RC 21027 (94165) November 17, 1997
Computer Science/Mathematics

IBM Research Report

Approximation Algorithms for Grouping Problems

Henning Biermann
Jayant R. Kalagnanam

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York 10598

Research Division
Almaden · Austin · T.J. Watson · Beijing · Haifa · Tokyo · Zurich

Approximation Algorithms for Grouping Problems

Henning Biermann*, Jayant Kalagnanam†

International Business Machines
T. J. Watson Research Center
Yorktown Heights, NY 10598

May 14, 1998

Abstract

Grouping is encountered frequently in material design problems where an order book is used to create manufacturable units. Typically, these problems can be formulated as variations of bin packing problems. In the standard problem we have to pack a given set of items into as few bins as possible considering a weight constraint.

We study variants with additional assignment constraints for items and bins of certain types. The objective can be to minimize the number of used bins as well as minimizing the unused portions of the bins.

We apply some well known heuristics to these variants and study their performance. Moreover, we develop an approximation algorithm based on a set cover formulation of the problem. The performance of the method is discussed in a worst case study and some computational results for real world problem instances.

The algorithms are designed and implemented to fit into the framework of a workbench for rapid prototyping of optimization problems.

1 Introduction

A large number of optimization problems encountered in material design applications can be formulated as *grouping problems*. In a typical application, we are given an order book and have to group orders to create manufacturable units which can subsequently be scheduled for production.

*biermann@cs.nyu.edu
†jayant@watson.ibm.com

These real world problems are generally highly constrained since we can't group different orders arbitrarily. Moreover, the solution should optimize multiple objectives, such as minimizing the number of produced material units while maximizing the used portion of each material unit.

This paper is organized as follows: The next two sections give an overview about research on the related *bin packing* and *set cover* problems and study how various constraints and objectives translate to standard problems.

Afterwards, we design an algorithm with respect to a wide range of grouping problems. The complexity of this algorithm is analyzed and the performance is studied in general and some special cases.

Though this work was motivated by a slab design problem in steel production, the algorithms are flexible enough to be part of a software library for combinatorial optimization problems.

1.1 Problem Formulation

In the context of steel production, we encounter a *slab design problem* where the goal of this design is to minimize the number of units that need to be manufactured for a given order book. The maximum size of the slabs allowed in this design is constrained for order sets. This problem can be modeled as a grouping problem as follows:

The order book specifies what steel units are ordered in which amounts. Steel units (*items*) have certain size and "quality" requirements. We refer to orders of the same kind as *item types*.

Slabs are available in different sizes (*slab types*). We assume that the number of slabs per type is not limited. This idealizes the case of a large inventory.

The assignment of ordered items to slabs is restricted by *assignment constraints*: For each slab type there is a fixed set of compatible item types. Quality requirements are expressed in terms of *colors constraints*: Each item type specifies a color, and there is a limit on the number of different colors per slab.

We are looking for a *packing* of items into slabs, which minimizes the total number of slabs.

We also study the objective to minimize the total unused portion of the slabs.

2 Bin Packing

2.1 Formulation, Definitions, Results

In the classical bin packing problem we are given a sequence $L = (a_1, a_2, \ldots, a_n)$ of *items*, each with a *size* $s(a_i) \in (0, 1]$ and are asked to pack them into a minimum number of unit-capacity *bins*. This is to find a partition $B_1, \ldots, B_m$ such that $\sum_{a \in B_j} s(a) \leq 1, \ 1 \leq j \leq m$.

This problem has many real-world applications, from our slab-order problem to packing television commercials into station breaks [Bro71].

The bin packing problem is *NP*-complete (i.e. [GJ79]), and it was one of the first problems for which approximation algorithms have been studied.

Bin packing is extensively covered in the literature. The recent survey [CGJ97] describes the various results for the one dimensional problem covering online and offline algorithms, worst case analysis and average case analysis. An previous survey [CGJ84] elaborates on the variety of problems and covers work through 1984.

In this section, we describe a few insights about bin packing which are interesting from an implementation point of view.

2.2 Lower Bounds and Exact Solutions

While most of the research is done on approximation algorithms Marcello and Toth explored algorithms for exact solution. [MT89] explains how to get stronger lower bounds than the trivial relaxation based bound

$$\frac{\sum_{i=0}^{n} s(a_i)}{C} \leq OPT(L).$$

Also, a dominance criterion is developed to reduce the number of items. The authors apply these techniques in a branch and bound approach. The presented implementation finds an optimum packing very fast, even for large problem instances ($n = 1000 : \ t \leq 20s$).

