The *Local Ratio* Technique and its Application to Scheduling and Resource Allocation Problems

Reuven Bar-Yehuda[*] Keren Bendel[*] Ari Freund[†] Dror Rawitz[‡]

Abstract

We give a short survey of the local ratio technique for approximation algorithms, focusing mainly on scheduling and resource allocation problems.

1 Introduction

The *local ratio* technique is used in the design and analysis of approximation algorithms for NP-hard optimization problems. Since its first appearance in the early 1980's it has been used extensively, and recently, has been fortunate to have a great measure of success in dealing with scheduling problems. Being simple and elegant, it is easy to understand, yet has surprisingly broad applicability.

At the focus of this chapter are applications of the *local ratio* technique to scheduling problems, but we also give an introduction to the technique and touch upon some of the milestones in its path of development. We present a host of optimization problems and local-ratio approximation algorithms, some of which were developed using the technique, and others which are local ratio interpretations of pre-existing algorithms.

The basic concepts relating to optimization and approximation are as follows. (Precise definitions can be found in Section 2.) An *optimization problem* is a problem in which every possible solution is associated with a cost, and we seek a solution of minimum (or maximum) cost. For example, in the *minimum spanning tree* problem our objective is to find a minimum cost spanning tree in a given edge weighted graph. For this problem, the solutions are all spanning trees, and the cost of each spanning tree is its total edge weight. Although this particular problem is polynomial-time solvable, many other optimization problems are NP-hard, and for those, computing *approximate* solutions (efficiently) is of interest. (In fact, finding approximate solutions is also of interest in cases where this can be done faster than finding exact solutions.) A solution whose cost is within a factor of r of the optimum is said to be r*-approximate.* Thus, for example, a spanning tree whose weight is at most twice the weight of a minimum spanning tree is said to be 2-approximate. An r*-approximation* algorithm is one that is guaranteed to return r-approximate solutions.

Analyzing an r-approximation algorithm consists mainly in showing that it achieves the desired degree of approximation, namely, r (correctness and efficiency are usually straightforward). To do

[*]Department of Computer Science, Technion, Haifa 32000, Israel. E-mail: {reuven,bkeren}@cs.technion.ac.il.
[†]IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel. E-mail: arief@il.ibm.com.
[‡]School of Electrical Engineering, Tel-Aviv University, Tel-Aviv 69978, Israel. E-mail: rawitz@eng.tau.ac.il.

so we need to obtain a lower bound B on the optimum cost and show that cost of the solution is no more than $r \cdot B$. The local ratio technique uses a "local" variation of this idea as a design principle. In essence, the idea is to break down the cost W of the algorithm's solution into a sum of "partial costs" $W = \sum_{i=1}^{k} W_i$, and similarly break down the lower bound B into $B = \sum_{i=1}^{k} B_i$, and to show that $W_i \leq r \cdot B_i$ for all i. (In the maximization case, B is an upper bound and we need to show that $W_i \geq B_i/r$ for all i.) The breakdown of W and B is determined by the way in which the solution is constructed by the algorithm. In fact, the algorithm constructs the solution in such a manner as to ensure that such breakdowns exist. Put differently, at the ith step, the algorithm "pays" $W_i \leq r \cdot B_i$ and manipulates the problem instance so that the optimum cost drops by at least B_i.

To see how this works in practice we demonstrate the technique on the *vertex cover* problem. Given a graph $G = (V, E)$, a *vertex cover* is a set of vertices $C \subseteq V$ such that every edge has at least one endpoint in C. In the vertex cover problem we are given a graph $G = (V, E)$ and a non-negative cost function w on its vertices, and our goal is to find a vertex cover of minimum cost. Imagine that we have to actually purchase the vertices we select as our solution. Rather than somehow deciding on which vertices to buy and then paying for them, we adopt the following strategy. We repeatedly select a vertex and pay for it. However, the amount we pay need not cover its entire cost; we may return to the same vertex later and pay some more. In order to keep track of the payments, whenever we pay ϵ for a vertex, we lower its marked price by ϵ. When the marked price of a vertex drops to zero, we are free to take it, as it has been fully paid for.

The heart of the matter is the rule by which we select the vertex and decide on the amount to pay for it. Actually, we select two vertices each time and pay ϵ for each, in the following manner. We select any edge (u, v) whose two endpoints have non-zero cost, and pay $\epsilon = \min\{\text{price of } u, \text{price of } v\}$ for both u and v. As a result, the marked price of at least one of the endpoints drops to zero. After $O(|V|)$ rounds, prices drop sufficiently so that every edge has an endpoint of zero cost. Hence, the set of all zero-cost vertices is a vertex cover. We take this zero-cost set as our solution.

We formalize this by the following algorithm. We say that an edge is *positive* if both its endpoints have strictly positive cost.

```
Algorithm VC
    1.  While there exists a positive edge $(u, v)$:
    2.      Let $\epsilon = \min\{w(u), w(v)\}$.
    3.      $w(u) \leftarrow w(u) - \epsilon$.
    4.      $w(v) \leftarrow w(v) - \epsilon$.
    5.  Return the set $C = \{v \mid w(v) = 0\}$.
```

To analyze the algorithm, consider the ith iteration. Let (u_i, v_i) be the edge selected in this iteration, and let ϵ_i be the payment made for each endpoint. Every vertex cover must contain either u_i or v_i (in order to cover the edge (u_i, v_i)), and therefore decreasing the price of these vertices by ϵ_i lowers the cost of every vertex cover by at least ϵ_i. It follows that the optimum, denoted OPT, also decreases by at least ϵ_i. Thus, in the ith round we pay $2\epsilon_i$ and lower OPT by at least ϵ_i, so the *local ratio* between our payment and the drop in OPT is at most 2 in any given iteration. Summing over all iterations, we get that the ratio between our total payment and the total drop in OPT is at most 2 as well. Now, we know that the total drop in the optimal cost is OPT, since we end up with a vertex cover of zero cost, so our total payment is at most $2OPT$. Since this payment fully covers the solution's cost (in terms of the original cost function), the solution is 2-approximate.

It is interesting to note that the proof that the solution found is 2-approximate does not depend on the actual value of ϵ in any given iteration. In fact, any value between 0 and $\min\{w(u), w(v)\}$ would yield the same result (by the same arguments). We chose $\min\{w(u), w(v)\}$ for the sake of efficiency. This choice ensures that the number of vertices with positive cost strictly decreases with each iteration.

We also observe that the analysis of algorithm **VC** does not seem tight. The analysis bounds the cost of the solution by the sum of *all* payments, but some of these payments are made for vertices that do not end up in the solution. It might seem that trying to "recover" these wasteful payments could yield a better approximation ratio, but this is not true (in the worst case); it is easy to construct examples in which all vertices for which payments are made are eventually made part of the solution.

Finally, there might still seem to be some slack in the analysis, for in the final step of the algorithm *all* zero-cost vertices are taken to the solution, without trying to remove unnecessary ones. One simple idea is to prune the solution and turn it into a *minimal* (with respect to set inclusion) subset of C that is still a vertex cover. Unfortunately, it is not difficult to come up with worst-case scenarios in which C is minimal to begin with. Nevertheless, we shall see that such ideas are sometimes useful (and, in fact, necessary) in the context of other optimization problems.

1.1 Historical Highlights

The origins of the local ratio technique can be traced back to a paper by Bar-Yehuda and Even on *vertex cover* and *set cover* [16]. In this paper, the authors presented a linear time approximation algorithm for *set cover*, that generalizes Algorithm **VC**, and presented a primal-dual analysis of it. This algorithm was motivated by a previous algorithm of Hochbaum [42], which was based on LP duality, and required the solution of a linear program. Even though Bar-Yehuda and Even's primal-dual analysis contains an implicit local ratio argument, the debut of the local ratio technique occurred in a followup paper [17], where the authors gave a local ratio analysis of the same algorithm. They also designed a specialized $(2 - \frac{\log_2 \log_2 n}{2 \log_2 n})$-approximation algorithm for *vertex cover* that contains a local-ratio phase. The technique was dormant until Bafna, Berman, and Fujito [9] incorporated the idea of *minimal solutions* into the local ratio technique in order to devised a local ratio 2-approximation algorithm for the *feedback vertex set* problem. Subsequently, two generic algorithms were presented. Fujito [33] gave a unified local ratio approximation algorithm for node-deletion problems, and Bar-Yehuda [15] developed a local ratio framework that explained most local ratio (and primal-dual) approximation algorithms known at the time. At this point in time the local ratio technique had reached a certain level of maturity, but only in the context of minimization problems. No local ratio algorithms were known for maximization problems. This was changed by Bar-Noy et al. [11], who presented the first local ratio (and primal-dual) algorithms for maximization problems. These algorithm are based on the notion of *maximal* solutions rather than minimal ones. More recently, Bar-Yehuda and Rawitz [19] developed two approximation frameworks, one extending the generic local ratio algorithm from [15], and the other extending known primal-dual frameworks [38, 22], and proved that both frameworks are equivalent, thus merging these two seemingly independent lines of research. The most recent local ratio development, due to Bar-Yehuda et al. [12], is a novel extension of the local ratio technique called *fractional local ratio*.

1.2 Organization

The remainder of this chapter is organized as follows. In Section 2 we establish some terminology and notation. In Section 3 we state and prove the Local Ratio Theorem (for minimization problems) and formulate the local ratio technique as a design and analysis framework based on it. In Section 4 we formally introduce the idea of minimal solutions into the framework, making it powerful enough to encompass many known approximation algorithms for covering problems. Finally, in Section 5 we discuss local ratio algorithms for scheduling problems, focusing mainly on maximization problems. As a first step, we describe a local ratio framework for maximization problems, which is, in a sense, a mirror image of its minimization counterpart developed in Sections 3 and 4. We then survey a host of problems to which the local ratio technique has been successfully applied.

In order not to interrupt the flow of text we have removed nearly all citations and references from the running text, and instead have included at the end of each section a subsection titled *Background*, in which we cite sources for the material covered in the section and discuss related work.

1.3 Background

The *vertex cover* problem is known to be NP-hard even for planar cubic graphs with unit weights [36]. Håstad [41] shows, using PCP arguments[1], that *vertex cover* cannot be approximated within a factor of $\frac{7}{6}$ unless P=NP. Dinur and Safra [29] improve this bound to $10\sqrt{5}-21 \approx 1.36067$. The first 2-approximation algorithm for weighted *vertex cover* is due to Nemhauser and Trotter [57]. Hochbaum [43] uses this algorithm to obtain an approximation algorithm with performance ratio $2 - \frac{2}{d_{\max}}$, where $d_{\max}$ is the maximum degree of a vertex. Gavril (see [35]) gives a linear time 2-approximation algorithm for the non-weighted case. (Algorithm **VC** reduces to this algorithm on non-weighted instances.) Hochbaum [42] presents two 2-approximation algorithms, both requiring the solution of a linear program. The first constructs a vertex cover based on the optimal dual solution, whereas the second is a simple LP rounding algorithm. Bar-Yehuda and Even [16] present an LP based approximation algorithm for weighted *set cover* that does not solve a linear program directly. Instead, it constructs simultaneously a primal integral solution and a dual feasible solution without solving either the primal or dual programs. It is the first algorithm to operate in this method, a method which later became known as the *primal-dual schema*. Their algorithm reduces to Algorithm **VC** on instances that are graphs. In a subsequent paper, Bar-Yehuda and Even [17] provide an alternative local ratio analysis for this algorithm, making it the first local ratio algorithm as well. They also present a specialized $(2 - \frac{\log_2 \log_2 n}{2 \log_2 n})$-approximation algorithm for *vertex cover*. Independently, Monien and Speckenmeyer [56] achieved the same ratio for the unweighted case. Halperin [40] improved this result to $2 - (1 - o(1))\frac{2 \ln \ln d_{\max}}{\ln d_{\max}}$ using semidefinite programming.

[1]By "PCP arguments" we mean arguments based on the celebrated PCP Theorem and its proof. The PCP theorem [6, 5] and its variants state that certain suitably defined complexity classes are in fact equal to NP. A rather surprising consequence of this is a technique for proving lower bounds on the approximation ratio achievable (in polynomial time) for various problems. For more details see Arora and Lund [4] and Ausiello et el. [7].

