorem 4.2.11 *There exists AF algorithm, $\mathcal{A}_w$, such that $r_w \geq \frac{v}{c}$.*

Proof: Consider the weak AF algorithm, $\mathcal{A}_w$, that places a new item in the last (highest indexed) bin into which it can fit. Given c, for any $j = 0, 1, 2, ...$, let σ_j be the sequence of the c requests $c + j, 1, 2, \ldots, c - 1$. For any N_0, consider the sequence σ of length $N_0 v$ consisting of $\sigma_0, \sigma_1, \ldots$ (the last subsequence may be partial). $\mathcal{A}_w$ packs σ as follows: the c items of σ_0 are placed in the first bin; a new bin is opened for the first item of σ_1, which cannot fit into the first bin. All other items of σ_1 can fit into both the first and the second bin, however, $\mathcal{A}_w$ places them all in the second bin. Similarly, for each j, the items of σ_j are placed in a new bin. That is, $\mathcal{A}_w$ uses $\lceil \frac{n}{c} \rceil \geq \frac{n}{c}$ bins. On the other hand, for long enough sequences, by Property 4.1.2, ($N_0 \geq \frac{M_\sigma - 1}{c-1}$), an optimal algorithm can pack σ into $\lceil \frac{n}{v} \rceil = N_0$ bins (see Figure 4.2). Thus, $N_w(\sigma) \geq \frac{v}{c} N_{opt}(\sigma)$, which implies that $r_w \geq \frac{v}{c}$. ■

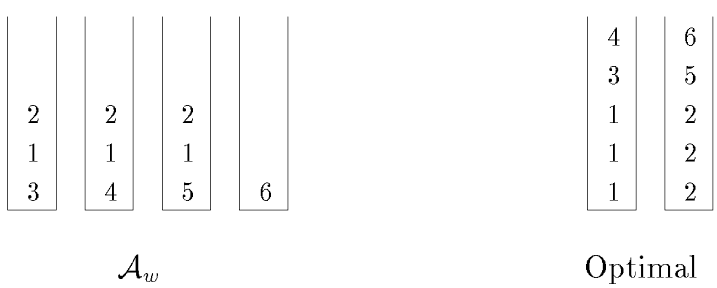

Figure 4.2: Lower bound for AF algorithms ($v = 5, c = 3$, and $N_0 = 2$)

Theorem 4.2.12 *First-fit is the only strong-AF algorithm.*

Proof: First note that $\mathcal{A}_F$ is an any-fit algorithm: we open a new bin for a new item only if no possible open bin exists for it. Assume towards contradiction that $\mathcal{A}_F$ is not a strong-AF algorithm. That is, assume that for some $a \in I_i$, the bin B_j, which is the first possible bin for a, does not have a compartment for I_i while some other bin, B_k, with $k > j$, has a compartment for I_i. Let b be the first item of I_i placed in B_k. Clearly, B_j was possible for b at the time it arrived. This contradicts to the way $\mathcal{A}_F$ packs b. Thus, $\mathcal{A}_F$ is a strong AF algorithm.

Let $\mathcal{A}_s$ be a strong AF algorithm. $\mathcal{A}_s$ never allocates a new compartment if an arriving item can be placed in some open compartment. Thus, for each color, at each moment, there is at most one open possible bin. Being the only possible bin, this bin is also the lowest indexed bin among the possible ones. Thus, the execution of $\mathcal{A}_s$ is identical to the one of $\mathcal{A}_F$. ∎

The Color-Sets Algorithm:

Consider a simple algorithm, $\mathcal{A}_{CS}$, which partitions the M_σ colors in σ into $\lceil \frac{M_\sigma}{c} \rceil$ color-sets and packs the items of each color-set greedily. Each color-set consists of c colors (excluding the last color-set that may contain fewer colors).

The partition into color-sets is determined online by the input sequence. That is, the first set, C_1, consists of the first c colors in σ, the second set, C_2, the next c colors in σ and so on. At any time there is one active bin for each color set. When an item a of color $i \in C_j$ arrives, it is placed in the active bin of C_j. If the active bin contains v items, we open a new bin for a and this is the new active bin of C_j. Since $|C_j| \leq c$, the color constraints can be ignored, and the resulting placement is legal.

Theorem 4.2.13 *For $\mathcal{A}_{CS}$, the color-sets algorithm, $r_{CS} < 2$.*

Proof: Assume that, when $\mathcal{A}_{CS}$ terminates there are ℓ active bins, including $x_1, x_2, \ldots, x_\ell$ items. Since we open a new active bin for some color-set only when the current active bin of that color set is full, we have

$$N_{CS}(\sigma) = \frac{n - (x_1 + x_2 +, \ldots, x_\ell)}{v} + \ell \leq \frac{n}{v} + \ell(1 - \frac{1}{v})$$

The above expression is maximized when ℓ is. Since $\ell \leq \lceil \frac{M_\sigma}{c} \rceil$, we have

$$N_{CS}(\sigma) \leq \frac{n}{v} + \lceil \frac{M_\sigma}{c} \rceil (1 - \frac{1}{v}) \leq 2N_{opt}(\sigma) - \frac{1}{v} \lceil \frac{M_\sigma}{c} \rceil.$$

Thus, $r_{CS} < 2$. ∎

The Next-fit Algorithm:

The well-known *next-fit* algorithm, denoted by $\mathcal{A}_N$, always packs a newly arriving item into the currently *active* bin. If the item cannot be placed in the active bin, then the currently active bin is closed (and never used again) and an empty bin is opened and becomes the new active bin. Since $\mathcal{A}_N$ never opens a new bin if the active bin is non-full and non-occupied, it fulfills Property 4.2.5. By Theorem 4.2.10, $r_N \leq \frac{v}{c}$. We show that this bound is tight.

Theorem 4.2.14 $r_N \geq \frac{v}{c}$.

Proof: Recall that $r_N = \limsup_{N_0 \to \infty}(\max_\sigma\{N_N(\sigma)/N_0 \mid N_{opt}(\sigma) = N_0\})$. Given v, c, for any $N_0 > 1$ consider an input sequence σ, of length $n = N_0 v$, consisting of repetitions of the subsequence $1, 2, \ldots, c+1$. That is, $\sigma = 1, 2, \ldots, c+1, 1, 2, \ldots, c+1, \ldots, 1, 2, \ldots$. Note that $\mathcal{A}_N$ must close each active bin after exactly c items. Thus, $N_N(\sigma) = \lceil \frac{n}{c} \rceil \geq \frac{n}{c}$. On the other hand, since $M_\sigma = c+1$ and $N_0 > 1$, an optimal algorithm packs σ in $\frac{n}{v} = N_0$ bins (see Figure 4.3). Thus, $N_N(\sigma) \geq \frac{v}{c} N_{opt}(\sigma)$, which implies that $r_N \geq \frac{v}{c}$. ■

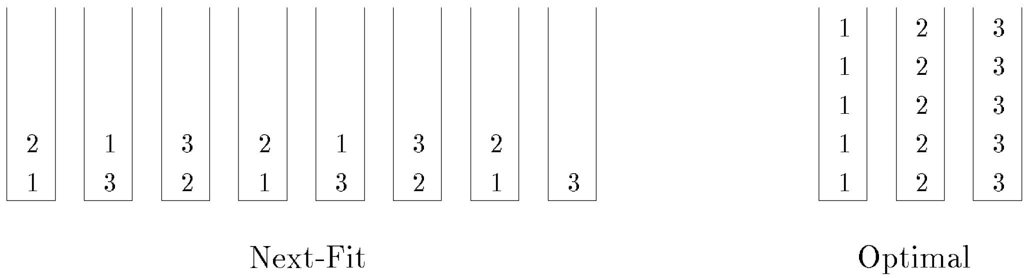

Figure 4.3: Lower bound for next-fit ($v = 5, c = 2$, and $N_0 = 3$)

4.3 Online Class-Constrained Packing of Temporary Items

In this section we consider a generalization of online class-constrained bin-packing to the case where items may leave the system after a while. We first consider the case in which a departure event is associated with specific item (of specific color, placed in specific bin). In Section 4.3.2 we discuss the case in which a departure event is associated with a *color*; thus, we may choose which item of that color to remove.

4.3.1 Departures of Specific Items

For the model where only arrival events are allowed, we showed (in Section 4.2.2) that first-fit is superior to other any-fit algorithms, and that its competitive-ratio is less than 2. When items can leave, this is no longer true. We show a lower bound of v/c on the competitive ratio of any deterministic algorithm. Then, we show that all any-fit algorithms achieve this bound. Note that, as in Section 4.2, we conclude that a-priori knowledge of n, M_σ cannot help.

Theorem 4.3.1 *Let $\mathcal{A}$ be an any-fit algorithm; then $r_A \geq \frac{v}{c}$.*

Proof: Given v, c, for any N_0 such that $k = \frac{1}{c}(N_0 - v + c)$ is an integer, we construct an input sequence, σ such that $N_A(\sigma) = kv$, while $N_{opt}(\sigma) = kc + v - c = N_0$. The construction of σ is done in k iterations. In the jth iteration, $1 \le j \le k$, we add and remove items of the colors $(j-1)c+1, \ldots, jc$. Specifically, the jth iteration in σ consists of:

1. Arrivals of v^2 items that fill the bins $(j-1)v+1, \ldots, jv$, such that each of these bins contain at least one item of each of the colors $(j-1)c+1, \ldots, jc$. This can be done, for example, by repeating v times a sub-sequence of arrivals of $(j-1)c+1, \ldots, jc$, followed by $v-c$ arrivals of additional items of these colors.

2. Departures of $(v-c)$ items from each of the bins $(j-1)v+1, \ldots, jv$, until each of these bins is left with one item of each of the colors $(j-1)c+1, \ldots, jc$.

Note that after each iteration, $1 \le j \le k$, each bin contains c items of c different colors. Thus, the arriving items in each iteration, that are of new colors, must be packed in new bins. It follows that after the jth, $1 \le j \le k$, iteration, $\mathcal{A}_A$ uses jv bins, each with c items.

Consider now an optimal algorithm, $\mathcal{A}_{opt}$, which knows the whole sequence, and in particular, the items that will leave during the second part of each iteration. $\mathcal{A}_{opt}$ will pack the vc items that will not be removed in the first available c bins, and the items that will leave, in the next $v-c$ bins. Thus, after the first part of iteration j, $1 \le j \le k$, $\mathcal{A}_{opt}$ uses $(j-1)c+v$ full bins, and during the second part of the jth iteration it retreats to jc full bins. The maximal number of active bin during the execution of $\mathcal{A}_{opt}$ is $(k-1)c+v$, in use after the first part of the last iteration.

We get that $r_A(\sigma) = \frac{kv}{kc+v-c}$. Recall that k was selected such that $N_0 = kc + v - c$. Thus, $r_A(\sigma) = \frac{\frac{v}{c}(N_0-v+c)}{N_0} = \frac{v}{c}(1 - \frac{v-c}{N_0})$, and $\limsup_{N_0 \to \infty}(\max_\sigma\{N_A(\sigma)/N_0 \mid N_{opt}(\sigma) = N_0\}) = \frac{v}{c}$. ∎

We now show a lower bound of v/c on the competitive ratio of *any* deterministic algorithm. Clearly, this bound is valid also for any AF algorithm; however, the sequences we use in the proof below are much longer than those used in the proof of Theorem 4.3.1.

Theorem 4.3.2 *For any deterministic algorithm $\mathcal{A}_d$, $r_d \ge \frac{v}{c}$.*

Proof: Given v, c, for any N_0 such that $k = \frac{1}{c}(N_0 - v + 1)$ is an integer, we construct an input sequence, σ such that $N_d(\sigma) = kv$, while $N_{opt}(\sigma) = kc + v - 1 = N_0$. The construction of σ is done in kc iterations. In the jth iteration, $1 \le j \le kc$, we add and remove items of color j. Specifically, the jth iteration in σ consists of:

1. Arrivals of $v(v-1)+1$ items of color j. These items must be placed in at least v different bins.

2. Departures of $v(v-2)+1$ items of color j, selected such that each of the v remaining items of color j is placed in a different bin.

Note that after the iteration kc, there are vkc items in the bins. Since the items of each color are placed in different bins, each bin holds at most c items. Thus, $\mathcal{A}_d$ uses at least kv bins.

Consider now an optimal algorithm, $\mathcal{A}_{opt}$, which knows the whole sequence, and in particular, the items that will leave during the second step in each iteration. For each color, $\mathcal{A}_{opt}$ will pack the v items that will not be removed in the first available bin, and the $v(v-2)+1$ items that are about to leave, in the next $v-1$ bins. Thus, after the first step of iteration j, $1 \leq j \leq ck$, $\mathcal{A}_{opt}$ uses $(j-1)+v$ bins, and during the second step of the jth iteration it retreats to j full bins. Hence, the maximal number of bins used by $\mathcal{A}_{opt}$ is $kc+v-1$.

We get that $r_d(\sigma) = \frac{kv}{kc+v-1}$. Recall that k was selected such that $N_0 = kc+v-1$. Thus, $r_d(\sigma) = \frac{\frac{v}{c}(N_0-v+1)}{N_0} = \frac{v}{c}(1-\frac{v+1}{N_0})$, and $\limsup_{N_0\to\infty}(\max_\sigma\{N_d(\sigma)/N_0 \mid N_{opt}(\sigma)=N_0\}) = \frac{v}{c}$. ∎

Theorem 4.3.3 *If $\mathcal{A}$ is an any-fit algorithm, then for any instance with bins having volume v and c compartments, $r_A \leq \frac{v}{c}$.*

Proof: Let σ be an input sequence for $\mathcal{A}$. Let N_A be the number of bins used by $\mathcal{A}$ when packing σ. When analyzing the competitive ratio of $\mathcal{A}$, we can assume w.l.o.g. that σ ends when bin number N_A is opened. Indeed, after this point, the number of bins used by an optimal algorithm can only increase, and we are interested in bounding the largest $\frac{N_A(\sigma)}{N_{opt}(\sigma)}$ ratio. Since $\mathcal{A}$ is an any-fit algorithm, when N_A is opened, all the other bins must be full or occupied, and therefore each open bin contains at least c items. Let n be the number of packed items at that time. It follows that $N_A(\sigma) \leq \frac{n}{c}+1$. By Claim 4.1.1, $N_{opt}(\sigma) \geq \frac{n}{v}$. Thus, $N_A(\sigma) \leq \frac{v}{c}N_{opt}(\sigma)+1$, and $r_A \leq \frac{v}{c}$. ∎

4.3.2 Departures of Items of Specific Colors

We show that any-fit algorithms achieve a poor ratio, even if departure events are associated only with a color, and the algorithm is free to select the item that will be removed from that color. Our result holds for *any* removal policy.

Theorem 4.3.4 *If $\mathcal{A}$ is an any-fit algorithm, then $r_A \geq \min(\frac{v}{c}, c-1)$.*

Proof: Given v, c, let $x = v-c+1$, and let $z = \max(\lceil\frac{xc}{v}\rceil, \lceil\frac{x}{c-1}\rceil)$. For any N_0 such that $k = \frac{1}{z}(N_0+z-x)$ is an integer, we construct a sequence, σ such that $N_A(\sigma) = kx$, while $N_{opt}(\sigma) = kz-z+x = N_0$.

