Local Ratio: A Unified Framework for Approximation Algorithms

REUVEN BAR-YEHUDA and KEREN BENDEL
Technion—Israel Institute of Technology
and
ARI FREUND
Caesarea Rotschild Institute, University of Haifa
and
DROR RAWITZ
Technion—Israel Institute of Technology

The *local ratio* technique is a methodology for the design and analysis of algorithms for a broad range of optimization problems. The technique is remarkably simple and elegant, and yet can be applied to several classical and fundamental problems (including covering problems, packing problems, and scheduling problems). The local ratio technique uses elementary math and requires combinatorial insight into the structure and properties of the problem at hand. Typically, when using the technique, one has to invent a weight function for a problem instance under which every "reasonable" solution is "good." The local ratio technique is closely related to the *primal-dual* schema, though it is not based on weak LP duality (which is the basis of the primal-dual approach) since it is not based on linear programming.

In this survey we introduce the local ratio technique and demonstrate its use in the design and analysis of algorithms for various problems. We trace the evolution path of the technique since its inception in the 1980's, culminating with the most recent development, namely, *fractional local ratio*, which can be viewed as a new LP rounding technique.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; *Sequencing and scheduling*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Approximation algorithms for NP-hard problems, dynamic storage allocation, general caching, scheduling, resource allocation

Most of the work on this survey was done while the third author was at the Technion Department of Computer Science.

Authors' addresses: R. Bar-Yehuda, K. Bendel, and D. Rawitz, Department of Computer Science, Technion, Haifa 32000, Israel, email: {reuven,bkeren,rawitz}@cs.technion.ac.il; A. Freund, Caesarea Rothschild Institute, University of Haifa, Haifa 31905, Israel, email: arief@cs.haifa.ac.il.

1. INTRODUCTION

The *local ratio* technique is an approximation paradigm for NP-hard optimization problems which has been enjoying a great measure of success since its inception in the early 1980's. Its main feature of attraction is its simplicity and elegance; it is very easy to understand, and has surprisingly broad applicability. The technique is based on exploiting the combinatorial structure of the problem at hand—often by making just one or two straightforward observations—and is therefore accessible to the non-specialist. In this sense the technique is not unlike *dynamic programming*; it is easily mastered, and can be used effectively in a wide variety of problems. Although it is often the case that employing heavier machinery can produce better results, the local ratio technique usually provides a good head start on the problem.

In this survey we introduce the local ratio technique and demonstrate its use in the design and analysis of algorithms for various problems. Among the multitude of algorithms we describe some are algorithms that were developed initially using the technique, and others are local ratio interpretations of pre-existing algorithms. Several of them appear here for the first time. Whenever possible we present a unified analysis for a number of similar algorithms. We cover minimization and maximization problems, approximation and exact optimization algorithms, classical results and more recent developments.

Before going any further, let us briefly introduce the basic notions of optimization and approximation, as the intended audience of this survey is the general Computer Science community. (We give more complete and precise definitions in Section 2.) An *optimization problem* is one where we seek a solution of minimum or maximum cost among a set of candidate solutions. For example, if the problem is to find the longest (simple) path between two given vertices in a graph, the candidate solutions are all simple paths connecting the two vertices, and the cost of each path is its length. Since this problem, and many others, are NP-hard, we are often interested in efficiently obtaining *approximate* solutions, i.e., solutions that, although suboptimal, are not "far" from optimal. A solution whose cost is within a factor of r of the optimum is said to be *r-approximate*. Thus, in the *longest path* example, a path whose length is at least one half the length of the longest path is 2-approximate. An *r-approximation* algorithm for a given problem is an algorithm that finds r-approximate solutions.

Say we want to construct an r-approximation algorithm for a minimization problem. A key step in the design of such an algorithm is to establish a good lower bound b on the cost of the optimal solution. This bound can later be used in the analysis to prove that the solution found by the algorithm is r-approximate by showing that its cost is no more than $r \cdot b$. The local ratio technique uses a "local" variation of this idea. In essence, the idea is to break down the cost w of the solution found by the algorithm into a sum of "partial costs" $w = w_1 + w_2 + \ldots + w_k$, and similarly break down the lower bound b into $b = b_1 + b_2 + \ldots + b_k$, and to show that $w_i \leq r \cdot b_i$ for all i. (For maximization problems, b is an upper bound and we show that $w_i \geq b_i/r$ for all i.) The breakdown of w and b is determined by the manner 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 a breakdown exist. Put differently, at the ith step, the algorithm "pays" $r \cdot b_i$ and manipulates

the problem instance so that the optimum cost drops by at least b_i.

To give a first taste of the local ratio technique, we demonstrate it on the well known *vertex cover* problem. In this problem we are given a graph $G = (V, E)$ and a non-negative weight function w on its vertices, and the objective is to find a minimum weight *vertex cover*, i.e., a minimum weight set of vertices $C \subseteq V$ such that every edge in the graph has at least one endpoint in C. Let us develop a 2-approximation algorithm for this problem in intuitive terms. Imagine that we are shopping for a vertex cover of the graph at the local branch of *Graphs'Я'Us*. The direct approach would be to somehow select a vertex cover and then pay for it, but rather than doing so, we agree with the shopkeeper on the following method of payment. We repeatedly select vertices and make certain down payments on them. The down payment we make each time may be less than the actual cost of the vertex, and we may make several payments on the same vertex. The shopkeeper, in turn, does not want to be bothered keeping track of all of our down payments, so he adopts the following strategy. Whenever we make a payment of ϵ on a vertex, he responds by simply decreasing the price of that vertex by ϵ.

More specifically, we conduct the business transaction in rounds. In each round we select an edge (u, v) whose two endpoints have non-zero cost, and make a down payment of $\epsilon = \min\{\text{price of } u, \text{price of } v\}$ on each of its endpoints. In response, the shopkeeper lowers the prices of u and v by ϵ each. With these new prices in effect, at least one of the edge's endpoints has zero cost. Thus after $O(|V|)$ rounds, prices will have dropped sufficiently for each edge to have at least one zero-cost endpoint. When that happens, the set of all zero-cost vertices is a vertex cover, and it is free. We take this set as our solution.

We formalize the above process as the following algorithm.

Algorithm VC

1. While there exists an edge (u, v) such that $\min\{w(u), w(v)\} > 0$:
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 $C = \{v \mid w(v) = 0\}$.

What is the cost of the solution in terms of the original weights, and how far is it from the optimal cost? Consider the ith round. Let (u_i, v_i) be the edge selected in this round, and let ϵ_i be the down payment made on each of its endpoints. Since every vertex cover must contain at least one of u_i and v_i (in order to cover the edge connecting them), decreasing both their prices by ϵ_i has the effect of lowering the optimal cost, denoted OPT, by at least ϵ_i. Thus, in the ith round we pay $2\epsilon_i$ and effect a drop of at least ϵ_i in OPT. Hence the *local ratio* between our payment and the drop in OPT is at most 2 in every round. It follows that the ratio between our total payment and the total drop in OPT, summed over all rounds, is at most 2. Clearly, the cost of the solution we find is fully covered by the sum of all down payments we make, and the total drop in the optimal cost is the original value of

OPT (as evidenced by the fact that we find a vertex cover of zero cost with respect to the final weights). Thus the cost of our solution is at most $2OPT$.

Note that our approximation ratio analysis makes no reference to the actual value of ϵ in any given iteration. The choice of $\epsilon = \min\{w(u), w(v)\}$ is not compulsory; any $0 \leq \epsilon \leq \min\{w(u), w(v)\}$ would do just fine and would result in a factor 2 approximation. We use $\epsilon = \min\{w(u), w(v)\}$ for reasons of efficiency. This choice ensures that the number of vertices with positive cost decreases with each iteration, thus limiting the number of iterations to less than $|V|$. (Note that although a naive implementation of the algorithm runs in $\Theta(|V||E|)$ time, the algorithm can be made to run in linear time by performing a single sweep through the edges in arbitrary order.)

Another point worth noting is that our analysis of algorithm **VC** contains some slack in that it bounds the cost of the solution by the sum of *all* down payments, rather than by the sum of down payments made for vertices that are actually taken to the solution. It might seem that a more refined analysis could yield a better approximation ratio, but unfortunately this is not true for the *vertex cover* problem; one can easily construct examples in which all vertices for which down payments are made are eventually taken to the solution. Another source of slack is that in the final step, all zero-cost vertices are taken to the solution, and no attempt is made to prune it by removing unnecessary vertices. One obvious idea is to return a *minimal* (with respect to set inclusion) subset of C that is still a vertex cover. This idea fails too, as it is easy to construct worst-case examples in which C is minimal to begin with. As we shall see, though, these ideas are useful (indeed, 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 [1981] on *vertex cover* and *set cover*. In this paper, the authors presented a generalization of Algorithm **VC** suited for *set cover*, and gave a primal-dual analysis of it. This linear time algorithm was motivated by a previous algorithm of Hochbaum [1982] which was based on LP duality[1] and required the solution of a linear program. Although Bar-Yehuda and Even's primal-dual analysis contains an implicit local ratio argument, the debut of the local ratio technique only occured in a followup paper [Bar-Yehuda and Even 1985] several years later, where the authors presented a local ratio analysis of the same algorithm. They also developed a specialized $(2 - \frac{\log_2 \log_2 n}{2 \log_2 n})$-approximation algorithm for *vertex cover* based on local-ratio principles. More than a decade later, Bafna et al. [1999] devised a local ratio 2-approximation algorithm for the *feedback vertex set* problem. In this landmark paper, they incorporated the idea of *minimal solutions* into the local ratio technique. Subsequently, Fujito [1998] presented a unified local ratio approximation algorithm for node-deletion problems, and later still, Bar-Yehuda [2000] developed a generic local ratio algorithm that explained most local ratio and primal-dual ap-

[1]LP stands for *linear programming*. The use of linear programming in the context of approximation algorithms is beyond the scope of this survey, as is the *primal-dual* schema, which is an approximation paradigm based on linear programming concepts. A good survey of the primal-dual schema can be found in [Williamson 2001].

proximation algorithms known at the time. At this point the local ratio technique had reached a certain level of maturity, but only in the context of minimization problems. No local ratio (or primal-dual) algorithms were known for maximization problems at the time. This changed in a paper by Bar-Noy et al. [2001], who presented the first local ratio algorithms for maximization problems. The key idea facilitating these algorithms was the use of *maximal* solutions rather than minimal ones. More recently, Bar-Yehuda and Rawitz [2001] developed two frameworks, one extending the generic algorithm from Bar-Yehuda [2000], and the other extending known primal-dual frameworks [Goemans and Williamson 1997; Bertsimas and Teo 1998], and showed that both are equivalent, thus merging two seemingly independent lines of research. The most recent development, due to Bar-Noy et al. [2002], is a novel extension of the local ratio technique called *fractional local ratio*.

1.2 Overview of this Survey

The survey is organized as follows. In Section 2 we define the basic notions that are used in the field of approximation algorithms. We also establish some terminology and notation to be used later in the survey. In Section 3 we begin the survey proper by presenting two introductory examples in the spirit of Algorithm **VC** above. In Section 4 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 5 we introduce the idea of minimal solutions into the framework, making it powerful enough to encompass many known approximation algorithms for covering problems (e.g., *feedback vertex set* and *network design*). In Section 6 we cover local ratio algorithms for scheduling problems, focusing mainly on maximization problems. As a first step, we develop a local ratio framework for maximization problems, which is, in a sense, a mirror image of its minimization counterpart developed in Sections 4 and 5. In Section 7 we present several algorithms that deviate from the standard local ratio approach. In particular we show local ratio analyses of exact optimization algorithms (for polynomial-time solvable problems). In the final section of the survey, Section 8, we describe a recent development, namely, the fractional local ratio technique.

Since this write-up is intended as a primer as well as survey, it is written somewhat textbook style. We have removed nearly all citations and references from the running text, and instead have included at the end of each major section a subsection titled *Background*, in which we cite sources for the material covered in the section and discuss related work. The first such subsection, discussing *vertex cover*, appears next.

1.3 Background

The *vertex cover* problem is known to be NP-complete even for planar cubic graphs with unit weights [Garey et al. 1976]. Håstad [1997] proves a lower bound on the approximability of the problem. He shows, using PCP arguments, that *vertex cover* cannot be approximated within a factor of $\frac{7}{6}$ unless P=NP. Dinur and Safra [2002] improve this bound to $10\sqrt{5} - 21 \approx 1.36067$. The first 2-approximation algorithm for the weighted version of *vertex cover* is due to Nemhauser and Trotter [1975]. Hochbaum [1983] 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 [Garey and Johnson 1979]) gives a linear time 2-approximation algorithm for the non-weighted case. (Algorithm **VC** reduces to this algorithm on non-weighted instances.) Hochbaum [1982] presents two approximation algorithms for *set cover*, both requiring the solution of a linear program. The first constructs a cover based on the optimal dual solution, whereas the second is a simple LP rounding algorithm. Her algorithms achieve an approximation guarantee of 2 on instances that are graphs (i.e., on *vertex cover* instances). Bar-Yehuda and Even [1981] present an LP based approximation algorithm for weighted *set cover* that does not solve a linear program directly. Rather, 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 [1985] 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 [1985] achieved the same ratio for the unweighted case. Recently, Halperin [2000] improved this result to $2 - (1 - o(1))\frac{2 \ln \ln d_{\max}}{\ln d_{\max}}$ using semidefinite programming.