2 Definitions and Notation

An optimization problem is comprised of a family of problem *instances*. Each instance is associated with (1) a collection of *solutions*, each of which is either *feasible* or *infeasible*, and (2) a *cost* function assigning a cost to each solution. We note that in the sequel we use the terms *cost* and *weight* interchangeably. Each optimization problem can be either a *minimization* problem or a *maximization* problem. For a given problem instance, a feasible solution is referred to as *optimal* if it is either minimal or maximal (depending, respectively, on whether the problem is one of minimization or maximization) among all feasible solutions. The cost of an optimal solution is called the *optimum value*, or simply, the *optimum*. For example, in the well known *minimum spanning tree* problem instances are edge-weighted graphs, solutions are subgraphs, feasible solutions are spanning trees, the cost of a given spanning tree is the total weight of its edges, and optimal solutions are spanning trees of minimum total edge weight.

Most of the problems we consider in this survey can be formulated as problems of selecting a subset (satisfying certain constraints) of a given set of objects. For example, in the *minimum spanning tree* problem we are required to select a subset of the edges that form a spanning tree. In such problems we consider the cost function to be defined on the objects, and extend it to subsets in the natural manner.

An approximation algorithm for an optimization problem takes an input instance and efficiently computes a feasible solution whose value is "close" to the optimum. The most popular measure of closeness is the *approximation ratio*. Recall that for $r \geq 1$, a feasible solution is called *r-approximate* if its cost is within a factor of r of the optimum. More formally, in the minimization case, a feasible solution X is said to be r-approximate if $w(X) \leq r \cdot w(X^*)$, where $w(X)$ is the cost of X, and X^* is an optimal solution. In the minimization case, X is said to be r-approximate if $w(X) \geq w(X^*)/r$. (Note that in both cases r is smaller, when X is closer to X^*.) An r-approximate solution is also referred to as an *r-approximation*. An algorithm that computes r-approximate solutions is said to achieve an *approximation factor* of r, and it is called an *r-approximation* algorithm. Also, r is said to be a *performance guarantee* for it. The *approximation ratio* of a given algorithm is $\inf\{r \mid r \text{ is a performance guarantee for the algorithm}\}$. Nevertheless, the term *approximation ratio* is sometimes used instead of *performance guarantee*.

We assume the following conventions, except where specified otherwise. All weights are non-negative and denoted by w. We denote by $w(x)$ the weight of element x, and by $w(X)$ the total weight of set X, i.e., $w(X) = \sum_{x \in X} w(x)$. We denote the optimum value of the problem instance at hand by OPT. All graphs are simple and undirected. A graph is denoted $G = (V, E)$, where $n \triangleq |V|$, and $m \triangleq |E|$. The degree of vertex v is denoted by $\deg(v)$.

3 The Local Ratio Theorem

In Algorithm **VC** we have paid $2 \cdot \epsilon$ for lowering OPT by at least ϵ in each round. Other local ratio algorithms can be explained similarly—one pays in each round at most $r \cdot \epsilon$, for some r, while lowering OPT by at least ϵ. If the same r is used in all rounds, the solution computed is r-approximate. This idea works well for several problems. However, it is not hard to see that this idea works mainly because we make a down payment on several items, and we are able to argue that OPT must drop by a proportional amount because every solution must involve some of these items. This localization of the payments is at the root of the simplicity and elegance of the analysis, but it is also the source of its weakness: how can we design algorithms for problems in which no single set of items is necessarily involved in every optimal solution? For example, consider the *feedback vertex set* problem, in which we are given a graph and a weight function on the vertices, and our goal is to remove a minimum weight set of vertices such that the remaining graph contains no cycles. Clearly, it is not always possible to find two vertices such that at least one of them is part of every optimal solution! The Local Ratio Theorem, which is given below, allows us to go beyond localized payments by focusing on the *changes* in the weight function, and treating these changes as weight functions in their own right. Indeed, this is essential in the local ratio 2-approximation algorithm for feedback vertex set that is given in the next section.

The Local Ratio Theorem is deceptively simple. It applies to optimization problems that can be formulated as follows.

> Given a *weight vector* $w \in \mathbb{R}^n$ and a set of *feasibility constraints* $\mathcal{F}$, find a *solution vector* $x \in \mathbb{R}^n$ satisfying the constraints in $\mathcal{F}$ that minimizes the inner product $w \cdot x$.

(This section discusses minimization problems. In Section 5 we deal with maximization problems.)

In this survey we mainly focus on optimization problems in which $x \in \{0,1\}^n$. In this case the optimization problem consists of instances in which the input contains a set I of n weighted elements and a set of feasibility constraints on subsets of I. Feasible solutions are subsets of I satisfying the feasibility constraints. The cost of a feasible solution is the total weight of the elements it contains. Such a minimization problem is called a *covering* problem if any extension of a feasible solution to any possible instance is always feasible. The family of covering problems contains a broad range of optimization problems, such as *vertex cover*, *set cover*, and *feedback vertex set*.

Theorem 1 (Local Ratio—Minimization Problems) *Let $\mathcal{F}$ be a set of feasibility constraints on vectors in $\mathbb{R}^n$. Let $w, w_1, w_2 \in \mathbb{R}^n$ be such that $w = w_1 + w_2$. Let $x \in \mathbb{R}^n$ be a feasible solution (with respect to $\mathcal{F}$) that is r-approximate with respect to w_1 and with respect to w_2. Then, x is r-approximate with respect to w as well.*

Proof. Let x^*, x_1^*, and x_2^* be optimal solutions with respect to w, w_1, and w_2, respectively. Clearly, $w_1 \cdot x_1^* \le w_1 \cdot x^*$ and $w_2 \cdot x_2^* \le w_2 \cdot x^*$. Thus,

$$w \cdot x = w_1 \cdot x + w_2 \cdot x \le r(w_1 \cdot x_1^*) + r(w_2 \cdot x_2^*) \le r(w_1 \cdot x^*) + r(w_2 \cdot x^*) = r(w \cdot x^*)$$

and we are done. ■

As we shall see, algorithms that are based on the Local Ratio Theorem are typically recursive and has the following general structure. If a zero-cost solution can be found, return one. Otherwise, find a decomposition of w into two weight functions w_1 and $w_2 = w - w_1$, and solve the problem recursively on w_2. We demonstrate this on the *vertex cover* problem. (Recall that a positive edge is an edge whose two endpoints have non-zero cost.)

Algorithm RecursiveVC(G,w)

```
1.  If all edges are non-positive, return the set $C = \{v \mid w(v) = 0\}$.
2.  Let $(u, v)$ be a positive edge, and let $\epsilon \triangleq \min\{w(u), w(v)\}$.
3.  Define $w_1(x) = \begin{cases} \epsilon & x = u \text{ or } x = v, \\ 0 & \text{otherwise,} \end{cases}$ and define $w_2 = w - w_1$.
4.  Return RecursiveVC($G$,$w_2$).
```

Clearly, Algorithm **RecursiveVC** is merely a recursive version of Algorithm **VC**. However, the recursive formulation is amenable to analysis that is based on the Local Ratio Theorem. We show that the the algorithm computes 2-approximate solutions by induction on the number of recursive calls (which is clearly finite). In the recursion base, the algorithm returns a zero-weight vertex cover, which is optimal. For the inductive step, consider the solution C. By the inductive hypothesis C is 2-approximate with respect to w_2. We claim that C is also 2-approximate with respect to w_1. In fact, *every* feasible solution is 2-approximate with respect to w_1. Observe that the cost (with respect to w_1) of every vertex cover is at most 2ϵ, while the minimum cost of a vertex cover is at least ϵ. Thus, by the Local Ratio Theorem C is 2-approximate with respect to w as well.

We remark that algorithms that follow the general structure outlined above differ from one another only in the choice of w_1. (Actually, the way they search for a zero-cost solution is sometimes different.) It is not surprising that these algorithms also share most of their analyses. Specifically, the proof that a given algorithm is an r-approximation is by induction on the number of recursive calls. In the base case, the solution has zero cost, and hence it is optimal (and also r-approximate). In the inductive step, the solution returned by the recursive call is r-approximate with respect to w_2 by the inductive hypothesis. And, it is shown that every solution is r-approximate with respect to w_1. This makes the current solution r-approximate with respect to w due to the Local Ratio Theorem. Thus, different algorithms are different from one another only in the choice of the weight function w_1 in each recursive call, and in the proof that every feasible solution is r-approximate with respect to w_1. We formalizes this notion by the following definition.

Definition 1 *Given a set of constraints $\mathcal{F}$ on vectors in $\mathbb{R}^n$ and a number $r \geq 1$, a weight vector $w \in \mathbb{R}^n$ is said to be* fully r-effective *if there exists a number b such that $b \leq w \cdot x \leq r \cdot b$ for all feasible solutions x.*

We conclude this section by demonstrating the above framework on the *set cover* problem. Since the analysis of algorithms in our framework boils down to proving that w_1 is r-effective, for an appropriately chosen r, we focus solely on w_1, and neglect to mention the remaining details explicitly. We remark that our algorithm for *set cover* can be formulated just as easily in terms of localized payments; the true power of the Local Ratio Theorem will become apparent in Section 4.

3.1 Set Cover

In the *set cover* problem we are given a collection of non-empty sets $\mathcal{C} = \{S_1, \ldots, S_n\}$ and a weight function w on the sets. A *set cover* is a sub-collection of sets that *covers* all elements. In other words, the requirement is that the union of the sets in the set cover be equal to $U \triangleq \bigcup_{i=1}^{n} S_i$. The objective is to find a minimum-cost set cover.

Let $\deg(x)$ be the number of sets in $\mathcal{C}$ that contain x, i.e., $\deg(x) = |\{S \in \mathcal{C} \mid x \in S\}|$. Let $d_{\max} = \max_{x \in U} \deg(x)$. We present a fully $d_{\max}$-effective weight function w_1. Let x be an element that is not covered by any zero-weight set, and let $\epsilon = \min\{w(S) \mid x \in S\}$. Define:

$$w_1(S) = \begin{cases} \epsilon & x \in S, \\ 0 & \text{otherwise.} \end{cases}$$

w_1 is $d_{\max}$-effective since (1) the cost of every feasible solution is bounded by $\epsilon \cdot \deg(x) \leq \epsilon \cdot d_{\max}$, and (2) every set cover must cover x, and therefore must cost at least ϵ.

Note that vertex cover can be seen as a set cover problem in which the sets are the vertices and the elements are the edges (and therefore $d_{\max} = 2$). Indeed, Algorithm **RecursiveVC** is a special case of the algorithm that is implied by the discussion above.

3.2 Background

For the unweighted *set cover* problem, Johnson [46] and Lovász [52] show that the greedy algorithm is an $H_{s_{\max}}$-approximation algorithm, where H_n the nth harmonic number, i.e., $H_n = \sum_{i=1}^{n} \frac{1}{i}$, and $s_{\max}$ is the maximum size of a set. This result was generalize by Chvátal [28] result to the weighted case. Hochbaum [42] gives two $d_{\max}$-approximation algorithms, both of which are based on solving a linear program. Bar-Yehuda and Even [16] suggest a linear time primal-dual $d_{\max}$-approximation algorithm. In subsequent work [17], they present the Local Ratio Theorem and provide a local ratio analysis of the same algorithm. (Their analysis is the one given in this section.) Feige [32] proves a lower bound of $(1 - o(1)) \ln |U|$ (unless NP$\subseteq$DTIME($n^{O(\log\log n)}$)). Raz and Safra [59] show that set cover cannot be approximated within a factor of $c \log n$ for some $c > 0$ unless P=NP.

4 A Framework for Covering Problems

In the problems we have seen this far, we were always able to identify a small subset of items (vertices or sets) and argue that every feasible solution must include at least one of them. We defined a weight function w_1 that associated a weight of ϵ with each of the items in this small subset and a weight of zero with all others. Thus, we were able to obtain an approximation ratio bounded by the size of the subset. There are many problems, though, where it is impossible to identify such a small subset, since no such subset necessarily exists.

An good example is the *partial set cover* problem (or *partial cover* problem, for short). This problem is similar to *set cover* except that not all elements should be covered. More specifically, the input consists of a collection of sets, a weight function on the sets, and a number k, and we want to find a minimum-cost collection of sets that covers at least k of the elements. The crucial difference between *set cover* and *partial set cover* is that in the latter, there is no single element that must be covered by all feasible solutions. Recall that the algorithm for *set cover* picked some element x and defined a weight function w_1 that associated a weight of ϵ with each of the sets that contains x. The analysis was based on the fact that, with respect to w_1, ϵ is a lower bound, since x

must be covered, while $\epsilon \cdot \deg(x)$ is an upper bound on the cost of every set cover. This approach fails for *partial set cover*—an optimal solution need not necessarily cover x, and therefore ϵ is no longer a lower bound. Thus if we use w_1, we will end up with a solution whose weight is positive, while OPT (with respect to w_1) may be equal to 0.