The construction of σ is done in k iterations. In the jth iteration $(1 \leq j \leq k)$ we handle items in the v colors $(j-1)v+1, \ldots, jv$. Specifically, the jth iteration in σ consists of:

1. Arrivals of xv items that fill the x bins $(j-1)x+1, \ldots, jx$ in the following way: For $i=1$ to x: σ includes a sub-sequence of arrivals of $(c-1)$ items in colors $(j-1)v+1, \ldots, (j-1)v+c-1$, followed by $v-c+1$ arrivals of items of color $(j-1)v+c+i-1$. Since $\mathcal{A}$ is any any-fit algorithm, there is at most one possible open bin for each item as it arrives, thus, the bins $(j-1)x+1, \ldots, jx$ are filled sequentially one after the other.

2. For $i=1$ to x: remove $v-c$ items of color $(j-1)v+c+i-1$. Now, regardless of the removal policy of the algorithm, we end up with each of the x bins $(j-1)x+1, \ldots, jx$ containing exactly c items, one in each of the colors $(j-1)v+1, \ldots, (j-1)v+c-1$, and one item whose color is in $(j-1)v+c, \ldots, jv$.

Note that after each iteration, $1 \leq j \leq k$, each bin contains c items of c different colors. Thus, the arriving items in each iteration, which are of new colors, must be packed in new bins. It follows that after the jth $(1 \leq j \leq k)$ iteration, $\mathcal{A}$ uses jx bins, each with c items.

Claim 4.3.5 *An optimal placement of σ uses $N_0 = kz - z + x$ bins.*

Proof: Consider an optimal algorithm, $\mathcal{A}_{opt}$, which knows the entire sequence, and in particular, the items that are will leave during the second step of each iteration. For each iteration, $\mathcal{A}_{opt}$ can pack the items that are not removed in the first available bins. We show that these permanent items can be packed in z bins. Note that this set of items consists of xc items of $x+c-1$ colors. We distinguish between two types of colors: (i) *repeated*, from each of which there are x items (in the jth iteration, these are the $(c-1)$ colors $(j-1)v+1, \ldots, (j-1)v+c-1$), and (ii) *single-items*, from each of which there is only one item (in the jth iteration, these are the x colors $(j-1)v+c, \ldots, jv$).

Assume that $\lceil \frac{xc}{v} \rceil \geq \lceil \frac{x}{c-1} \rceil$, then $\mathcal{A}_{opt}$ can pack the permanent items in $\lceil \frac{xc}{v} \rceil$ bins as follows: at first, each bin contains $c-1$ items of single-items colors, and $x = v-c+1$ items of some repeated color. After all the singles are packed, we fill the remaining bins greedily with the remaining items of the repeated colors. Since all the bins are filled to capacity v, $\mathcal{A}_{opt}$ uses $\lceil \frac{xc}{v} \rceil$ bins.

Assume that $\lceil \frac{xc}{v} \rceil < \lceil \frac{x}{c-1} \rceil$, then we can pack the permanent items in $\lceil \frac{x}{c-1} \rceil$ bins as follows: at first, we pack in each bin $c-1$ single items, and $x = v-c+1$ items of some repeated color. Once all the items of the repeated colors are packed, we use the remaining bins greedily to pack c single items in each. Since each bin contains at least $c-1$ single items, and there are x such items, all the single items are packed. Also, since $\lceil \frac{xc}{v} \rceil < \lceil \frac{x}{c-1} \rceil$, there is enough capacity for the other items.

Thus, after the first step of iteration j, $1 \leq j \leq k$, $\mathcal{A}_{opt}$ uses $(j-1)z+x$ bins, and during the second step of the jth iteration it retreats to jz full bins. The maximal number of bins is $(k-1)z+x = N_0$, used during the last iteration. ∎

We get that $r_A(\sigma) \geq \frac{kx}{kz-z+x}$, and $r_A = \limsup_{N_0 \to \infty} (\max_\sigma \{N_{CS}(\sigma)/N_0 \mid N_{opt}(\sigma) = N_0\}) \geq \frac{x}{z} \geq \min(\frac{v}{c}, c-1)$. ∎

The Color-Sets Algorithm:

We now show that any algorithm which is based on color-sets packing achieves the worst possible ratio, which converges to v, even if departure events are associated only with a color. Our result holds for any removal policy and any algorithm which packs items by color-sets.

Theorem 4.3.6 *Let $\mathcal{A}_{CS}$ be a color-sets algorithm with any removal policy; then, $r_{CS} \geq v$.*

Proof: Recall that $\mathcal{A}_{CS}$ partitions the M_σ colors of σ into $\lceil \frac{M_\sigma}{c} \rceil$ color-sets, where the jth set, C_j, contains the jth set of c colors in σ. The items of each color-set are packed greedily, since the color-constraint can be ignored.

Given v, c, for any N_0, let $k = N_0 - v + 1$. We construct a sequence, σ such that $N_A(\sigma) = kv$, while $N_{opt}(\sigma) = k + v - 1 = N_0$. W.l.o.g., the colors are numbered by the order of their (first) appearance in σ; thus, for each $1 \leq j \leq k$, $C_j = \{(j-1)c + 1, \ldots, jc\}$. The construction of σ is done in k iterations. In the jth iteration, $1 \leq j \leq k$, we add and remove items whose colors are in C_j. Specifically, the jth iteration consists of two steps:

1. Arrivals of v^2 items whose colors are in C_j. This sequence consists of repeating v times a sub-sequence of arrivals of a single item of color cj followed by $v-1$ arrivals of items of colors $(j-1)c + 1, \ldots, jc - 1$. This sequence must contains at least one item in each of the colors in C_j (so $\mathcal{A}_{CS}$ can define C_j).

2. Departures of the $v(v-1)$ items in the colors $(j-1)c + 1, \ldots, jc - 1$.

Note that in the first step of iteration j, $\mathcal{A}_{CS}$ fills the bins $(j-1)v + 1, \ldots, jv$ and after the second step of iteration j, regardless of the removal policy of $\mathcal{A}_{CS}$, we end up with each of these bins containing a single item (of color cj). However, these bins cannot be used by $\mathcal{A}_{CS}$ in the next iterations, since the arriving items belong to different color-sets. Thus, in each iteration, the arriving items must be packed in new bins. It follows that after the kth iteration, $\mathcal{A}_{CS}$ has kv open bins, each contains a single item.

Consider now an optimal algorithm, $\mathcal{A}_{opt}$, which knows the entire sequence, and in particular, the items that will depart in the second step of each iteration. In each iteration, $\mathcal{A}_{opt}$ packs the v items that are not removed in the first available bin, and the items that will be removed in the next $v-1$ bins. Thus, after the first step of iteration j, $1 \leq j \leq k$, $\mathcal{A}_{opt}$ has $j - 1 + v$ full bins, and during the second step of the jth iteration it retreats to j full bins. The maximal number of bins is $k + v - 1$, used during the last iteration.

We get that $r_{CS}(\sigma) = \frac{kv}{k+v-1}$. Since $N_0 = k + v - 1$, $r_{CS}(\sigma) = \frac{N_0 v - v^2 + v}{N_0} = v(1 - \frac{v^2 - v}{N_0})$, and $\limsup_{N_0 \to \infty}(\max_\sigma\{N_{CS}(\sigma)/N_0 \mid N_{opt}(\sigma) = N_0\}) = v$. ∎

4.4 The Online Class-Constrained Multiple Knapsack Problem

Recall that in the CCMK problem we have N identical knapsacks, of volume v and c compartments, and our goal is to maximize the number of packed items from the input sequence σ. For a placement algorithm $\mathcal{A}$, and an input sequence σ, let $n_A(\sigma, N)$ denote the number of items in σ packed by $\mathcal{A}$. Let $n_{opt}(\sigma, N)$ denote the maximal number of items in σ that can be packed in N knapsacks. The *competitive ratio* of $\mathcal{A}$ is

$$r_A = \min_{\sigma, N} \frac{n_A(\sigma, N)}{n_{opt}(\sigma, N)}.$$

We show a lower bound of c/v for deterministic algorithms. This bound is achieved by any greedy algorithm, which never rejects an item that can be packed. That is, an arriving item is rejected only if it cannot be packed in any of the knapsacks. We show that the competitive ratio of any greedy algorithm matches our lower bound, regardless of the way we pack the items. For the temporary CCMK problem, note that the number of packed items is not bounded by Nv. We show that no deterministic algorithm is competitive.

Theorem 4.4.1 *For any $v \geq c$, and any deterministic algorithm, $\mathcal{A}_d$, for CCMK, $r_d \leq \frac{c}{v}$.*

Proof: Consider the following sequence that is constructed online by the adversary, according to the behavior of $\mathcal{A}_d$:

1. For all $i = 1, \ldots, Nc$, repeat items of color i until one such item is packed, or until Nv items of color i are rejected. In the later case, the sequence ends.

2. Nv items of color $Nc + 1$.

Consider first the case where no color is rejected Nv times. An optimal algorithm can pack the Nv items of color $Nc + 1$. On the other hand, each of the knapsacks filled by $\mathcal{A}_d$ contains exactly c items of c different colors, thus, only mc items can be packed by $\mathcal{A}_d$. If some color, i, is persistently rejected, then an optimal algorithm can pack the Nv rejected items of color i, while $\mathcal{A}_d$ packs only $i - 1$ items of the colors $1, \ldots, i - 1$. In both cases $r_d \leq \frac{Nc}{Nv} = \frac{c}{v}$. ∎

A greedy algorithm never rejects an item that can be packed. That is, an arriving item is rejected only if there is no possible knapsack for it. We show that the competitive ratio of any greedy algorithm matches our lower bound, independent of the way we pack the items.

Theorem 4.4.2 *For any greedy algorithm $\mathcal{A}_G$, the competitive ratio of $\mathcal{A}_G$ is $r_G \geq \frac{c}{v}$.*

Proof: For any $N > 0$ and sequence σ, consider the knapsacks when σ terminates. If some knapsack contains less than c items, then, since $\mathcal{A}_G$ is greedy, no item was rejected and $r_G(\sigma, N) = 1$. Otherwise, there are at least c items in each of knapsack, and $n_G(\sigma, N) \geq cN$. Since $n_{opt}(\sigma, N) \leq vN$, the ratio is immediate. ■

Next we show that when temporary items are allowed, then no algorithm is competitive.

Theorem 4.4.3 *For any $v \geq c$, any deterministic algorithm, $\mathcal{A}_d$, for CCMK^t, and any $r > 0$, the competitive ratio of $\mathcal{A}_d$ is $r_d \geq r$.*

Proof: Consider a sequence, σ, that first blocks all the N knapsacks (as in the proof of Theorem 4.4.1). After all the knapsacks are full or occupied, σ includes a sequence, of arbitrary length, r, of pairs of arrival-departure of some item, (of a new color). Clearly, these repeated arrivals can not be accepted by the algorithms, while an optimal algorithm can gain r packed items. Any ratio can be achieved by picking a large enough value of r. ■

Chapter 5

Scheduling with Machine Allotment and Parallelism Constraints

In this Chapter we study the SPAC problem. Recall, that given a set of n jobs and m uniform machines, we need to assign the jobs to the machines subject to parallelism and machine allotment constraints, such that the overall completion time of the schedule (or *makespan*) is minimized.

We present a $(1 + \alpha)$-approximation algorithm for this problem, where $\alpha \in (0, 1]$ depends on the minimal number of machine allotments and the minimal parallelism allowed for any job. Also, we show that when the maximal number of machines that can share the execution of a job is some fixed constant, our problem has a PTAS; for other special cases we give optimal polynomial time algorithms. Finally, through the relation of our problem to the classic *preemptive* scheduling problem on multiple machines, we show that the power of preemption is dramatically reduced when we limit the way the preemption may occur along the schedule.

5.1 Hardness of the SAC Problem

In this section we derive hardness results for the SAC problem. The case where $\forall\ j,\ a_j = 1$ is known to be strongly NP-hard. When $\forall\ j,\ a_j > 1$, some preemptions are allowed for each of the jobs, and we may expect that the problem becomes easy to solve (as the classic makespan problem with preemptions). We show that the SAC problem is strongly NP-hard even for instances with no parallelism constraints, 'weak' allotment constraints, and with identical machines/jobs. In other words, unless $P = NP$, it cannot admit an FPTAS already for these instances.

We consider three classes of instances of the SAC problem. For each, we explain why it may seem to be 'easy-to-solve' and follow this with a proof of hardness. The three classes are:

1. Identical machines, where each job can be allotted to at least c machines, for any $c > 1$.

2. Identical machines, where the *total* number of allotments is unbounded.

3. Identical jobs, where the total number of allotments is at least $\frac{3}{2}m$.

We derive our hardness results using reductions from *3-partition*, which is strongly NP-hard [20]. An instance of 3-partition is defined as follows.
Input: a finite set A of $3q$ elements, a bound $B \in Z^+$, and a size $s(x)$ for each $x \in A$, such that each $s(x)$ satisfies $B/4 < s(x) < B/2$ and such that $\sum_{x \in A} s(x) = qB$.
Output: Is there a partition of A into q disjoint sets, $S_1, S_2, \ldots, S_q$, such that, for $1 \leq i \leq q, \sum_{x \in S_i} s(x) = B$? (Note that the above constraints on the element sizes imply that every such S_i must contain exactly three elements from A).

5.1.1 Identical Machines and Any Number of Allotments per Job

We first consider instances with identical machines and any number of allotments per job. We show that there is no constant c, such that if each job could be allotted to c identical machines, then SAC admits an FPTAS.

Theorem 5.1.1 *The SAC problem is strongly NP-hard, even if $\forall 1 \leq j \leq n, a_j \geq c$, for any given $c > 1$ and the machines are identical.*

Proof: Let $c > 1$ be an integer. Given an instance of 3-partition, we construct an input, I, for the makespan problem with identical machines and $\forall 1 \leq j \leq n, a_j \geq c$, such that $w_{OPT}(I) = 1$ if and only if A has a 3-partition.

The input for the makespan problem consists of $m = (c-1)3q+q$ machines with the same rates: $u_1 = u_2 = \ldots = u_{(c-1)3q+q} = B$; and $n = 3q$ jobs with $t_j = (c-1)B + s(x_j)$, $a_j = c$, $\forall 1 \leq j \leq 3q$. Thus, we have that $\sum_j t_j = \sum_i u_i = (c-1)3qB + qB$. Since $\sum_j t_j = \sum_i u_i$, $w_{OPT}(I) \geq 1$.

Assume that A has a 3-partition to the sets $S_1, S_2, \ldots, S_q$. This induces the following schedule of I, whose makespan equals to 1. $\forall 1 \leq j \leq 3q$, J_j is processed for a single time unit on arbitrary vacant $c-1$ machines out of $M_1, \ldots, M_{(c-1)3q}$. That is, $(c-1)B$ processing units are allocated to $c-1$ segments of J_j. In addition, if $x_j \in S_k$ (k is the index of the triple to which x_j belongs in the partition), then the last (c-th) segment of J_j is allocated $s(x_j)$ processing units on $M_{(c-1)3q+k}$.