2. DEFINITIONS AND NOTATION

2.1 Optimization Problems

An optimization problem is a family of problem *instances*, i.e., possible inputs. With each instance is associated a collection of *solutions*, each of which is either *feasible* or *infeasible*, and a *cost* function assigning a cost to each solution. Each optimization problem is classified as either a *maximization* problem or a *minimization* problem. For a given problem instance, an *optimal* solution is a feasible solution whose cost is optimal, which is to say that 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 referred to as the *optimum value*, or simply the *optimum*. For example, the familiar *minimum spanning tree* problem is a minimization problem in which the instances are edge-weighted graphs, solutions are subgraphs, feasible solutions are spanning trees, the cost of a given solution is the total weight of its edges, and optimal solutions are spanning trees of minimum total edge weight. Throughout this survey we use the terms *cost* and *weight* interchangeably.

Many optimization problems can be formulated as problems of selecting a subset (satisfying certain constraints) of a given set of weighted objects. For example, *minimum spanning tree* can be viewed as the problem of selecting a subset of the edges forming 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. Nearly all of the problems we shall encounter in this survey are problems of this type.

2.2 Approximation Algorithms

An approximation algorithm for an optimization problem is an algorithm that takes an input instance and finds a feasible solution for it. The word *approximation* in

the term *approximation algorithm* expresses the idea that our goal in the design of such an algorithm is to make it so that the algorithm will always return a feasible solution whose value approximates the optimum in some sense. Approximation algorithms are of interest chiefly in the context of NP-hard problems, where finding optimal solutions efficiently is not possible (unless P=NP), but finding approximate solutions efficiently quite often is.

The most popular measure of closeness to the optimum is *approximation ratio*, defined as follows. For $r \geq 1$, a feasible solution is said to be *r-approximate* if its cost is within a factor of r of the optimum. More specifically, let $w(X)$ be the cost of a given feasible solution X and let w^* be the optimum value. Then, in the case of minimization, X is said to be r-approximate if $w(X) \leq r \cdot w^*$, and in the case of maximization, it is said to be r-approximate if $w(X) \geq w^*/r$. (Note that for both types of problems the smaller r is, the closer X is to being optimal, and if $r = 1$, X is optimal.) In the context of approximation algorithms we only consider problems in which the optimum is non-negative for all input instances. An r-approximate solution is also called an *r-approximation*. An algorithm that always returns 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 as a synonym for *performance guarantee*.

2.3 Conventions and Notation

In the sequel 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 for a given problem instance by OPT.

—Graphs are simple and undirected. A graph is denoted $G = (V, E)$. Whenever a digraph is specified, it is denoted $G = (N, A)$. We denote the number of vertices/nodes by n and the number of edges/arcs by m. We denote the degree of vertex v by $\deg(v)$.

—We denote by H_n the nth harmonic number, that is, $H_n = \sum_{i=1}^{n} \frac{1}{i}$.

3. TWO INTRODUCTORY EXAMPLES

In this section we demonstrate two easy applications of the local ratio technique. We describe approximation algorithms for the *prize collecting* version of *vertex cover* and for a certain *graph editing* problem. Our treatment of these problems is similar to the treatment of *vertex cover* in the introduction.

3.1 Prize Collecting Vertex Cover

The *prize collecting vertex cover* problem is a generalization of *vertex cover* in which we are not obligated to cover all edges, but must pay a penalty for those left uncovered. More specifically, both vertices and edges have non-negative weights; every set of vertices is a feasible solution; and the cost of a feasible solution is the total weight of its vertices plus the total weight of the edges it does not cover. Our algorithm for this problem is very similar to Algorithm **VC**.

Algorithm PCVC

1. While there exists an edge $e = (u, v)$ such that $\min\{w(u), w(v), w(e)\} > 0$:
2. Let $\epsilon = \min\{w(u), w(v), w(e)\}$.
3. $w(u) \leftarrow w(u) - \epsilon$.
4. $w(v) \leftarrow w(v) - \epsilon$.
5. $w(e) \leftarrow w(e) - \epsilon$.
6. Return the feasible solution $C = \{v \mid w(v) = 0\}$.

The analysis here is similar to that of *vertex cover*. First, observe that when the algorithm halts, every edge not covered by C has zero cost, and thus the cost of C is zero. Next, consider the ith iteration, in which edge $e_i = (u_i, v_i)$ is involved and ϵ_i is the amount by which the weights are reduced. We pay $3\epsilon_i$ and effect a drop of at least ϵ_i in the optimal cost (since every feasible solution must either contain one (or both) of u_i and v_i, or else pay for the edge e_i). Thus, a factor of 3 is immediate. However, we can tighten the analysis by taking into account only payments that actually contribute towards covering the cost of C. Of the $3\epsilon_i$ we seem to pay, only $2\epsilon_i$ are actually "consumed," since we must either pay for e_i or for one or both of u_i and v_i, but not for all three. Thus the approximation ratio is 2.

We remark that our analysis still holds if we set $\epsilon = \min\{w(u), w(v), \frac{1}{2}w(e)\}$ in Line 2, and decrease $w(e)$ by 2ϵ in Line 5. The ratio of 2 remains intact because OPT still drops by at least ϵ and we still pay in actuality at most 2ϵ. In fact, we can set $\epsilon = \min\{w(u), w(v), \frac{1}{\alpha}w(e)\}$, and decrease $w(e)$ by $\alpha \cdot \epsilon$, for any $1 \leq \alpha \leq 2$.

3.2 Graph Editing

A *graph editing* problem asks to transform, by means of *editing operations*, a given graph into a new graph possessing some desired property. The editing operations we consider are:

Vertex deletion. Delete a vertex (and all incident edges).

Edge deletion. Delete an edge.

Non-edge addition. Connect two non-adjacent vertices by an edge. We refer to a pair of non-adjacent vertices as a *non-edge*, and to the act of connecting them by an edge as *adding* the corresponding non-edge to the graph.

The input consists of a graph with weights associated with its vertices, edges, and non-edges, collectively known as the *graph elements*. The cost of a transformation is defined as the total cost of the graph elements involved, that is, the total cost of edges removed (including edges removed as part of vertex deletion operations), vertices removed, and non-edges added. Different graph editing problems may impose restrictions on the graph-element weights or on the types of permitted editing operations. For example, the *vertex cover* problem is a graph editing problem in which all edge weights are zero, only vertex deletion operations are allowed, and the desired property of the target graph is *independent set*, i.e., it must contain no edges.

As another example, consider the *prize collecting vertex cover* problem. One might be tempted to think that it is a graph editing problem (for the *independent set* property) in which the permitted editing operations are vertex deletion and edge deletion. This is not the case, however, because in the graph editing framework, the cost of deleting a vertex is its weight plus the total weight of its incident edges, whereas in the context of *prize collecting vertex cover*, the cost of a vertex is only its own weight. Nevertheless, the *prize collecting vertex cover* problem can be viewed as a graph editing problem by means of the following reduction. Given a vertex-and-edge weighted graph G, construct the *complement graph* G', that is, the graph obtained from G by turning every edge into a non-edge and every non-edge into an edge. Define weights on the graph element of G' as follows. The weight of a vertex in G' is the same as its weight in G; the weight of a non-edge in G' is the same as the weight of the corresponding edge in G; the weights of all edges in G' are zero. It can be easily seen that the problem of finding a prize collecting vertex cover in G is equivalent to the graph editing problem on G' in which the desired property is *clique* (i.e., the target graph must be complete).

An important graph editing problem is that of (t, k)*-graph editing for the complete multi-partite property*. Here we are given a t-partite graph $G = (V, E)$ together with a partitioning of G into $t \geq 2$ (non-empty) sides $V = \bigcup_{i=1}^{t} V_i$, a weight function w on G's elements, and a number k. The objective is to find a minimum cost transformation yielding a complete t'-partite graph, for some $t' \leq k$, whose sides are subsets of the sides of G. We stress that t' is not part of the input, but rather is defined by the target graph. In the remainder of this section we develop a 2-approximation algorithm for the special case $k = t$. The algorithm can be extended to general k, but we omit the details of this.

To analyze the problem, let us first introduce some terminology. We define the *extended weight* of a vertex u, denoted $\overline{w}(u)$, as the sum of $w(u)$ and the total weight of edges incident on u. We say that a non-edge $e = (u, v)$ is *costly* if $\min\{\overline{w}(u), \overline{w}(v), w(e)\} > 0$. Finally, we use the term *non-edge* as an abbreviation of *non-edge between two vertices belonging to different sides of the partition.*

Consider a non-edge (u, v). If we delete u or v from the graph, we will not be able to add (u, v). On the other hand, if we leave them both in the graph, we will be forced to add (u, v) (since the target graph must be complete t'-partite). Similarly, if (u, v) is an edge, it will be deleted if and only if at least one of its endpoints is deleted. Thus, every feasible solution is completely defined by the set of vertices which it removes from the graph, and, conversely, every set of vertices defines a feasible solution in a similar manner. We can therefore identify feasible solutions with sets of vertices.

OBSERVATION 1. *The optimal cost is* 0 *if and only if the graph contains no costly non-edges, and if that is the case, the set* $\{v \mid \overline{w}(v) = 0\}$ *is an optimal solution.*

The following algorithm uses a similar idea to the one behind Algorithm **PCVC**.

Algorithm (t,t)-Graph Editing

1. While there exists a costly non-edge $e=(u,v)$:
2. Let $\epsilon = \min\{\overline{w}(u), \overline{w}(v), w(e)\}$.
3. $\overline{w}(u) \leftarrow \overline{w}(u) - \epsilon$.
4. $\overline{w}(v) \leftarrow \overline{w}(v) - \epsilon$.
5. $w(e) \leftarrow w(e) - \epsilon$.
6. Return the feasible solution $\{v \mid \overline{w}(v) = 0\}$.

There is a subtle point regarding the implementation of Lines 3 and 4. Recall that the extend weight $\overline{w}(x)$ of vertex x is defined as the sum of $w(x)$ and the total weight of edges incident on x. When we say "$\overline{w}(x) \leftarrow \overline{w}(x) - \epsilon$" we actually mean "decrease $w(x)$ and the weights of the edges incident on x by some non-negative (but not necessarily equal) amounts, such that $\overline{w}(x)$ drops by ϵ as a result." We do not care exactly how much is subtracted from the weight of each of the graph elements involved, only that the total decrease amounts to ϵ and the weights of the graph elements remain non-negative. Note that the effect of this is not only to decrease $\overline{w}(x)$ by ϵ, but possibly to also decrease the extended weight of some of x's neighbors (as a result of decreasing the weights of the edges connecting them to x). The algorithm must keep track of these changes. Also note that u and v (in Lines 3 and 4) are non-adjacent, so decreasing the extended weight of one of them has no effect on the extended weight of the other.

The analysis is similar to the analysis of Algorithm **PCVC**. In the ith iteration we pay $3\epsilon_i$ and manage to lower the optimal cost by at least ϵ_i (because every feasible solution must either delete at least one of the two vertices, or else add the non-edge). Of this payment, at least ϵ_i can be recovered, since the solution returned will either delete one or both of the vertices, or else add the non-edge, but will not do both. Thus the solution returned is 2-approximate.

3.3 Background

Prize collecting vertex cover. The *prize collecting vertex cover* problem (also known as *generalized vertex cover*) was introduced and studied by Hochbaum [1997b], who presented an $O(nm \log \frac{n^2}{m})$-time 2-approximation algorithm. Algorithm **PCVC** shown here is due to Bar-Yehuda and Rawitz [2001]. The lower bounds of Håstad [1997] and Dinur and Safra [2002] for *vertex cover* apply to the prize collecting version because the plain *vertex cover* problem is a special case of the prize collecting version (where edge costs are infinite).

Graph editing. The *graph editing* problem discussed in this section has its roots in a number of node and edge deletion problems considered by Hochbaum [1998]. She approximates these problems by reducing them to max-flow problems. Bar-Yehuda and Rawitz [2002] generalize the problems and use the local ratio technique to approximate them. They consider three variants of the following optimization problem: given a graph and a non-negative weight function on the vertices and edges, find a minimum weight set of vertices and edges whose removal from the graph leaves a complete k-partite graph. The *(t,k)-graph editing* problem [Freund and Rawitz 2002] further generalizes one of these variants, namely, the

(t,k)-*deletion* problem.

4. THE LOCAL RATIO THEOREM

As we have seen, the local ratio technique is based on the idea of paying in each iteration at most $r \cdot \epsilon$, for some r, in order to achieve a reduction of at least ϵ in OPT. If the same r is used in all iterations, the solution returned is r-approximate. This idea has a very intuitive appeal, and works well in the examples we have encountered in the previous section. However, a closer look at these examples reveals that the idea worked mainly because we were able to make a very localized payment and argue that OPT must drop by a proportional amount because every solution must involve some of the items we paid for. For example, in *vertex cover* the payment was localized to a single edge; we paid ϵ for each of its two endpoints and argued that OPT must drop by at least ϵ because every vertex cover must include at least one of the endpoints. This localization of the payments is at the root of the simplicity and elegance of the analysis, but it is also a source of weakness: how can we deal with problems in which no single set of items is necessarily involved in every optimal solution? Consider for example the *feedback vertex set* problem, in which we are given a vertex-weighted graph and are asked to remove a minimum weight set of vertices such that the remaining graph will contain no cycles. As we shall see, there is a 2-approximation local ratio algorithm for this problem, yet surely 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 we present below, allows us to go beyond localized payments by shifting the focus of attention from the weight function itself to the *changes* in the weight function, and treating these changes as weight functions in their own right. A consequence of this is that algorithms based on the theorem are more readily formulated as recursive algorithms, rather than iterative ones. This, in turn, leads to the idea of concentrating on minimal solutions in a very natural manner. Minimal solutions are an essential ingredient in many applications of the technique. We explain and demonstrate all this in this section and subsequent ones.