We cope with such hard situations by extending the same upper-bound/lower-bound idea. Even if we cannot identify a small subset of items that must contribute to all solutions, we know that the set of *all* items must surely do so (since, otherwise, the empty set is an optimal solution). Thus, to prevent OPT from being equal to 0, we can assign a positive weight to every item (or at least to many items). This takes care of the lower bound, but raises the question of how to obtain a non-trivial upper bound. Clearly, we cannot hope that the cost of every feasible solution will always be within some reasonable factor of the cost of a single item. However, in some cases it is enough to obtain an upper bound only for *minimal solutions*. A *minimal solution* is a feasible solution that is minimal with respect to set inclusion, i.e., a feasible solution all of whose proper subsets are not feasible. Minimal solutions arise naturally in the context of *covering* problems, which are the problems for which feasible solutions have the property of being monotone inclusion-wise, that is, the property that adding items to a feasible solution cannot render it infeasible. (For example, adding a set to a set cover yields a set cover, so *set cover* is a covering problem. In contrast, adding an edge to a spanning tree does not yield a tree, so *minimum spanning tree* is not a covering problem.) The idea of focusing on minimal solutions leads to the following definition.

Definition 2 *Given a set of constraints $\mathcal{F}$ on vectors in $\mathbb{R}^n$ and a number $r \geq 1$, a weight vector $w \in \mathbb{R}^n$ is said to be* r-effective *if there exists a number b such that $b \leq w \cdot x \leq r \cdot b$ for all minimal feasible solutions x.*

Note that any fully r-effective weight function is also r-effective, while the opposite direction is not always true.

If we can prove that our algorithm uses r-effective weight functions and returns minimal solutions, we will have essentially proved that it is an r-approximation algorithm. Designing an algorithm to output minimal solutions is not hard. Most of the creative effort is therefore concentrated in finding an r-effective weight function (for a small r).

In this section we present local ratio algorithms for the *partial cover* and *feedback vertex set* problems. Both algorithms depend on obtaining minimal solutions. We describe and analyze the algorithm for *partial set cover* in full detail. We then outline a general local ratio framework for covering problems, and discuss the algorithm for *feedback vertex set* informally with reference to this framework.

4.1 Partial Set Cover

In the *partial set cover* problem the input consists of a collection of non-empty sets $\mathcal{C} = \{S_1, \ldots, S_n\}$, a weight function w on the sets, and a number k. The objective is to find a minimum-cost subcollection of $\mathcal{C}$ that covers at least k elements in $U \triangleq \bigcup_{i=1}^{n} S_i$. We assume that a feasible solution exists, i.e., that $k \leq |U|$. The *partial cover* problem generalizes *set cover* since in the set cover problem k is simply set to $|U|$.

Next, we present a $\max\{d_{\max}, 2\}$-approximation algorithm. (Recall that $d_{\max} = \max_{x \in U} \deg(x)$, where $\deg(x) = |\{S \in \mathcal{C} \mid x \in S\}|$.)

```
Algorithm PSC($U$, $\mathcal{C}$, $w$, $k$)
  1.  If $k \leq 0$, return $\emptyset$.
  2.  Else, if there exists a set $S \in \mathcal{C}$ such that $w(S) = 0$ do:
  3.      Let $U'$, $\mathcal{C}'$ be the instance obtained by removing $S$.
  4.      $P' \leftarrow$ PSC($U'$, $\mathcal{C}'$, $w$, $k - |S|$).
  5.      If $P'$ covers at least $k$ elements in $U$:
  6.          Return the solution $P = P'$.
  7.      Else:
  8.          Return the solution $P = P' \cup \{S\}$.
  9.  Else:
 10.      Let $\epsilon$ be maximal such that $\epsilon \cdot \min\{|S|, k\} \leq w(S)$ for all $S \in \mathcal{C}$.
 11.      Define the weight functions $w_1(x) = \epsilon \cdot \min\{|S|, k\}$ and $w_2 = w - w_1$.
 12.      Return PSC($U$, $\mathcal{C}$, $w_2$, $k$).
```

Note the slight abuse of notation in Line 4. The weight function in the recursive call is not w itself, but rather the restriction of w to $\mathcal{C}'$. We will continue to silently abuse notation in this manner.

Let us analyze the algorithm. We claim that the algorithm finds a minimal solution that is $\max\{d_{\max}, 2\}$-approximate. Intuitively, Lines 3–8 ensure that the solution returned is minimal, while the weight decomposition in Lines 9–12 ensures that every minimal solution is $\max\{d_{\max}, 2\}$-approximate. This is done by associating with each set S a weight that is proportional to its "covering power," which is the number of elements in S, but not more than k, since covering more than k elements is no better than covering k elements.

Proposition 2 *Algorithm **PSC** returns a feasible minimal solution.*

Proof. The proof is by induction on the recursion. At the recursion basis the solution returned is the empty set, which is both feasible (since $k \leq 0$) and minimal. For the inductive step, $k > 0$ and there are two cases to consider. If Lines 9–12 are executed, then the solution returned is feasible and minimal by the inductive hypothesis. Otherwise, Lines 3–8 are executed. By the inductive hypothesis P' is minimal and feasible with respect to $(U', \mathcal{C}', k - |S|)$. If $P' = \emptyset$ then $|S| \geq k$ and $P = P' \cup \{S\}$ is clearly feasible and minimal. Otherwise, P' covers at least $k - |S|$ elements in $U \setminus S$, and by minimality, for all $T \in P'$, the collection $P' \setminus \{T\}$ covers less than $k - |S|$ elements that are not contained in S. (Note that $P' \neq \emptyset$ implies $k > |S|$.) Consider the solution P. Either $P = P'$, which is the case if P' covers at least k or more elements, or else $P = P' \cup \{S\}$, in which case P covers at least $k - |S|$ elements that are not contained in S and an additional $|S|$ elements that are contained in S. In either case P covers at least k elements and is therefore feasible. It is also minimal (in either case), since for all $T \in P$, if $T \neq S$, then $P \setminus \{T\}$ covers less than $k - |S|$ elements that are not contained in S and at most $|S|$ elements that are contained in S, for a total of less than k elements, and if $T = S$, then $S \in P$, which implies $P = P' \cup \{S\}$, which is only possible if $P \setminus \{S\} = P'$ covers less than k elements. ■

Proposition 3 *The weight function w_1 used in Algorithm **PSC** is $\max\{d_{max}, 2\}$-effective.*

Proof. In terms of w_1, every feasible solution costs at least $\epsilon \cdot k$, since it either contains a set whose cost is $\epsilon \cdot k$, or else consists solely of sets whose size is less than k, in which case the cost of the solution equals ϵ times the total sizes of the sets in the solution, which must be at least k in order to cover k elements. To prove that every minimal solution costs at most $\epsilon \cdot k \cdot \max\{d_{\max}, 2\}$, consider a non-empty minimal feasible solution P. If P is a singleton, its cost is at most ϵk, and the claim follows. Otherwise, we prove the claim by showing that $\sum_{S \in P} |S| \leq k \cdot \max\{d_{\max}, 2\}$. We say that an element x is *covered r times* by P if $|\{S \in P \mid x \in S\}| = r$. We bound the total number of times elements are covered by P, since this number is equal to $\sum_{S \in P} |S|$. Clearly every element x may be covered at most $\deg(x) \leq d_{\max}$ times. Thus, if t is the number of elements that are covered by P twice or more, these elements contribute at most $t \cdot d_{\max}$ to the count. As for the elements that are covered only once, they contribute exactly $\sum_{S \in P} |S|_1$, where $|S|_1$ is the number of elements covered by S but not by any other member of P. Let $S^* = \arg\min\{|S|_1 \mid S \in P\}$. Then (by the choice of S^* and the fact that P is not a singleton) $|S^*|_1 \leq \sum_{S \in P \setminus \{S^*\}} |S|_1$. In addition, $t + \sum_{S \in P \setminus \{S^*\}} |S|_1 < k$ by the minimality of P. Thus the elements that are covered only once contribute $|S^*|_1 + \sum_{S \in P \setminus \{S^*\}} |S|_1 \leq 2 \sum_{S \in P \setminus \{S^*\}} |S|_1 < 2(k - t)$, and the total is less than $t \cdot d_{\max} + 2(k - t) \leq k \cdot \max\{d_{\max}, 2\}$, where the inequality follows from the fact that $t \leq k$ (which is implied by the minimality of P). ■

Theorem 4 *Algorithm **PSC** returns* $\max\{d_{max}, 2\}$*-approximate solutions.*

Proof. The proof is by induction on the recursion. In the base case the solution returned is the empty set, which is optimal. For the inductive step, if Lines 3–8 are executed, then P' is $\max\{d_{\max}, 2\}$-approximate with respect to $(U', \mathcal{C}', w, k - |S|)$ by the inductive hypothesis. Since $w(S) = 0$, the cost of P equals that of P' and the optimum for $(U, \mathcal{C}, w, k)$ cannot be smaller than the optimum value for $(U', \mathcal{C}', w, k - |S|)$ because if P^* is an optimal solution for $(U, \mathcal{C}, w, k)$, then $P^* \setminus \{S\}$ is a feasible solution of the same cost for $(U', \mathcal{C}', w, k - |S|)$. Hence P is $\max\{d_{\max}, 2\}$-approximate with respect to $(U, \mathcal{C}, w, k)$. If, on the other hand, Lines 10–12 are executed, then by the inductive hypothesis the solution returned is $\max\{d_{\max}, 2\}$-approximate with respect to w_2, and by Proposition 3 it is also $\max\{d_{\max}, 2\}$-approximate with respect to w_1. Thus by the Local Ratio Theorem it is $\max\{d_{\max}, 2\}$-approximate with respect to w as well. ■

4.2 A Framework

A local ratio algorithm for a covering problem typically consists of a three-way *if* condition that directs execution to one of the following three primitives: *computation of optimal solution*, *problem size reduction*, or *weight decomposition*. The top-level description of the algorithm is:

1. If a zero-cost minimal solution can be found, do: *computation of optimal solution.*
2. Else, if the problem contains a zero-cost element, do: *problem size reduction.*
3. Else, do: *weight decomposition.*

The three types of primitives are:

Computation of optimal solution. Find a zero-cost minimal solution and return it. (Typically, the solution is simply the empty set.) This is the recursion basis.

Problem size reduction. This primitive consists of three parts.

1. Pick a zero-cost item, and assume it is taken into the solution. Note that this changes the problem instance, and may entail further changes to achieve consistency with the idea that the item has been taken temporarily to the solution. For example, in the *partial set cover* problem we selected a zero-cost set and assumed it is part of our solution. Hence, we have deleted all elements contained in it and reduced the covering requirement parameter k by the size of this set. Note that the modification of the problem instance may be somewhat more involved. For example, a graph edge might be eliminated by *contracting* it. As a result, two vertices are "fused" together and the edges incident on them are merged or deleted. In other words, the modification may consist of removing some existing items and introducing new ones. This, in turn, requires that the weight function be modified to cover the new items. However, it is important to realize that the modification of the weight function amounts to a re-interpretation of the old weights in terms of the new instance and not to an actual change of weights.
2. Solve the problem recursively on the modified instance.
3. If the solution returned (when re-interpreted in terms of the original instance) is feasible (for the original instance), return it. Otherwise, add the deleted zero-cost item to the solution, and return it.

Weight decomposition. Construct an r-effective weight function w_1 such that $w_2 = w - w_1$ is non-negative, and solve the problem recursively using w_2 as the weight function. Return the solution obtained.

We note that the above description of the framework should not be taken too literally. Each branch of the three-way *if* statement may actually consist of several sub-cases, only one of which is to be executed. We shall see an example of this in the algorithm for *feedback vertex set* (Section 4.3).

The analysis of an algorithm that follows the framework is similar to our analysis for *partial set cover*. It consists of proving the following three claims.

Claim 1. The algorithm outputs a minimal feasible solution.

This claim is proven by induction on the recursion. In the base case the solution is feasible and minimal by design. For the inductive step, if the algorithm performs *weight decomposition*, then by the inductive hypothesis the solution is feasible and minimal. If the algorithm performs *problem size reduction*, the claim follows from the fact that the solution returned by the recursive call is feasible and minimal with respect to the modified instance (by the inductive hypothesis) and it is extended only if it is infeasible with respect to the current instance. Although the last argument seems straightforward, the details of a rigorous proof tend to be slightly messy, since they depend on the way in which the instance is modified.

Claim 2. The weight function w_1 is r-effective.

The proof depends on the combinatorial structure of the problem at hand. Indeed, the key to the design of a local ratio algorithm is understanding the combinatorial properties of the problem and finding the "right" r and r-effective weight function (or functions).

Claim 3. The algorithm computes an r-approximate solution.