Thus, to each job we allocate exactly $t_j = (c-1)B + s(x_j)$ processing units on c different machines. Since $\forall 1 \leq k \leq q$, $\sum_{j \in S_k} s(x_j) = B$, each of the last q machines allocates exactly $u_i = B$ processing units, and the makespan equals to 1.

Assume that a schedule whose makespan equals 1 exists for I. We show that A has a 3-partition. For each machine, M_i, let n_i denote the number of jobs scheduled on M_i, and let $J_{i_1}, J_{i_2}, \ldots, J_{i_{n_i}}$ be the list of these jobs, such that, w.l.o.g, $i_1 < i_2 < \ldots < i_{n_i}$. The following graph, $G = (V, E)$, is induced by the schedule:

V : there is a vertex, J_j, for each job, $1 \leq j \leq 3q$.

E : each machine M_i contributes to E the edges of the path $J_{i_1}, J_{i_2}, \ldots, J_{i_{n_i}}$

Note that each machine contributes exactly $n_i - 1$ edges to E. Therefore, the graph G has $\sum_{i=1}^{m} n_i - m$ edges. Recall that J_j can be executed by at most a_j machines. In other words, J_j can appear on at most a_j paths, meaning that $\sum_i n_i \leq \sum_j a_j$. Therefore, the number of edges in G is at most $\sum_j a_j - m = 3qc - (c-1)3q - q = 2q$. Having $n = 3q$ vertices and at most $2q$ edges, G consists of at least q connected components.

Assume that G has ℓ connected components: $D_1, D_2, \ldots, D_\ell$. Consider a component $D_k = (V_{D_k}, E_{D_k})$. V_{D_k} is a set of jobs. D_k is connected, therefore, for each machine, M_i, the path contributed by M_i is either completely contained or not contained in E_{D_k}. Thus, E_{D_k} determines the subset of machines which process the jobs in V_{D_k}.

Claim 5.1.2 *For each component D_k, $\forall 1 \leq k \leq \ell$, there exists an integer $r_k > 0$ such that $\sum_{j \in V_{D_k}} t_j = ((c-1)|V_{D_k}| + r_k)B$.*

Proof: Recall that $\sum_j t_j = \sum_i u_i$; thus, in any schedule with makespan $= 1$, no machine is idle. It means that *all* the processing units of E_{D_k} are allocated to V_{D_k}. The rate of each machine is B. Thus, the total number of processing units allocated to the jobs in V_{D_k} is a multiple of B. Since $\forall j, t_j > (c-1)B$, there exists an integer $r_k > 0$ such that $\sum_{j \in V_{D_k}} t_j = ((c-1)|V_{D_k}| + r_k)B$. ∎

We now prove that $\ell = q$, that is, G consists of exactly q connected components.

Claim 5.1.3 *The graph G has exactly q connected components.*

Proof: From Claim 5.1.2 there exist positive integers $r_1, \ldots, r_\ell$ such that $((c-1)|V_{D_1}| + r_1)B + \ldots + ((c-1)|V_{D_\ell}| + r_\ell)B = ((c-1)3q + q)B$. There are $3q$ jobs, therefore, $\sum_{k=1}^{\ell} |V_{D_k}| = 3q$, and we get that $\sum_{i=1}^{\ell} r_k = q$. Since the r_k's are positive integers and $\ell \geq q$ we must have $\ell = q$ and $r_1 = \cdots = r_q = 1$. ∎

Now, given that G consists of $D_1 \cup D_2 \cup \cdots \cup D_q$, define the following partition: for all $1 \leq j \leq 3q$, $j \in S_k$ if and only if $j \in V_{D_k}$. By Claim 5.1.3, $r_k = 1$, $\forall 1 \leq k \leq q$, thus, $\sum_{j \in V_{D_k}} t_j = ((c-1)|V_{D_k}| + 1)B$, meaning that $\sum_{j \in S_k} s(x_j) = B$ for all $1 \leq k \leq q$. ∎

5.1.2 Identical Machines and any Total Number of Allotments

For the preemptive scheduling problem on identical machines, an optimal schedule can be obtained using at most $m-1$ preemptions (For example, the greedy algorithm 'splits' only the last job on each machine). It means that the total number of allotments of jobs to machines is at most

$n + m - 1$. In our second hardness result we show that, for any given c, even if the machines are identical, and the *total* number of allotments may be larger than $c(m+n)$, the makespan problem is strongly NP-hard.

Theorem 5.1.4 *The SAC problem is strongly NP-hard, even if* $\sum_{j=1}^{n} \min(m, a_j) \geq c(n+m)$, *and the machines are identical* [1]. *This holds for any* $c \geq 1$.

Proof: For a given c, we show a reduction from the 3-*partition problem*, with $|A| = 3q \geq 15c$. We construct an input, I, for the makespan problem with identical machines and $\sum_j \min(m, a_j) \geq c(n+m)$, such that $w_{OPT}(I) = 1$ if and only if A has a 3-partition.

The input I consists of $m = q$ machines with the same rates: $u_1 = u_2 = \ldots = u_m = q^2 B + q$; and $n = 4q$ jobs:

- $3q$ jobs with $t_j = q^2 s(x_j)$, $a_j = 1$, $\forall 1 \leq j \leq 3q$. We call these jobs *integral.*
- q jobs with $t_j = q$, $a_j = q$, $\forall 3q < j \leq 4q$. we call these jobs *additional.*

Note that $\sum_i u_i = q^3 B + q^2 = \sum_j t_j$. In addition, for this instance, $\sum_{j=1}^{n} a_j = q^2 + 3q$; $m + n = q + 4q$. Thus, $\sum_j \min(m, a_j) = q^2 + 3q > c(5q)$, for any $q \geq 5c$. Since $\sum_j t_j = \sum_i u_i$, $w_{OPT}(I) \geq 1$.

We show that A has a 3-partition if and only if $w_{OPT}(I) = 1$. Assume that A has a 3-partition to the sets $S_1, S_2, \ldots, S_q$. The following is a schedule whose makespan equals 1.

1. One processing unit of each machine $M_i, 1 \leq i \leq m$ is allocated to each of the additional jobs (i.e., overall M_i allocates q processing units). Thus, the execution of each additional job is shared among $a_j = q$ machines.
2. The remaining $q^2 B$ processing units of M_i are allocated to the integral jobs $\{J_{i_i}, J_{i_2}, J_{i_3}\}$ such that $S_i = \{x_{i_i}, x_{i_2}, x_{i_3}\}$.

Since $\forall i, \sum_{j \in S_i} s(x_j) = B$, the total number of processing units allocated by M_i in the second step is $q^2 B$. Therefore, the above is a schedule whose makespan equals 1.

Assume that there exists a schedule whose makespan equals to 1 for I.

Claim 5.1.5 *In any such schedule, exactly q processing units of each machine are allocated to* additional *jobs.*

[1] We take the minimum between m and a_j since we gain nothing if a job can be allotted to more than m machines. This makes our result stronger.

Proof: Let n_i denote the number of processing units of M_i that are allocated to the additional jobs. The remaining $u_i - n_i$ processing units are allocated to integral jobs. The integral jobs cannot split, thus, in any such schedule, $u_i - n_i$ is a multiple of q^2. Since $u_i = q^2B + q$ and the total processing time of the additional jobs is q^2 we get that $n_i = q$ for all i. ∎

Now, given that q processing units of each machine are allocated to additional jobs, we conclude that the remaining q^2B processing units of each machine are allocated to integral jobs, and a 3-partition of A is induced by the schedule. ∎

5.1.3 Identical Jobs

When $\sum_j a_j = m$ and the jobs are identical, the SAC problem is strongly NP-hard: this can be shown by a simple reduction from 3-partition (as mentioned in [24]). We show that SAC remains strongly NP-hard, even if the jobs are identical, and the set of possible partitions is larger, more precisely, $\sum_j a_j \geq \alpha m$, for $\alpha = \frac{3}{2}$.

Theorem 5.1.6 *The SAC problem is strongly NP-hard even if the jobs are identical and $\sum_j a_j \geq \frac{3}{2}m$.*

Proof: Given an instance for 3-partition, we construct an input, I, for the makespan problem with $\sum_j a_j \geq \frac{3}{2}m$, such that $w_{OPT}(I) = 1$ if and only if A has a 3-partition. In this reduction we adapt some ideas from the hardness proof given in [24].

The input I consists of $m = 4q$ machines with the following rates: for the first $3q$ machines, $u_i = K - s(x_i)$, $1 \leq i \leq 3q$, where $K > 3qB$ is a large constant; for the other q machines $u_i = 3K + B$, $3q < i \leq 4q$. There are $n = 3q$ identical jobs with $t_j = 2K, a_j = 2$, $\forall 1 \leq j \leq 3q$. For this instance, $\sum_j t_j = \sum_i u_i = 6qK$, and as needed, $\sum_j a_j = 6q = \frac{3}{2}m$. Since $\sum_j t_j = \sum_i u_i$, $w_{OPT}(I) \geq 1$.

We show that A has a 3-partition if and only if $w_{OPT}(I) = 1$. Assume that A has a 3-partition to the sets $S_1, S_2, \ldots, S_q$. Let $S_k = \{x_{k_1}, x_{k_2}, x_{k_3}\}$, $\forall 1 \leq k \leq q$. The following is a schedule whose makespan equals 1: $\forall 1 \leq k \leq q$, the four machines $M_{k_1}, M_{k_2}, M_{k_3}, M_{3q+k}$ process the three jobs $J_{k_1}, J_{k_2}, J_{k_3}$. Specifically, M_{k_i} allocates $K - s(x_{k_i})$ processing units to J_{k_i}, $1 \leq i \leq 3$, and M_{3q+k} allocates $K + s(x_{k_i})$ to J_{k_i} $1 \leq i \leq 3$. Since $s(x_{k_1}) + s(x_{k_2}) + s(x_{k_3}) = B$, and $u_{3q+k} = 3K + B$, the completion time equals to 1. Note also that the allotment constraints are preserved: for any $1 \leq j \leq 3q$, if $x_j \in S_k$, then J_j is processed by the two machines M_j and M_{3q+k}.

Now, given a schedule whose makespan equals to 1, we find a 3-partition of A. Denote by *slow* the set of the first $3q$ machines $\{M_1, \ldots, M_{3q}\}$. Note that for any pair, M_{i_1}, M_{i_2}, of slow machines, the total number of processing units provided by M_{i_1} and M_{i_2} is less than $2K$. Since $t_j = 2K$, if some J_j is scheduled only on these two machines it cannot be completed on time. Since $a_j = 2$ for $1 \leq j \leq n$, this implies that each job, J_j, is processed by at most *one* slow machine. In addition, since there are $3q = n$ slow machines, and no idle time is possible (since $\sum_i u_i = \sum_j t_j$), each slow machine processes exactly one job. Assume w.l.o.g that $\forall 1 \leq i \leq 3q$, M_i is the slow machine allocated to J_i (for the whole duration of the schedule). For each such job, since $a_j = 2$, the remaining $K + s(x_i)$ processing units are allocated by a single machine. Consider a machine $M_i, i > 3q$. Since K was selected such that $K >> s(x)$, for any $x \in A$, it follows that M_i processes exactly three jobs (since $u_i = 3K + B$, and the remainders of any four jobs require more than u_i processing units). Let J_{k_1}, J_{k_2} and J_{k_3} be the three jobs scheduled on M_{3q+k}. We get that $3K + B = 3K + (s(x_{k_1}) + s(x_{k_2}) + s(x_{k_3}))$, meaning that x_{k_1}, x_{k_2} and x_{k_3} form a triple for the partition. Since all the jobs are scheduled, the whole schedule of the jobs on the last q machines, induces a valid 3-partition of A. ∎

5.2 Approximation Algorithm for the SPAC Problem

5.2.1 Uniform Machines

In this section we present an approximation algorithm for the SPAC problem on uniform machines. Denote by $w_{OPT}(I)$ the length of an optimal schedule of an instance I. First, we show that an optimal schedule can be obtained by relaxing the parallelism constraint[2] of each job by one. Specifically, if for all j, we allow to run J_j in parallel on $\rho_j + 1$ (instead of ρ_j) machines, we can obtain a schedule of length $w_{OPT}(I)$. Then, we transform the above infeasible-optimal schedule into a feasible one, whose makespan is $w \leq \max_j(1 + \frac{1}{\rho_j})w_{OPT}(I)$.

Relaxing the Parallelism Constraint

Theorem 5.2.1 *Given an instance I of the SPAC problem, denote by I^+ the instance in which the parallelism parameter of each job is increased by one (i.e., $\rho_j^+ = \rho_j + 1$, $\forall j$); then we can find in $O(\max(m \log m, n \log n))$ steps a schedule for I^+, whose makespan is at most the minimal possible makespan for I.*

Given the instance I^+ derived from I, we present a polynomial time algorithm which finds a legal schedule of I^+. The algorithm, denoted by $\mathcal{A}_r$, proceeds by scheduling J_j on at most $\rho_j^+ = \rho_j + 1$ machines, $\forall 1 \leq j \leq n$. The length of the schedule generated for I^+ by $\mathcal{A}_r$ is at most the length of an optimal schedule of I.

We renumber the machines in non-increasing order by their speeds, i.e., $u_1 \geq u_2 \geq, \ldots, \geq u_m$, and the jobs in a non-increasing order by their *processing ratios*, i.e., $\frac{t_1}{\rho_1} \geq \frac{t_2}{\rho_2} \geq, \ldots, \geq \frac{t_n}{\rho_n}$. For each $1 \leq \ell \leq n$, let $\hat{\rho}_\ell = \sum_{j=1}^{\ell} \rho_j$. Let

$$w = \max\{\frac{\sum_{j=1}^{n} t_j}{\sum_{i=1}^{m} u_i}, \max_{\{\ell | \hat{\rho}_\ell \leq m\}} \frac{\sum_{j=1}^{\ell} t_j}{\sum_{i=1}^{\hat{\rho}_\ell} u_i}\}. \tag{5.1}$$

[2]Note that when $\rho_j = a_j$, we relax at the same time the parallelism and the allotment constraint of J_j.

To prove the theorem, we show that the makespan of any schedule of I is at least w, and that $\mathcal{A}_r$ generates for I^+ a schedule of length w. Note that, as illustrated in Example 1.2.1, w is not tight; that is, for some instances $w_{OPT}(I) > w$.

We first show that w is a lower bound on the length of any legal schedule of I.