The Local Ratio Theorem is deceptively simple[2]. 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* $\mathcal{C}$, find a *solution vector* $x \in \mathbb{R}^n$ satisfying the constraints in $\mathcal{C}$ that minimizes the inner product $w \cdot x$.

(For convenience, we deal in this section with minimization problems only. In Section 6 we discuss maximization problems.)

The most common type of optimization problem that can be formulated as above consists of instances in which the input contains a set I of n weighted elements and a specification 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 comprises. In terms of the above formulation, the weight function on the elements is expressed by the weight vector (where the ith

[2]Indeed, some authors refer to it as the Local Ratio *Lemma*. We prefer *Theorem* in light of its fundamental role.

component of the vector is the weight of the ith element), and subsets are expressed as 0–1 vectors (where the ith component of the vector is 1 if the ith element is in the subset, and is 0 otherwise). The inner product $w \cdot x$ is then the weight of the subset corresponding to x.

THEOREM 2 LOCAL RATIO—MINIMIZATION PROBLEMS. *Let $\mathcal{C}$ 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 $\mathcal{C}$) 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, $w_1 \cdot x_1^* \leq w_1 \cdot x^*$ and $w_2 \cdot x_2^* \leq w_2 \cdot x^*$. Thus, $w \cdot x = w_1 \cdot x + w_2 \cdot x \leq r(w_1 \cdot x_1^*) + r(w_2 \cdot x_2^*) \leq r(w_1 \cdot x^*) + r(w_2 \cdot x^*) = r(w \cdot x^*)$. □

(Note that the theorem holds even when negative weights are allowed.)

The Local Ratio Theorem leads naturally to the formulation of recursive algorithms with the following general structure.

(1) If a zero-cost solution can be found, return one.
(2) Otherwise, find a suitable decomposition of w into two weight functions w_1 and $w_2 = w - w_1$, and solve the problem recursively using w_2 as the weight function in the recursive call.

We demonstrate this on *vertex cover*.

Algorithm RecursiveVC(G,w)

1. If $\min\{w(u), w(v)\} = 0$ for all edges (u, v):
2. Return the set $\{v \mid w(v) = 0\}$.
3. Else:
4. Let (u, v) be an edge such that $\epsilon \triangleq \min\{w(u), w(v)\} > 0$.
5. Define $w_1(x) = \begin{cases} \epsilon & x = u \text{ or } x = v, \\ 0 & \text{otherwise,} \end{cases}$ and define $w_2 = w - w_1$.
6. Return **RecursiveVC(G,w_2)**.

Algorithm **RecursiveVC** is clearly just a recursive formulation of Algorithm **VC**. However, the recursive formulation is amenable to direct analysis based on the Local Ratio Theorem. We prove that the solution returned by the algorithm is 2-approximate by induction on the recursion. In the base case, the algorithm returns a vertex cover of zero cost, which is optimal. For the inductive step, consider the solution returned by the recursive call. By the inductive hypothesis it is 2-approximate with respect to w_2. We claim that it is also 2-approximate with respect to w_1. In fact, we claim that *every* feasible solution is 2-approximate with respect to w_1. To see this, observe that the cost (with respect to w_1) of every vertex cover is at most 2ϵ, since only u and v have non-zero cost and they each cost ϵ. On

the other hand, the minimum cost of a cover is at least ϵ, since every vertex cover must include at least one of u and v in order to cover the edge (u, v). Thus, by the Local Ratio Theorem the solution returned is 2-approximate with respect to w.

Note that different algorithms (for different problems) conforming to the general structure outlined above differ from one another only in the decomposition of w, and this decomposition is determined completely by the choice of w_1. (Actually, they might also differ in the way they search for a zero-cost solution, but most often the only candidate that needs to be examined is the set of all zero-cost elements.) Accordingly, such algorithms also share most of their analyses. Specifically, the proof that a given algorithm is an r-approximation algorithm is by induction on the recursion. In the base case the solution is optimal (and therefore r-approximate) because it has zero cost, and in the inductive step, the solution returned by the recursive call is r-approximate with respect to w_2 by the inductive hypothesis. Thus, different algorithms differ from one another only in the choice of w_1 and in the proof that every feasible solution is r-approximate with respect to w_1. (Obviously, they may also differ in the value of the approximation ratio r.) The following definition formalizes this notion.

DEFINITION 1. *Given a set of constraints $\mathcal{C}$ 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.*

The analysis of algorithms in our framework boils down to proving that w_1 is r-effective. Proving this amounts to proving that b is a lower bound on the optimum value and $r \cdot b$ is an upper bound on the cost of every feasible solution, and thus every feasible solution is r-approximate (all with respect to w_1). For this reason we will henceforth focus solely on w_1, and neglect to mention the remaining details explicitly.

In the remainder of this section we demonstrate the above framework on two problems: *hitting set* and *feedback vertex set in tournaments*. We remark that these algorithms can be formulated just as easily in terms of localized payments; the true power of the Local Ratio Theorem will become apparent in the next section.

4.1 Hitting Set

The input in a *hitting set* instance is a collection of non-empty sets $\mathcal{C} = \{S_1 \ldots S_m\}$ and a weight function w on the sets' elements $U = \bigcup_{i=1}^{m} S_i$. An element $x \in U$ is said to *hit* a given set $S_i \in \mathcal{C}$ if $x \in S_i$. A subset $H \subseteq U$ is said to *hit* a given set $S_i \in \mathcal{C}$ if $H \cap S_i \neq \emptyset$. The objective is to find a minimum-cost subset of U that hits all sets $S_i \in \mathcal{C}$.

Let $s_{\max} = \max_{1 \leq i \leq m} |S_i|$. The following decomposition of w achieves an approximation factor of $s_{\max}$. Let i be any index such that S_i contains no zero-weight elements, and let $\epsilon = \min\{w(x) \mid x \in S_i\}$. Define:

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

We claim that w_1 is $s_{\max}$-effective. Clearly, the cost of every feasible solution is bounded by $\epsilon \cdot |S_i| \leq \epsilon \cdot s_{\max}$. On the other hand, every feasible solution must hit

S_i, i.e., must contain at least one element of S_i. Thus every feasible solution costs at least ϵ.

Remarks. The *hitting set* problem generalizes many covering problems. For example, *vertex cover* can be seen as a *hitting set* problem in which the sets are the edges and the elements are the vertices. Indeed, Algorithm **RecursiveVC** is just a special case of the algorithm for *hitting set* we have just described. (Note that $s_{\max} = 2$ for the formulation of *vertex cover* in terms of *hitting set*.)

We also wish to comment on the relation between *hitting set* and *set cover*. In the *set cover* problem we are given a collection of sets $\mathcal{C} = \{S_1 \ldots S_m\}$, as in *hitting set*, but this time the weight function is on the sets, not on the elements. The objective is to find a minimum-cost collection of sets that "covers" all elements. In other words, the union of the sets in the solution must be equal to $\bigcup_{i=1}^{m} S_i$. It is well known that *set cover* and *hitting set* are equivalent problems in the sense that each is obtained from the other by switching the roles of sets and elements. In *set cover* terms, the approximation ratio obtained by the above algorithm is the maximum *degree* of an element, where the degree of an element is the number of sets containing it as a member.

4.2 Feedback Vertex Set in Tournaments

A *tournament* is an orientation of a complete (undirected) graph, i.e., it is a directed graph with the property that for every unordered pair of distinct nodes $\{u, v\}$ it either contains the arc (u, v) or the arc (v, u), but not both. The *feedback vertex set in tournaments* problem is the following. Given a tournament and a weight function w on its nodes, find a minimum-weight set of nodes whose removal leaves a graph containing no directed cycles.

An immediate observation, expressed in the following lemma, is that we may restrict our attention to cycles of length 3. (Note that the smallest possible cycle length in a tournament is 3.) We refer to the set of (three) nodes on a directed cycle of length 3 as a *triangle*. We say that a triangle is *positive* if all of its nodes have strictly positive weights.

LEMMA 3. *A tournament contains a directed cycle if and only if it contains a triangle.*

PROOF. The existence of a triangle implies the existence of a directed cycle by definition. The existence of a directed cycle implies the existence of a triangle by the following argument. Suppose the minimum length of a directed cycle in the tournament is $k \geq 3$. If $k = 3$, we are done. Otherwise, the tournament contains some directed cycle $v_1, v_2, v_3, \ldots, v_k, v_1$. If the tournament contains the arc (v_3, v_1), then it contains the directed cycle v_1, v_2, v_3, v_1 of length 3. Otherwise, it contains the arc (v_1, v_3) and therefore it contains the directed cycle $v_3, \ldots, v_k, v_1, v_3$ of length $k - 1$. In either case we reach a contradiction with the minimality of k. □

Lemma 3 implies that the set of all zero-cost nodes is an optimal solution (of zero cost) if and only if the tournament contains no positive triangles. Thus we obtain a 3-approximation algorithm by means of the following fully 3-effective weight function. Let $\{v_1, v_2, v_3\}$ be a positive triangle, and let $\epsilon = \min\{w(v_1), w(v_2), w(v_3)\}$.

Define:

$$w_1(v) = \begin{cases} \epsilon & v \in \{v_1, v_2, v_3\}, \\ 0 & \text{otherwise.} \end{cases}$$

The maximum cost, with respect to w_1, of a feasible solution is clearly at most 3ϵ, while the minimum cost is at least ϵ, since every feasible solution must contain at least on of v_1, v_2, v_3.

Cai et al. [2001] present an algorithm that finds an optimal solution in any tournament T that does not contain *forbidden* sub-tournaments, i.e. sub-tournaments of the forms shown in Figure 1 (where the two arcs not shown in T_1 may take any direction). They use this algorithm to obtain a 2.5-approximation algorithm for the general case, employing the following fully 2.5-effective weight function. Let F be a subset of five positive-weight vertices inducing a forbidden sub-tournament and let $\epsilon = \min\{w(v) \mid v \in F\}$. Define:

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

This function is fully 2.5-effective since the cost of every feasible solution is clearly at most 5ϵ, whereas the minimum cost is at least 2ϵ (as every four vertices in F contain a triangle). At the basis of the recursion—the case where every forbidden sub-tournament in T contains at least one zero-weight vertex—an optimal solution is found by removing the set Z of all zero-weight vertices (thereby eliminating all forbidden sub-tournaments), solving the problem optimally on the remaining graph, and adding Z to the solution found.

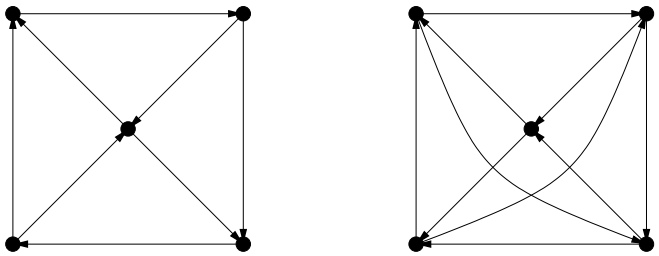

Fig. 1. Forbidden sub-tournaments.

4.3 Background

Hitting set. For the unweighted *hitting set* (or equivalently *set cover*) problem, Johnson [1974] and Lovász [1975] show that the greedy algorithm is an $H(s_{\max})$-approximation algorithm. Chvátal [1979] generalizes this result to the weighted case. His analysis has a *dual fitting* interpretation (see Jain et al. [2002]). Hochbaum [1982] gives two $s_{\max}$-approximation algorithms, both of which are based on solving a linear program. Bar-Yehuda and Even [1981] suggest a linear time primal-dual $s_{\max}$-approximation algorithm. In subsequent work [Bar-Yehuda and Even 1985],

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 [1996] proves a lower bound of $(1 - o(1)) \ln |U|$ (unless NP$\subseteq$DTIME$(n^{O(\log \log n)})$).

Feedback vertex set in tournaments. The 2.5-approximation algorithm for *feedback vertex set in tournaments* discussed in this section is due to Cai et al. [2001].

5. MINIMAL SOLUTIONS

In the previous section we introduced the idea of weight decomposition as a means of going beyond localized payments, but the examples we gave of its use did not in fact depart from that paradigm. All we did was translate the "payment" arguments into weight decomposition language. In each case we identified a small subset of elements such that every feasible solution had to contain at least one of them, and by associating a weight of ϵ with each of these elements and a weight of zero with all others, we were able to obtain an approximation ratio bounded by the size of the subset. There are many problems, though, where identifying such a small subset is impossible, simply because none such subset necessarily exists.

A case in point is the *partial hitting set* problem. This problem is similar to *hitting set* except that not all sets need to be hit. More specifically, the input consists of a collection of sets, a weight function on the sets' elements, and a number k. The objective is to find a minimum-cost subset of elements that hits at least k of the sets. The crucial difference between *hitting set* and *partial hitting set* is that in the latter, there is no single set that must be hit by all feasible solutions. Recall that the algorithm for *hitting set* picked some set S_i and associated a weight of ϵ with each of its elements. The analysis was based on the fact that $\epsilon \cdot |S_i|$ is an upper bound on the cost of every feasible solution, while ϵ is a lower bound since S_i must be hit. This approach fails for *partial hitting set* because ϵ is no longer a lower bound—an optimal solution need not necessarily hit S_i. Thus if we use the above weight function, we will end up with a solution whose weight with respect to w_1 is positive, while the optimum value may be equal to 0.