The proof of this claim is also by induction on the recursion, based on the previous two claims. In the base case the *computation of optimal solution* ensures that the solution returned is r-approximate. For the inductive step, we have two options. In the *problem size reduction* case, the solution found recursively for the modified instance is r-approximate by the inductive hypothesis, and it has the same cost as the solution generated for the original instance, since the two solutions may only differ by a zero-weight item. This, combined with the fact that the optimum can only decrease because of instance modification, yields the claim. In the *weight decomposition* case, the claim follows by the inductive hypothesis and the Local Ratio Theorem, since the solution is feasible and minimal, and w_1 is r-effective.

4.3 Feedback Vertex Set

A set of vertices in an undirected graph is called a *feedback vertex set* (FVS for short) if its removal leaves an acyclic graph (i.e., a forest). Another way of saying this is that the set intersects all cycles in the graph. The *feedback vertex set* problem is: given a vertex-weighted graph, find a minimum-weight FVS. In this section we describe and analyze a 2-approximation algorithm for the problem following our framework for covering problems.

The algorithm is as follows.

```
Algorithm FVS($G, w$)
 1.  If $G$ is empty, return $\emptyset$.
 2.  If there exists a vertex $v \in V$ such that $\deg(v) \leq 1$ do:
 3.      return FVS($G \setminus \{v\}, w$).
 4.  Else, if there exists a vertex $v \in V$ such that $w(v) = 0$ do:
 5.      $F' \leftarrow$ FVS($G \setminus \{v\}, w$).
 6.      If $F'$ is an FVS with respect to $G$:
 7.          Return $F'$.
 8.      Else:
 9.          Return $F = F' \cup \{v\}$.
10.  Else:
11.      Let $\epsilon = \min_{v \in V} \frac{w(v)}{\deg(v)}$.
12.      Define the weight functions $w_1(v) = \epsilon \cdot \deg(v)$
         and $w_2 = w - w_1$.
13.      Return FVS($G, w_2$)
```

The analysis follows the pattern outlined above—the only interesting part is showing that w_1 is 2-effective. For a given set of vertices X let us denote $\deg(X) = \sum_{v \in X} \deg(v)$. Since $w_1(F) = \epsilon \cdot \deg(F)$ for any FVS F, it is sufficient to demonstrate the existence of a number b such that for all minimal solutions F, $b \leq \deg(F) \leq 2b$. Note that the weight decomposition is only applied to graphs in which all vertices have degree at least 2, so we shall henceforth assume that our graph $G = (V, E)$ is such a graph.

Consider a minimal feasible solution F. The removal of F from G leaves a forest on $|V| - |F|$ nodes. This forest contains less than $|V| - |F|$ edges, and thus the number of edges deleted to obtain it is greater than $|E| - (|V| - |F|)$. Since each of these edges is incident on some vertex in F, we get $|E| - (|V| - |F|) < \deg(F)$ (which is true even if F is not minimal). Let F^* be a minimum cardinality FVS, and put $b = |E| - (|V| - |F^*|)$. Then $b < \deg(F)$, and it remains to prove that $\deg(F) \le 2b = 2|E| - 2(|V| - |F^*|)$. We show equivalently that $\deg(V \setminus F) \ge 2(|V| - |F^*|)$. To do so we select for each vertex $v \in F$ a cycle C_v containing v but no other member of F (the minimality of F ensures the existence of such a cycle). Let P_v denote the path obtained from C_v by deleting v, and let V' denote the union of the vertex sets of the P_vs. Consider the connected components of the subgraph induced by V'. Each connected component fully contains some path P_v. Since the cycle C_v must contain a member of the FVS F^*, either P_v contains a vertex of $F^* \setminus F$ or else $v \in F^* \cap F$. Thus there are at most $|F^* \setminus F|$ connected components that contain vertices of F^* and at most $|F^* \cap F|$ that do not, for a total of at most F^* connected components. Hence, the subgraph contains at least $|V'| - |F^*|$ edges, all of which have both endpoints in V'. In addition, G contains at least $2|F|$ edges with exactly one endpoint in V'—the two edges connecting each $v \in F$ with the path P_v (to form the cycle C_v). It follows that $\deg(V') \ge 2(|V'| - |F^*|) + 2|F|$. Thus, bearing in mind that $F \subseteq V \setminus V'$ and that the degree of every vertex is at least two, we see that

$$\begin{aligned}\deg(V \setminus F) &= \deg(V') + \deg((V \setminus V') \setminus F)\\ &\ge 2(|V'| - |F^*|) + 2|F| + 2(|V| - |V'| - |F|)\\ &= 2(|V| - |F^*|).\end{aligned}$$

4.4 Background

Partial set cover. The *partial set cover* problem was first studied by Kearns [48] in relation to learning. He proves that the performance ratio of the greedy algorithm is at most $2H_n + 3$, where n is the number of sets. (Recall that H_n is the nth harmonic number.) Slavík [60] improves this bound to H_k. The special case in which the cardinality of every set is exactly 2 is called the *partial vertex cover* problem. This problem was studied by Bshouty and Burroughs [25], who obtained the first polynomial time 2-approximation algorithm for it. The $\max\{d_{\max}, 2\}$-approximation algorithm for *partial set cover* given in this section (Algorithm **PSC**) is due to Bar-Yehuda [14]. In fact, his approach can be used to approximate an extension of the partial cover problem in which there is a *length* l_i associated with each element x, and the goal is to cover elements of total length at least k. (The plain *set cover* problem is the special case where $l_i = 1$ for all i.) Gandhi et al. [34] present a multi-phase primal-dual algorithm for *partial cover* achieving a performance ratio of $\max\{d_{\max}, 2\}$.

Minimal solutions and feedback vertex set. Minimal solutions first appeared in a local ratio algorithm for FVS. FVS is NP-hard [47] and MAX SNP-hard [53], and at least as hard to approximate as vertex cover [51]. An $O(\log n)$-approximation algorithm for unweighted FVS is implies by a lemma due to Erdös and Pósa [31]. Monien and Shultz [55] improve the ratio to $\sqrt{\log n}$. Bar-Yehuda et al. [18] present a 4-approximation algorithm for unweighted FVS, and an $\mathrm{O}(\log n)$-approximation algorithm for weighted FVS. Bafna, Berman, and Fujito [9] present a local ratio 2-approximation algorithm for weighted FVS, whose weight decomposition is somewhat similar to the one used in Algorithm **FVS**. Their algorithm is the first local ratio algorithm to make use of minimal solutions (although this concept was used earlier in primal-dual algorithms for *network design* problems [58, 1, 37]). At about the same time, Becker and Geiger [20] also obtained a

2-approximation algorithm for FVS. Algorithm **FVS** is a recursive local ratio formulation of their algorithm. Chudak et al. [26] suggest yet another 2-approximation algorithm, and give primal-dual analyses of the three algorithms. Fujito [33] proposes a generic local ratio algorithm for a certain type of node-deletion problems. Algorithm **RecursiveVC**, Algorithm **FVS**, and the algorithm from [9] can be seen as applications of Fujito's generic algorithm. Bar-Yehuda [15] presents a unified local ratio approach for covering problems. He present a short generic approximation algorithm which can explain many known optimization and approximation algorithms for covering problems (including Fujito's generic algorithm). Later, Bar-Yehuda and Rawitz [19] devised a framework that extends the one from [15]. The notion of effectiveness of a weight function first appeared in [15]. The corresponding primal-dual notion appeared earlier in [22].

5 Scheduling Problems

In this section we turn to applications of the local ratio technique in the context of resource allocation and scheduling problems. Resource allocation and scheduling problems are immensely popular objects of study in the field of approximation algorithms and combinatorial optimization, owing to their direct applicability to many real-life situations and their richness in terms of mathematical structure. Historically, they were among the first to be analyzed in terms of worst-case approximation ratio, and research into these problems continues actively to this day.

In very broad terms, a scheduling problem is one in which *jobs* presenting different demands vie for the use of some limited *resource*, and the goal is to resolve all conflicts. Conflicts are resolved by scheduling different jobs at different times and either enlarging the amount of available resource to accommodate all jobs or accepting only a subset of the jobs. Accordingly, we distinguish between two types of problems. The first type is when the resource is fixed but we are allowed to reject jobs. The problem is then to maximize the number (or total weight) of accepted jobs, and there are two natural ways to measure the quality of a solution: in *throughput maximization* problems the measure is the total weight of accepted jobs (which we wish to maximize), and in *loss minimization* problems it is the total weight of rejected jobs (which we wish to minimize). While these two measures are equivalent in terms of optimal solutions, they are completely distinct when one considers approximate solutions. The second type of problem is *resource minimization.* Here we must satisfy all jobs, and can achieve this by increasing the amount of resource. The objective is to minimize the cost of doing so.

Our goal in this section is twofold. First, we demonstrate the applicability of the local ratio technique in an important field of research, and second, we take the opportunity of tackling throughput maximization problems to develop a local ratio theory for maximization problems in general. We begin with the latter. We present a local ratio theorem for maximization problems and sketch a general framework based on it in Section 5.1. We apply this framework to a collection of throughput maximization problems in Section 5.2. Following that, we consider loss minimization in Section 5.3, and resource minimization in Section 5.4.

5.1 Local Ratio for Maximization Problems

The Local Ratio Theorem for maximization problems is nearly identical to its minimization counterpart. It applies to optimization problems that can be formulated as follows.

Given a *weight vector* $w \in \mathbb{R}^n$ and a set of *feasibility constraints* $\mathcal{F}$, find a *solution vector* $x \in \mathbb{R}^n$ satisfying the constraints in $\mathcal{F}$ that maximizes the inner product $w \cdot x$.

Before stating the Local Ratio Theorem for maximization problems, we remind the reader of our convention that a feasible solution to a maximization problem is said to be r-approximate if its weight is at least $1/r$ times the optimum weight (so approximation factors are always greater than or equal to 1).

Theorem 5 (Local Ratio—Maximization Problems) *Let $\mathcal{F}$ be a set of feasibility constraints on vectors in $\mathbb{R}^n$. Let $w, w_1, w_2 \in \mathbb{R}^n$ be such that $w = w_1 + w_2$. Let $x \in \mathbb{R}^n$ be a feasible solution (with respect to $\mathcal{F}$) that is r-approximate with respect to w_1 and with respect to w_2. Then, x is r-approximate with respect to w as well.*

Proof. Let x^*, x_1^*, and x_2^* be optimal solutions with respect to w, w_1, and w_2, respectively. Clearly, $w_1 \cdot x_1^* \geq w_1 \cdot x^*$ and $w_2 \cdot x_2^* \geq w_2 \cdot x^*$. Thus, $w \cdot x = w_1 \cdot x + w_2 \cdot x \geq \frac{1}{r}(w_1 \cdot x_1^*) + \frac{1}{r}(w_2 \cdot x_2^*) \geq \frac{1}{r}(w_1 \cdot x^*) + \frac{1}{r}(w_2 \cdot x^*) = \frac{1}{r}(w \cdot x^*)$. ■

The general structure of a local ratio approximation algorithm for a maximization problem is similar to the one described for the minimization case in Section 4.2. It too consists of a three-way *if* condition that directs execution to one of three main options: *optimal solution*, *problem size reduction*, or *weight decomposition*. There are several differences though. In contrast to what is done in the minimization case, we make no effort to keep the weight function non-negative, i.e., in *weight decomposition* steps we allow w_2 to take on negative values. Also, in *problem size reduction* steps we usually remove an element whose weight is either zero or negative. Finally and most importantly, we strive to construct *maximal* solutions rather than minimal ones. This affects our choice of w_1 in *weight decomposition* steps. The weight function w_1 is chosen such that every *maximal* solution (a feasible solution that cannot be extended) is r-approximate with respect to it[2]. In accordance, when the recursive call returns in *problem size reduction* steps, we extend the solution if possible (rather than if necessary), but we attempt to do so only for zero-weight elements (not negative weight ones).

As in the minimization case, we use the notion of effectiveness.

Definition 3 *In the context of maximization problems, a weight function w is said to be* r-effective *if there exists a number b such that $b \leq w \cdot x \leq r \cdot b$ for all maximal feasible solutions x.*

5.2 Throughput Maximization Problems

Consider the following general problem. The input consists of a set of *activities*, each requiring the utilization of a given limited *resource*. The amount of resource available is fixed over time; we normalize it to unit size for convenience. The activities are specified as a collection of sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Each set represents a single activity: it consists of all possible *instances* of that activity. An instance $I \in \mathcal{A}_i$ is defined by the following parameters.

1. A half-open time interval $[s(I), e(I))$ during which the activity will be executed. We call $s(I)$ and $e(I)$ the *start-time* and the *end-time* of the instance.