2.3 Heuristics

Next Fit

This is perhaps the simplest heuristic, first described in [Joh73]. It's a bounded-space online algorithm, which considers the items in input order L, and keeps track of one partially filled ("open") bin.

We put the current item a into the open bin if its residual capacity is at least $s(a)$. Otherwise, we close this bin, open a new one and put a into this bin. We continue with the next item until L is processed completely.

Therefore, Next Fit (NF) has a running time linear in the number of items. It's a bounded space, online algorithm.

It's not difficult to see that for all lists L, $NF(L) \leq 2OPT(L) - 1$, this means that Next Fit finds a solution which is less than twice the optimum for the list L. The key observation is that the sum of the item sizes in two adjacent bins is more than the bin capacity. Thus, $NF(L)/2 < \sum_{i=1}^{n} s(a_i) \leq OPT(L)$.

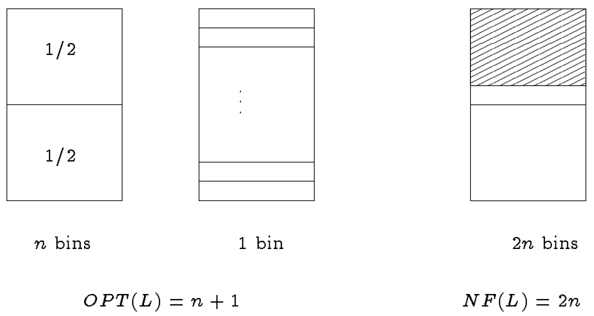

Figure 1: Worst case examples for Next Fit: $L = \left(\frac{1}{2}, \frac{1}{2n}, \frac{1}{2}, \ldots, \frac{1}{2}, \frac{1}{2n}\right)$

First Fit

The algorithm First Fit (FF) considers the items in input order $L = (a_1, \ldots, a_n)$ and keeps a list of all bins used so far. Whenever we have to pack an item a, we scan the bins in the order in which they were opened. Item a is put into the first bin which has a residual capacity of at least $s(a)$. In case there is no such bin, a new one is opened.

First Fit can be implemented to take $O(n \log n)$ time. In contrast to NF the required space isn't bounded for this online algorithm.

It can be shown that for all lists L, $FF(L) \leq \lceil (17/10) OPT(L) \rceil$ [GGJY76].

First Fit Decreasing and Variants

The algorithm First Fit Decreasing (FFD) performs FF after preprocessing the input list L, where the items are sorted in decreasing size order. Therefore, FFD has the same time complexity as FF, but it's an offline algorithm.

The asymptotic approximation ratio of this algorithm is known to be $11/9$ due to [Joh73]. Examples of worst case lists L, for with $FFD(L) = 11/9\, OPT(L)$ can be found in [CGJ97].

There are some similar offline algorithm (Best Fit Decreasing, ...) with the same approximation ratio.

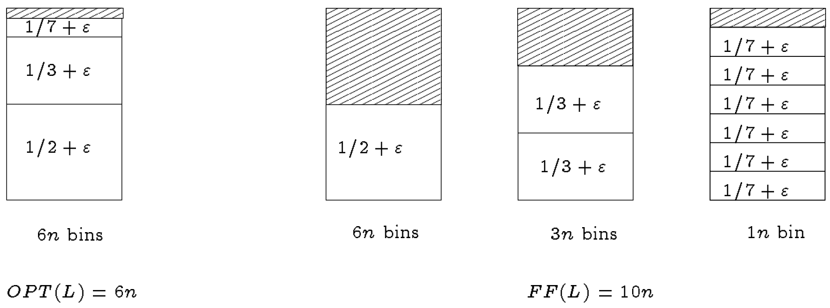

Figure 2: Bad case example for First Fit: The approximation ratio is 10/6. $L = (\frac{1}{7}+\varepsilon, \ldots, \frac{1}{7}+\varepsilon, \frac{1}{3}+\varepsilon, \ldots, \frac{1}{3}+\varepsilon, \frac{1}{2}+\varepsilon, \ldots, \frac{1}{2}\varepsilon)$

2.4 Variations

Variable Bin Size

In this generalization of bin packing we are allowed to use bins of different capacities. A problem instance consists of the item list L and the list of available bin capacities $C_1, \ldots, C_m$. The objective is to find a packing that minimizes the unused portion of the bins.

This bin packing problem has been studied in [FL86].