Lemma 5.2.2 *For any instance, I, $w_{OPT}(I) \geq w$.*

Proof: Consider a schedule of length w. When no machine is idle at any time during the schedule, the total processing potential of the machines is $w \sum_i u_i$. Thus, $w_{OPT}(I)$ is at least the left term in the right-hand side of (5.1). The execution time of J_1 is minimized when it runs in parallel on the ρ_1 fastest machines for the whole duration of the schedule. Similarly, we cannot do better than scheduling the first ℓ jobs on the $\hat{\rho}_\ell$ fastest machines for the whole duration of the schedule. Thus, the right term in (5.1) is a lower bound for $w_{OPT}(I)$. ∎

We now turn to describe and analyze the algorithm $\mathcal{A}_r$. $\mathcal{A}_r$ adapts some ideas from the approximation algorithm for the class constrained multiple knapsack problem, presented in Section 2.3. In each stage we represent by Q_i the *potential* of the machine M_i, that is, the number of processing units that M_i can still allocate. Initially, $Q_i = wu_i$. $\mathcal{A}_r$ maintains a list, L, of the machines, sorted by their potential in non-decreasing order. That is, $Q_{L[1]} \leq Q_{L[2]} \leq \ldots$ (where $L[k]$ denotes the machine at position k in L). The list L is updated along the execution of the algorithm. Specifically, when M_i allocates processing units to some job, its potential decreases, and its position in L may be updated. Once M_i has allocated $u_i w$ processing units, it is removed from L. Given a pair of machines M_{i_1}, M_{i_2}, we say that M_{i_1} is weaker (stronger) than M_{i_2}, if $Q_{i_1} \leq Q_{i_2}$ ($Q_{i_1} \geq Q_{i_2}$).

The jobs, sorted in non-increasing order by their processing ratios, are scheduled one after the other. The job J_j is scheduled on the first (i.e., weakest) consecutive sequence of $\rho_j + 1$ or less machines, $L[k_1], \ldots, L[k_2]$, whose total potential is at least t_j. All the potential of $L[k_1], \ldots, L[k_2 - 1]$ and some of the potential of $L[k_2]$ is allocated to J_j, such that the total number of processing units allocated to J_j is t_j. We show that such a sequence of machines always exists. The selection of this sequence is done as follows. We first examine $L[1]$, which is the weakest machine. If $Q_{L[1]} \geq t_j$ we schedule J_j on $L[1]$ and update the potential of this machine; else, we check whether $Q_{L[1]} + Q_{L[2]} \geq t_j$, and so on until either we find a sequence of machines with sufficient potential ($\geq t_j$), or the total potential of the first $\rho_j + 1$ machines is less than t_j. In the latter case, we proceed to examine the next *window* of $\rho_j + 1$ machines, $L[2], \ldots, L[\rho_j + 2]$, and so on, until our window covers $\rho_j + 1$ machines, $L[k], \ldots, L[k + \rho_j]$, such that $t_j > Q_{L[k-1]} + \ldots + Q_{L[k+\rho_j-1]}$ and $t_j \leq Q_{L[k]} + \ldots + Q_{L[k+\rho_j]}$ (see Figure 5.1). At this stage we can clearly allocate to J_j all the potential of the machines $L[k], \ldots, L[k + \rho_j - 1]$, and complete the execution of J_j by allocating to it also some of the potential of the machine $L[k + \rho_j]$.

In fact, $\mathcal{A}_r$ only determines the amount of processing units allocated to each job by each of the machines. The order of the jobs on each machine is arbitrary. Note that J_j is allotted to at most $\rho_j + 1$ machines. (Thus, at most $\rho_j + 1$ machines process it *simultaneously*).

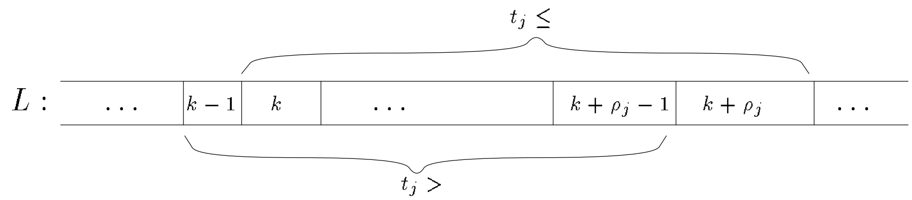

Figure 5.1: The window scanning the machine-list L

Let J_b be the first job such that, when J_b is scheduled, the list L contains at most ρ_b machine indices, or the total processing potential of the ρ_b weakest machines is at least t_b. We distinguish between two phases of $\mathcal{A}_r$:

1. The jobs $J_1, \ldots, J_{b-1}$ are scheduled.
2. The jobs $J_b, \ldots, J_n$ are scheduled.

We show that in each phase the corresponding set of jobs is scheduled legally. Note that jobs scheduled in the first phase are scheduled using the *moving-window*, each on exactly $\rho_j + 1$ machines. For this phase, we need to show that we never fail to find a subset of $\rho_j + 1$ machines that can complete J_j. Specifically, we show below that for each $j < b$, when J_j is scheduled, the total potential of the strongest $\rho_j + 1$ machines in L is at least t_j. For the second phase of $\mathcal{A}_r$ we show that for each $b \le j \le n$, J_j is allocated t_j processing units from at most $\rho_j + 1$ machines.

For simplicity, assume that whenever we use the moving-window to schedule a job, the list L is scanned from left to right. That is, the index of the weakest machine, $L[1]$, is the leftmost, and the index of the strongest machine is the rightmost in L. The window moves from left to right until for some k, which is the index in L of the weakest machine in the window, we get that the $\rho_j + 1$ machines $L[k-1], \ldots, L[k+\rho_j]$ can complete the execution of J_j. When J_j is scheduled, the machines $L[k], \ldots, L[k+\rho_j-1]$ are removed from L, and the machine $L[k+\rho_j]$ is possibly moved to a new position in the list L, according to its remaining potential. Note that this new position of $L[k+\rho_j]$ in L is left to its original position.

We can view the removed sequence of machines as a *hole* in L. Each job J_j creates a hole of ρ_j machines in L, and one additional machine (the (ρ_j+1)th) is moved left to the hole. By analyzing these holes we conclude that $\mathcal{A}_r$ never fails to schedule jobs during its first phase.

Lemma 5.2.3 *Each job, J_j, scheduled by $\mathcal{A}_r$ during the first phase, is scheduled on exactly ρ_j+1 machines and is allocated t_j processing units.*

Proof: Assume that for some job J_g, the moving window procedure fails to complete J_g. That is, the window reaches the rightmost position in L, but the total potential of the $\rho_g + 1$ strongest machines covered by the window is less than t_g.

Let us examine the sequence of holes created in L by the time J_g is scheduled. We first show that when we fail to schedule J_g, its partial schedule creates at the right end of L a hole, which is the union of holes created by previously scheduled jobs.

Claim 5.2.4 *There exists a set $S \subseteq \{J_1, \ldots, J_g\}$, such that the hole created by J_g unites the holes created by the jobs in S, into a single hole positioned at the right end of L.*

Proof: Consider the hole in L that contains the $\rho_g + 1$ machines on which J_g is scheduled and all the holes that are united when these $\rho_g + 1$ machines are removed. Let H denote this united hole. Since each job J_j, $j < g$ creates a hole of ρ_j *consecutive* machines in L, then for each such job, either *all* or *none* of these ρ_j machines is contained in H (we ignore the $(\rho_j + 1)$th machine: J_j may only partially use this machine, in which case the machine is not removed from L).

Let S be the set of all the jobs, J_j, that contribute ρ_j or $\rho_j + 1$ machines to H. By definition, the hole H unites the holes created by the jobs in S into a single hole. In addition, since J_g is not completed, H must include the strongest $\rho_g + 1$ available machines, and in particular the rightmost one. Therefore, H is positioned at the right end of L. ∎

We conclude that $\mathcal{A}_r$ allocates to the jobs in S at least all the potential of the $\sum_{J_j \in S} \rho_j$ strongest machines.

Claim 5.2.5 *The total potential of the $\sum_{J_j \in S} \rho_j$ fastest machines is at least $\sum_{J_j \in S} t_j$.*

Proof: Let $\rho_S = \sum_{J_j \in S} \rho_j$. Recall that for any $1 \le \ell \le n$, $\hat{\rho}_\ell = \sum_{j=1}^{\ell} \rho_j$. Let J_ℓ be the job such that $\hat{\rho}_{\ell-1} < \rho_S \le \hat{\rho}_\ell$. Note that $\ell \le g$ since $S \subseteq \{J_1, \ldots, J_g\}$. By the definition of w, $w \cdot \sum_{i=1}^{\hat{\rho}_{\ell-1}} u_i \ge \sum_{j=1}^{\ell-1} t_j$ and $w \cdot \sum_{i=1}^{\hat{\rho}_\ell} u_i \ge \sum_{j=1}^{\ell} t_j$. That is, the first $\hat{\rho}_{\ell-1}$ machines are strong enough to complete the first $\ell - 1$ jobs, and the first $\hat{\rho}_\ell$ machines are strong enough to complete the first ℓ jobs. Since the machines are sorted such that $u_i \ge u_{i+1}$, we conclude that for any integer $0 < x \le \rho_\ell$, the first $\hat{\rho}_{\ell-1} + x$ machines can complete the first $\ell - 1$ jobs and an $\frac{x}{\rho_\ell}$-fraction from J_ℓ. Formally, $w \cdot \sum_{i=1}^{\hat{\rho}_{\ell-1}+x} u_i \ge \sum_{j=1}^{\ell-1} t_j + x\frac{t_\ell}{\rho_\ell}$.

In particular, for $x = \rho_S - \hat{\rho}_{\ell-1}$, we have that

$$w \sum_{i=1}^{\rho_S} u_i \ge \sum_{j=1}^{\ell-1} t_j + (\rho_S - \hat{\rho}_{\ell-1}) \frac{t_\ell}{\rho_\ell}. \tag{5.2}$$

For a set, Y, of jobs $J_{i_1}, J_{i_2}, \ldots$, consider the vector $\vec{v}_Y$ consisting of ρ_{i_1} entries with the value $\frac{t_{i_1}}{\rho_{i_1}}$, followed by ρ_{i_2} entries with the value $\frac{t_{i_2}}{\rho_{i_2}}$, and so on. For the set $\mathcal{J}$ of all the jobs in our instance, consider the vector $\vec{v}_1$ consisting of the first ρ_S entries of $\vec{v}_{\mathcal{J}}$. Since the jobs are sorted such that $\frac{t_j}{\rho_j} \ge \frac{t_{j+1}}{\rho_{j+1}}$, and since S is a subset of $\mathcal{J}$, $\vec{v}_1 \ge \vec{v}_S$. That is, for any index i, $v_1^i \ge v_S^i$. Therefore, $\sum_{i=1}^{\rho_S} v_1^i \ge \sum_{i=1}^{\rho_S} v_S^i$. However, $\sum_{i=1}^{\rho_S} v_1^i = \sum_{j=1}^{\ell-1} t_j + (\rho_S - \hat{\rho}_{\ell-1}) \frac{t_\ell}{\rho_\ell}$, and $\sum_{i=1}^{\rho_S} v_S^i = \sum_{J_j \in S} t_j$. Thus, $\sum_{j=1}^{\ell-1} t_j + (\rho_S - \hat{\rho}_{\ell-1}) \frac{t_\ell}{\rho_\ell} \ge \sum_{J_j \in S} t_j$.

From Equation (5.2), we get that the ρ_S strongest machines can complete the jobs in S. ■

This contradicts our assumption that $\mathcal{A}_r$ fails to schedule $J_g \in S$ during the first phase. ■

We turn to show that all the jobs that are scheduled during the second stage of $\mathcal{A}_r$ are scheduled legally.

Lemma 5.2.6 *Each job, J_j, scheduled by $\mathcal{A}_r$ during the second phase, is scheduled on at most $\rho_j + 1$ machines and is allocated t_j processing units.*

Proof: $\mathcal{A}_r$ reaches the second phase if, for the next job to be scheduled , J_b, the list L contains at most ρ_b machine indices, or if the total processing potential of the ρ_b weakest machines is at least t_b.

In the analysis of the second phase, we need to show that all the jobs are completed, and that at most $\rho_j + 1$ machines participate in the schedule of J_j. For showing that all jobs are completed, note that when the algorithm starts, we have $w \sum_{i=1}^{m} u_i \geq \sum_{j=1}^{n} t_j$, that is, the total processing potential is at least the total processing requirement of the jobs. This is due to the fact that during the first phase no job is allocated more than t_j processing units). This guarantees that when we schedule J_j greedily, we never run out of processing potential.

Clearly, at most ρ_b machines participate in the greedy schedule of J_b. The last machine on which J_b is scheduled may have additional processing potential. This machine, M_ℓ, is now the weakest machine (since it belonged to the set of weakest machines before J_b was scheduled, and all the weaker machines in this set are now omitted from L). i.e., $L[1] = \ell$. We now proceed to schedule the remaining jobs. In order to show that for any $j > b$, at most $\rho_j + 1$ machines share the execution of J_j, we first prove the following claim.

Claim 5.2.7 *After J_b is scheduled, for any $j > b$, the list L contains at most ρ_j machines, or the total potential of the ρ_j machines $L[2], \ldots, L[\rho_j + 1]$ is at least t_j.*

Proof: Assume that J_b is scheduled on x machines. Clearly, $x \leq \rho_b$; thus, the potential of the strongest machine among these x machines is at least $\frac{t_b}{x} \geq \frac{t_b}{\rho_b} \geq \frac{t_j}{\rho_j}$, for any $j > b$. Following the schedule of J_b, the list L is updated, and the weakest $x - 1$ machines are removed; the remaining potential of the next (x-th) machine becomes $Q_{L[1]}$. Since the machines in L are sorted in non-decreasing order by their potential, the potential of each of the machines $L[2], \ldots, L[\rho_j + 1]$ (assuming $|L| > \rho_j$), is at least $\frac{t_j}{\rho_j}$, meaning that the total potential of these ρ_j machines is at least t_j. ■

Consider a job $J_j, j > b$. If, following the schedule of J_b, the list L contains at most ρ_j machines, then, clearly, J_j will be scheduled on at most ρ_j machines; otherwise, note that if the ρ_j machines $L[2], \ldots, L[\rho_j + 1]$ are strong enough to complete J_j, then any set of ρ_j machines, not including $L[1]$, is strong enough for J_j.

We show that $\mathcal{A}_r$ never uses more than ρ_j+1 machines for processing J_j. Once J_b is scheduled, we turn to schedule J_{b+1}. Consider the subset of the $\rho_{b+1}+1$ weakest machines. It consists of $L[1]$ and additional ρ_{b+1} machines. From the above discussion, the total potential of the additional ρ_{b+1} machines is at least t_{b+1}, therefore (even if $Q_{L[1]}$ is small), the total potential of the weakest $\rho_{b+1}+1$ machines is at least t_{b+1}, and we can allocate to J_{b+1} exactly t_{b+1} processing units, by using at most $\rho_{b+1}+1$ machines. Again, the last machine may have remaining potential. The same argument holds for all the remaining jobs. That is, every job J_j will be allocated exactly t_j processing units, using at most ρ_j+1 machines. ∎

Proof of Theorem 5.2.1: From Lemmas 5.2.3 and 5.2.6 we get that $\mathcal{A}_r$ assigns to each of the jobs, J_j, t_j processing units, on at most ρ_j+1 machines. In addition, from Lemma 5.2.2, the length of the schedule is $w \leq w_{OPT}(I)$.