[2]We actually impose a somewhat weaker condition, as described in Section 5.2.

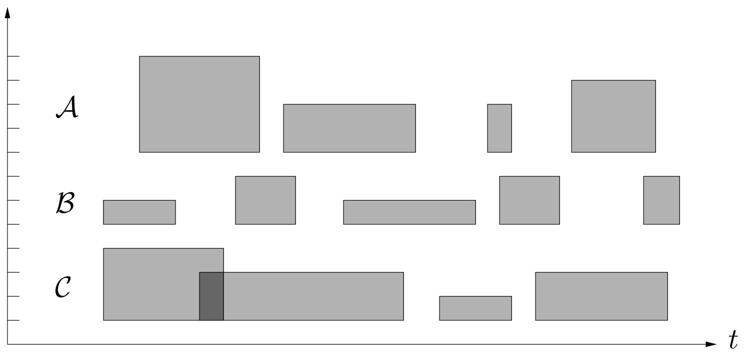

Figure 1: An example of activities and instances.

2. The amount of resource required for the activity. We refer to this amount as the *width* of the instance and denote it $d(I)$. Naturally, $0 < d(I) \leq 1$.

3. The *weight* $w(I) \geq 0$ of the instance. It represents the profit to be gained by scheduling this instance of the activity.

Different instances of the same activity may have different parameters of duration, width, or weight. A *schedule* is a collection of instances. It is *feasible* if (1) it contains at most one instance of every activity, and (2) for all time instants t, the total width of the instances in the schedule whose time interval contains t does not exceed 1 (the amount of resource available). The goal is to find a feasible schedule that maximizes the total weight of instances in the schedule. For example, consider the problem instance depicted in Figure 1. The input consists of three activities, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, each comprising several instances, depicted as rectangles in the respective rows. The projection of each rectangle on the t axis represents the corresponding instance's time interval. The height of each rectangle represents the resource requirement (the instance's width) on a 1:5 scale (e.g., the leftmost instance of activity $\mathcal{C}$ has width 0.6). The weights of the instances are not shown. Numbering the instances of each activity from left to right, the schedule $\{\mathcal{A}(1), \mathcal{C}(3), \mathcal{C}(4)\}$ is infeasible because activity $\mathcal{C}$ is scheduled twice; $\{\mathcal{A}(1), \mathcal{C}(1)\}$ is infeasible because both instances overlap and their total width is 1.4; $\{\mathcal{A}(1), \mathcal{B}(1), \mathcal{C}(4)\}$ is feasible.

In the following sections we describe local ratio algorithms for several special cases of the general problem. We use the following notation. For a given activity instance I, $\mathcal{A}(I)$ denotes the activity to which I belongs and $\mathcal{I}(I)$ denotes the set of all activity instances that intersect I but belong to activities other than $\mathcal{A}(I)$.

5.2.1 Interval Scheduling

In the *interval scheduling* problem we must schedule jobs on a single processor with no preemption. Each job consists of a finite collection of time intervals during which it may be scheduled. The problem is to select a maximum weight subset of non-conflicting intervals, at most one interval for each job. In terms of our general problem, this is simply the special case where every activity consists of a finite number of instances and the width of every instance is 1.

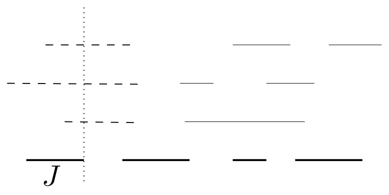

Figure 2: J, $\mathcal{A}(J)$, and $\mathcal{I}(J)$: heavy lines represent $\mathcal{A}(J)$; dashed lines represent $\mathcal{I}(J)$.

To design the weight decomposition for this problem, we examine the properties of maximal schedules. Let J be the activity instance with minimum end-time among all activity instances of all activities (breaking ties arbitrarily). The choice of J ensures that all of the intervals intersecting it intersect each other (see Figure 2). Consider a maximal schedule $\mathcal{S}$. Clearly $\mathcal{S}$ cannot contain more than one instance from $\mathcal{A}(J)$, nor can it contain more than one instance from $\mathcal{I}(J)$, since all of these instances intersect each other. Thus $\mathcal{S}$ contains at most two intervals from $\mathcal{A}(J) \cup \mathcal{I}(J)$. On the other hand, $\mathcal{S}$ must contain at least one instance from $\mathcal{A}(J) \cup \mathcal{I}(J)$, for otherwise it would not be maximal (since J could be added to it). This implies that the weight function

$$w_1(I) = \epsilon \cdot \begin{cases} 1 & I \in \mathcal{A}(J) \cup \mathcal{I}(J), \\ 0 & \text{otherwise,} \end{cases}$$

is 2-effective for any choice of $\epsilon > 0$, and we can expect to obtain a 2-approximation algorithm based on it.

A logical course of action is to fix $\epsilon = \min\{w(I) : I \in \mathcal{A}(J) \cup \mathcal{I}(J)\}$ and to solve the problem recursively on $w - w_1$, relying on two things: (1) w_1 is 2-effective; and (2) the solution returned is maximal. However, we prefer a slightly different approach. We show that w_1 actually satisfies a stronger property than 2-effectiveness. For a given activity instance I, we say that a feasible schedule is *I-maximal* if either it contains I, or it does not contain I but adding I to it will render it infeasible. Clearly, every maximal schedule is also I-maximal for any given I, but the converse is not necessarily true. The stronger property satisfied by the above w_1 is that every J-maximal schedule is 2-approximate with respect to w_1 (for all $\epsilon > 0$). To see this, observe that no optimal schedule may contain more than two activity instances from $\mathcal{A}(J) \cup \mathcal{I}(J)$, whereas every J-maximal schedule must contain at least one (if it contains none, it cannot be J-maximal since J can be added). The most natural choice of ϵ is $\epsilon = w(J)$.

Our algorithm for *interval scheduling* is based on the above observations. The initial call is $\mathbf{IS}(\mathcal{A}, w)$, where $\mathcal{A}$ is the set of jobs, which we also view as the set of all $\cup_{i=1}^{m} \mathcal{A}_i$.

```
Algorithm IS($\mathcal{A}, w$)
  1.  If $\mathcal{A} = \emptyset$, return $\emptyset$.
  2.  If there exists an interval $I$ such that $w(I) \leq 0$ do:
  3.      Return IS($\mathcal{A} \setminus \{I\}, w$).
  4.  Else:
  5.      Let $J$ be the instance with minimum end-time in $\mathcal{A}$.
  6.      Define $w_1(I) = w(J) \cdot \begin{cases} 1 & I \in \mathcal{A}(J) \cup \mathcal{I}(J), \\ 0 & \text{otherwise,} \end{cases}$
          and let $w_2 = w - w_1$.
  7.      $\mathcal{S}' \leftarrow$ IS($\mathcal{A}, w_2$).
  8.      If $\mathcal{S}' \cup \{J\}$ is feasible:
  9.          Return $\mathcal{S} = \mathcal{S}' \cup \{J\}$.
 10.      Else:
 11.          Return $\mathcal{S} = \mathcal{S}'$.
```

As with similar previous claims, the proof that Algorithm **IS** is 2-approximation is by induction on the recursion. At the basis of the recursion (Line 1) the schedule returned is optimal and hence 2-approximate. For the inductive step there are two possibilities. If the recursive call is made in Line 3, then by the inductive hypothesis the schedule returned is 2-approximate with respect to $(\mathcal{A} \setminus \{I\}, w)$, and since the weight of I is non-positive, the optimum for $(\mathcal{A}, w)$ cannot be greater than the optimum for $(\mathcal{A} \setminus \{I\}, w)$. Thus the schedule returned is 2-approximate with respect to $(\mathcal{A}, w)$ as well. If the recursive call is made in Line 7, then by the inductive hypothesis $\mathcal{S}'$ is 2-approximate with respect to w_2, and since $w_2(J) = 0$ and $S \subseteq S' \cup \{J\}$, it follows that $\mathcal{S}$ too is 2-approximate with respect to w_2. Since $\mathcal{S}$ is J-maximal by construction (Lines 8–11), it is also 2-approximate with respect to w_1. Thus, by the Local Ratio Theorem, it is 2-approximate with respect to w as well.

5.2.2 Independent Set in Interval Graphs

Consider the special case of *interval scheduling* in which each activity consists of a single instance. This is exactly the problem of finding a maximum weight independent set in an interval graph (each instance corresponds to an interval), and it is well known that this problem can be solved optimally in polynomial time (see, e.g., [39]). We claim that Algorithm **IS** solves this problem optimally too, and to prove this it suffices to show that every J-maximal solution is optimal with respect to w_1. This is so because at most one instance from $\mathcal{A}(J) \cup \mathcal{I}(J)$ may be scheduled in any feasible solution (since $\mathcal{A}(J) = \{J\}$), and every J-maximal solution schedules one.

5.2.3 Scheduling on Parallel Identical Machines

In this problem the resource consists of k parallel identical machines. Each activity instance may be assigned to any of the k machines. Thus $d(I) = 1/k$ for all I.

In order to approximate this problem we use Algorithm **IS**, but with a different choice of w_1, namely,

$$w_1(I) = w(J) \cdot \begin{cases} 1 & I \in \mathcal{A}(J), \\ 1/k & I \in \mathcal{I}(J), \\ 0 & \text{otherwise.} \end{cases}$$

The analysis of the algorithm is similar to the one used for the case $k = 1$ (i.e., interval scheduling). It suffices to show that every J-maximal schedule is 2-approximate with respect to w_1. This is so because every J-maximal schedule either contains an instance from $\mathcal{A}(J)$ or a set of instances intersecting J that prevent J from being added to the schedule. In the former case, the weight of the schedule with respect to w_1 is at least $w(J)$. In the latter case, since k machines are available but J cannot be added, the schedule must already contain k activity instances from $\mathcal{I}(J)$, and its weight (with respect to w_1) is therefore at least $k \cdot \frac{1}{k} \cdot w(J) = w(J)$. Thus the weight of every J-maximal schedule is at least $w(J)$. On the other hand, an optimal schedule may contains at most one instance from $\mathcal{A}(J)$ and at most k instances from $\mathcal{I}(J)$ (as they all intersect each other), and thus its weight cannot exceed $w(J) + k \cdot \frac{1}{k} \cdot w(J) = 2w(J)$.

Remark. Our algorithm only finds a set of activity instances that can be scheduled, but does not construct an actual assignment of instances to machines. This can be done easily by scanning the instances (in the solution found by the algorithm) in increasing order of end-time, and assigning each to an arbitrary available machine. It is easy to see that such a machine must always exist. Another approach is to solve the problem as a special case of scheduling on parallel unrelated machines (described next).

5.2.4 Scheduling on Parallel Unrelated Machines

Unrelated machines differ from identical machines in that a given activity instance may be assignable only to a subset of the machines, and furthermore, the profit derived from scheduling it on a machine may depend on the machine (i.e., it need not be the same for all allowable machines). We can assume that each activity instance may be assigned to precisely one machine. (Otherwise, simply replicate each instance once for each allowable machine.) We extend our scheduling model to handle this problem as follows. We now have k types of unit quantity resource (corresponding to the k machines), each activity instance specifies the (single) resource type it requires, and the feasibility constraint applies to each resource type separately. Since no two instance may be processed concurrently on the same machine, we set $d(I) = 1$ for all instances I.

To approximate this problem, we use Algorithm **IS** but define $\mathcal{I}(J)$ slightly differently. Specifically, $\mathcal{I}(J)$ is now defined as the set of instances intersecting J that belong to other activities *and can be scheduled on the same machine as J*. It is easy to see that again, every J-maximal schedule is 2-approximate with respect to w_1, and thus the algorithm is 2-approximation.

5.2.5 Bandwidth Allocation of Sessions in Communication Networks

Consider a scenario in which the bandwidth of a communication channel must be allocated to sessions. Here the resource is the channel's bandwidth, and the activities are sessions to be routed through the channel. A session is specified as a list of intervals in which it can be scheduled, together with a width requirement and a weight for each interval. The goal is to find the most profitable set of sessions that can utilize the available bandwidth.

To approximate this problem we first consider the following two special cases.

Special Case 1 All instances are *wide*, i.e., $d(I) > 1/2$ for all I.

Special Case 2 All activity instances are *narrow*, i.e., $d(I) \leq 1/2$ for all I.

In the case of wide instances the problem reduces to interval scheduling since no pair of intersecting instances may be scheduled together. Thus, we use Algorithm **IS** to find a 2-approximate schedule with respect to the wide instances only.