Color Constraints

In the context of the slab-order problem we are dealing with *color constraints*. Each item a_i, $1 \leq i \leq n$, is assigned a color $c(a_i) \in \{1, \ldots, k\}$, where k is the number of all possible colors.

We want to pack the items into bins $B_1, \ldots, B_m$ such that we meet the capacity constraints. Moreover, and the number of colors in each bin is at most a given constant K. This means,

$$\left| \cup_{a_i \in B_j} c(a_i) \right| \leq K, \quad 1 \leq j \leq m$$

.

The objective is to use as few bins as possible.

All heuristics described above can be applied to this problem by taking the color constraints into account. It's easy to see that for any list L,

$$FF_{col}(L) \leq 2\ OPT(L) + k.$$

We consider the bins which have a residual capacity of at least $1/2$. Each item color can appear in at most one of these bins. Therefore the number of half filled bins is at most k.

This inequation bounds the performance of First Fit for color constraints by comparing it to the optimum in the uncolored case. It's not clear that this bound is tight at all.

Assignment Restrictions

Assignment restrictions between bins and items can be represented by a bipartite graph. Items and bins correspond to vertices. Edges connect bins and items which can be assigned.

It's easy to see that assignment restrictions change the problem drastically. The presented heuristics can't guarantee any approximation ratio for the restricted case, as illustrated in figure 3.

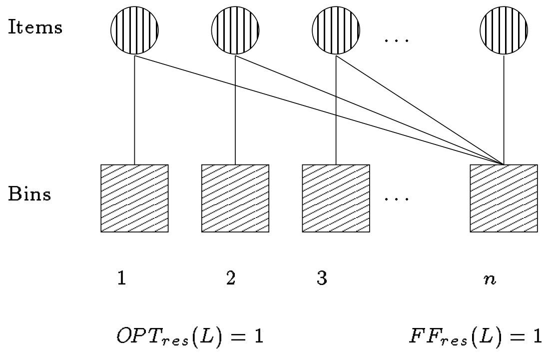

Figure 3: Bin packing with assignment restriction: unbounded approximation ratio for FF. $L = (1/n, \ldots, 1/n)$.

The same example suggests the set cover approach for the general grouping problem, which is explained in section 4.

2.5 Results and Questions

It's interesting to see how the colored bin packing problem is related to the other bin packing problems. Depending on the maximum number of colors per bin K, the problem is quite different:

- K is larger than the maximum number of items fitting into a single bin: the color constraints don't matter at all.
- $k = n$, no two items have the same color: The color constraints correspond a limited number of K items per bin. This problem is studied in [KSS75]. By modifying FFD it's possible to obtain an asymptotic approximation ratio 2.

- $K = 1$: The colored problem reduces to k instances of the bin packing problem, by partitioning the items corresponding to their color.

This observation leads to the question how the solution changes depending on K for a fixed set of k colors. A bad case example can be derived from figure 2: Assume that items of different sizes are colored differently. For $K = 3$ we need $6n$ bin. At least $7n$ bins are necessary for $K = 2$, and $K = 1$ allows packing with at least $10n$ bins.

Some computational results for random instances are listed in figure 4. As expected, the results for colored and uncolored instances are very similar if the item set is large. An intuitive explanation is the uniform distribution of colors and item sizes: There are items of similar size for every color.

Similarly, one could ask for upper bounds: What is the worst coloring of the items of a given instance L, i.e. which coloring maximizes the number of necessary bins.

Lower bounds are interesting for theoretical as well as practical reasons. Possible applications are branch and bound algorithms. Relaxing the integrality of item sizes leads to an integer program where items can be split among multiple bins, while bins still suffice the color constraints. It's not clear how to relax these constraints, but solving an integer program at every node in the search tree isn't very desirable.

Moreover, it would be useful to develop a dominance criterion similar to the one in [MT89], which allows cutting branches in the search tree very efficiently.

n	$K = \infty$		$K = 2$		$K = 1$	
	FF	FFD	FF	FFD	FF	FFD
50	20	18	19	21	26	26
100	31	30	34	33	39	38
200	62	60	65	64	71	72
400	132	129	135	133	149	141
800	252	247	258	250	250	252

Figure 4: Performance of bin packing heuristics for random instances. All sizes and colors are chosen uniformly at random: $s(a) \in \{20, \ldots, 100\}, c(a) \in \{1, \ldots, 20\}$. The bin capacity is $C = 200$. Shown are the numbers of bins used by FF and FFD considering color constraints for $K = 1, 2, \infty$ (uncolored case).