We now turn to compute the running time of $\mathcal{A}_r$. The algorithm $\mathcal{A}_r$ can be implemented in $O(\max(m\log m,\ n\log n))$ steps: $O(m\log m)+O(n\log n)$ steps are required for sorting the lists and calculating w. Given that the lists are sorted, the total time for scheduling the jobs is $O(m+n)+O(n\log m)$. The first phase of the algorithm, in which we schedule the jobs by scanning the list L with the moving-window, can be implemented in $O(m)+O(n\log m)$ steps. The idea is that there is no need to scan the list L from the beginning for each job. Note that the schedule of any job $J_j, j>1$ is done on exactly ρ_j+1 machines. Thus, the location in L of these ρ_j+1 consequent machines must encircle the strongest machine with potential at most $\frac{t_j}{\rho_j+1}$. Since L is sorted, finding this machine can be done (e.g., using skip-lists) in $O(\log m)$ steps. We can now find in $O(\rho_j)$ steps the set of $\rho+1$ machines that will process J_j. Finally, after we schedule J_j, we have to reposition in L the $(\rho+1)$th machine in the window according to its remaining potential. Since L is sorted, this can be implemented in $O(\log m)$ steps.

Let P_1 denote the set of jobs scheduled in the first phase. Recall that when the list L includes less than ρ_j machines (when J_j is the next job to be scheduled), we move to the second phase. Thus, $\sum_{J_j\in P_1}\rho_j \leq m$, and therefore, the total time for locating the windows and schedule all the jobs in P_1 is $O(n\log m)+O(m)$. During the second phase of the algorithm, we schedule the jobs greedily, in $O(m+n)$ steps.

A $\max_j(1+\frac{1}{\rho_j})$-Approximation

The algorithm $\mathcal{A}_r$ yields the following approximation algorithm, $\mathcal{A}_1$, for the SPAC problem. Given an instance, I,

1. Use $\mathcal{A}_r$ to find a schedule of length w for I^+.
2. For each job, $J_j, j=1,\ldots,n$,
   If J_j is scheduled on a_j+1 machines or if ρ_j+1 machines process J_j simultaneously:
   - Let M_s be the machine which allocated the minimal number of processing units to J_j.

- Omit J_j from the set of jobs scheduled on M_s.
- Any other machine, M_i, which processed J_j for x time units, will now process J_j for $x(1+\frac{1}{\rho_j})$ time units.

In other words, we transform the non-feasible schedule into a feasible one by splitting, for each job, J_j, the processing of J_j on the least-contributing machine among the other ρ_j machines that process J_j. As shown below, this extends the makespan of the schedule by a fraction which depends on the minimal parallelism parameter of any job.

Theorem 5.2.8 $w_{\mathcal{A}_1}(I) \leq \max_j(1+\frac{1}{\rho_j})w_{OPT}(I)$.

Proof: We first show that each job, J_j, is allocated at least t_j processing units. Clearly, the machine M_s allocated to J_j at most $\frac{t_j}{\rho_j+1}$ processing units. Thus, the other ρ_j machines allocate to J_j at least $\frac{\rho_j t_j}{\rho_j+1}$ processing units. The execution of J_j on each of these machines is 'stretched' by a factor $1+\frac{1}{\rho_j}$. Hence, the total allocation of processing units to J_j on these machines is increased to be at least $\frac{\rho_j t_j}{\rho_j+1}(1+\frac{1}{\rho_j}) = t_j$.

To bound the resulting makespan, note that in the worst case, there exists a machine that has to compensate for all the jobs that it executes, meaning that its processing time is stretched from w to at most $\max_j w(1+\frac{1}{\rho_j})$. By Theorem 5.2.1, $\mathcal{A}_r$ generates a schedule of length $w \leq w_{OPT}(I)$, thus, the makespan obtained by our algorithm is at most $\max_j(1+\frac{1}{\rho_j})w_{OPT}(I)$. ∎

In particular, if the allotment and parallelism limits of each job are at least b, for some $b \geq 1$, that is, $\rho_j \geq b$ for all $1 \leq j \leq n$, the above approximation algorithm yields the makespan $(1+\frac{1}{b})w_{opt}(I)$.

5.2.2 Identical Machines

We show that for the special case where the machines are identical, the algorithm $\mathcal{A}_r$ is a $\max_j(1+\frac{1}{2\rho_j-1})$-approximation to the optimal. Assume w.l.o.g. that all machines have the rate $u=1$. Thus, $w = \max\{\sum_j t_j/m, \max_j \frac{t_j}{\rho_j}\}$.

When $\mathcal{A}_r$ is executed on an instance with identical machines, since $t_1 \leq w\rho_1$, we schedule greedily all the jobs. Thus, each job, J_j, is scheduled on a set of consecutive machines. (see Figure 5.2). Let T_1 and T_2 denote the lengths of the time intervals allocated to J_j on the 'extreme' machines. (e.g., M_i and $M_{i+\rho_j}$ in Figure 5.2). Note that if J_j is scheduled on ρ_j+1 machines then $t_j = (\rho_j-1)w + T_1 + T_2$ and $T_1+T_2 \leq w$ (otherwise, J_j is allocated more than $w\rho_j \geq t_j$ processing units). As in the case of uniform machines, we transform the non-feasible schedule into a feasible one, by splitting, for each job, J_j, the processing of J_j on the least-contributing machine among the other ρ_j machines that process J_j. However, as we show below, the 'stretching' factor of each machine can be reduced to be $1+\frac{1}{2\rho_j-1}$. Thus, the resulting algorithm, $\mathcal{A}_2$, has better approximation ratio.

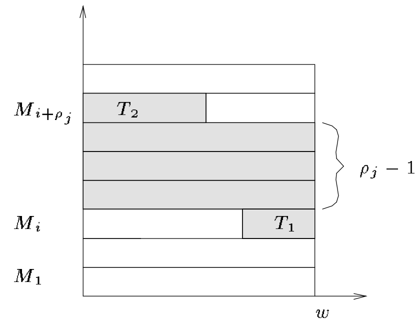

Figure 5.2: The schedule of J_j on identical machines (in proof of Theorem 5.2.9)

Theorem 5.2.9 $w_{\mathcal{A}_2}(I) \leq \max_j(1 + \frac{1}{2\rho_j - 1})w_{OPT}(I)$.

Proof: Assume w.l.o.g. that $T_1 \leq T_2$. Since $T_1 + T_2 \leq w$, we have that $T_1 \leq \frac{w}{2}$. In the 'stretched' schedule, J_j is allocated x_j processing units.

$$x_j = (w(\rho_j - 1) + T_2)(1 + \frac{1}{2\rho_j - 1}) = w(\rho_j - 1) + T_2 + (w(\rho_j - 1) + T_2)\frac{1}{2\rho_j - 1}.$$

Since $t_j = w(\rho_j - 1) + T_2 + T_1$, the additional $(w(\rho_j - 1) + T_2)\frac{1}{2\rho_j - 1}$ processing units need to compensate for the T_1 units that were omitted. Indeed,

$$\begin{aligned}
\frac{w(\rho_j - 1) + T_2}{2\rho_j - 1} - T_1 &= \frac{w(\rho_j - 1) + T_2 - (2\rho_j - 1)T_1}{2\rho_j - 1} \\
&\geq \frac{w(\rho_j - 1) + T_1 - (2\rho_j - 1)T_1}{2\rho_j - 1} \\
&= \frac{w(\rho_j - 1) - (2\rho_j - 2)T_1}{2\rho_j - 1} \\
&= \frac{\rho_j - 1}{2\rho_j - 1}(w - 2T_1) \quad \geq \quad 0.
\end{aligned}$$

Thus, $x_j \geq w(\rho_j - 1) + T_2 + T_1 = t_j$, meaning that J_j is allocated at least t_j processing units. ■

5.3 A PTAS for Scheduling with Allotment Constraints

In this section we present a PTAS for the SAC problem on uniform machines. We assume that the maximal allotment parameter of any job is some fixed constant. In Section 5.1 we have shown that SAC is strongly NP-hard in this case, even for instances with identical machines (Theorem 5.1.1) or identical jobs (Theorem 5.1.6).

Our PTAS consists of two stages. In the first stage we guess a partition of each job, J_j, to at most a_j segments. In the second stage we consider each job segment as a separate job. The resulting instance has at most na_{max} jobs. We run on this instance a PTAS, P^*, for multiprocessor scheduling on uniform machines (e.g., [30, 15]). Note that some segments of the same job may be scheduled by P^* on the same machine: this is equivalent to sharing the execution of J_j among fewer machines.

An immediate problem which arises when trying to apply this, is that the number of possible partitions is *exponential.* We show that it suffices to examine only a polynomial number of possible partitions in order to approximate the optimal schedule. This subset of partitions can be found and described efficiently.

We first show how to reduce the number of partitions that need to be considered, when the jobs are *identical.* Next, we extend this technique to instances with a *fixed* number of job types. Finally, we show that an *arbitrary* instance can be converted into one in which small jobs cannot split at all, and non-small jobs can be replaced by jobs of a fixed number of types. Each of these steps extends the makespan by factor $(1+\varepsilon)$. An additional $(1+\varepsilon)$-extension is caused by P^*.

5.3.1 Identical Jobs

Assume first that all jobs have the same length, t, and the same allotment parameter, $a > 1$. Thus, the execution of each job can be shared among at most a machines. Given $\varepsilon > 0$, let $\delta = \frac{\varepsilon}{a}$.

Lemma 5.3.1 *Any schedule of I of length C, can be transformed into one of length at most $(1+\varepsilon)C$, in which all job segments are larger than $\delta\frac{t}{a}$, and their lengths are multiples of $\delta^2\frac{t}{a}$.*

Proof: We say that a job segment is *small*, if is has length smaller than $\delta\frac{t}{a}$. We first describe how the schedule is modified; then, we show that the total load on each machine is increased by at most a factor $1+\varepsilon$. For each job, J_j,

1. Add all the small segments of J_j to the longest one and round up the resulting length to the next multiple of $\delta^2\frac{t}{a}$.

2. Round the length of any other segment to the next multiple of $\delta^2\frac{t}{a}$.

3. Shorten the longest segment by a multiple of $\delta^2\frac{t}{a}$ such that the total sum of the segment lengths is at least t.

Clearly, in the resulting schedule all the segments are larger than $\delta\frac{t}{a}$, their lengths are multiples of $\delta^2\frac{t}{a}$, and the processing time of each job is at least t. In addition, since we only group the small segments and add them to the longest one, the allotment constraint is preserved.

We show that for each machine M_i and job J_j, if M_i processed J_j for t_1 time units, it now processes J_j for at most $(1+\varepsilon)t_1$ time units; thus, the makespan of the schedule is at most $(1+\varepsilon)C$.

For each job, each non-small segment, excluding the longest one, has length $p > \delta\frac{t}{a}$ and its new length is at most $p + \delta^2\frac{t}{a} \leq p + \delta p < p(1+\varepsilon)$. The longest segment of a job must have length $p \geq \frac{t}{a}$. Even if the length of this (long) segment is not reduced in step 3, its new length is now at most $p + (a-1)\delta\frac{t}{a} + \delta^2\frac{t}{a} \leq p + \delta t \leq p + \delta a p = p(1+\varepsilon)$. ■

Recall, that our PTAS guesses the partition of the n jobs, each into at most a segments, and uses P^* for the guessed partitions. By Lemma 5.3.1 we conclude that we pay only ε for considering only a subset of the possible partitions. We now show that an optimal partition can be guessed efficiently. That is, the number of possible partitions is polynomial in n.

Denote by S_δ the set of partitions of a number t into at most a numbers which are all larger than $\delta\frac{t}{a}$, and their values are multiples of $\delta^2\frac{t}{a}$. In the following we compute the size of S_δ, denoted by h_δ. We use in the computation the next result, given in [10].

Lemma 5.3.2 *Let f be the number of g-tuples of non-negative integers such that the sum of tuple coordinates is equal to d, for some $d \geq 1$. Then $f = \binom{d+g-1}{g-1}$. If $d + g \leq \alpha g$, for some $\alpha \geq 1$, then $f = O(\alpha^{\alpha g})$.*

Now, we bound h_δ in terms of a.

Lemma 5.3.3 *The size of S_δ is $h_\delta = O((2e)^a)$, where the symbol e denotes the base of the natural logarithm.*

Proof: Note that we can describe a partition of a given job to at most a segments by an $a/(\delta^2)$-tuple: each coordinate, i, gives the number of segments of length $i \cdot (\delta^2 t)/a$, $1 \leq i \leq a/\delta^2$; the sum of the coordinates is at most a. By Lemma 5.3.2, taking $d = a$, $g = a/\delta^2$ and $\alpha = 1 + \delta^2$, we get that the number of such tuples is

$$\binom{a + a/\delta^2 - 1}{a/\delta^2 - 1} = O((1+\delta^2)^{(1+\delta^2)a/\delta^2}) = O((2e)^a)$$

The last equality follows from the standard bound $(1+x)^{1/x} \leq e$, for $0 < x < 1$, and the assumption that $(1+\delta^2) \leq 2$. ■

Each item in S_δ describes a partition of a single job. To describe a partition of the n jobs we use a vector of length h_δ, whose ith entry specifies for how many jobs we adapt the ith partition vector. The number of possible vectors is less than n^{h_δ}.

5.3.2 Fixed Number of Job Types

Assume that there are T different job types, where $T \geq 1$ is some constant. All the n_k jobs of the kth type, $1 \leq k \leq T$ have length t_k and allotment parameter a_k. Note that the proof of

Lemma 5.3.1 considers the extension of each segment on each machine *separately*. Thus, choosing $\delta_k = \frac{\varepsilon}{a_k}$, we can extend Lemma 5.3.1 as follows:

Lemma 5.3.4 *Any schedule of I of length C can be transformed into one of length at most $(1+\varepsilon)C$, in which all the segments of jobs of the kth type are larger than $\delta \frac{t_k}{a_k}$ and their lengths are multiples of $\delta^2 \frac{t_k}{a_k}$.*

Let h_{δ_k} be the *constant* number of possible partitions of one job of the kth type. In order to describe a possible partition of the n jobs we use a vector of length $h_{\delta_1} + h_{\delta_2} + \cdots + h_{\delta_T}$, whose entries specify how many jobs of each type are partitioned in a certain way. The number of possible vectors is less than $\prod_{k=1}^{T} n_k^{h_{\delta_k}} = O(n^{\sum_{i=1}^{T} h_{\delta_k}})$.

5.3.3 Arbitrary Jobs

Given an arbitrary instance, our idea is to distinguish between small and large jobs. For the subset of large jobs we pay ε in order to convert it into one with a fixed number of job types - for which, as we showed in Section 5.3.2, we need to examine only a polynomial number of possible partitions. For the small jobs we show that we may pay at most factor ε from reducing all their allotment parameters to one. Any job, J_j, for which $a_j = 1$ need not participate in the 'guessing partition' process, i.e., J_j is given to P^* as a single segment.

Let t_{max} be the maximal length of any job in I. Let a_1 be the minimal allotment parameter among the jobs with length t_{max}. For a given ε, we say that a job is *small*, if its length is less than $\varepsilon \frac{t_{max}}{a_1}$.