Our method of handling such situations is a logical extension of the same upper-bound/lower-bound idea. Roughly speaking, we do not know of any single subset that must contribute to all solutions, but the set of *all* elements must surely do so (otherwise, the empty set is feasible and optimal). Thus, to prevent OPT from being equal to 0, we can assign a positive weight to every element. This takes care of the lower bound but opens up the question of how to obtain a non-trivial upper bound—clearly, it is almost never the case that the cost of every feasible solution is within some reasonable factor of the cost of a single element. It turns out that the simple idea of considering only *minimal* solutions works well in many situations. By *minimal solution* we mean a feasible solution that is minimal with respect to set inclusion, i.e., a feasible solution whose proper subsets are all infeasible. Minimal solutions are meaningful mainly in the context of *covering* problems. (In our context, *covering* problems are problems for which feasible solutions are monotone inclusion-wise, i.e., if a set X is a feasible solution, then so is every superset of X. For example, adding a vertex to a vertex cover yields a vertex cover, so *vertex 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, which is an adaptation of Definition 1 to minimal solutions.

DEFINITION 2. *Given a set of constraints $\mathcal{C}$ 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.*

If we can show that our algorithm uses an r-effective w_1 and returns minimal solutions, we will have essentially proved that it is an r-approximation algorithm. Designing an algorithm to return minimal solutions is quite easy. Most of the creative effort is therefore expended in finding an r-effective weight function (for a small r).

In this section we present local ratio algorithms for the problems of *partial hitting set*, *network design*, and *feedback vertex set*, algorithms that depend on obtaining minimal solutions. We describe and analyze the first of these algorithms in full detail. We then outline the general framework of local ratio algorithms obtaining minimal solutions, and discuss the subsequent algorithms informally with reference to this framework.

5.1 Partial Hitting Set

In the *partial hitting set* problem, the input consists of a collection of non-empty sets $\mathcal{C} = \{S_1 \ldots S_m\}$, a weight function w on the sets' elements $U = \bigcup_{i=1}^{m} S_i$, and a number $k \leq m$. The objective is to find a minimum-cost subset of U that hits at least k sets in $\mathcal{C}$.

Given an instance of *partial hitting set*, let $\mathcal{S}(x) = \{S_i \in \mathcal{C} \mid x \in S_i\}$ for $x \in U$. Define the *degree* of an element $x \in U$ as $d(x) = |\mathcal{S}(x)|$. We present an $s_{\max}$-approximation algorithm. (Recall that $s_{\max} = \max_{1 \leq i \leq m} |S_i|$ by definition.)

Algorithm PHS($\mathcal{C}, w, k$)

1. If $k \leq 0$, return $\emptyset$.
2. Else, if there exists an element $x \in U$ such that $w(x) = 0$ do:
3. $\quad H' \leftarrow$ **PHS**$(\mathcal{C} \setminus \mathcal{S}(x), w, k - d(x))$.
4. $\quad$ If H' hits at least k sets in $\mathcal{C}$:
5. $\quad\quad$ Return the solution $H = H'$.
6. $\quad$ Else:
7. $\quad\quad$ Return the solution $H = H' \cup \{x\}$.
8. Else:
9. $\quad$ Let ϵ be maximal such that $\epsilon \cdot \min\{d(x), k\} \leq w(x)$ for all $x \in U$.
10. $\quad$ Define the weight functions $w_1(x) = \epsilon \cdot \min\{d(x), k\}$ and $w_2 = w - w_1$.
11. $\quad$ Return **PHS**$(\mathcal{C}, w_2, k)$.

Note the slight abuse of notation in Line 3. The weight function in the recursive call is not w itself, but rather the restriction of w to $\bigcup(\mathcal{C} \setminus \mathcal{S}(x))$. 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 $s_{\max}$-approximate. Intuitively, Lines 2–7 ensure that the solution returned is minimal, while the weight decomposition in Lines 8–11 ensures that every minimal solution is $s_{\max}$-approximate. This is done by associating with each element x a weight that is proportional to its "covering power," which is the number of sets it hits, but not more than k, since hitting more than k sets is no better than hitting k sets.

PROPOSITION 4. *Algorithm **PHS** 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 9–11 are executed, then the solution returned is feasible and minimal by the inductive hypothesis. Otherwise, Lines 3–7 are executed. By the inductive hypothesis H' is minimal and feasible with respect to $(\mathcal{C} \setminus \mathcal{S}(x), k - d(x))$. If $H' = \emptyset$ then $d(x) \geq k$ and $H = H' \cup \{x\}$ is clearly feasible and minimal. Otherwise, H' hits at least $k - d(x)$ sets in $\mathcal{C}$ that do not contain x, and by minimality, for all $y \in H'$, the set $H' \setminus \{y\}$ hits less than $k - d(x)$ sets that do not contain x. (Note that $H' \neq \emptyset$ implies $k > d(x)$.) Consider the solution H. Either $H = H'$, which is the case if H' hits at least k or more sets in $\mathcal{C}$, or else $H = H' \cup \{x\}$, in which case H hits at least $k - d(x)$ sets that do not contain x and an additional $d(x)$ sets that do contain x. In either case H hits at least k sets and is therefore feasible. It is also minimal (in either case), since for all $y \in H$, if $y \neq x$, then $H \setminus \{y\}$ hits less than $k - d(x)$ sets that do not contain x and exactly $d(x)$ sets that do contain x, for a total of less than k sets, and if $y = x$, then $x \in H$, which implies $H = H' \cup \{x\}$, which is only possible if $H \setminus \{y\} = H'$ hits less than k sets. □

PROPOSITION 5. *The weight function w_1 used in Algorithm **PHS** is s_{max}-effective.*

PROOF. In terms of w_1, every feasible solution costs at least $\epsilon \cdot k$, since it either contains an element whose cost is $\epsilon \cdot k$, or else consists solely of elements with degree less than k, in which case the cost of the solution equals ϵ times the total degree of elements in the solution, which must be at least k in order to hit k sets. To prove that every minimal solution costs at most $\epsilon \cdot s_{\max} \cdot k$, consider a non-empty minimal feasible solution H. The minimality of H implies that every $x \in H$ hits at least one set that no other element in H hits. Thus every subset of k elements from H hits at least k different sets, and therefore $|H| \leq k$ (by minimality). If H contains an element y such that $d(y) \geq k$, then by minimality, $H = \{y\}$, and therefore $w_1(H) = w_1(y) = \epsilon \cdot k$. Otherwise, $w_1(x) = \epsilon \cdot d(x) \leq \epsilon \cdot s_{\max}$ for all $x \in H$, and therefore $w_1(H) \leq \epsilon \cdot s_{\max} \cdot |H| \leq \epsilon \cdot s_{\max} \cdot k$. □

THEOREM 6. *Algorithm **PHS** returns s_{max}-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 3–7 are executed, then H' is $s_{\max}$-approximate with respect to $(\mathcal{C} \setminus \mathcal{S}(x), w, k - d(x))$ by the inductive hypothesis. Since $w(x) = 0$, the cost of H equals that of H'

and the optimum value for $(\mathcal{C}, w, k)$ cannot be smaller than the optimum value for $(\mathcal{C} \setminus \mathcal{S}(x), w, k - d(x))$ because if H^* is an optimal solution for $(\mathcal{C}, w, k)$, then $H^* \setminus \{x\}$ is a feasible solution of the same cost for $(\mathcal{C} \setminus \mathcal{S}(x), w, k - d(x))$. If, on the other hand, Lines 9–11 are executed, then by the inductive hypothesis the solution returned is $s_{\max}$-approximate with respect to w_2, and by Proposition 5 it is also $s_{\max}$-approximate with respect to w_1. Thus by the Local Ratio Theorem it is $s_{\max}$-approximate with respect to w as well. □

5.2 A Framework

The general structure of a local ratio algorithm obtaining minimal solutions consists of a three-way *if* condition that directs execution to one of the following three options: *optimal solution*, *problem size reduction*, and *weight decomposition*. The top-level description of the algorithm is:

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

The three types of steps are:

Optimal solution. Find a zero-cost minimal solution and return it. (Typically, the solution will simply be the empty set.) This is the recursion basis.

Problem size reduction. This option consists of three steps.

(1) Reduce the problem size by deleting a zero-cost element. This changes the problem instance, and may entail further changes (to achieve consistency with the idea that the element has potentially been taken to the solution). For example, in the *partial hitting set* problem we select a zero-cost element and delete all sets containing it, and reduce the hitting requirement parameter k by the number of these sets. Sometimes the problem instance must be adjusted more radically to reflect the deletion of the element. For example, a graph edge might be eliminated by *contracting* it. As a result, two vertices get "fused" together and the edges incident on them undergo changes and possibly fusions. In other words, the modification consists of eliminating some existing elements and introducing new ones. This, in turn, requires that the weight function be extended to cover the new elements. It is important to realize, though, that the adjustment of the weight function amounts to a re-interpretation of the existing weights in terms of the new instance and not to an actual change of weights.
(2) Solve the problem recursively on the new instance.
(3) If the solution returned (when re-interpreted in terms of the original instance) is feasible (for the original instance), return it. Otherwise, extend it by adding the deleted zero-cost element, and return this solution.

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

The above formulation of the generic algorithm 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. For example, Line 2 might hypothetically represent something like: *If there is a vertex of degree less than 2, take appropriate action. Otherwise, if there is a zero-weight vertex, take appropriate action.* In both cases, *appropriate action* would fall in the category of *problem size reduction*. We shall see examples of this in the sequel.

The analysis of an algorithm in the above framework follows the pattern of our analysis for *partial hitting set*. It consists of proving the following three claims.

(1) The algorithm returns a minimal feasible solution.

(2) The weight function w_1 is r-effective.

(3) The algorithm returns an r-approximate solution.

The proof of the first claim is 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 the solution is feasible and minimal by the inductive hypothesis. If the algorithm performs *problem size reduction*, the claim follows intuitively 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 original instance. Although the argument is straightforward, the details of a rigorous proof tend to be slightly messy, as they depend heavily on the precise manner in which the instance is modified.

The proof of the second claim follows from the combinatorial structure of the problem. There is no fixed pattern here. Indeed, the key to the design of a local ratio algorithm is understanding the combinatorial properties of the problem under study and finding the right r and an r-effective weight function.

The proof of the third claim is also by induction on the recursion, based on the first two claims. In the *optimal solution* case it is true by design. 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 element. This, combined with the fact that OPT can only decrease as a result of the instance modification, yields the claim. (Again, the details of a rigorous proof tend to be somewhat messy.) In the *weight decomposition* case, the claim follows by the inductive hypothesis and the Local Ratio Theorem, based on the fact that the solution is feasible and minimal, and that w_1 is r-effective.

In the next two sections we describe the algorithms for the *network design* and *feedback vertex set* problems in the context of this framework.

5.3 Network Design

The *network design* problem is defined as follows. Given an edge weighted graph $G = (V, E)$ and a *demand* function $f : 2^V \rightarrow \mathbb{N}$, find a minimum weight set of edges F that for all $S \subseteq U$, contains at least $f(S)$ edges from $\delta(S)$, where $\delta(S)$ is the set of edges in the cut defined by S (i.e., edges with exactly one endpoint in S). We assume $f(V) = f(\emptyset) = 0$, for otherwise no solution exists. Furthermore, we assume

the existence of a polynomial-time oracle that provides $f(S)$ when given a subset S. We make this assumption because the number of subsets $S \subseteq V$ is exponential, and enumerating $f(S)$ for all of them in the input would be unreasonable. Finally, to simplify the exposition, we generalize the problem and allow G to be a multi-graph, i.e., G may contain multiple edges between pairs of vertices.

Many familiar problems can be formulated as network design problems with a *0-1 demand functions*, i.e., demand functions with range $\{0,1\}$. For example, in the *shortest path* problem $f(S) = 1$ if $|S \cap \{s,t\}| = 1$ and $f(S) = 0$ otherwise; in the *minimum spanning tree* problem, $f(S) = 1$ if $\emptyset \neq S \subsetneq V$ and $f(S) = 0$ otherwise; and in the *Steiner tree* problem, $f(S) = 1$ if $\emptyset \neq S \subsetneq T$ and $f(S) = 0$ otherwise. The *Steiner tree* problem is: given a graph $G = (V, E)$ and a set of *terminals* $T \subseteq V$, find a minimum weight set of edges that induces a connected subgraph containing all terminals (and possibly other vertices as well). The key property of 0-1 demand functions is that every minimal solution induces a forest, for if a feasible solution contains a cycle, removing a single edge from this cycle cannot destroy the solution's feasibility. (Of course, multiple edges are redundant too.)

In our treatment of the problem we restrict our attention to 0-1 demand functions. We further restrict our attention to two families of 0-1 demand functions for which deciding whether a given set of edges constitutes a feasible solution is easy. Specifically, we consider 0-1 *proper* functions and 0-1 *downwards monotone* functions.

0-1 proper functions.. A 0-1 demand function f is *proper* if it satisfies two conditions.

Symmetry. $f(S) = f(V \setminus S)$ for all $S \subseteq V$.

Maximality. For all pairs of disjoint sets A and B, if $f(A) = f(B) = 0$, then $f(A \cup B) = 0$.

Note that the examples given above for applications of 0-1 functions are actually applications of 0-1 proper functions.

0-1 downwards monotone functions.. A 0-1 demand function f is *downwards monotone* if $f(S) = 1$ implies $f(S') = 1$ for all nonempty $S' \subseteq S$. Notice that a downwards monotone function satisfies maximality, but not necessarily symmetry.