3 Set Cover

3.1 Formulation, Definitions, Results

In the classical set cover problem we are given a set F of subsets of a "target set" U, i.e. $F = \{S_1, \ldots, S_n\}, \quad S_i \subset U, \quad 1 \leq i \leq n$. The question is to find a minimum collection $B \subset F$ such that every element of U is *covered* by B:

$$|B| = \min\left\{|V| : V \subset F, \ \bigcup_{S \in V} S = U\right\}$$

.

This problem can be generalized to the *weighted set cover*. In this problem we are additionally given a weight w_i for each set S_i, $1 \leq i \leq n$. The objective is to find a minimum weight cover, i.e.

$$\min \sum_{S \in V} w(S) \quad \text{subject to } V \subset F, \bigcup_{S \in V} S = U.$$

The set cover problem is know to be *NP*-complete, i.e. [GJ79]. Moreover, it's Max-$\mathcal{SNP}$-complete, which means that there are no polynomial approximation schemes, unless $P = NP$.

Set cover problems are related to other cover problems like *vertex cover, independent subset, set packing, multicover*. A recent survey of summarizes the research on approximation algorithms for covering and packing problems and focuses on a linear programming approach in particular [Hoc97].

In our application we make use of Chvátal's *Greedy Set Cover* algorithm [Chv79], which can be easily implemented and guarantees a best possible approximation ratio.

3.2 Greedy Algorithm

The greedy algorithm starts with an empty set B. In every iteration it picks largest set S contained in the selection F. S is added to the cover B and all items of S are removed from F. This is repeated until all items are removed from F, i.e. $F = \{\emptyset\}$. Therefore, the greedy algorithm always chooses set which contains the largest amount of "uncovered" items.

It can be shown that this heuristic is an $\mathcal{H}(d)$ approximation, where d denotes the size of the largest set in F. The approach was extended to find a weighted matching in [Chv79]. For this objective, we pick subsets with minimum ratio of weight to the number of its "uncovered" items during the iteration.

A different approximation algorithm was devised for the set cover problem by Hochbaum [Hoc82]. The algorithm is based on the dual of the linear programming formulation of the problem. It can be shown that these algorithms have the same approximation ratio $\mathcal{H}(d)$.

These results match the recently proved lower bound for approximating the set cover [Fei95].

4 Grouping

This section finally addresses the general grouping problem. We derive a set cover formulation for the slab-order problem and use the greedy algorithm to find an approximation.

4.1 Set Cover Formulation

Consider a given instance of the slab-order problem with

- item-types $O = \{a_1, \ldots, a_N\}$,
- slab-types $S = \{B_1, \ldots, B_M\}$,
- sizes $s(a_i)$ and required amounts $n(a_i)$ of items $1 \leq i \leq N$,
- slab capacities $C(B_j)$ and assignment constraints $A(B_j) \subset O, 1 \leq j \leq M$.

We want to cover the set U which contains $n(a_i)$ items of type $a_i, \quad 1 \leq i \leq N$. (The upper index points out that we have multiple copies of the same element in U.)

$$U = \{a_1^1, \dots, a_1^{n_1}, \dots, a_N^1, \dots, a_N^{m_N}\}$$

The collection F of item sets which fit into single slabs is given by

$$F = \left\{ V \subset U \mid \exists B_j : \bigcup_{a \in V} a \subset A(B_j), \sum_{a \in V} s(a) \leq C(B_j) \right\}$$

The objective is to find a minimum size subset S of F which covers U. This corresponds to minimizing the number of used slabs. It's important to consider that set F could be very large, since capacity constrains are expressed by enumerating all subsets which meet this constraint, possibly exponentially many.

4.2 Approximation Algorithm

We approximate the solution for a slab-order instance using the greedy algorithm for it's equivalent set cover formulation. The greedy algorithm described in section 3.2, but it's possible to implement this method using an implicit compact representation of F.

In each iteration, the greedy algorithm chooses the largest set $V \in F$ for the cover. This corresponds to picking the slab type $B \in \{B_1, \dots B_M\}$ which can contain the largest set of items $V \subset U$.

It's easy to determine a maximal item set V_j for a given bin type B_j: We arrange all items $a \in U$ which are allowed for B_j in nondecreasing size order. Now, we add item for item to the initially empty set V as long as the total size of V is less than $C(B_j)$.

After we have determined the maximum slab type B_j, all items in V_j are packed into a slab of this type and removed from U. These steps are repeated until all elements in U are packed (i.e. $U = \emptyset$).