Lemma 5.3.5 *Any schedule of I of length C can be transformed into one of length at most $(1+\varepsilon)C$, in which all the small jobs are not partitioned at all (that is, have $a_j = 1$).*

Proof: Given a schedule of I, let W_{s_i} be the total number of processing units allocated to segments of small jobs on M_i. We reallocate $W_{s_1}, W_{s_2}, \ldots$ to the small jobs sequentially, starting from the first small job on the first machine. When a small job is completed we move to the next small job; when all the W_{s_i} processing units are allocated, we complete the active small job and move to the next machine and to the next small job. Clearly, since we allocate at least $\sum_i W_{s_i}$ processing units, all the small jobs are completed. Also, since we always complete a job on the machine on which its processing starts, we do not split small jobs. Finally, the makespan can increase at most by factor $1+\varepsilon$: in the original schedule, some machine must have load at least $\frac{t_{max}}{a_1}$. Thus, $C \geq \frac{t_{max}}{a_1}$. The total load on each machine is extended by at most $\varepsilon \frac{t_{max}}{a_1}$. Therefore, the resulting makespan is at most $C + \varepsilon \frac{t_{max}}{a_1} \leq C + \varepsilon C = (1+\varepsilon)C$. ■

From Lemma 5.3.5, for the purpose of guessing a partition of jobs to segments, we may assume that our instance consists of *large* jobs. Let a_{max} be the maximal allotment parameter of any job in I. We replace the large jobs by $T = a_{max}(\frac{a_1}{\varepsilon^2} - \frac{1}{\varepsilon})$ sets of jobs such that the jobs in each set are identical. For $1 \leq a \leq a_{max}, \frac{1}{\varepsilon} < \ell \leq \frac{a_1}{\varepsilon^2}$, let $N_{a,\ell}$ be the number of jobs with $a_j = a$ and $t_j \in ((\ell - 1)\varepsilon^2 \frac{t_{max}}{a_1}, \ell\varepsilon^2 \frac{t_{max}}{a_1}]$.

Our new instance, I', consists of $N_{a,\ell}$ jobs with $a_j = a$ and $t_j = \ell\varepsilon^2 \frac{t_{max}}{a_1}$. As in other PTASs that use interval partition (e.g. [30, 15, 2]), we have

Lemma 5.3.6 *Any schedule of I of length C can be replaced by a schedule of I' of length $(1+\varepsilon)C$*

Proof: Each long job from I with $a_j = a$ and $t_j \in ((\ell - 1)\varepsilon^2 \frac{t_{max}}{a_1}, \ell\varepsilon^2 \frac{t_{max}}{a_1}]$, contributes to I' a job with the same allotment parameter and of length $t_j + \Delta$, $\Delta \leq \varepsilon^2 \frac{t_{max}}{a_1}$. This extension of Δ can split among the segments of J_j as follows. For each machine M_i and job J_j, if M_i processes a fraction α of J_j, $0 \leq \alpha \leq 1$, M_i will now process a fraction α of the extended job. Summing over all the segments of J_j, we get that the extended job is fully processed. The job J_j is long, thus, $t_j \geq \varepsilon \frac{t_{max}}{a_1}$. Therefore, $\alpha(t_j + \Delta) \leq \alpha(t_j + \varepsilon^2 \frac{t_{max}}{a_1}) \leq \alpha(t_j + \varepsilon t_j) \leq \alpha t_j(1 + \varepsilon)$, and the total processing time of each job on each machine is extended by at most a factor $1 + \varepsilon$. ∎

We summarize in the next result.

Theorem 5.3.7 *The SAC problem with fixed allotment parameters admits a PTAS, whose running time is $O(n^{\frac{a_{max}a_1}{\varepsilon^2}(2e)^{a_{max}}})$.*

Proof: The PTAS described above consists of four steps:

1. Distinguishing between small and large jobs.
2. Replacing the large jobs by jobs of $T \leq a_{max}(\frac{a_1}{\varepsilon^2})$ sets of identical jobs, as described in Section 5.3.3.
3. Guessing a partition of the resulting jobs to segments.
4. Running the PTAS P^* on the resulting sets of segments and small jobs.

By Lemmas 5.3.4, 5.3.5, 5.3.6, and since P^* is a PTAS for the multiprocessing scheduling problem, we get that each stage may extend the makespan by factor $(1+\varepsilon)$. W.l.o.g. we assume that $\varepsilon < 1$; thus, by running these steps with $\hat{\varepsilon} = \varepsilon/9$ we get a total extension of factor $(1+\varepsilon)$. As discussed in Section 5.3.2, in the third step we examine $O(n^{\sum_{i=1}^{T} h_{\delta_k}}) = O(n^{a_{max}(\frac{a_1}{\varepsilon^2})(2e)^{a_{max}}})$ partitions. Finally, P^* can be implemented in $O(n^{\frac{1}{\varepsilon^2}})$ (see [15]).

Thus, we get that the overall running time is at most $O(n^{\frac{a_{max}a_1}{\varepsilon^2}(2e)^{a_{max}} + \frac{1}{\varepsilon^2}})$ which yields the statement of the theorem. ∎

5.4 Solving SPAC for Parallel-dominated Instances

In this section we consider instances in which $\forall j,\ \rho_j < a_j$. We call such instances *parallel-dominated*. We show that for these instances the SPAC problem is optimally solvable on identical machines. For uniform machines the solvability of our problem depends on the differences $(a_j - \rho_j)$. Recall that preemptive scheduling on multiple machines can be viewed as the SPAC problem on parallel-dominated instances, where $\rho_j = 1$ and $a_j = m,\ \ \forall 1 \leq j \leq n$. Our results imply that when we bound the number of machines that can share the execution of each job, the preemptive scheduling problem is

1. Solvable on identical machines with any allotment constraints.
2. Strongly NP-hard on uniform machines and identical jobs, where each job can run on at most two machines.
3. Solvable on uniform machines where the jobs are identical, and each job can run on at least three machines.

5.4.1 An Optimal Algorithm for Identical Machines

We now show that SPAC is polynomially solvable on identical machines, for any parallel-dominated instance.

Theorem 5.4.1 *The SPAC problem is solvable in $O(n+m)$ steps on identical machines, for instances where $\rho_j < a_j$, $\forall j$.*

Proof: Assume w.l.o.g that all the machines have the same rate $u = 1$. Recall that $w = \max\{\sum_j t_j/m, \max_j \frac{t_j}{\rho_j}\}$ is a lower bound on the length of an optimal schedule.

Consider the simple greedy algorithm, $\mathcal{A}_u$, based on McNaughton rule [48]. $\mathcal{A}_u$ proceeds by scheduling the jobs one after the other on the machines. It uses each machine, M_i, for w time units, and then moves to M_{i+1}; it moves to the next job J_{j+1}, once the job J_j is allocated t_j processing units. Thus, each job J_j is scheduled on a consecutive set of machines.

We first show that the parallelism constraints are preserved. Assume by contradiction that there exists a job J_j such that at time $t_0 \in [0, w]$, at least $\rho_j + 1$ machines process J_j (see Figure 5.3).

Suppose that the first machine used for processing J_j is M_i. At time t_0, the machines $M_i, \ldots, M_{i+\rho_j}$ process J_j; Each of the machines $M_{i+1}, \ldots, M_{i+\rho_j-1}$ allocates to J_j w processing units; M_i allocates T_1 units and $M_{i+\rho_j}$ allocates T_2 units. Since both M_i and $M_{i+\rho_j}$ process J_j at time t_0, we get that $T_1 + T_2 > w$. Consequently, J_j is allocated $(\rho_j - 1)w + T_1 + T_2 > \rho_j w$

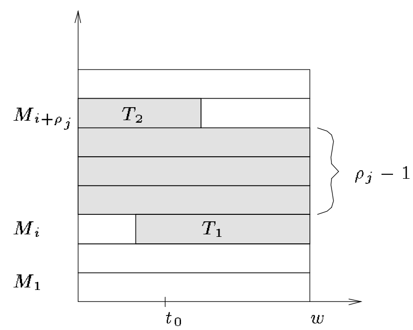

Figure 5.3: The greedy schedule of J_j (in proof of Theorem 5.4.1)

processing units. Since $w \geq \frac{t_j}{\rho_j}$, we get that J_j was allocated more than t_j processing units, in contradiction to the way $\mathcal{A}_u$ proceeds.

To see that all the jobs are scheduled within an interval of length w, note that $\mathcal{A}_u$ proceeds to the next machine only when the current machine is 'saturated'. Thus, it can allocate mw processing units. By definition, $w \geq \sum_j t_j/m$. That is, $mw \geq \sum_j t_j$.

Note that in the schedule produced by $\mathcal{A}_u$, each job is processed by at most $\rho_j + 1$ machines. Thus, since $\forall j, \rho_j < a_j$, the allotment constraints are preserved. ■

5.4.2 Parallel-dominated Instances and Uniform Machines

We now show that when the machines may have different speeds, the SPAC problem is strongly NP-hard on parallel-dominated instances, even if all jobs are identical.

Theorem 5.4.2 *The SPAC problem is strongly NP-hard even if all jobs are identical and* $\forall j, \rho_j < a_j$.

Proof: Given an instance of 3-partition, we construct an input, I, for the makespan problem with $\forall j, \rho_j = 1, a_j = 2$, such that $w_{OPT}(I) = 1$ if and only if A has a 3-partition. Note that this is an instance of the preemption problem in which the execution of each job can be shared by at most two machines.

The input I consists of $m = 4q$ machines with the following rates: for the first $3q$ machines, $u_i = (K + s(x_i))(1 - \frac{K - s(x_i)}{3K-B})^{-1}$, $1 \leq i \leq 3q$, where $K > 3qB$ is a large constant. These machines are denoted *slow*. The other q machines are *fast* with $u_i = 3K - B$, $3q < i \leq 4q$. There are $n = 3q$ identical jobs with $t_j = 2K, \rho_j = 1, a_j = 2$, $\forall 1 \leq j \leq 3q$.

Claim 5.4.3 *For any slow machine,* M_i, $1 \leq i \leq 3q$, $\frac{1}{2}u_i < K$.

Proof: By definition of the 3-partition problem, $\forall x \in A,\ B/4 < s(x) < B/2$. Thus,

$$\frac{1}{2}u_i < \frac{1}{2}(K + \frac{B}{2})(1 - \frac{K - \frac{B}{4}}{3K - B})^{-1} < \frac{1}{2}(K + \frac{B}{2})\frac{5}{3} = \frac{5}{6}K + \frac{5}{12}B < K.$$

The last inequality follows from fact that K was selected such that $3B < K$, therefore, $\frac{5}{12}B < \frac{1}{2}B < \frac{1}{6}K$. ■

Assume that A has a 3-partition to the sets $S_1, S_2, \ldots, S_q$. $\forall 1 \le k \le q$, let $S_k = \{x_{k_1}, x_{k_2}, x_{k_3}\}$. The following is a schedule whose makespan equals 1: $\forall 1 \le k \le q$, the four machines $M_{k_1}, M_{k_2}, M_{k_3}, M_{3q+k}$ process the three jobs $J_{k_1}, J_{k_2}, J_{k_3}$. Specifically, as illustrated in Figure 5.4, M_{k_i} executes J_{k_i}, $1 \le i \le 3$ for $1 - \frac{K - s(x_{k_i})}{3K-B}$ time units and thus allocates to it $K + s(x_{k_i})$ processing units. The fast machine M_{3q+k} executes J_{k_i}, $1 \le i \le 3$ for $\frac{K-s(x_{k_i})}{3K-B}$ time units and thus allocates to it $K - s(x_{k_i})$ processing units. Since $s(x_{k_1}) + s(x_{k_2}) + s(x_{k_3}) = B$, and $u_{3q+k} = 3K - B$, the completion time equals to 1. Note that the allotment constraints are preserved: the execution of J_j is shared by the two machines M_j and M_{3q+k} such that $x_j \in S_k$. Also, the parallelism constraints are satisfied: M_{k_i} is idle while J_{k_i} is executed on M_{3q+k}. Finally, J_{k_i} is allocated exactly t_{k_i} processing units, since $(3K - B)\frac{K-s(x_{k_i})}{3K-B} + u_{k_i}(1 - \frac{K-s(x_{k_i})}{3K-B}) = 2K$.

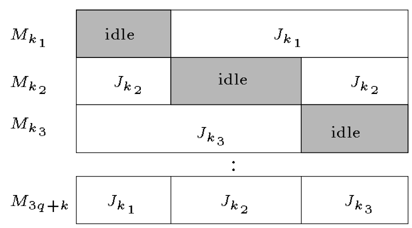

Figure 5.4: The schedule of $J_{k_1}, J_{k_2}, J_{k_3}$ (in proof of Theorem 5.4.2)

Now, suppose that we have a schedule of I whose makespan equals to 1. We show that A has a 3-partition. First, note that any job, J_j, has to be scheduled on at least one fast machine. This is due to the fact that $\rho_j = 1$ and, by Claim 5.4.3, $u_i < t_j$ for any slow machine; thus, any combination of slow machines can provide to J_j less than t_j processing units. Since J_j is scheduled on at least one fast machine and $a_j = 2$, J_j is scheduled on at most one slow machine. Let t_{f_j} be the total time allocated to J_j on fast machines, and let t_{i_j} be the time allocated to J_j on (at most one) slow machine. Denote by M_{i_j} the slow machine processing J_j (if J_j is processed only by fast machines then $t_{i_j} = 0$ and M_{i_j} is undefined). Let x_{i_j} be the item with the index i_j in the input for the 3-partition problem.

Lemma 5.4.4 *For any job, J_j, the processing of J_j is shared by one fast machine and one slow machine, M_{i_j} such that $t_{f_j} = \frac{K-s(x_{i_j})}{3K-B}$ and $t_{i_j} = 1 - \frac{K-s(x_{i_j})}{3K-B}$.*

Proof: By definitions of t_{f_j}, t_{i_j}, and M_{i_j},

$$\begin{cases} t_{f_j} + t_{i_j} \leq 1 \\ (3K-B)t_{f_j} + (K + s(x_{i_j}))(1 - \frac{K - s(x_{i_j})}{3K-B})^{-1} t_{i_j} = 2K \end{cases} \tag{5.3}$$

Claim 5.4.5 *for any job, J_j, $t_{f_j} \geq \frac{K-s(x_{i_j})}{3K-B}$.*

Proof: Since $\rho_j = 1\ \forall j$, for any $\varepsilon > 0$, if $t_{f_j} = \frac{K-s(x_{i_j})}{3K-B} - \varepsilon$, then from equation (5.3) we get that $t_{i_j} \leq 1 - \frac{K-s(x_{i_j})}{3K-B} + \varepsilon$, and the overall number of processing units allocated to J_j is less than $2K$. ∎

Recall that there are $3q$ jobs and $3q$ slow machines. We show now that each job is processed by exactly one slow machine, and that each slow machine processes exactly one job. This is proved by the following claim:

Claim 5.4.6 $\forall j, t_{i_j} > \frac{1}{2}$

Proof: Assume by contradiction that $t_{i_j} \leq \frac{1}{2}$ for $y > 0$ jobs. Denote by S_1 the set of $3q - y$ remaining jobs, i.e., $S_1 = \{j \mid t_{i_j} > \frac{1}{2}\}$. Then the jobs in S_1 are scheduled on exactly $3q - y$ slow machines, each running a single job in this set. Since $\forall x \in A, s(x) > \frac{B}{4}$, we have $\sum_{j \in S_1} s(x_{i_j}) < Bq - y\frac{B}{4}$. From equation (5.3), for any job $(3K-B)t_{f_j} + u_{i_j} t_{i_j} = 2K$; thus, for each of the y jobs not in S_1, $t_{f_j} \geq \frac{2K - \frac{1}{2}u_{i_j}}{3K-B}$. Also, by Claim 5.4.5 for each of the $3q-y$ jobs in S_1, $t_{f_j} \geq \frac{K-s(x_{i_j})}{3K-B}$.