Downwards monotone functions can be used to model several familiar problems. For example, the *edge covering* problem is that of selecting a minimum weight set of edges spanning the entire vertex set. This problem (which is polynomial time solvable [Grötschel et al. 1988]) can be modeled by: $f(S) = 1$ if and only if $|S| = 1$, which is downwards monotone. Another example is the *lower capacitated tree partitioning* problem. Here we must find a minimum weight set of edges F such that (V, F) is a forest in which each tree contains at least k vertices, for some parameter k. This problem can be modeled by: $f(S) = 1$ if and only if $0 < |S| < k$, which again is downwards monotone.

Notice that if a 0-1 demand function f is proper or downwards monotone, we can check efficiently whether an edge set F is feasible by checking if $f(C) = 0$ for every connected component C of (V, F). If $f(C) = 1$ for some component C, then F is clearly infeasible. Otherwise, F is feasible since every $S \subseteq V$ that is a union

of components obeys $f(S) = 0$ by maximality, and every S that is not a union of components is satisfied by some edge in F.

5.3.1 *The Algorithm.* We now present a 2-approximation algorithm. We begin with some definitions and an observation, and then describe and analyze the algorithm.

Definitions.

(1) The vertices v such that $f(\{v\}) = 1$ are called *terminals*.

(2) We denote by $\tau(e)$ the number of endpoints of edge e that are terminals. (Thus, $\tau(e) \in \{0, 1, 2\}$ for all e.)

(3) In the course of its execution, the algorithm modifies the graph by performing *edge contractions*. Contracting an edge consists in "fusing" its two endpoints u and v into a single (new) vertex z. The edge (or multiple edges) connecting u and v are deleted and every other edge incident on u or v becomes incident on z instead. In addition, the demand function f is replaced with a new demand function f', defined by

$$f'(S) = \begin{cases} f((S \setminus z) \cup \{u, v\}) & z \in S, \\ f(S) & z \notin S. \end{cases}$$

Clearly, if f is 0-1 proper or downwards monotone then so is f'.

OBSERVATION 7. *Let I be an instance of the problem and let I' be the problem instance obtained from it by contracting a single edge e. Then:*

(1) The optimum value for I is not less than the optimum value for I'.

(2) If F' is a minimal feasible solution for I', then either F' is minimal feasible for I, or else it is infeasible for I but $F' \cup \{e\}$ is minimal feasible for I.

The algorithm follows.

Algorithm ND(G, w, f)

1. If G contains no terminals, return $\emptyset$.
2. Else, if there exists an edge e such that $w(e) = 0$ do:
3. Let (G', w', f') be the instance obtained by contracting e.
4. $F' \leftarrow$ **ND**(G', w', f').
5. If F' is a feasible solution with respect to G:
6. Return the solution $F = F'$.
7. Else:
8. Return the solution $F = F' \cup \{e\}$.
9. Else:
10. Let $\epsilon = \min\{w(e)/\tau(e) \mid \tau(e) \geq 1\}$.
11. Define the weight functions $w_1(e) = \epsilon \cdot \tau(e)$ and $w_2 = w - w_1$.
12. Return **ND**(G, w_2, f).

In terms of our framework, Line 1 implements the *optimal solution* case (correctness follows from the maximality property of f), Lines 2–8 implement the *problem size reduction* case (correctness follows from Observation 7), and Lines 9–12 implement the *weight decomposition* case.

We claim that the solution returned by the algorithm is 2-approximate, and to prove this we only need to show that w_1 is 2-effective.

LEMMA 8. *Let F be a minimal feasible solution. Then every tree in the forest induced by F contains at most one non-terminal leaf (and all other leaves are terminals).*

PROOF. We first consider the case that f is 0-1 proper. Suppose there exists some leaf v that is not a terminal. Let T_v be the tree containing v. Let F' be the set of edges obtained from F by removing the edge incident on v. Since F is minimal, F' is infeasible. Thus there exists $\emptyset \neq S \subsetneq V$ such that $f(S) = 1$ and $\delta(S) \cap F' = \emptyset$. By symmetry, we can assume $v \in S$ without loss of generality. The fact that $\delta(S) \cap F' = \emptyset$ implies that S is a union of trees in the forest induced by F' (we consider trees to be sets of vertices). The trees in this forest are exactly the trees in the forest induced by F, except that T_v is split into two trees $\{v\}$ and $T_v \setminus \{v\}$. Since F is feasible, every tree T in the forest induced by it satisfies $f(T) = 0$, and by maximality, so does the union of any number of them. Thus, $f(S \cap T_v) = 1$, for otherwise maximality would imply $f(S) = 0$. We know that S is a union of trees in the forest induced by F', and $\emptyset \neq S \cap T_v \neq T_v$ since $f(S \cap T_v) = 1$. Thus, either $S \cap T_v = T_v \setminus \{v\}$ or $S \cap T_v = \{v\}$. The latter is impossible since v is not a terminal, so $S \cap T_v = T_v \setminus \{v\}$. But this contradicts our assumption that $v \in S$. Thus in the case of 0-1 proper functions, *all* leaves are terminals.

Now consider the case that f is 0-1 downwards monotone. Again, suppose some leaf v is a non-terminal. The proof proceeds as before, except that this time we cannot assume $v \in S$ (since symmetry does not necessarily hold). Thus in the final step of the proof, we conclude that $S \cap T_v = T_v \setminus \{v\}$, and there is no contradiction. Hence, $f(T_v \setminus \{v\}) = 1$, which implies (by downwards monotonicity) that all vertices in T_v except for v are terminals. We remark that the proof can be extended quite easily to show that in fact if a tree contains *any* non-terminal (not necessarily a leaf) then all other vertices in the tree are terminals. □

The proof that w_1 is 2-effective follows from Lemma 8. Consider a minimal feasible solution F. We claim that $\epsilon|V_t| \leq w_1(F) \leq 2\epsilon|V_t|$, where V_t denotes the set of terminals. To show this, we prove the equivalent claim $|V_t| \leq \sum_{e \in F} \tau(e) \leq 2|V_t|$. For the first inequality, observe that the feasibility of F implies that every terminal must be adjacent to at least one of the edges in F, and thus $\sum_{e \in F} \tau(e) \geq |V_t|$. Turning to the second inequality, let V_F be the set of vertices in the forest induced by F, let $\deg_F(v)$ denote the degree of vertex v in the forest, and let k be the

number of trees in the forest. Then,

$$\begin{aligned}\sum_{e\in F}\tau(e) &= \sum_{v\in V_t}\deg_F(v)\\ &= \sum_{v\in V_F}\deg_F(v) - \sum_{v\in V_F\setminus V_t}\deg_F(v)\\ &= 2(|V_F|-k) - \sum_{v\in V_F\setminus V_t}\deg_F(v).\end{aligned}$$

By Lemma 8, all but one of the non-terminals in a given tree are not leaves, so each has degree at least 2 in the forest. Thus,

$$\sum_{v\in V_F\setminus V_t}\deg_F(v) \geq 2|V_F\setminus V_t| - k = 2|V_F| - 2|V_t| - k,$$

and thus $\sum_{e\in F}\tau(e) \leq 2|V_t| - k \leq 2|V_t|$.

In a slightly more refined analysis we can take into account the $-k$ term in the last inequality and see that w_1 is actually $(2-1/|V_t|)$-effective (since $k\geq 1$). Furthermore, if f is 0-1 proper, then *all* leaves are terminals, and we can replace the $-k$ by $-2k$, so in this case w_1 is even $(2-2/|V_t|)$-effective. (Note that V_t decreases as the recursion proceeds to ever deeper levels, so in the above expressions we must understand $|V_t|$ to represent the number of terminals in the *initial* graph.)

In particular, the ratio is 1 for *shortest path*, since this problem is modeled by a 0-1 proper function and $|V_t| = 2$. In fact, Algorithm **ND** can be viewed as a recursive implementation of a variant of Dijkstra's algorithm using bi-directional search (see Nicholson [1966]). Also, in the case of *minimum spanning tree*, w_1 is really 1-effective, since every minimal solution costs exactly $2\epsilon(n-1)$. For this problem Algorithm **ND** can be seen as a recursive implementation of Kruskal's minimum spanning tree algorithm [Kruskal 1956].

5.4 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 minimal-solutions framework.

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. Since $w_1(F) = \epsilon \cdot \sum_{v \in F} \deg(v)$ for any FVS F, it is sufficient to demonstrate the existence of a number b such that for all minimal solutions F, $b \leq \sum_{v \in F} \deg(v) \leq 2b$. Note that the weight decomposition is only applied to graphs in which all vertices have degree at least 2. We shall henceforth assume that our graph $G = (V, E)$ is such a graph.

Consider a feasible solution F. There is a clear connection between $\sum_{v \in F} \deg(v)$ and the number of edges deleted by removing F from the graph, so it makes sense to reason about these edges. Consider the graph obtained by removing F from G. This graph is a forest on $|V| - |F|$ nodes, and it therefore contains precisely $|V| - |F| - k$ edges, where k is the number of its connected components (trees). The number of edges deleted by the removal of F is thus $|E| - |V| + |F| + k > |E| - |V| + |F|$, and since each of these edges contributes to the degree of some vertex in F, we get the following lower bound: $|E| - |V| + |F| < \sum_{v \in F} \deg(v)$.

Let us fix some feasible solution F^* that minimizes $|E| - |V| + |F^*|$, i.e., let F^* be a minimum cardinality FVS. We claim that all minimal solutions F satisfy $|E| - |V| + |F^*| < \sum_{v \in F} \deg(v) \leq 2(|E| - |V| + |F^*|)$, which proves that w_1 is 2-effective. Let F be a minimal solution. The first inequality follows immediately from the discussion above and our choice of F^*. The second inequality, however, is not immediate, and to prove it we must first explore the structure of G.

Consider a vertex $v \in F$. The minimality of F implies that there exists at least one cycle in G such that v is the only vertex in F contained in this cycle. Denote this cycle by C_v (if more than one such cycle exist, choose one arbitrarily) and let P_v be the path obtained by removing v from C_v. Let $\mathcal{P} = \{P_v \mid v \in F\}$. Now consider the connected components of the graph G' obtained by removing F from G. Some of these connected components contain vertices from the paths $\mathcal{P}$. We refer to them as *vpath-components*. Let V_1 be the set of vertices in the vpath-components and let V_2 be the remaining vertices in G'. The original graph G is

thus partitioned into three parts, as illustrated in Figure 2. (Although F and V_2 are each drawn in the figure as a single oval, the subgraphs they induce are not necessarily connected.) Observe that there are no edges between V_1 and V_2 since such edges would appear in G', but V_1 is a union of connected components in G'. Let $n_1 = |V_1|$ and $n_2 = |V_2|$. Let m_1 be the number of edges in G that are incident on vertices in V_1 (i.e., edges between vertices in V_1 and edges connecting vertices in V_1 with vertices in F). Similarly, let m_2 be the number of edges in G that are incident on vertices in V_2. Let m_1' be the number of edges connecting vertices in F with vertices in V_1 and let m_2' be the number of edges connecting vertices in F with vertices in V_2. Let m_F be the number of edges in the subgraph induced by F. Finally, let κ be the number of vpath-components.

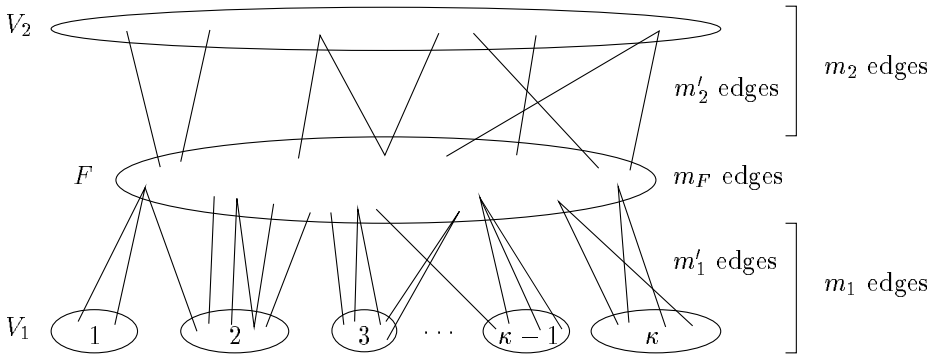

Fig. 2. Partition of G.

With the above notation in place, we are ready to bound $\sum_{v \in F} \deg(v)$. First note that $|V| = n_1 + n_2 + |F|$ and $|E| = m_1 + m_2 + m_F$. Also,

$$\sum_{v \in F} \deg(v) = m_1' + m_2' + 2m_F.$$

To achieve the desired bound on $\sum_{v \in F} \deg(v)$, we bound m_1' and m_2', and sum these bounds together with $2m_F$. We will also need a bound on $2|F|$.