4.2.1 Complexity

Memory

We need to store the size and the ordered amout of items and some capacity information for the slab types. This requires $O(M+N)$ space.

Also, assignment constraints between slab types and item types need to be represented. Each slab type has a list of item types which can be assigned. This is similar to an adjacency list representation of a bipartite graph. The storage requirement depends on the number of edges in the graph, but is at most $O(MN)$

Running Time

The algorithm takes $O(N \log N)$ steps to sort the N item types. Each iteration takes $O(MN)$ steps, since the maximum item set is determined for each of the M slabs types. It's possible to implement the algorithm such that it's performing at most N iterations. Thus, the total running time is $O(N^2 M + N \log N)$.

This means that the running time is depending on the number of slab and item *types* and not the total number of *items*.

This implementation is based on the following observation: while iterating, we choose the same slab type as long as U contains the corresponding maximum subset. This means that the coice of the slab type changes only if an element type is exhausted. Only then, we have to search for a new maximum slab type.

4.2.2 Approximation

We have explained that the greedy heuristic for set cover is a $\mathcal{H}(d)$-approximation algorithm. This result ensures that the algorithm above for the slab-order problem is also an $\mathcal{H}(d)$-approximation. In other words, the result is at most $O(\log d)$ worse than the optimum, where d denotes the maximum number of items which can be packed into any item type.

Approximability

This ratio matches the lower bound for approximating the solution, since we can formulate set cover problem instances $(U,\ F = \{S_1, \ldots, S_M\})$ in our framework: $(n(a) = 1, s(a) = \varepsilon$ for $a \in U, \quad A(B_j) = S_j, C(B_j) = \infty,$ for $1 \leq j \leq M)$.

Set Cover Case

We sketch a situation where the greedy grouping algorithm uses $\mathcal{H}(d)$ times more slabs than the optimum solution. This is also a worst case example for set cover.

Let U be a set containing $n = dm$ elements. Moreover, U is partitioned into d sets $U_1, \ldots, U_d$ of m elements each. We construct the subset collection F from which we have to choose the cover. The set F will be the union of two sets F_1 and F_2, each of them a seperate cover.

For F_1, we partition each U_i into subsets of size i, $1 \leq i \leq d$. All these subsets are elements of F_1. Therefore, $|F_1| \leq \mathcal{H}(d) \cdot n$.

The set F_2 consists of d element subsets of U, where every subset contains exactly one element of $U_1, \ldots, U_d$. Thus, $|F_2| = n$.

Given the set cover instance (U, F) the greedy algorithm finds the cover F_1, assuming that it's unluckily picking elements in F_1 if there are maximum size subsets in both F_1 and F_2.

This can be established by an inductive argument: Consider the situation where the greedy algorithm has picked all sets in F_1 containing at least $j+1$ elements. This means that every set in F_2 contains $d - j$ elements which are already covered and j uncovered elements. Thus, we pick the next set from F_1 according to our convention, since both, F_1 and F_2, contain subsets of size j.

Bin Packing Case

Assuming that there are only a few different slab types, we can derive stronger bounds: The heuristic ensures that at most one slab per slab type is half filled. Otherwise, there were two slabs of the same type and the unused protion of each of them was more than a half its capacity. Thus, we could pack the items contained in both into just one slab. A case like this can't happen since the greedy algorithm always packs many items as possible into each slab.

If all slab capacities were the same we would get the bound for any given problem instance I: $ALG(I) \leq 2\ OPT(I) + M$. It's possible to extend this formula to allow different slab capacities; let $C_{\max}$ and $C_{\min}$ be the maximum and minimum capacities:

$$ALG(I) \leq 2\ C_{\max}/C_{\min} OPT(I) + M.$$

Real World Data

We applied the algorithm to some large data sets from steel production. Comparing the results with relaxation based lower bounds, we found that the approximation ratio is better than 1.3 on average. The largest gap between lower bound and approximation was a factor 1.8.

The algorithms run fairly fast for uncolored instances, we could pack 1000 items in less than 2 seconds.

The good approximation ratio is not very surprising: Typically, item types were compatible with a large part of the bin types (about $1/4$ of the types), and for most item types, multiple items were available. Therefore, the situation was more like the bin packing case analyzed above, where it's not hard to find a good packing.

It would be interesting to see how the algorithm behaves for different input data, e.g. in cases where there are not too many items per type and the bin sizes differ much.