Summing up the time intervals allocated to all jobs on the fast machines, we get that

$$\begin{aligned}
\sum_{j=1}^{3q} t_{f_j} &\geq \sum_{j \in S_1} \frac{K - s(x_{i_j})}{3K-B} + \sum_{j \notin S_1} \frac{2K - \frac{1}{2}u_{i_j}}{3K-B} \\
&\geq \frac{1}{3K-B}((3q-y)K - Bq + y\frac{B}{4} + 2Ky - \sum_{j \notin S_1} \frac{1}{2}u_{i_j}) \\
&= \frac{1}{3K-B}(3Kq - Bq - Ky + y\frac{B}{4} + 2Ky - \sum_{j \notin S_1} \frac{1}{2}u_{i_j}) \\
&= q + \frac{1}{3K-B}(y(K + \frac{B}{4}) - \sum_{j \notin S_1} \frac{1}{2}u_{i_j}) \quad > \quad q.
\end{aligned}$$

The last inequality follows from Claim 5.4.3 and from the assumption that there are $y > 0$ jobs for which $t_{i_j} \leq \frac{1}{2}$. However, since the makespan is 1, the total execution time on the fast q machines cannot exceed q. Thus, y equals zero and $\forall j, t_{i_j} > \frac{1}{2}$. ∎

It follows that each slow machine processes a different job. Hence, $\sum_{j=1}^{3q} s(x_{i_j}) = \sum_{i=1}^{3q} s(x_i) = qB$. Assume by contradiction that for some j, $t_{f_j} > \frac{K-s(x_{i_j})}{3K-B}$. Summing up the time intervals allocated on the fast machines, we get that

$$\sum_{j=1}^{3q} t_{f_j} > \frac{3qK - \sum_j s(x_{i_j})}{3K-B} = \frac{3qK - qB}{3K-B} = q,$$

which is, again, a contradiction to the length of the schedule.

Assigning $t_{f_j} = \frac{K-s(x_{i_j})}{3K-B}$ in equation (5.3), we get that $\forall j, t_{i_j} = 1 - \frac{K-s(x_{i_j})}{3K-B}$. ∎

Thus, in any schedule whose makespan equals to 1, J_j is allocated exactly $K+s(x_{i_j})$ processing units on some slow machine, M_{i_j}, and exactly $K - s(x_{i_j})$ on some fast machine. Given that each slow machine processes exactly one job, assume w.l.o.g that $\forall 1 \leq i \leq 3q$, M_i processes only J_i. Also, note that each fast machine must have load $3K - B$ since the total load on the q identical fast machines is $\sum_{i=1}^{3q} K - s(x_i) = (3K - B)q$.

Denote by *fast-segment* the part of a job that is processed on a fast machine. Consider a fast machine $M_i, i > 3q$. Since K was selected such that $K >> s(x) \quad \forall x \in A$, M_i has to process exactly three fast-segments (Otherwise, there exists a fast machine, M_i, which processes at least four fast-segments, but any four fast-segments require more than $u_i = 3K - B$ processing units). For each $3q < k \leq 4q$, let J_{k_1}, J_{k_2} and J_{k_3} be the three jobs whose fast-segments are scheduled on M_{3q+k}. It follows that $3K - B = 3K - (s(x_{k_1}) + s(x_{k_2}) + s(x_{k_3}))$, meaning that x_{k_1}, x_{k_2} and x_{k_3} form a triple for the partition. Since all the jobs are scheduled, the schedule of the fast-segments on the fast machines induces a valid 3-partition of A. ∎

In the case where the jobs are identical, and $\forall j$, $\rho_j < a_j - 1$, an optimal algorithm exists.

Theorem 5.4.7 *The SPAC problem has an $O(m \log m)$ optimal algorithm when the jobs are identical and $\forall j, \rho_j < a_j - 1$.*

Proof: Assume that for all jobs $t_j = t$, $\rho_j = \rho$, $a_j = a$, where $a \geq \rho + 2$. Let $w = nt/\sum_{i=1}^{m} u_i$. By Lemma 5.2.2, w is a lower bound for $w_{OPT}(I)$. We give an algorithm, $\mathcal{A}_d$, which outputs a schedule of I of length w; each job is scheduled on at most $\rho + 2$ machines.

For each machine, M_i, let Q_{M_i} denote the *potential* of M_i, that is, the number of processing units that M_i can still allocate. Initially, for each of the m machines $Q_{M_i} = u_i w$. Generally, $Q_{M_i} = u_i w'$, where w' is the total length of intervals in $[0, w]$ in which M_i is idle. Given a pair of machines M_1, M_2, we say that M_1 is weaker (stronger) than M_2, if $Q_{M_1} \leq Q_{M_2}$ ($Q_{M_1} \geq Q_{M_2}$).

Definition 5.4.1 *A* merged machine*, M', is a pair of machines M_{i_1}, M_{i_2} such that $i_1 < i_2$, and the machine M_{i_1} is idle exactly from 0 to w_0, M_{i_2} is idle exactly from w_0 to w, for some $w_0 \in [0, w]$. The potential of a merged machine is $Q_{M'} = u_{i_1} w_0 + u_{i_2}(w - w_0)$.*

Similar to the algorithm $\mathcal{A}_r$, described in Section 5.2.1, the algorithm $\mathcal{A}_d$ maintains a list, L, of the machines, sorted by their potential in non-decreasing order. That is, $Q_{L[1]} \leq Q_{L[2]} \leq \ldots$ ($L[k]$ denotes the machine at position k in L). The list L is updated along the algorithm, according to the current available machines.

The jobs are scheduled one after the other. We first describe the schedule of J_1, and then the schedule of J_j, $j > 1$. The job J_1 is scheduled on a consecutive set of machines, selected as follows. First, we examine $L[1]$, which is the weakest machine. If $Q_{L[1]} \geq t$ we schedule J_1 on $L[1]$ and update the potential of this machine; else, we check whether $Q_{L[1]} + Q_{L[2]} \geq t$, and so on until either we find a sequence of at most ρ machines with sufficient potential ($\geq t$), or the total potential of the first ρ machines is less than t. In the former case, we schedule J_1 greedily on the weakest machines until it is completed. In the latter case, we proceed to examine the next *window* of ρ machines, $L[2], \ldots, L[\rho+1]$, and so on, until our window covers ρ machines, $L[k], \ldots, L[k+\rho-1]$, such that $Q_{L[k-1]} + \ldots + Q_{L[k+\rho-2]} < t$ and $Q_{L[k]} + \ldots + Q_{L[k+\rho-1]} \geq t$ (see Figure 5.5). At this stage, we allocate to J_1 all the potential of the machines $L[k], \ldots, L[k+\rho-2]$, and complete the execution of J_1 by allocating to it also some of the potential of the machines $L[k-1]$ and $L[k+\rho-1]$ in the following way. Note that we have $w(\sum_{k-1 \leq i \leq k+\rho-2} u_{L[i]}) < t$ and $w(\sum_{k \leq i \leq k+\rho-1} u_{L[i]}) \geq t$; thus, $wu_{L[k-1]} < \frac{t}{\rho}$ and $wu_{L[k+\rho-1]} \geq \frac{t}{\rho}$, and there exists some $w_0 \in [0, w]$ such that the execution of J_1 can be completed if it is scheduled on $L[k-1]$ in the interval $[w_0, w]$ and on $L[k+\rho-1]$ in the interval $[0, w_0]$ (see Figure 5.5). Therefore, J_1 is scheduled on $\rho+1$ machines, and in each moment, at most ρ machines process it simultaneously.

The machines $Q_{L[k]}, \ldots, Q_{L[k+\rho-2]}$ are removed from L. The two machines $L[k-1]$ and $L[k+\rho-1]$ are partially used and form together one *merged machine*, M', which is idle in $[0, w]$ and whose potential is $Q_{M'} = u_{L[k-1]}w_0 + u_{L[k+\rho-1]}(w - w_0)$. This merged machine replaces $L[k-1]$ and $L[k+\rho-1]$ in L.

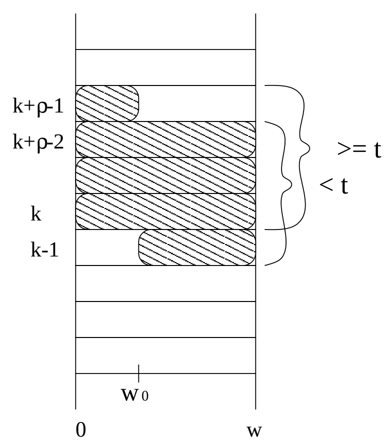

Figure 5.5: Scheduling the first job (in proof of Theorem 5.4.7)

For any job J_j, $j > 1$: if the total potential of the weakest ρ machines is at least t, then J_j and all the remaining jobs, are scheduled greedily starting from the weakest machine; otherwise, as in the schedule of J_1, we scan the list L using a moving window that covers ρ machines, until we find the weakest consecutive set of ρ machines that can complete the execution of J_j (the merged machine is considered as a single machine in L). We show that at any stage of the algorithm, the list L contains at most one merged machine.

Let J_b be the first job such that, when J_b is scheduled, the total potential of the weakest ρ machines is at least t (that is, $J_b, \ldots, J_n$ are scheduled greedily). We distinguish between two phases of $\mathcal{A}_d$:

1. The jobs $J_1, \ldots, J_{b-1}$ are scheduled.

2. The jobs $J_b, \ldots, J_n$ are scheduled.

Claim 5.4.8 *For all $1 < j \leq b$, before J_j is scheduled, the list L contains at most one merged machine composed of idle-segments of two machines, M_{i_1} and M_{i_2}, such that $wu_{i_1} < \frac{t}{\rho}$ and $wu_{i_2} \geq \frac{t}{\rho}$.*

Proof: The proof is by induction on j. For $j = 2$, we showed that the claim holds after J_1 is scheduled. For the induction step, assume that the claim holds for all $j \leq k$. Let M_{i_1} and M_{i_2} be the two machines composing the merged machine before J_k is scheduled. We show that the schedule of J_k must include some intervals on both M_{i_1} and M_{i_2}. As illustrated in Figure 5.6(a) and (b), the only possible ways to schedule J_k is to select a set of machines that encircle the "hole" created by the machines that were previously used, and omitted from L (The hole is shaded with lines). This follows from the fact that J_k is scheduled on the first possible consecutive set of machines in L. Assume that J_k is scheduled only on machines that are weaker than M_{i_1} (Figure 5.6(c)). By the induction hypothesis $wu_{i_1} < \frac{t}{\rho}$ and thus, the execution of J_k is done solely on machines whose rates are less than $\frac{t}{\rho}$. Clearly, no combination of such ρ machines can complete J_k in w time units. Similarly, the schedule of J_k cannot be done solely on machines which are stronger than M_{i_2} (Figure 5.6(d)), since $wu_{i_2} \geq \frac{t}{\rho}$. We note that if $wu_{i_2} = \frac{t}{\rho}$ and the ρ machines following M_{i_2} in L have the rate $\frac{t}{w\rho}$, then we may not schedule J_j on M_{i_1}, but in this case the schedule of J_k will be concatenated to that of J_{k-1} and the claim holds.

Given that the window contains both M_{i_1} and M_{i_2}, we get that the new merged-machine must replace the old one. The new-merged machine is composed of two machines, such that the slower one has rate at most u_{i_1} and the faster one has rate at least u_{i_2}, therefore, the claim holds also before the schedule of J_{k+1}. ■

We turn now to consider the number of machine allotments of a job J_j, $j > 1$. If J_j is scheduled greedily, then, at most ρ machines in L share its execution. By Claim 5.4.8, at most one of these machines may be a merged-machine; hence, we get that J_j is scheduled on at most

$\rho + 1$ machines (no merged-machines are composed during the second phase). Assume that J_j is scheduled in the first phase, using the moving window. Let $L[k], \ldots, L[k+\rho-1]$ be the set of ρ machines such that $Q_{L[k-1]} + \ldots + Q_{L[k+\rho-2]} < t$ and $Q_{L[k]} + \ldots + Q_{L[k+\rho-1]} \geq t$. As in the schedule of J_1, we now allocate to J_j all the potential of the machines $L[k], \ldots, L[k+\rho-2]$, and complete its execution by allocating to it also some of the potential of the machines $L[k-1]$ and $L[k+\rho-1]$. This allocation is done such that the idle intervals of $L[k-1]$ and $L[k+\rho-1]$ form a merged machine. Overall, at most $\rho+1$ machines from L participate in the execution of J_j. By Claim 5.4.8 at most one of these machines may be a merged one, thus, J_j is scheduled on at most $\rho + 2$ machines.

Also, the parallelism constraint of J_j is preserved: if J_j is scheduled in the first phase, then it runs on $\rho - 1$ machines $L[k], \ldots, L[k+\rho-2]$ in the interval $[0, w]$; also, J_j is scheduled on $L[k-1]$ and $L[k+\rho-1]$ in non-overlapping time intervals. Finally, if one of the machines is 'merged', then J_j is scheduled on the corresponding two machines in two distinct time intervals. It follows that at any time, J_j is processed in parallel by at most ρ machines. The argument is similar for jobs J_j that were scheduled greedily.

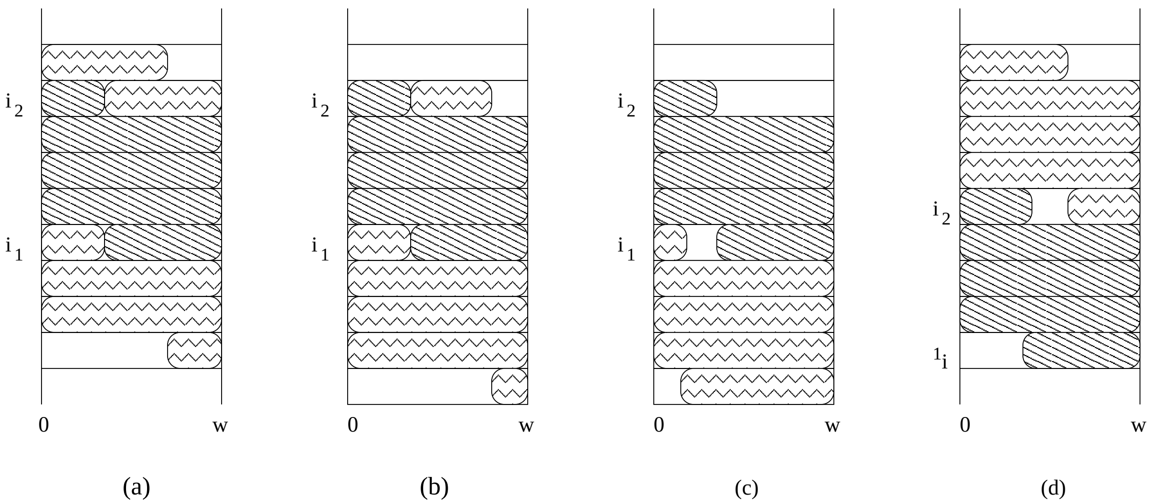

Figure 5.6: Scheduling a job J_j, $j > 1$ (in proof of Theorem 5.4.7)

Finally, we show that we never fail to schedule a job, that is, the set of strongest ρ machines in L can always complete the execution of a job. Recall that $w = nt/\sum_{i=1}^{m} u_i$. Therefore, for each $1 \leq j \leq n$, $w \geq jt/\sum_{i=1}^{j\rho} u_i$ (assuming that $u_1 \geq u_2 \geq, \ldots, \geq u_m$). This implies that for any j, the set of first j jobs can be completed by the $j\rho$ fastest machines. In terms of $\mathcal{A}_d$, the window never has to move beyond the ρ fastest available machines.