As we have already pointed out, the removal of F leaves a forest and thus each of the vpath-components is a tree. Therefore, the number of edges in the subgraph induced by V_1 is $n_1 - \kappa$, and thus $m_1' = m_1 - n_1 + \kappa$. Let us bound κ. Consider a path $P_v \in \mathcal{P}$. This path is obtained by deleting v from the cycle C_v. By definition, C_v contains exactly one vertex in F, namely v, so P_v contains no vertices from F, and therefore exists in the graph G'. It follows that P_v is fully contained in exactly one vpath-component. To bound the number of vpath-components, observe that for all $v \in F$, the cycle C_v contains at least one vertex of F^* (since F^* is an FVS), so either $v \in F^*$, in which case $v \in F^* \cap F$, or else P_v contains a vertex of F^*, in which case this vertex belongs to $F^* \setminus F$. Thus each vpath-component either contains a path P_v such that $v \in F^* \cap F$ or else contains a vertex of $F^* \setminus F$. Thus $\kappa \leq |F^* \cap F| + |F^* \setminus F| = |F^*|$, and hence

$$m_1' \leq m_1 - n_1 + |F^*|.$$

The last expression also bounds $2|F|$ since every vertex $v \in F$ is connected by at least two edges to vertices in V_1 (the two edges connecting it to its neighbors in the cycle C_v), and thus $2|F| \leq m_1'$, which implies

$$m_1 - n_1 + |F^*| \geq 2|F|.$$

To bound m_2', recall that the degree of every vertex in the graph is at least 2. Thus $2n_2 \leq \sum_{u \in V_2} \deg(u) = 2(m_2 - m_2') + m_2' = 2m_2 - m_2'$, which implies

$$m_2' \leq 2m_2 - 2n_2.$$

Putting the pieces together, we get

$$\begin{aligned}
\sum_{v \in F} \deg(v) &= m_1' + m_2' + 2m_F \\
&\leq m_1 - n_1 + |F^*| + 2m_2 - 2n_2 + 2m_F \\
&= 2(m_1 + m_2 + m_F) - 2n_1 - 2n_2 - (m_1 - n_1 + |F^*|) + 2|F^*| \\
&\leq 2|E| - 2n_1 - 2n_2 - 2|F| + 2|F^*| \\
&= 2(|E| - |V| + |F^*|).
\end{aligned}$$

Remark. In addition to the above choice of w_1, the following two alternative weight decompositions have been proposed in the literature, and both have been shown to be 2-effective. The first is obtained by replacing Lines 11 and 12 of Algorithm **FVS** with the following:

If G contains a *semi-disjoint* cycle[3] C:
  Let $\epsilon = \min_{v \in C} w(v)$.
  Define the weight functions $w_1(v) = \begin{cases} \epsilon & v \in C, \\ 0 & \text{otherwise,} \end{cases}$ and $w_2 = w - w_1$.
Else:
  Let $\epsilon = \min_{v \in V} w(v)/(\deg(v) - 1)$.
  Define the weight functions $w_1(v) = \epsilon \cdot (\deg(v) - 1)$
  and $w_2 = w - w_1$.

The second is obtained by:

Choose an *endblock*[4] B, and let $\epsilon = \min_{v \in B} w(v)/(\deg(v) - 1)$.
Define the weight functions $w_1(v) = \begin{cases} \epsilon \cdot (\deg(v) - 1) & v \in B, \\ 0 & \text{otherwise,} \end{cases}$
and $w_2 = w - w_1$.

5.5 Background

Minimal solutions and feedback vertex set. The use of minimal solutions in the local ratio setting appeared for the first time in the context of *feedback vertex set*. This problem is NP-hard [Karp 1972] and MAX SNP-hard [Lund and Yannakakis 1993], and at least as hard to approximate as vertex cover [Lewis and Yannakakis

[3] A cycle C is *semidisjoint* if there exists a vertex $x \in C$ such that $\deg(u) = 2$ for every vertex $u \in C \setminus \{x\}$.
[4] An *endblock* is a biconnected component containing at most one articulation point.

1980]. An approximation algorithm for the unweighted case that achieves a performance ratio of $2\log_2 n$ is essentially contained in a lemma due to Erdös and Pósa [1962]. Monien and Shultz [1981] improve the ratio to $\sqrt{\log n}$. Bar-Yehuda et al. [1998] give a 4-approximation algorithm for the unweighted case, and an $O(\log n)$-approximation algorithm for the weighted case. Bafna et al. [1999] present a local ratio 2-approximation algorithm for the weighted case (using the first of the two alternative weight decompositions presented above). Their algorithm is the first local ratio algorithm to make use of the concept of minimal solutions (although this concept was used earlier in primal-dual algorithms [Ravi and Klein 1993; Agrawal et al. 1995; Goemans and Williamson 1995]). At about the same time, Becker and Geiger [1996] also obtained a 2-approximation algorithm for the problem. Algorithm **FVS** is a recursive formulation of their algorithm. Chudak et al. [1998] interpret the above two algorithms in primal-dual terms, and suggest a third algorithm, whose local ratio interpretation is described by the second alternative weight decomposition given above. Fujito [1998] proposes a generic local ratio algorithm for node-deletion problems with nontrivial and hereditary graph properties[5]. Algorithm **RecursiveVC**, Algorithm **FVS**, and the algorithm of Bafna et al. [1999] can be seen as instances of Fujito's generic algorithm. Bar-Yehuda [2000] presents a unified local ratio approach for covering problems. His presentation contains a short generic approximation algorithm which can explain many known exact optimization and approximation algorithms for covering problems. Bar-Yehuda and Rawitz [2001] devise a framework that extends the generic algorithm from Bar-Yehuda [2000]. The notion of effectiveness of a weight function first appeared in Bar-Yehuda [2000]. A similar idea appeared earlier in the primal-dual setting [Bertsimas and Teo 1998].

Partial hitting set. The *partial hitting set* problem generalizes *hitting set*, and as we have remarked in Section 4.1, *hitting set* and *set cover* are equivalent problems. For this reason, the *partial hitting set* problem is often described in the literature in *set cover* terms and is referred to as *partial covering*. The problem was first studied by Kearns [1990] in relation to learning. He proves that the performance ratio of the greedy algorithm is at most $2H_m + 3$. Slavík [1997] shows that it is actually bounded by 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 [1998], who obtained the first polynomial time 2-approximation algorithm for it. The $s_{\max}$-approximation algorithm for *partial hitting set* given in this section (Algorithm **PHS**) is due to Bar-Yehuda [2001]. In fact, his approach can be used to approximate an extension of the problem in which there is a *length* l_i associated with each set S_i, and the goal is to hit sets of total length at least k. (The plain *hitting set* problem is the special case where $l_i = 1$ for all i.) Gandhi et al. [2001] present a multi-phase primal-dual algorithm for *partial hitting set* achieving a performance ratio of $s_{\max}$.

Network design. As we have seen, Algorithm **ND** can be applied to problems that can be modeled by 0-1 downwards monotone functions. These include *lower capacitated partitioning* problems [Goemans and Williamson 1995; Imielińska et al.

[5]A graph property π is *nontrivial* if it is true for infinitely many graphs and false for infinitely many graphs; it is *hereditary* if every subgraph of a graph satisfying π also satisfies π.

1993] and *location design* problems [Laporte 1988; Laporte et al. 1983]. (For further discussion see Goemans and Williamson [1997].) Algorithm **ND** has evolved from work on the *generalized Steiner network* problem (also called the *survival network design* problem), in which we are given a graph $G = (V, E)$, edge costs, and a nonnegative integer r_{ij} for each pair of vertices i and j. Our goal is to find a minimum weight subset of edges $E' \subseteq E$ such that there are at least r_{ij} edge-disjoint paths between each pair of vertices $i, j \in V$ in the graph (V, E'). This problem can be modeled as a *network design* problem with the proper function[6] $f(S) = \max_{i \in S, j \notin S} r_{ij}$. Note that the *Steiner tree* problem is a special case of this problem where $r_{ij} = 1$ if i and j are terminals, and $r_{ij} = 0$ otherwise. Agrawal et al. [1995] describe an approximation algorithm for a variant of the *generalized Steiner network* problem in which an edge can be selected (and paid for) multiple times. The performance ratio of their algorithm is $2\lceil \log_2(r_{\max} + 1) \rceil$, where $r_{\max} = \max_{ij} r_{ij}$. Goemans and Williamson [1995] present a primal-dual approximation framework for network design problems in which the demand function is 0-1 proper. The approximation ratio of their generic algorithm is $2 - \frac{2}{|V_t|}$. (In the case of *Steiner tree* their algorithm simulates the algorithm of Agrawal et al. [1995].) Algorithm **ND** is a local ratio version of this algorithm. Williamson et al. [1995] present the first approximation algorithm for general proper functions. Their algorithm finds a solution within a factor of $\max\{2f_{\max} - 1, 2\}$ of the optimal, where $f_{\max} = \max_S f(S)$. In addition to the primal-dual approach their algorithm uses the idea of satisfying f in "phases," which was introduced by Ravi and Klein [1993] (in the context of a 3-approximation algorithm for proper functions with range $\{0, 2\}$). Gabow et al. [1993] improve the efficiency of this algorithm. Goemans et al. [1994] improve the ratio for the general case to $2H_{f_{\max}}$. Jain [1998] considers *weakly super-modular* functions, that is, functions f satisfying (1) $f(V) = 0$, and (2) $f(A) + f(B) \leq \max\{f(A \setminus B) + f(B \setminus A), f(A \cap B) + f(A \cup B)\}$ for all $A, B \subseteq V$. (Note that proper functions are weakly super-modular.) He describes a 2-approximation algorithm based on LP rounding.

6. 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

[6] A function that satisfies symmetry and maximality is called proper.

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 6.1. We apply this framework to a collection of throughput maximization problems in Section 6.2. Following that, we consider loss minimization in Section 6.3, and resource minimization in Section 6.4.

6.1 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* $\mathcal{C}$, find a *solution vector* $x \in \mathbb{R}^n$ satisfying the constraints in $\mathcal{C}$ 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 9 LOCAL RATIO—MAXIMIZATION PROBLEMS. *Let $\mathcal{C}$ 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 $\mathcal{C}$) 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 5.2. It too consists of a three-way *if* condition that directs execution to one of three main options: *optimal solution*, *problem size reduction*, and *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[7]. 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 (and not for 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.*

6.2 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) 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.

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 $\mathcal{A}(I)$.

6.2.1 *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.

[7]We actually impose a somewhat weaker condition, as described in Section 6.2.

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). Note that the choice of J ensures that all of the intervals intersecting it intersect each other at J's right endpoint (see Figure 3). 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.

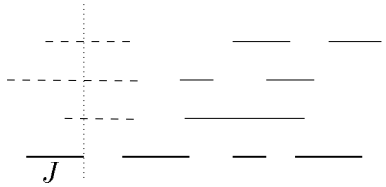

Fig. 3. J, $\mathcal{A}(J)$, and $\mathcal{I}(J)$: heavy lines represent $\mathcal{A}(J)$; dashed lines represent $\mathcal{I}(J)$.

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 instances, i.e., $\cup_{i=1}^{m} \mathcal{A}_i$.

Algorithm IS$(\mathcal{A}, w)$

1. If $\mathcal{A} = \emptyset$, return $\emptyset$.
2. If there exists an instance 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 the weight functions
$$w_1(I) = w(J) \cdot \begin{cases} 1 & I \in \mathcal{A}(J) \cup \mathcal{I}(J), \\ 0 & \text{otherwise,} \end{cases} \text{ and } w_2 = w - w_1.$$
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.

6.2.2 *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., Golumbic [1980]). 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.

6.2.3 *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).

6.2.4 *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 the instance once for each allowable machine.) We extend our scheduling model to handle this problem as follows. We now have k types of unit size 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.

6.2.5 *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 can use Algorithm **IS** to find a 2-approximate schedule.

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 3-approximation 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 $w(J) \cdot 2 \cdot \frac{1}{2} = w(J)$. 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.

6.2.6 *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 $O(n^2/\epsilon)$. Bar-Noy et al. [2001] provide the full details.

6.2.7 *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 one 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 all possible batches in which it is represented. All of these batches conflict with each other (in jobs), 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 possible 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 idea 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. [2002], 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.

6.3 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 6.2) 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 $(D, \mathcal{A}, w)$ denote the input, where D is the resource quantity function, $\mathcal{A}$ is the set of activity instances, and w is the weight function. If the set of all activity instances 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 $(D, \mathcal{A}, w_2)$.

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 maximum such that $w_2(I) \geq 0$ 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 10. *Every collection of instances from $\mathcal{I}(t^*)$ whose total width is at least Δ^* has total weight at least $\epsilon\Delta^*$.*

OBSERVATION 11. *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 10, 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 12. *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,

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

Hence, $\sum_{I \in \overline{\mathcal{M}}(t) \setminus \{J\}} d(I) < \Delta^*$, and therefore, by Observation 11,

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

□

To complete the analysis, we consider two cases. 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 some 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 12, the total weight of these instances is less than $4\epsilon\Delta^*$.

6.3.1 *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 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 completes the description of the reduction. Note that the reduction is cost preserving: there is a one-to-one correspondence between solutions to the caching problem and solutions to our scheduling problem, and corresponding solutions have the same cost. Thus the reduction implies a 4-approximation algorithm for the general caching problem via our 4-approximation algorithm for loss minimization.

6.4 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 resource cost. 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 (or fully 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.

6.4.1 *Bandwidth Trading.* In the *bandwidth trading* problem we are given a set of machine *types* $\mathcal{T} = \{T_1, \ldots, T_m\}$ and a set of *jobs* $J = \{1, \ldots, n\}$. 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 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 4).
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 later in the text).
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 this 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 $3k$.

Based on these facts, we now prove that the algorithm is 3-approximation. 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 the optimum cost for $(\mathcal{T}, J, w)$ is the same as for $(\mathcal{T} \setminus \{T\}, J \setminus J', w)$ and $w(S) = w(S')$. Therefore S is 3-approximate with respect to $(\mathcal{T}, J, w)$. 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 Fact 1 above, $w_2(S) \leq w_2(S')$, and therefore S too is 3-approximate with respect to w_2. By Fact 2 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.