Also, future work could compare the set cover strategie, to maximize the number of items per bin, with other approaches. Bin packing motivated ideas would pack largest items first and fill up the residual capacity with smaller items. This could be formalized as picking a bin with the smallest residual capacity and the fewest items inside.

It should be clear that these strategies don't give any approximation ratio in the general case. Nevertheless, they could have a good behaviour for our problem instances as well.

4.3 Variations

4.3.1 Minimizing the unused Portions of the Slabs

By posing this problem as a weighted set cover, it's possible to minimize the unused portion of the slabs:

We assign set $S \in F$ the weight $w(S) = C(B_j)$ if all elements in S fit into a single slab of type B_j. In case there is more than one possible slab type B_j we assign $w(S)$ to be the minimum of all possible slab capacities. This corresponds to choosing the smallest possible slab for the items in S.

The objective is to find a minimum weight collection of sets of F which covers all items.

Using the greedy algorithm for weighted set cover we can find an $\mathcal{H}(d)$ approximation, where d denotes the maximum number of items which can be packed into a single slab. The running time is the same as in the unweighted case.

4.3.2 Colored Items

We can extend this approach to the problem variant with color constraints, where

- $c(a_i)$ is the color of items of type a_i, $1 \leq i \leq N$, and
- $K > 1$ is the maximum number of colors per slab.

We construct set U as above and cover it with a collection of sets of F. Each set $V \in F$ has to suffice the the color constraints additionally to the weight and assignment constraints: $|\cup_{a \in V} c(a)| \leq K$.

This has the consequence that the algorithm needs to look at each slab type in all color combinations in order to pick the one which can contain the most elements.

This will increase the running time of the greedy algorithm by a factor of $\binom{k}{K}$, where k is the total number of colors in this instance. Assuming that K doesn't belong to the input variables the running time is still polynomial $O(\binom{k}{K} N^2 M + N \log N)$.

The same argumentation as above establishes that the algorithm is a $\mathcal{H}(d)$-approximation.

5 Conclusion

Grouping - between Bin Packing and Set Cover

We designed an approximation algorithm for the grouping problems which has a worst case ratio of $H(d)$. This ratio is tight for a specific set cover type instance. Grouping instances which are similar to "bin packing problems" are approximated much better. We could show the approximation ratio 2 (for constant many bin types).

Moreover, we implemented the algorithm and were able to find good packings for our real world instances in short time.

The observations above raise the questions for the average case behavior for the algorithm. The gap in the approximation ratio for the two extreme cases seems to be very large. It would be interesting to see how the algorithm does for situations in between, but it's not clear which problem instances to look at.

Also, the worst case ratio in the bin packing case doesn't seem to be very tight.

Exact Solutions

This paper doesn't address the how to find exact solutions for the grouping problem. It would be necessary to develop stronger lower bounds for a branch and bound approach. Another possible approach uses graph partitioning for a divide and conquer strategy: In order to divide the problem into separate subproblems, we split the assignment graph into two partitions and determine values for the "variables" which correspond to crossing edges. This idea could be a part of a general search approach.

References

[Bro71] A. R. Browny. *Optimum Packing and Depletion*. American Elsevier, New York, 1971.

[CGJ84] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing: An updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49—106, Wien, 1984. Springer-Verlag.

[CGJ97] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 46–93, Boston, 1997. PWS Publishing Company.

[Chv79] V. Chvátal. A greedy heuristic for the set-covering problem. *Math. of Oper. Res.*, 4(3):233–235, 1979.

[Fei95] U. Feige. A threshold of ln n for approximating set cover. *Manuscript*, 1995.

[FL86] D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM J. Comput.*, (15):222–230, 1986.

[GGJY76] M. R. Garey, R. L. Graham, D. S. Johnson, and A. C. Yao. Resource constrained scheduling as generalized bin packing. *J. Comb. Th. Ser. A*, (21):257–289, 1976.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, 1979.

[Hoc82] D. S. Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM J. Comput.*, 11(3), 1982.

[Hoc97] D. S. Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set and related problems. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 46–93, Boston, 1997. PWS Publishing Company.

[Joh73] D. S. Johnson. Near-optimal bin packing algorithms. Technical report, Massachusetts Institute of Technology, 1973.

[KSS75] K. L. Krause, Y. Y. Shen, and H. D. Schwertman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *J. Assoc. Comput.*, (22):522–550, 1975.

[MT89] S. Martello and P. Toth. *Knapsack Problems*. John Wiley & Sons Ltd., New York, 1989.