The algorithm $\mathcal{A}_d$ can be implemented in time $O(\max(m \log m, n))$. $O(m \log m)$ steps are needed for sorting the machines. Next, the total time for scheduling the jobs is $O(m+n)$. Recall that the schedule of each job J_j is done on machines that encircle in L the hole created by the machines that process J_{j-1}. Hence, we do not need to scan the list L from the beginning for each job. Also, when the window covers the $\rho_j + 1$ machines that will share the execution of J_j, the calculation of the new merged-machine and the updates in L take $O(1)$. ∎

Chapter 6

Summary

6.1 The Contribution of this Work

In this work we introduced the problems of class-constrained packing, and scheduling with machine allotment and parallelism constraints. We gave the first results for these problems. We showed that these results can be used to solve some fundamental resource allocation problems in multimedia-on-demand systems, in production planning, and in scheduling parallelizable jobs.

In most cases, our results are the best one could expect. Specifically, we gave optimal algorithms for instances that are polynomially solvable (e.g., packing with uniform capacity ratio, or scheduling on identical machines); FPTASs, for problems that are weakly NP-hard (such as 0-1 class-constrained knapsack); and PTASs, for those that are strongly NP-hard (e.g., CCBP, CCMK, and SPAC). For the online problems, we derived tight lower and upper bounds.

Some of the technical tools developed in this work can be useful in future related work. In particular, the *moving window* algorithm is generic: it is used here to solve both packing and scheduling problems (For instances with uniform capacity ratio, this algorithm was shown to be optimal [24]). The technique of eliminating small items by grouping, presented in Chapter 3 is suitable for any packing problem in which handling the small items is complex; another idea is to transform any instance of multiple knapsack to an instance that contains $O(\log n/\varepsilon)$ distinct bin types.

Some of the results obtained in this study show a clear and sometimes surprising distinction between classical packing and scheduling problems, and their class-constrained versions. Most notably, simple greedy algorithms which are known to be optimal in traditional packing problems were shown to perform poorly in the presence of class-constraints (e.g., 'pack small items first', when all the items have the same value, or any-fit algorithms for online packing). Our scheduling problem was studied in both *uniform* and *identical* machine environments. For many fundamental scheduling problems, similar solvability/approximability results are known for these two environments [1]. Our study distinguishes between these two environments with respect to the solvability of the SPAC problem.

Finally, through the relation of the SPAC problem to classic *preemptive* scheduling on multiple machines, this work sheds some fresh light on what is known in scheduling folklore as the *power of preemption*. Specifically, we showed that, even if all the jobs or the machines are *identical*, with the addition of machine allotment constraints, preemptive scheduling becomes strongly NP-hard.

6.2 Future Work

We leave open several interesting avenues for future work:

- In Chapter 3, we presented a PTAS for CCBP and CCMK with items of arbitrary sizes, and an FPTAS for a single knapsack. The PTASs are suitable for instances in which M is fixed. On the other hand, our FPTAS is suitable also for instances with $M = \Omega(|I|)$. Is there a PTAS for instances with multiple knapsacks, where M is accepted as part of the input?
- For the generalized class-constrained problems, we gave in Section 3.4 APX-hardness proofs. It would be interesting to study offline and online algorithms for these problems.
- For the online CCBP and CCMK we considered only instances with *unit-sized* items. Can similar lower and upper bounds be obtained for instances in which items are of *arbitrary sizes*?
- Can randomization help in solving online class constrained packing problems?
- In Section 5.3, we gave a PTAS for the SAC problem. Our PTAS can be used for instances in which the maximal allotment parameter of any job is some fixed constant. Is there a PTAS for instances with arbitrary allotment parameter?
- In the SPAC problem, determining the minimal difference $(a_j - \rho_j)$, required by any efficient algorithm, for *general* instances, remains an open question. In particular, can we optimally solve the classic preemptive scheduling problem, when $\forall j,\ a_j = 3$? for larger *constant* values of a_j?

The following issues refer to the application of our results to storage management:

- The CCMK and the CCFP can be applied for the *static* assignment of files to the disks. Static assignment is only the first component of a MOD resource allocation scheme, in which the initial state of the system is defined; it is then followed by a *dynamic phase*, in

[1] For example, the makespan problem is optimally solvable in both environments when preemptions are allowed [48, 31, 26], and has a PTAS in both environments when preemptions are not allowed [29, 30, 15].

which customer requests arriving to the system need to be serviced. During the dynamic phase, the popularities of the various files can change. Such changes are natural, e.g., when dealing with video data available on the world-wide-web. In response, the MOD system should support operations such as file deletions or replications, as well as reallocations of load. This introduces a semi-online version of each of our packing problems, where the sizes of the sets that we would like to pack can change dynamically, and we need to update the placement accordingly. The transition from one placement to another should be done with minimal number of reallocations of items to compartments.

- We have not considered the offline CCBP problem with unit-sized items (the online CCBP is studied in Chapter 4). In terms of storage management in multimedia systems, we get a *system configuration* problem; that is, assuming that the total number of requests for broadcasts, and the popularities of the video files are known in advance, we need to satisfy all the requests, using as few (identical) disks as possible. In the more general problem, the bins may be of several different types, where each type is characterized by specific volume, number of compartments, and a fixed cost. Our objective is to pack all the items in a set of bins at minimal cost.

- Finally, storage management schemes should address also reliability issues, e.g., the case where a disk crashes (a knapsack is removed), and all the load that it receives (the items it contains) need to be transferred to other disks (to be packed in other knapsacks).

Publications

Journal Papers

i. Shachnai H. and Tamir T., *On Two Class-Constrained Versions of the Multiple Knapsack Problem*, **Algorithmica**, vol. 29, pp. 442–467, 2001.

ii. Shachnai H. and Tamir T., *Polynomial Time Approximation Schemes for Class-Constrained Packing Problems.* To appear in the **Journal of Scheduling** (invited CONF 2000 issue).

iii. Shachnai H. and Tamir T., *Multiprocessor Scheduling with Machine Allotment and Parallelism Constraints.* To appear in Algorithmica.

Conferences

iv. Shachnai H. and Tamir T., *Noah bagels - Some Combinatorial Aspects*, **International Conference on FUN with Algorithms (FUN)**, Isola d'Elba, June 1998 (A preliminary version of [*i*]).

v. Shachnai H. and Tamir T., *Polynomial Time Approximation Schemes for Class-Constrained Packing Problems* **The 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)**, Saarbrücken, September 2000 (A preliminary version of [*ii*]).

Bibliography

[1] C. Aggarwal, J. Wolf, and P.S. Yu. On optimal piggyback merging policies for video-on-demand systems. In *Proc. Sigmetrics*, pages 200–209, 1996.

[2] N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid. Approximation schemes for scheduling. In *Proc. of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 493–500, 1997.

[3] L. Babel, B.Chen, H.Kellerer, and V.Kotov. On-line algorithms for cardinality constraint bin packing problems. Technical report, Institut fuer Statistik und Operations Research, Universitaet Graz, 2001.

[4] J. Blazewick, M. Drabowski, and J. Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Comput*, 35(C):389–393, 1986.

[5] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, 1998.

[6] P. Brucker and S. Knust. Complexity results of scheduling problems. http://www.mathematik.uni-osnabrueck.de/research/OR/class/.

[7] A. Caprara, H. Kellerer, U. Pferschy, and D. Pisinger. Approximation algorithms for knapsack problems with cardinality constraints. *European Journal of Operations Research*, 123:333–345, 2000.

[8] A.K. Chandra, D.S. Hirschberg, and C.K. Wong. Approximate algorithms for some generalized knapsack problems. *Theoretical Computer Science*, 3:293–304, 1976.

[9] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *Proc. of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 185–194, 1999.

[10] C. Chekuri and S. Khanna. A ptas for the multiple knapsack problem. In *Proc. of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000.

[11] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. Raid: High performance, reliable secondary storage. *ACM computing surveys*, 26(2):145–185, 1994.

[12] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proc. of the second ACM International Conference on Multimedia*, October 1994.

[13] W. J. Davis, D. L. Setterdahl, J. G. Macro, V. Izokaitis, and B. Bauman. Recent advances in the modeling, scheduling and control of flexible automation. In *Proc. of the Winter Simulation Conference*, pages 143–155, 1993.

[14] X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *Proc. of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 159–167, 1996.

[15] L. Epstein and J. Sgall. Approximation schemes for scheduling on uniformly related and identical parallel machines. In *Proc. of the 7th European Symposium on Algorithms*, volume 1643 of *Lecture Notes in Computer Science*, pages 151–162. Springer-Verlag, 1999.

[16] C.E. Ferreira, A. Martin, and R. Weismantel. Solving multiple knapsack problems by cutting planes. *SIAM J. on Opt*, 6:858 –877, 1996.

[17] A. Fiat and G.J. Woeginger. *Online Algorithms: The State of the Art.* LNCS. 1442, Springer-Verlag, 1998.

[18] A.M. Frieze and M.R.B. Clarke. Approximation algorithms for the m-dimensional $0-1$ knapsack problem: Worst-case and probabilistic analyses. *European Journal of Operational Research*, 15:100–109, 1984.

[19] M.R. Garey and D.S. Johnson. Strong np-completeness results: Motivations, examples, and implications. *Journal of the ACM*, 25:499–508, 1978.

[20] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[21] M.R. Garey, R.L.Graham, and J.D.Ullman. Worst-case analysis of memory allocation algorithms. In *Proc. of the 4th ACM Symposium on theory of Computing*, 1972.

[22] G.V. Gens and E.V. Levner. Computational complexity of approximation algorithms for combinatorial problems. In *Proc. of the 8th International Symp. on Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 292–300. Springer-Verlag, 1979.

[23] S. Ghandeharizadeh and R.R. Muntz. Design and implementation of scalable continuous media servers. *Parallel Computing Journal*, 24(1):91–122, 1998.

[24] L. Golubchik, S. Khanna, S. Khuller, R. Thurimella, and A. Zhu. Approximation algorithms for data placement on parallel disks. In *Proc. of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 223–232, 2000.

[25] L. Golubchik, J.C.S Lui, and R.R. Muntz. Adaptive piggybacking: a novel technique for data sharing in video-on-demand storage servers. *ACM Multimedia Systems Journal*, 4(3):140–155, 1996.

[26] T. Gonzalez and S. Sahni. Preemptive scheduling of uniform processor systems. *Journal of the ACM*, 25:92–101, 1978.

[27] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:263–269, 1969.

[28] D.S. Hochbaum. *Approximation Algorithms for NP-Hard Problems.* PUS Publishing Company, 1995.

[29] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *Journal of the ACM*, 34(1):144–162, 1987.

[30] D.S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal of Computing*, 17(3):539–551, 1988.

[31] E.G. Horvath, S. Lam, and R.Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24:32–43, 1977.

[32] M.S. Hung and J.C. Fisk. A heuristic routine for solving large loading problems. *Naval Research Logistical Quarterly*, 26(4):643–650, 1979.

[33] T. Ibaraki and N. Katoh. *Resource Allocation Problems - Algorithmic Approaches.* The MIT Press, 1988.

[34] O.H. Ibarra and C.E. Kim. Fast approximation for the knapsack and the sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[35] G. Ingargiola and J. F. Korsh. An algorithm for the solution of 0-1 loading problems. *Operations Research*, 23(6):110–119, 1975.

[36] D.S. Johnson. *Near-optimal bin packing algorithms.* PhD thesis, MIT, Cambridge, MA, 1973.

[37] D.S. Johnson. Fast algorithms for bin packing. *J. Comput. System Sci.*, 8:272–314, 1974.

[38] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithm. *SIAM Journal of Computing*, 3:256–278, 1974.

[39] M. Kamath, K. Ramamritham, and D. Towsley. Continuous media sharing in multimedia database systems. In *Proceedings of the fourth Intl. Conference on Database Systems for Advanced Applications*, Singapore, 1995.

[40] V. Kann. Maximum bounded 3-dimensional matching is max snp-complete. *Information Processing Letters*, 37:27–35, 1991.

[41] N. Karmakar and R.M. Karp. An efficient approximation scheme for the one-dimensional bin packing problem. In *Proc. of the 23rd Symp. on Foundations of Computer Science*, pages 312–320, 1982.

[42] H. Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In *Proc. of APPROX'99*, Lecture Notes in Computer Science, pages 51–62. Springer-Verlag, 1999.

[43] H. Kellerer and U. Pferschy. Cardinality constrained bin-packing problems. *Annals of Operations Research*, 92:335–349, 1999.

[44] K.L. Krause, V.Y. Shen, and H.D. Schwetman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *Journal of the ACM*, 22(4):522–550, October 1975.

[45] P.W.K. Lie, J.C.S. Lui, and L. Golubchik. Threshold-based dynamic replication in large-scale video-on-demand systems. In *Proceedings of the Eighth International Workshop on Research Issues in Database Engineering (RIDE)*, pages 52–59, Orlando, February 1998.

[46] E. Y-H Lin. A bibliographical survey on some well-known non-standard knapsack problems. *Information Systems and Oper. Research*, 36(4):274–317, 1998.

[47] S. Martello and P. Toth. Algorithms for knapsack problems. *Annals of Discrete Math.*, 31:213–258, 1987.

[48] R. McNaughton. Scheduling with deadlines and loss functions. *Manage. Sci.*, 6:1–12, 1959.

[49] R. Motwani. Lecture notes on approximation algorithms. Technical report, Dept. of Computer Science, Stanford Univ., CA, 1992.

[50] H. Shachnai and T. Tamir. Noah bagels - some combinatorial aspects. In *Proc. of the International Conference on FUN with Algorithms (FUN)*, Isola d'Elba, June 1998.

[51] D.B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming A*, 62:461–474, 1993.

[52] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison-Wesley, 5th edition, 1998.

[53] A. van Vliet. On the asymptotic worst case behavior of harmonic fit. *J. of Algorithms*, 20:113–136, 1996.

[54] W.F. Vega and G.S. Leuker. Bin packing can be solved within $1 + \varepsilon$ in linear time. *Combinatorica*, 1:349–355, 1981.

[55] J.L. Wolf, P.S. Yu, and H. Shachnai. Scheduling issues in video-on-demand systems. *Multimedia Information Storage and Management*, pages 183–207, 1996.

[56] J.L. Wolf, P.S. Yu, and H. Shachnai. Disk load balancing for video-on-demand systems. *ACM Multimedia Systems Journal*, 5:358–370, 1997.

[57] A. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980.