In the case of *narrow* instances we find a 3-approximate schedule by a variant of Algorithm **IS** in which w_1 is defined as follows:

$$w_1(I) = w(J) \cdot \begin{cases} 1 & I \in \mathcal{A}(J), \\ 2 \cdot d(I) & I \in \mathcal{I}(J), \\ 0 & \text{otherwise.} \end{cases}$$

To prove that the algorithm is a 3-approximation algorithm it suffices to show that every J-maximal schedule is 3-approximate with respect to w_1. (All other details are essentially the same as for interval scheduling.) A J-maximal schedule either contains an instance of $\mathcal{A}(J)$ or contains a set of instances intersecting J that prevent J from being added to the schedule. In the former case the weight of the schedule is at least $w(J)$. In the latter case, since J cannot be added, the combined width of activity instances from $\mathcal{I}(J)$ in the schedule must be greater than $1 - d(J) \geq 1/2$, and thus their total weight (with respect to w_1) must be greater than $\frac{1}{2} \cdot 2w(J) = w(J)$. Thus, the weight of every J-maximal schedule is at least $w(J)$. On the other hand, an optimal schedule may contain at most one instance from $\mathcal{A}(J)$ and at most a set of instances from $\mathcal{I}(J)$ with total width 1 and hence total weight $2w(J)$. Thus, the optimum weight is at most $3w(J)$, and therefore every J-maximal schedule is 3-approximate with respect to w_1.

In order to approximate the problem in the general case where both narrow and wide activity instances are present, we solve it separately for the narrow instances and for the wide instances, and return the solution of greater weight. Let OPT be the optimum weight for all activity instances, and let OPT_n and OPT_w be the optimum weight for the narrow instance and for the wide instances, respectively. Then, the weight of the schedule found is at least $\max\left\{\frac{1}{3}OPT_n, \frac{1}{2}OPT_w\right\}$. Now, either $OPT_n \geq \frac{3}{5}OPT$, or else $OPT_w \geq \frac{2}{5}OPT$. In either case the schedule returned is 5-approximate.

5.2.6 Continuous Input

In our treatment of the above problems we have tacitly assumed that each activity is specified as a finite set of instances. We call this type of input *discrete input*. In a generalization of the problem we can allow each activity to consist of infinitely many instances by specifying the activity as a finite collection of *time windows*. A *time window* $\mathcal{T}$ is defined by four parameters: *start-time* $s(\mathcal{T})$, *end-time* $e(\mathcal{T})$, *instance length* $l(\mathcal{T}) \leq e(\mathcal{T}) - s(\mathcal{T})$, and *weight* $w(\mathcal{T})$. It represents the set of all instances defined by intervals of length $l(\mathcal{T})$ contained in the interval $[s(\mathcal{T}), e(\mathcal{T}))$ with associated profit $w(\mathcal{T})$. We call this type of input *continuous input.* The ideas underlying our algorithms for discrete input apply equally well to continuous input, and we can achieve the same approximation guarantees. However, because infinitely many intervals are involved, the running times of the algorithms might become super-polynomial (although they are guaranteed to be finite). To obtain efficiency we can sacrifice an additive term of ϵ in the approximation guarantee in return for an implementation whose worst case time complexity is $O(n^2/\epsilon)$. Reference [11] contains the full details.

5.2.7 Throughput Maximization with Batching

The main constraint in many scheduling problems is that no two jobs may be scheduled on the same machine at the same time. However, there are situations in which this constraint is relaxed, and *batching* of jobs is allowed. Consider, for example, a multimedia-on-demand system with a fixed number of channels through which video films are broadcast to clients (e.g., through a cable TV network). Each client requests a particular film and specifies several alternative times at which he or she would like to view it. If several clients wish to view the same movie at the same time, their requests can be *batched* together and satisfied simultaneously by a single transmission. In the throughput maximization version of this problem, we aim to maximize the revenue by deciding which films to broadcast, and when, subject to the constraint that the number of channels is fixed and only a single movie may be broadcast on a given channel at any time.

Formally, the batching problem is defined as follows. We are given a set of jobs, to be processed on a system of parallel identical machines. Each job is defined by its *type*, its *weight*, and a set of *start-times.* In addition, each job type has a *processing time* associated with it. (In terms of our video broadcasting problem, machines correspond to channels, jobs correspond to clients, job types correspond to movies, job weights correspond to revenues, start-times correspond to alternative times at which clients wish to begin watching the films they have ordered, and processing times correspond to movie lengths.) A *job instance* is a pair (J, t) where J is a job and t is one of its start-times. Job instance (J, t) is said to *represent* job J. We associate with it the time interval $[t, t + p)$, where p is the processing time associated with J's type. A *batch* is a set of job instances such that all jobs represented in the set are of identical type, no job is represented more than once, and all time intervals associated with the job instances are identical. We associate with the batch the time interval associated with its job instances. Two batches *conflict in jobs* if there is a job represented in both; they *conflict in time* if their time intervals are not disjoint. A *feasible schedule* is an assignment of batches to machines such that no two batches in the schedule conflict in jobs and no two batches conflicting in time are assigned to the same machine. The goal is to find a maximum-weight feasible schedule. (The weight of a schedule is the total weight of jobs represented in it.)

The batching problem can be viewed as a variant of the scheduling problem we have been dealing with up till now. For simplicity, let us consider the single machine case. For every job, consider the set of batches in which it is represented. All of these batches conflict with each other, and we may consider them instances of a single activity. In addition, two batches with conflicting time intervals also conflict with each other, so it seems that the problem reduces to *interval scheduling.* There is a major problem with this interpretation, though, even disregarding the fact that the number of batches may be exponential. The problem is that if activities are defined as we have suggested, i.e., each activity is the set of all batches containing a particular job, then activities are not necessarily disjoint sets of instances, and thus they lack a property crucial for our approach to *interval scheduling*. Nevertheless, the same basic approach can be still be applied, though the precise details are far too complex to be included in a survey such as this. We refer the reader to Bar-Noy et al. [12], who describe a 4-approximation algorithm for *bounded* batching (where there is an additional restriction that no more than a fixed number of job instances may be batched together), and a 2-approximation algorithm for unbounded batching.

5.3 Loss Minimization

In the previous section we dealt with scheduling problems in which our aim was to maximize the profit from scheduled jobs. In this section we turn to the dual problem of minimizing the loss due to rejected jobs.

Recall that in our general scheduling problem (defined in Section 5.2) we are given a limited resource, whose amount is fixed over time, and a set of activities requiring the utilization of this resource. Each activity is a set of instances, at most one of which is to be selected. In the loss minimization version of the problem considered here, we restrict each activity to consist of a single instance, but we allow the amount of resource to vary in time. Thus the input consists of the activity specification as well as a positive function $D(t)$ specifying the amount of resource available at every time instant t. Accordingly, we allow arbitrary positive instance widths (rather than assuming that all widths are bounded by 1). A schedule is feasible if for all time instants t, the total width of instances in the schedule containing t is at most $D(t)$. Given a feasible schedule, the feasible solution it defines is the set of all activity instances *not* in the schedule. The goal is to find a minimum weight feasible solution.

We now present a variant of Algorithm **IS** achieving an approximation guarantee of 4. We describe the algorithm as one that finds a feasible schedule, with the understanding that the feasible solution actually being returned is the schedule's complement. The algorithm is as follows. Let $(\mathcal{A}, w)$ denote the input, where $\mathcal{A}$ is the description of the activities excluding their weights, and w is the weight function. If the set of all activity instances in $\mathcal{A}$ constitutes a feasible schedule, return this schedule. Otherwise, if there is a zero-weight instance I, delete it, solve the problem recursively to obtain a schedule $\mathcal{S}$, and return either $\mathcal{S} \cup \{I\}$ or $\mathcal{S}$, depending (respectively) on whether $\mathcal{S} \cup \{I\}$ is a feasible schedule or not. Otherwise, decompose w by $w = w_1 + w_2$ (as described in the next paragraph), where $w_2(I) \geq 0$ for all activity instances I, with equality for at least one instance, and solve recursively for $(\mathcal{A}, w_2)$.

Let us define the decomposition of w by showing how to compute w_1. For a given time instant t, let $\mathcal{I}(t)$ be the set of activity instances containing t. Define $\Delta(t) = \sum_{I \in \mathcal{I}(t)} d(I) - D(t)$. To compute w_1, find t^* maximizing $\Delta(\cdot)$ and let $\Delta^* = \Delta(t^*)$. Assuming $\Delta^* > 0$ (otherwise the schedule containing all instances is feasible), let

$$w_1(I) = \epsilon \cdot \begin{cases} \min\{\Delta^*, d(I)\} & I \in \mathcal{I}(t^*), \\ 0 & \text{otherwise}, \end{cases}$$

where ϵ (which depends on t^*) is the unique scaler resulting in $w_2(I) \geq 0$ for all I and $w_2(I) = 0$ for at least one I. A straightforward implementation of this algorithm runs in time polynomial in the number of activities and the number of time instants at which $D(t)$ changes value.

To prove that the algorithm is 4-approximation, it suffices (by the usual arguments) to show that w_1 is 4-effective. In other words, it suffices to show that every solution defined by a maximal feasible schedule is 4-approximate with respect to w_1. In the sequel, when we say *weight*, we mean weight with respect to w_1.

Observation 6 *Every collection of instances from $\mathcal{I}(t^*)$ whose total width is at least Δ^* has total weight at least $\epsilon\Delta^*$.*

Observation 7 *Both of the following evaluate to at most $\epsilon\Delta^*$: (1) the weight of any single instance, and (2) the total weight of any collection of instances whose total width is at most Δ^*.*

Consider an optimal solution (with respect to w_1). Its weight is the total weight of instances in $\mathcal{I}(t^*)$ that are not in the corresponding feasible schedule. Since all of these instances intersect at t^*, their combined width must be at least Δ^*. Thus, by Observation 6, the optimal weight is at least $\epsilon\Delta^*$. Now consider the complement of a maximal feasible schedule $\mathcal{M}$. We claim that it is 4-approximate because its weight does not exceed $4\epsilon\Delta^*$. To prove this, we need the following definitions. For a given time instant t, let $\overline{\mathcal{M}}(t)$ be the set of instances containing t that are not in the schedule $\mathcal{M}$, (i.e., $\overline{\mathcal{M}}(t) = \mathcal{I}(t) \setminus \mathcal{M}$). We say that t is *critical* if there is an instance $I \in \overline{\mathcal{M}}(t)$ such that adding I to $\mathcal{M}$ would violate the width constraint at t. We say that t is critical *because of I*. Note that a single time instant may be critical because of several different activity instances.

Lemma 8 *If t is a critical time instant, then $\sum_{I\in\overline{\mathcal{M}}(t)} w_1(I) < 2\epsilon\Delta^*$.*

Proof. Let J be an instance of maximum width in $\overline{\mathcal{M}}(t)$. Then, since t is critical, it is surely critical because of J. This implies that $\sum_{I\in\mathcal{M}\cap\mathcal{I}(t)} d(I) > D(t) - d(J)$. Thus,

$$\begin{aligned}
\sum_{I\in\overline{\mathcal{M}}(t)} d(I) &= \sum_{I\in\mathcal{I}(t)} d(I) - \sum_{I\in\mathcal{M}\cap\mathcal{I}(t)} d(I) \\
&= D(t) + \Delta(t) - \sum_{I\in\mathcal{M}\cap\mathcal{I}(t)} d(I) \\
&< d(J) + \Delta(t) \\
&\le d(J) + \Delta^*.
\end{aligned}$$

Hence, $\sum_{I\in\overline{\mathcal{M}}(t)\setminus\{J\}} d(I) < \Delta^*$, and therefore, by Observation 7,

$$\sum_{I\in\overline{\mathcal{M}}(t)} w_1(I) = \sum_{I\in\overline{\mathcal{M}}(t)\setminus\{J\}} w_1(I) + w_1(J) \le \epsilon\Delta^* + \epsilon\Delta^* = 2\epsilon\Delta^*.$$

■

Thus there are two cases to consider. If t^* is a critical point, then the weight of the solution is $\sum_{I\in\overline{\mathcal{M}}(t^*)} w_1(I) < 2\epsilon\Delta^*$ and we are done. Otherwise, let $t_L < t^*$ and $t_R > t^*$ be the two critical time instants closest to t^* on both sides (it may be that only one of them exists). The maximality of the schedule implies that every instance in $\overline{\mathcal{M}}(t^*)$ is the cause of criticality of at least one time instant. Thus each such instance must contain t_L or t_R (or both). It follows that $\overline{\mathcal{M}}(t^*) \subseteq \overline{\mathcal{M}}(t_L) \cup \overline{\mathcal{M}}(t_R)$. Hence, by Lemma 8, the total weight of these instances is less than $4\epsilon\Delta^*$.