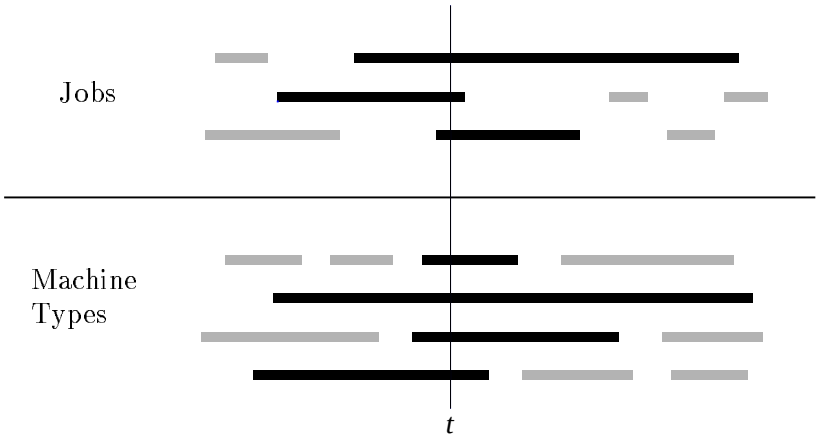

Fig. 4. Jobs containing time t (top, dark), and machine types available at time t (bottom, dark).

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 4.) 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 contain time t). Let J_c be the set of all jobs scheduled by S' 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 5.) 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 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 by 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 all contain the

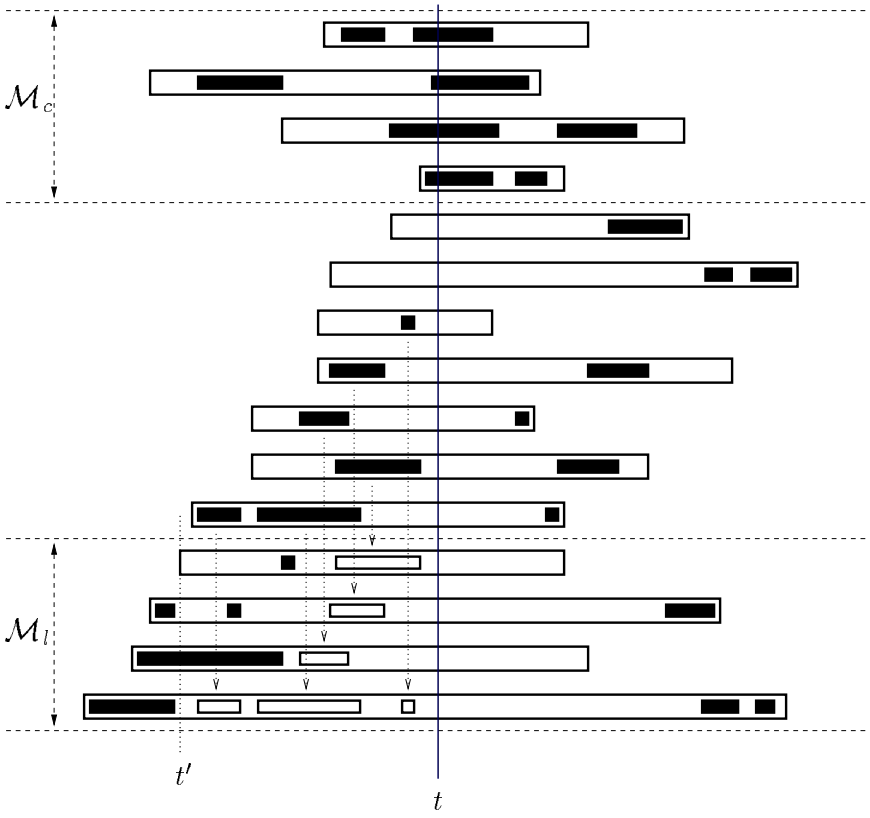

Fig. 5. Rescheduling jobs that were assigned to $\mathcal{M}_t$ and exist before time t.

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 containing any single time instant.

(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 our treatment of J_l above.

We thus 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.

6.5 Background

Throughput maximization. As mentioned earlier, single machine scheduling with one instance per activity is equivalent to *maximum weight independent set in interval graphs* and hence polynomial-time solvable [Golumbic 1980]. Arkin and Silverberg [1987] 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 [Spieksma 1999] or if instances may require arbitrary amounts of the resource. (In the latter case the problem is NP-hard as it contains *knapsack* [Garey and Johnson 1979] as a special case in which all time intervals intersect.) Spieksma [1999] 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. [2001] 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. [2001], 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 Bar-Noy et al. [2001] either in the approximation factor or in the running time complexity. Their algorithms can also be interpreted within the primal-dual schema (see also Bar-Yehuda and Rawitz [2001]) and are the first local ratio (or primal-dual) algorithms for maximization problems. Sections 6.2 and 6.3, with the exception of Section 6.2.7, are based on Bar-Noy et al. [2001]. Independently, Berman and DasGupta [2000] also improve upon the algorithms given in Bar-Noy et al. [2001]. They develop an algorithm for interval scheduling that is nearly identical to the one from Bar-Noy et al. [2001]. Furthermore, they employ the same rounding idea used in Bar-Noy et al. [2001] in order to contend with time windows. In addition to single machine scheduling, they also consider scheduling on parallel machines, both identical (obtaining an approximation guarantee of $(k+1)^k/((k+1)^k - k^k)$) and unrelated (obtaining an approximation guarantee of 2).

Chuzhoy et al. [2001] 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 6.2.7 is from Bar-Noy et al. [2002]. 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 Bar-Noy et al. [2002] for more details.)

Loss minimization. Section 6.3 is based on Bar-Noy et al. [2001]. Albers et al. [1999] 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. In other words, 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 6.3 is also from Albers et al. [1999].

Resource minimization. Section 6.4 is based on a write-up by Bhatia et al. [2003]. 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* [Bhatia et al. 2003; Jansen 1994]. Kolen and Kroon [1991; 1994] show that versions of the general problem considered in Bhatia et al. [2003] are NP-hard.

7. ODDS AND ENDS

In this section we present three local ratio algorithms that do not fit properly in the frameworks introduced previously. Each algorithm deviates from these frameworks in a different manner, but they are all based on the Local Ratio Theorem and related ideas. The first two algorithms are interpretations of known algorithms for *longest path in a DAG* and *minimum s-t cut*. These problems are polynomial-time solvable, and the algorithms presented here demonstrate the use of the local ratio technique in the design of 1-approximation algorithms and the utilization of negative weights in this context. The third problem we consider is *capacitated vertex cover*, for which we describe a 2-approximation algorithm. The interesting feature of this NP-hard problem is that feasible solutions are not necessarily 0-1 vectors.

7.1 Longest Path in a DAG

The *longest path* problem is, given an edge-weighted graph and two distinguished vertices s and t, find a simple path from s to t of maximum *length*, where the *length* of a path is defined as the sum of weights of its edges. For general graphs (either directed or undirected) the problem is NP-hard, but for directed acyclic graphs (DAGs) it is solvable in linear time by a Dijkstra-like algorithm that processes the nodes in topological order. In this section we present a local ratio interpretation of this algorithm. We allow negative arc weights, and we assume without loss of generality that s is a root, i.e., that every node is reachable from s. (Otherwise, simply delete all unreachable nodes.)

Let us first introduce some terminology. We say that node u is a *proper successor* of node v if u is entered by exactly one arc, and this arc is (v, u).

OBSERVATION 13. *Let s be root of a DAG on two or more nodes. Then s has at least one proper successor in the graph. (For example, the second node in a topological sort of the graph is a proper successor of s.)*

We also define the act of *contracting* an arc (u, v) as the following three-step process:

(1) Delete the arc (u, v).
(2) Redirect every arc leaving (or entering) u to leave (or enter) v instead. This modification may create pairs of parallel arcs. For each such pair, delete the arc of lesser weight (breaking ties arbitrarily).
(3) Delete u.

Let G' be the graph obtained from G by contracting an arc (u, v), and let $P' = v, v_1, \ldots, v_k$ be a path leaving v in G'. The arc (v, v_1) in G' may have originated

from (u, v_1) or from (v, v_1) in G. Depending on which the case is, we define the path P in G corresponding to P' to be either the path $u, v_1, \ldots, v_k$ or the path $u, v, v_1, \ldots, v_k$.

OBSERVATION 14. *Let G be an arc-weighted DAG with root s. Let v be a proper successor of s, and suppose $w(s, v) = 0$. Let G' be the graph obtained from G by contracting (s, v). Then:*

(1) G' is a DAG with root v.

(2) The maximum length of a path from s to t in G is equal to the maximum length of a path from v to t in G'.

(3) If P' is a longest path from v to t in G', then the corresponding path P in G is a longest path from s to t.

The algorithm is based on Observations 13 and 14. It repeatedly contracts the arc connecting the DAG's root to one of its proper successors.

Algorithm LPDAG(G, s, t, w)

1. If $s = t$ return the path consisting of the single node s.
2. Else, if s has a proper successor v such that $w(s, v) = 0$ do:
3. Let G' be the graph obtained by contracting (s, v).
4. $P' \leftarrow$ **LPDAG**(G', v, t, w).
5. Return the path in G corresponding to P'.
6. Else:
7. Let x be a proper successor of s and let $\epsilon = w(s, x)$.
8. Define the weight functions $w_1(u, v) = \begin{cases} \epsilon & u = s, \\ 0 & \text{otherwise,} \end{cases}$ and $w_2 = w - w_1$.
9. Return **LPDAG**(G, s, t, w_2).

It is easy to see that w_1 is strongly 1-effective for all ϵ (even negative!) Hence, Algorithm **LPDAG** solves the problem optimally.

7.2 Minimum s-t Cut

The *minimum s-t cut* problem is defined as follows. Given an arc-weighted directed graph $G = (N, A)$ and a pair of distinct distinguished nodes s and t, a *directed s-t cut* (or simply *cut* for short) is a set of arcs whose removal disconnects all directed paths from s to t. The goal is to find a *minimum* cut, i.e., a cut of minimum weight. Note that there always exists a minimum cut which is also minimal (in the sense of set inclusion), since arc weights are non-negative. Thus we restrict our attention to minimal cuts, and therefore have the following equivalent definition (which is more common in the literature). A cut is a partition of the graph nodes into two sets S and $\bar{S}$ such that $s \in S$ and $t \in \bar{S}$, and the *cut arcs* are the arcs leaving S and entering $\bar{S}$. We use the two definitions of *cut* interchangeably. We remark that the undirected version of the problem can be reduced to the directed case by replacing

each edge by two arcs (in opposite directions) with the same weight as the original edge.

It is well known that the *minimum s-t cut* problem can be solved by max-flow techniques, particularly, by the Ford and Fulkerson algorithm for max-flow [Ford and Fulkerson 1956]. In this section we present a local ratio interpretation of the Ford and Fulkerson algorithm. Apart from the fact that the algorithm below solves an optimization problem optimally, it is also unique in that it is the first (and only, as far as we know) local ratio algorithm that uses negative weights in w_1[8]. Because of these negative weights, there are cases where zero weight arcs become positive again, and therefore we cannot delete zero-weight arcs as in previous algorithms.

In the algorithm below, we assume that for every arc (u, v) in the graph, the arc (v, u) exists as well. (If this is not the case we can simply add the missing arcs with zero weight.) The algorithm is based on the following observation. We say that an arc is *unsaturated* if its weight is strictly positive. A path is *unsaturated* if it consists of unsaturated arcs only. Let P be an unsaturated path from s to t and let $\overleftarrow{P}$ be its reverse (from t to s). Define

$$w_1(u,v) = \begin{cases} \epsilon & (u,v) \in P, \\ -\epsilon & (u,v) \in \overleftarrow{P}, \\ 0 & \text{otherwise.} \end{cases}$$

We claim that w_1 is 1-effective because the cost (with respect to w_1) of every cut is precisely ϵ. To see this, let C be any minimal feasible solution, i.e., let C be a set of arcs defined by a cut $(S, \bar{S})$. Consider the path P. It starts at s and proceeds initially within S. At some point it crosses over to $\bar{S}$ and then possibly crosses the cut back and forth several times, ending finally in $\bar{S}$. It then continues to t within $\bar{S}$. Thus, if $k \geq 1$ is the number of times P crosses the cut from S to $\bar{S}$, then $k-1$ is the number of times it crosses the cut in the reverse direction. Thus C contains precisely k edges of P and $k-1$ edges of $\overleftarrow{P}$, and thus its weight with respect to w_1 is $\epsilon \cdot k - \epsilon(k-1) = \epsilon$. (See Figure 6 for an illustration.)

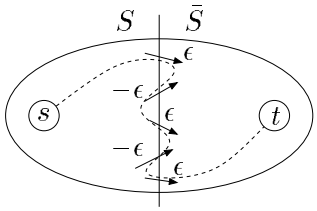

Fig. 6. The weight of a cut with respect to w_1.

[8]Although we have allowed negative weights in Algorithm **LPDAG** (Section 7.1), that algorithm may be easily modified to avoid them. In contrast, negative weights are inherent in the present algorithm.

Algorithm stCut(G, s, t, w)

1. Let S be the set of nodes reachable from s by unsaturated paths.
2. If $t \notin S$, return the cut $(S, \bar{S})$.
3. Else:
4. Let P be an unsaturated path from s to t and let $\overleftarrow{P}$ be its reverse.
5. Let $\epsilon = \min\{w(u,v) \mid (u,v) \in P\}$.
6. Define the weight functions $w_1(u,v) = \begin{cases} \epsilon & (u,v) \in P, \\ -\epsilon & (u,v) \in \overleftarrow{P}, \\ 0 & \text{otherwise,} \end{cases}$ and $w_2 = w - w_1$.
7. Return **stCut**(G, s, t, w_2).