5.3.1 Application: General Caching

In the *general caching* problem a replacement schedule is sought for a cache that must accommodate pages of varying sizes. The input consists of a fixed cache size $D > 0$, a collection of pages $\{1, 2, \ldots, m\}$, and a sequence of n requests for pages. Each page j has a *size* $0 < d(j) \leq D$ and a weight $w(j) \geq 0$, representing the cost of loading it into the cache. We assume for convenience that time is discrete and that the ith request is made at time i. (These assumptions cause no loss of generality as will become evident from our solution.) We denote by $r(i)$ the page being requested at time i. A *replacement schedule* is a specification of the contents of the cache at all times. It must satisfy the following condition. For all $1 \leq i \leq n$, page $r(i)$ is present in the cache at time i and the sum of sizes of the pages in the cache at that time is not greater than the cache size D. The initial contents of the cache (at time 0) may be chosen arbitrarily. Alternatively, we may insist that the cache be empty initially. The weight of a given replacement schedule is $\sum w(r(i))$ where the sum is taken over all i such that page $r(i)$ is absent from the cache at time $i - 1$. The objective is to find a minimum weight replacement schedule.

Observe that if we have a replacement schedule that evicts a certain page at some time between two consecutive requests for it, we may as well evict it immediately after the first of these requests and bring it back only for the second request. Thus, we may restrict our attention to schedules in which for every two consecutive requests for a page, either the page remains present in the cache at all times between the first request and the second, or it is absent from the cache at all times in between. This leads naturally to a description of the problem in terms of time intervals, and hence to a reduction of the problem to our loss minimization problem, as follows. Given an instance of the general caching problem, define the resource amount function by $D(i) = D - d(r(i))$ for $1 \leq i \leq n$, and $D(0) = D$ (or $D(0) = 0$ if we want the cache to be empty initially). Define the activity instances as follows. Consider the request made at time i. Let j be the time at which the previous request for $r(i)$ is made, or $j = -1$ if no such request is made. If $j + 1 \leq i - 1$, we define an activity instance with time interval $[j + 1, i - 1]$, weight $w(r(i))$, and width $d(r(i))$. This reduction implies a 4-approximation algorithm for the general caching problem via our 4-approximation algorithm for loss minimization.

5.4 Resource Minimization

Until now we have dealt with scheduling problems in which the resource was limited, and thus we were allowed to schedule only a subset of the jobs. In this section we consider the case where all jobs must be scheduled, and the resource is not limited (but must be paid for). The objective is to minimize the cost of the amount of resource in the solution. We present a 3-approximation algorithm for such a problem. We refer to the problem as the *bandwidth trading* problem, since it is motivated by bandwidth trading in next generation networks. (We shall not discuss this motivation here, as it is rather lengthy.)

The algorithm we present here is somewhat unusual in that it does not use an r-effective weight function in the weight decomposition steps. Whereas previous algorithms prune (or extend), if possible, the solution returned by the recursive call in order to turn it into a "good" solution, i.e., one that is minimal (or maximal), the algorithm we present here uses a weight function for which good solutions are solutions that satisfy a certain property different from minimality or maximality. Accordingly, it modifies the solution returned in a rather elaborate manner.

5.4.1 Bandwidth Trading

In the *bandwidth trading* problem we are given a set of machine *types* $\mathcal{T} = \{T_1, \dots, T_m\}$ and a set of *jobs* $J = \{1, \dots, n\}$. Each machine type T_i is defined by two parameters: a time interval $I(T_i)$ during which it is *available*, and a weight $w(T_i)$, which represents the cost of allocating a machine of this type. Each job j is defined by a single time interval $I(j)$ during which it must be processed. We say that job j *contains* time t if $t \in I(j)$. A given job j may be *scheduled feasibly* on a machine of type T if type T is available throughout the job's interval, i.e., if $I(j) \subseteq I(T)$. A *schedule* is a set of machines together with an assignment of each job to one of them. It is *feasible* if every job is assigned feasibly and no two jobs with intersecting intervals are assigned to the same machine. The cost of a feasible schedule is the total cost of the machines it uses, where the cost of a machine is defined as the weight associated with its type. The goal is to find a minimum-cost feasible schedule. We assume that a feasible schedule exists. (This can be checked easily.)

Our algorithm for the problem follows.

```
Algorithm BT($\mathcal{T}, J, w$)
  1.  If $J = \emptyset$, return $\emptyset$.
  2.  Else, if there exists a machine type $T \in \mathcal{T}$ such that $w(T) = 0$ do:
  3.      Let $J'$ be the set of jobs that can be feasibly scheduled on
          machines of type $T$, i.e., $J' = \{j \in J \mid I(j) \subseteq I(T)\}$.
  4.      $S' \leftarrow \mathbf{BT}(\mathcal{T} \setminus \{T\}, J \setminus J', w)$.
  5.      Extend $S'$ to all $J$ by allocating $|J'|$ machines of type $T$ and
          scheduling one job from $J'$ on each.
  6.      Return the resulting schedule $S$.
  7.  Else:
  8.      Let $t$ be a point in time contained in a maximum number of jobs, and
          let $\mathcal{T}_t$ be the set of machine types available at time $t$ (see Figure 3).
  9.      Let $\epsilon = \min\{w(T) \mid T \in \mathcal{T}_t\}$.
 10.      Define the weight functions $w_1(T) = \begin{cases} \epsilon & T \in \mathcal{T}_t, \\ 0 & \text{otherwise,} \end{cases}$
          and $w_2 = w - w_1$.
 11.      $S' \leftarrow \mathbf{BT}(\mathcal{T}, J, w_2)$.
 12.      Transform $S'$ into a new schedule $S$ (in a manner described below.
 13.      Return $S$.
```

To complete the description of the algorithm we must describe the transformation of S' to S referred to in Line 12. We shall do so shortly, but for now let us just point out two facts relating to the transformation.

1. For all machine types T, S does not use more machines of type T than S'.
2. Let k be the number of jobs containing time t (Line 8). The number of machines used by S whose types are in $\mathcal{T}_t$ is at most $3k$.

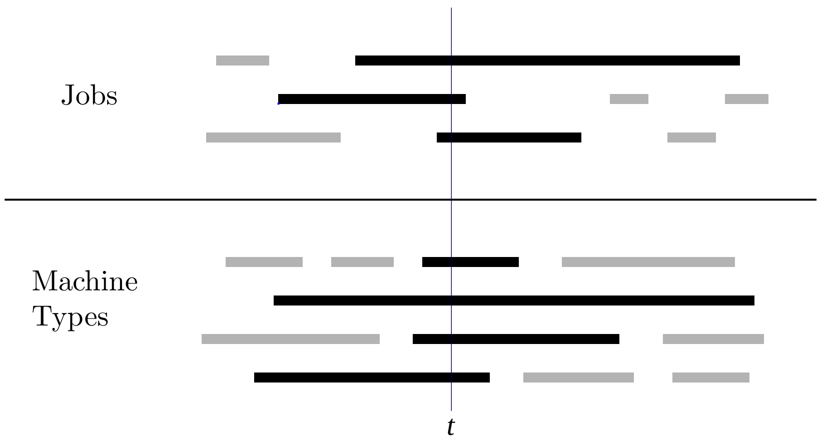

Figure 3: Jobs containing time t (top, dark), and machine types available at time t (bottom, dark).

Based on these facts, we now prove that the algorithm is a 3-approximation algorithm. The proof is by induction on the recursion. In the base case ($J = \emptyset$), the schedule returned is optimal and therefore 3-approximate. For the inductive step there are two cases. If the recursive invocation is made in Line 4, then by the inductive hypothesis, S' is 3-approximate with respect to $(\mathcal{T} \setminus \{T\}, J \setminus J', w)$. In addition, no job in $J \setminus J'$ can be scheduled feasibly on a machine of type T; all jobs in J' can be scheduled feasibly on machines of type T; and machines of type T are free ($w(T) = 0$). Thus $w(S) = w(S')$, and the optimum cost for $(\mathcal{T}, J, w)$ is the same as for $(\mathcal{T} \setminus \{T\}, J \setminus J', w)$. Therefore S is 3-approximate. The second case in the inductive step is that the recursive call is made in Line 11. In this case, S' is 3-approximate with respect to w_2 by the inductive hypothesis. By the first fact above, $w_2(S) \leq w_2(S')$, and therefore S too is 3-approximate with respect to w_2. By the second fact above, $w_1(S) \leq 3k\epsilon$, and because there are k jobs containing time t—each of which can be scheduled only on machines whose types are in $\mathcal{T}_t$, and no two of which may be scheduled on the same machine—the optimum cost is at least $k\epsilon$. Thus S is 3-approximate with respect to w_1. By the Local Ratio Theorem, S is therefore 3-approximate with respect to w.

It remains to describe the transformation of S' to S in Line 12. Let $J_t \subseteq J$ be the set of jobs containing time t, and recall that $k = |J_t|$. An example (with $k = 3$) is given in Figure 3. The strips above the line represent jobs, and those below the line represent machine types. The darker strips represent the jobs in J_t and the machine types in $\mathcal{T}_t$. Let $\mathcal{M}_t \subseteq \mathcal{M}_S$ be the set of machines that are available at time t and are used by S, and let $J_{\mathcal{M}_t}$ be the set of jobs scheduled by S on machines in $\mathcal{M}_t$. ($J_{\mathcal{M}_t}$ consists of the jobs J_t and possibly additional jobs.) If $|\mathcal{M}_t| \leq 3k$ then $S' = S$. Otherwise, we choose a subset of $\mathcal{M}'_t \subseteq \mathcal{M}_t$ of size at most $3k$ and reschedule all of the jobs in $J_{\mathcal{M}_t}$ on these machines. The choice of $\mathcal{M}'_t$ and the construction of S' are as follows.

1. Let $\mathcal{M}_c \subseteq \mathcal{M}_t$ be the set of k machines to which the k jobs in J_t are assigned. (Each job must be assigned to a different machine since they all exist at time t). Let J_c be the set of all jobs scheduled on machines in $\mathcal{M}_c$. We schedule these jobs the same as in S.

2. Let $\mathcal{M}_l \subseteq \mathcal{M}_t \setminus \mathcal{M}_c$ be the set of k machines in $\mathcal{M}_t \setminus \mathcal{M}_c$ with leftmost left endpoints. (See example in Figure 4.) Let $J_l \subseteq J_{\mathcal{M}_t}$ be the set of jobs in $J_{\mathcal{M}_t}$ that lie completely to the left of time point t and are scheduled by S on machines in $\mathcal{M}_t \setminus \mathcal{M}_c$. We schedule these jobs

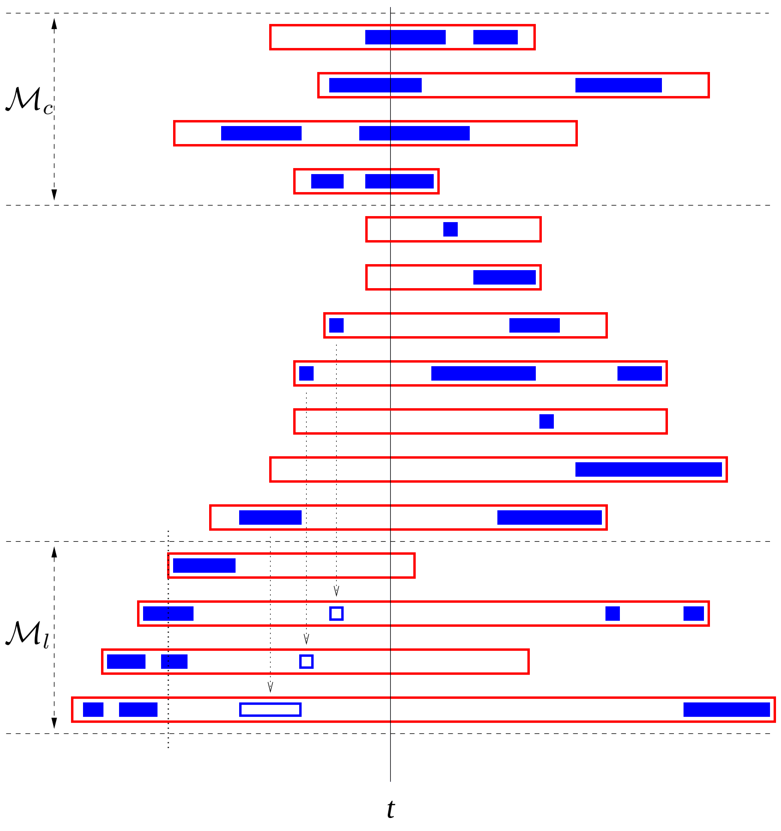

Figure 4: Rescheduling jobs that were assigned to $\mathcal{M}_t$ and exist before time t.

on machines from $\mathcal{M}_l$ as follows. Let t' be the rightmost left endpoint of a machine in $\mathcal{M}_l$. The jobs in J_l that contain t' must be assigned in S to machines from $\mathcal{M}_l$. We retain their assignment. We proceed to schedule the remaining jobs in J_l greedily by order of increasing left endpoint. Specifically, for each job j we select any machine in $\mathcal{M}_l$ on which we have not already scheduled a job that conflicts with j and schedule j on it. This is always possible since all k machines are available between t' and t, and thus if a job cannot be scheduled, its left endpoint must be contained in k other jobs that have already been assigned. These $k+1$ jobs coexist at the time instant defining the left endpoint of the job that cannot be assigned, in contradiction with the fact that k is the maximal number jobs coexisting at a any time.

3. Let $\mathcal{M}_r \subseteq \mathcal{M}_t \setminus \mathcal{M}_c$ be the set of k machines in $\mathcal{M}_t \setminus \mathcal{M}_c$ with rightmost right endpoints. ($\mathcal{M}_r$ and $\mathcal{M}_l$ and not necessarily disjoint.) Let $J_r \subseteq J_{\mathcal{M}_t}$ be the set of jobs in $J_{\mathcal{M}_t}$ that lie completely to the right of time point t and are scheduled by S on machines in $\mathcal{M}_t \setminus \mathcal{M}_c$. We schedule these jobs on machines from $\mathcal{M}_r$ in a similar manner to the above.

We have thus managed to schedule $J_c \cup J_l \cup J_r = J_{\mathcal{M}_t}$ on no more than $3k$ machines from $\mathcal{M}_t$, as desired.

5.5 Background

Throughput maximization. Single machine scheduling with one instance per activity is equivalent to *maximum weight independent set in interval graphs* and hence polynomial-time solvable [39]. Arkin and Silverberg [3] solve the problem efficiently even for unrelated multiple machines. The problem becomes NP-hard (even in the single machine case) if multiple instances per activity are allowed [61] (i.e., the problem is *interval scheduling*) or if instances may require arbitrary amounts of the resource. (In the latter case the problem is NP-hard as it contains *knapsack* [35] as a special case in which all time intervals intersect.) Spieksma [61] studies the unweighted interval scheduling problem. He proves that it is Max-SNP-hard, and presents a simple greedy 2-approximation algorithm. Bar-Noy et al. [13] consider *real-time scheduling*, in which each job is associated with a release time, a deadline, a weight, and a processing time on each of the machines. They give several constant factor approximation algorithms for various variants of the throughput maximization problem. They also show that the problem of scheduling unweighted jobs on unrelated machine is Max-SNP-hard.

Bar-Noy et al. [11], present a general framework for solving resource allocation and scheduling problems that is based on the local ratio technique. Given a resource of fixed size, they present algorithms that approximate the maximum throughput or the minimum loss by a constant factor. The algorithms apply to many problems, among which are: real-time scheduling of jobs on parallel machines; bandwidth allocation for sessions between two endpoints; general caching; dynamic storage allocation; and bandwidth allocation on optical line and ring topologies. In particular, they improve most of the results from [13] either in the approximation factor or in the running time complexity. Their algorithms can also be interpreted within the primal-dual schema (see also [19]) and are the first local ratio (or primal-dual) algorithms for a maximization problems. Sections 5.2 and 5.3, with the exception of Section 5.2.7, are based on [11].

Independently, Berman and DasGupta [21] also improve upon the algorithms given in [13]. They develop an algorithm for interval scheduling that is nearly identical to the one from [11]. Furthermore, they employ the same rounding idea used in [11] in order to contend with time windows. In addition to single machine scheduling, they also consider scheduling on parallel machines, both identical and unrelated.

Chuzhoy et al. [27] consider the unweighted *real-time scheduling* problem and present an $(e/(e-1)+\epsilon)$-approximation algorithm. They generalize this algorithm to achieve a ratio of $(1+e/(e-1)+\epsilon)$ for unweighted *bandwidth allocation*.

The batching problem discussed in Section 5.2.7 is from Bar-Noy et al. [12]. In the case of bounded batching, they describe a 4-approximation algorithm for discrete input and a $(4+\epsilon)$-approximation algorithm for continuous input. In the case on unbounded batching, their approximation factors are 2 and $2+\epsilon$, respectively. However, in the discrete input case the factor 2 is achieved under an additional assumption. (See [12] for more details.) In the *parallel batch processing* model all jobs belong to the same family, and any group of jobs can be batched together. A batch is completed when the largest job in the batch is completed. This model was studied by Brucker et al. [24] and by Baptiste [10]. The model discussed in [12] is called *batching with incompatible families.* This model was studied previously with different objective functions such as *weighted sum of completion times* [62, 30, 8] and *total tardiness* [54].

Loss minimization. Section 5.3 is based on [11]. Albers et al. [2] consider the *general caching* problem. They achieve a "pseudo" O(1)-approximation factor (using LP rounding) by increasing the size of the cache by O(1) times the size of the largest page, i.e., their algorithm finds a solution using the enlarged cache whose cost is within a constant factor of the optimum for for the original cache size. When the cache size may not be increased, they achieve an $O(\log(M+C))$ approximation factor, where M and C denote the cache size and the largest page reload cost, respectively. The reduction of *general caching* to the loss minimization problem discussed in Section 5.3 is also from [2].

Resource minimization. Section 5.4 is based on a write-up by Bhatia et al. [23]. The general model of the resource minimization problem, where the sets of machine types on which a job can be processed are arbitrary, is essentially equivalent (approximation-wise) to *set cover* [23, 45]. Kolen and Kroon [49, 50] show that versions of the general problem considered in [23] are NP-hard.

Acknowledgments

We thank the anonymous reviewer for many corrections and suggestions. One of the suggestions was a proof of the 2-effectiveness of the weight function used in the context of *feedback vertex set* (Section 4.3) which was considerably simpler than our original proof. This proof inspired the proof that now appears in the text.

References

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, 24(3):440–456, 1995.

[2] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 31–40, 1999.

[3] E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18:1–8, 1987.

[4] S. Arora and C. Lund. Hardness of approximations. In Hochbaum [44], chapter 10, pages 399–446.

[5] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[6] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, 1998.

[7] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation; Combinatorial optimization problems and their approximability properties.* Springer Verlag, 1999.

[8] M. Azizoglu and S. Webster. Scheduling a batch processing machine with incompatible job families. *Computer and Industrial Engineering*, 39(3–4):325–335, 2001.

[9] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM Journal on Discrete Mathematics*, 12(3):289–297, 1999.

[10] P. Baptiste. Batching identical jobs. *Mathematical Methods of Operations Research*, 52(3):355–367, 2000.

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

[12] A. Bar-Noy, S. Guha, Y. Katz, J. Naor, B. Schieber, and H. Shachnai. Throughput maximization of real-time scheduling with batching. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 742–751, 2002.

[13] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Approximating the throughput of multiple machines in real-time scheduling. *SIAM Journal on Computing*, 31(2):331–352, 2001.

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

[15] R. Bar-Yehuda. One for the price of two: A unified approach for approximating covering problems. *Algorithmica*, 27(2):131–144, 2000.

[16] R. Bar-Yehuda and S. Even. A linear time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981.

[17] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46, 1985.

[18] R. Bar-Yehuda, D. Geiger, J. Naor, and R. M. Roth. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference. *SIAM Journal on Computing*, 27(4):942–959, 1998.

[19] R. Bar-Yehuda and D. Rawitz. On the equivalence between the primal-dual schema and the local-ratio technique. In *4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, volume 2129 of *LNCS*, pages 24–35, 2001.

[20] A. Becker and D. Geiger. Optimization of Pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. *Artificial Intelligence*, 83(1):167–188, 1996.

[21] P. Berman and B. DasGupta. Multi-phase algorithms for throughput maximization for real-time scheduling. *Journal of Combinatorial Optimization*, 4(3):307–323, 2000.

[22] D. Bertsimas and C. Teo. From valid inequalities to heuristics: A unified view of primal-dual approximation algorithms in covering problems. *Operations Research*, 46(4):503–514, 1998.

[23] R. Bhatia, J. Chuzhoy, A. Freund, and J. Naor. Algorithmic aspects of bandwidth trading. In *30th International Colloquium on Automata, Languages, and Programming*, volume 2719 of *LNCS*, pages 751–766, 2003.

[24] P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. N. Potts, T. Tautenhahn, and S. L. van de Velde. Scheduling a batching machine. *Journal of Scheduling*, 1(1):31–54, 1998.

[25] N. H. Bshouty and L. Burroughs. Massaging a linear programming solution to give a 2-approximation for a generalization of the vertex cover problem. In *15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *LNCS*, pages 298–308. Springer, 1998.

[26] F. A. Chudak, M. X. Goemans, D. S. Hochbaum, and D. P. Williamson. A primal-dual interpretation of recent 2-approximation algorithms for the feedback vertex set problem in undirected graphs. *Operations Research Letters*, 22:111–118, 1998.

[27] J. Chuzhoy, R. Ostrovsky, and Y. Rabani. Approximation algorithms for the job interval selection problem and related scheduling problems. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 348–356, 2001.

[28] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[29] I. Dinur and S. Safra. The importance of being biased. In *34th ACM Symposium on the Theory of Computing*, pages 33–42, 2002.

[30] G. Dobson and R. S. Nambimadom. The batch loading and scheduling problem. *Operations Research*, 49(1):52–65, 2001.

[31] P. Erdös and L. Pósa. On the maximal number of disjoint circuits of a graph. *Publ. Math. Debrecen*, 9:3–12, 1962.

[32] U. Feige. A threshold of $\ln n$ for approximating set cover. In *28th Annual Symposium on the Theory of Computing*, pages 314–318, 1996.

[33] T. Fujito. A unified approximation algorithm for node-deletion problems. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 86:213–231, 1998.

[34] R. Gandhi, S. Khuller, and A. Srinivasan. Approximation algorithms for partial covering problems. In *28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *LNCS*, pages 225–236, 2001.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[36] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[37] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

[38] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Hochbaum [44], chapter 4, pages 144–191.

[39] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[40] E. Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In *11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 329–337, 2000.

[41] J. Håstad. Some optimal inapproximability results. In *29th Annual ACM Symposium on the Theory of Computing*, pages 1–10, 1997.

[42] D. S. Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on Computing*, 11(3):555–556, 1982.

[43] D. S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6:243–254, 1983.

[44] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problem.* PWS Publishing Company, 1997.

[45] K. Jansen. An approximation algorithm for the license and shift class design problem. *European Journal of Operational Research*, 73:127–131, 1994.

[46] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[47] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[48] M. Kearns. *The Computational Complexity of Machine Learning.* M.I.T. Press, 1990.

[49] A. W. J. Kolen and L. G. Kroon. On the computational complexity of (maximum) class scheduling. *European Journal of Operational Research*, 54:23–38, 1991.

[50] A. W. J. Kolen and L. G. Kroon. An analysis of shift class design problems. *European Journal of Operational Research*, 79:417–430, 1994.

[51] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary problems is NP-complete. *Journal of Computer and System Sciences*, 20:219–230, 1980.

[52] L. Lovász. On the ratio of optimal integral and fractional covers. *Discreate Mathematics*, 13:383–390, 1975.

[53] C. Lund and M. Yannakakis. The approximation of maximum subgraph problems. In *20th International Colloquium on Automata, Languages and Programming*, volume 700 of *LNCS*, pages 40–51, July 1993.

[54] S. V. Mehta and R. Uzsoy. Minimizing total tardiness on a batch processing machine with incompatable job famalies. *IIE Transactions*, 30(2):165–178, 1998.

[55] B. Monien and R. Shultz. Four approximation algorithms for the feedback vertex set problem. In *7th conference on graph theoretic concepts of computer science*, pages 315–390, 1981.

[56] B. Monien and E. Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22:115–123, 1985.

[57] G. L. Nemhauser and L. E. Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[58] R. Ravi and P. Klein. When cycles collapse: A general approximation technique for constrained two-connectivity problems. In *3rd Conference on Integer Programming and Combinatorial Optimization*, pages 39–56, 1993.

[59] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *29th ACM Symposium on the Theory of Computing*, pages 475–484, 1997.

[60] P. Slavík. Improved performance of the greedy algorithm for partial cover. *Information Processing Letters*, 64(5):251–254, 1997.

[61] F. C. R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2(5):215–227, 1999.

[62] R. Uzsoy. Scheduling batch processing machines with incompatible job families. *International Journal of Production Research*, 33:2685–2708, 1995.