Algorithm **stCut** is actually Ford and Fulkerson's algorithm [Ford and Fulkerson 1956]. Path P is an augmentation path, and w_2 defines the residual weights. As with the original algorithm, the time complexity of the algorithm depends on how P is chosen in Line 4. If it is chosen poorly, the algorithm might require pseudopolynomial time (or even fail to terminate if irrational weights are allowed).

7.3 Capacitated Vertex Cover

The *capacitated vertex cover* problem is similar to plain *vertex cover*, but now each vertex u has, in addition to its weight $w(u)$, also a *capacity* $c(u) \in \mathbb{N}$ that determines the number of edges it may cover. Specifically, vertex u may cover up to $c(u)$ incident edges. However, multiple "copies" of u can be used to cover additional edges[9]. Thus a feasible solution is an assignment of every edge to one of its endpoints, subject to the constraint that no edge is assigned to a zero-capacity vertex. Note that the presence of zero-capacity vertices may render the problem infeasible, but detecting this is easy: the problem is infeasible if and only if there is an edge whose two endpoints have zero capacity. We will assume henceforth that a feasible solution exists. Also note that the special case in which $c(u) \geq \deg(u)$ for every vertex u is the ordinary *vertex cover* problem.

To define the cost of a feasible solution A, let us introduce some notation. Denote by $A(u)$ the set of edges assigned to vertex u and by $\alpha(u)$ the number of copies of u required to cover the edges $A(u)$. (If $A(u) = \emptyset$, then $\alpha(u) = 0$. Otherwise, $\alpha(u) = \lceil |A(u)|/c(u) \rceil$. Note that in the latter case the ratio $|A(u)|/c(u)$ is well defined, since $c(u)$ must be positive.) The cost of assignment A is $\sum_u \alpha(u)w(u)$. Thus, although feasible solutions are defined as assignments of edges to vertices, when considering their cost, they can be viewed as vectors of α values, and thus can be treated within the local ratio framework.

In the description of the recursive algorithm below, we use $N(u)$ to denote the set of vertex u's neighbors, we denote by V_+ the set of vertices with non-zero capacity

[9]Such capacities are called *soft*. With *hard* capacities only one copy of every vertex is allowed.

and non-zero degree, and we denote $\tilde{c}(u) = \min\{\deg(u), c(u)\}$.

Algorithm CVC$(G = (V, E), c, w)$

1. If $E = \emptyset$, return the assignment $A(v) = \emptyset$ for all $v \in V$.
2. Else, if there exists a vertex $u \in V_+$ such that $w(u) = 0$ do:
3. Let G' be the graph obtained from G by deleting u.
4. $A \leftarrow \mathbf{CVC}(G', c, w)$.
5. For every $v \in N(u)$ do:
6. If $\deg(v) \leq c(v)$ and $A(v) \neq \emptyset$ do:
7. $A(v) \leftarrow A(v) \cup \{(u, v)\}$.
8. Else:
9. $A(u) \leftarrow A(u) \cup \{(u, v)\}$.
10. Return A.
11. Else:
12. Let $\epsilon = \min_{u \in V_+}\{w(u)/\tilde{c}(u)\}$.
13. Define the weight functions $w_1(v) = \epsilon \cdot \tilde{c}(v)$ and $w_2 = w - w_1$.
14. Return $\mathbf{CVC}(G, c, w_2)$.

Consider the recursive call made in Line 4. In order to distinguish between the assignment obtained by this recursive invocation and the assignment returned in Line 10, we denote the former by A' and the latter by A. Assignment A' is for graph G', and we denote the corresponding parameters α' and $\tilde{c}'$. Similarly, we use α and $\tilde{c}$ for the parameters of A and G.

To analyze the algorithm, we use a charging scheme by which the cost of the solution is charged to the graph edges. More specifically, we imagine that each edge is alloted two coins, which it may distribute between its endpoints. We aim to show that there is a way to distribute the coins so that the number of coins given to any vertex u is at least $\alpha(u)\tilde{c}(u)$. The key observation is the following.

OBSERVATION 15. *Suppose a recursive call is made in Line 4. Let $v \in N(u)$. Then:*

(1) If $\deg(v) \leq c(v)$ and $A'(v) \neq \emptyset$, then $\alpha(v) = \alpha'(v) = 1$ and $\tilde{c}(v) = \tilde{c}'(v) + 1$. Thus $\alpha(v)\tilde{c}(v) = \alpha'(v)\tilde{c}'(v) + 1$.

(2) If $\deg(v) > c(v)$ or $A'(v) = \emptyset$, then $A(v) = A'(v)$ and therefore $\alpha(v) = \alpha'(v)$. Moreover, there are two (not mutually exclusive) cases: either $\alpha(v) = \alpha'(v) = 0$ or $\deg(v) \geq c(v) + 1$. In the latter case the degree of v in G' is at least $c(v)$, and therefore $\tilde{c}(v) = \tilde{c}'(v) = c(v)$. Thus in either case, $\alpha(v)\tilde{c}(v) = \alpha'(v)\tilde{c}'(v)$.

LEMMA 16. *There exists a charging scheme (with respect to graph G and assignment A) in which every vertex u is given at least $\alpha(u)\tilde{c}(u)$ coins. Thus $\sum_u \alpha(u)\tilde{c}(u) \leq 2|E|$.*

PROOF. The proof is by induction on the recursion. For the base case $E = \emptyset$ it is trivial. For the inductive step there are two cases. If the recursive call is made in

Line 14, the claim follows by the inductive hypothesis. Otherwise, the recursive call is made in Line 4, and by the inductive hypothesis there exists a charging scheme for (G', A'). We extend this scheme to (G, A) as follows. First note that the transition from (G', A') to (G, A) consists of adding a single vertex u and a collection of edges incident on it, and assigning each of these edges to either u or its other endpoint. Thus we need only take care of u and its neighbors; these are the only vertices whose α and $\tilde{c}$ values may change. All other vertices are already satisfied by the charging scheme for (G', A'). Let $v \in N(u)$. If $\deg(v) \leq c(v)$ and $A'(v) \neq \emptyset$, then by Observation 15, $\alpha(v)\tilde{c}(v) - \alpha'(v)\tilde{c}(v) = 1$. Edge (u, v) (which is assigned to v) passes one of its coins to v to cover the difference, and passes its other coin to u. If, on the other hand, $\deg(v) > c(v)$ or $A'(v) = \emptyset$, then by Observation 15, v is already satisfied by the charging scheme for (G', A'). Edge (u, v) (which is assigned to u) passes both its coins to u. Thus we ensure that all of u's neighbors are satisfied, and it remains to show that u is satisfied as well. Consider the coins given to u by its incident edges. Each edge assigned to u contributes two coins and each edge not assigned to u contributes one coin. Thus the number of coins is

$$\begin{aligned} |A(u)| + \deg(u) &\geq |A(u)| + \tilde{c}(u) \\ &= \left(\frac{|A(u)|}{\tilde{c}(u)} + 1\right) \cdot \tilde{c}(u) \\ &\geq \left(\frac{|A(u)|}{c(u)} + 1\right) \cdot \tilde{c}(u) \\ &> \alpha(u)\tilde{c}(u). \end{aligned}$$

□

THEOREM 17. *Algorithm CVC returns 2-approximate solutions.*

PROOF. The proof is by induction on the recursion. In the base case the algorithm returns an empty assignment, which is optimal for the empty graph. For the inductive step there are two cases. If the recursive invocation is made in Line 4, then by the inductive hypothesis the assignment A' it returns is 2-approximate with respect to G'. Clearly, the optimum value for G can only be greater than the optimum value for G', and because $w(u) = 0$ and $\alpha(v) = \alpha'(v)$ for all $v \in V \setminus \{u\}$ (by Observation 15), the cost of A equals the cost of A'. Thus A is 2-approximate with respect to G. If the recursive call is made in Line 14, then by the inductive hypothesis the assignment is 2-approximate with respect to w_2. By Lemma 16 its cost with respect to w_1 is at most $2\epsilon|E|$, and clearly the optimal cost is at least $\epsilon|E|$. Thus the assignment is 2-approximate with respect to w_1 too, and by the Local Ratio Theorem it is 2-approximate with respect to w as well. □

7.4 Background

The problem of finding the longest path in a DAG (also called *critical path*) arises in the context of PERT (Program Evaluation and Review Technique) charts. For more details see Cormen et al. [1990, page 538] or Even [1979, pp. 138–142]. Algorithm **stCut**, the local ratio interpretation of Ford and Fulkerson's Algorithm [Ford and Fulkerson 1956], is from Bar-Yehuda and Rawitz [2001]. The algorithm for *ca-*

pacitated vertex cover (Algorithm **CVC**) is a local ratio version of the primal-dual algorithm given by Guha et al. [2002].

8. FRACTIONAL LOCAL RATIO

As we have seen, the standard local ratio approach is to use a weight decomposition that guarantees that the solution constructed by the algorithm will be r-approximate with respect to w_1 (for some desired r). In essence, the analysis consists of comparing at each level of the recursion the solution found in that level and an optimal solution for the problem instance passed to that level, where the comparison is made with respect to w_1 and with respect to w_2. Thus, in each level of the recursion there are potentially two optima (one with respect to w_1 and one with respect to w_2) against which the solution is compared, and in addition, different optima are used at different recursion levels. The *fractional local ratio* paradigm takes a different approach. It uses a single solution x^* to the *original* problem instance as the yardstick against which all intermediate solutions (at all levels of the recursion) are compared. In fact, x^* is not even feasible for the original problem instance but rather for a relaxation of it. Typically, x^* will be an optimal fractional solution to a linear programming[10] (LP) relaxation.

The fractional local ratio technique is based on a "fractional" version of the Local Ratio Theorem. Recall that both previous versions of the theorem (Theorems 2 and 9) apply to problems that can be formulated as finding an optimal vector $x \in \mathbb{R}^n$ subject to a set of feasibility constraints $\mathcal{C}$. In all previous applications we have assumed that the constraints describe the problem to be solved precisely, and in particular, one of the constraints is that x must be a 0-1 (or at least an integral) vector. In contrast, applications of the fractional local ratio technique consider relaxations of the actual problems—typically, LP relaxations of their integer programming (IP) formulations. This is the context in which we formulate the new version of the theorem. Particularly, we consider non-integral vectors, hence the term *fractional*. We precede the statement of the theorem with the following useful definition.

DEFINITION 4. *Let $r \geq 1$ and let $w \in \mathbb{R}^n$ be a vector of weights in the context of an optimization problem. Let x and x^* be vectors in $\mathbb{R}^n$. Vector x is said to be* r-approximate relative to x^* *(with respect to w) if $w \cdot x \leq r(w \cdot x^*)$ (for a minimization problem) or $w \cdot x \geq (w \cdot x^*)/r$ (for a maximization problem).*

The theorem is nothing more than the following straightforward observation.

THEOREM 18 FRACTIONAL LOCAL RATIO. *Let $w, w_1, w_2 \in \mathbb{R}^n$ be weight vectors in the context of an optimization problem such that $w = w_1 + w_2$. Let x^* and x be vectors in $\mathbb{R}^n$ such that x is r-approximate relative to x^* with respect to w_1 and with respect to w_2. Then, x is r-approximate relative to x^* with respect to w as well.*

The first step in the fractional local ratio technique is to obtain an optimal solution x^* to an LP relaxation of the problem. This solution is then used to drive a

[10] This section assumes familiarity with the concepts of integer programming, linear programming, and LP relaxations.

recursive algorithm that constructs a feasible solution (to the actual problem, not the relaxation) in a manner quite similar to the standard local ratio method. At each level of the recursion, either the problem size is reduced or the weight function is decomposed. The novelty of the fractional approach is that in weight decomposition steps, the construction of w_1 is based on x^* and the analysis compares the weight (with respect to w_1) of the solution returned with $w \cdot x^*$. Thus the fractional local ratio technique is a combination of LP rounding and the standard local ratio approach.

Intuitively, the drawback of the standard local ratio technique is that at each level of the recursion we attempt to approximate the weight of a solution that is optimal with respect to the weight function present at that level. The superposition of these "local optima" may be significantly worse than the "global optimum," i.e., the optimum for the original problem instance. Conceivably, we could obtain a better bound if at each level of the recursion we approximated the weight of a solution that is optimal with respect to the original weight function. This is the idea behind the fractional local ratio approach.

The best way to explain the technique is by means of an example. In the next section we demonstrate the technique on *maximum weight independent set*, and following that we apply it to *maximum weight independent set in t-interval graphs*, which is somewhat more involved.

8.1 Maximum Weight Independent Set

In the *maximum weight independent set* problem we are given a vertex-weighted graph $G = (V, E)$, and are asked to find a maximum weight *independent set* of vertices. An *independent set* of vertices is a subset of V in which no two vertices are adjacent.

Consider the following weight function. For a given vertex v, let $N[v]$ denote the set consisting of vertex v and all its neighbors. Fix some vertex v and let $\epsilon > 0$. Define

$$w_1(u) = \begin{cases} \epsilon & u \in N[v], \\ 0 & \text{otherwise.} \end{cases}$$

Let $\Delta = \max_{u \in V} \max\{1, \deg(u)\}$. Clearly, w_1 is Δ-effective since every feasible solution has weight at most $\epsilon \cdot \max\{1, \deg(v)\} \leq \epsilon \cdot \Delta$ and every maximal solution has weight at least ϵ. We can therefore use this weight function as the basis for a Δ-approximation algorithm in the standard local ratio framework. We can do better, however, by employing the fractional local ratio technique. Specifically, we can achieve an approximation guarantee of $\frac{1}{2}(\Delta + 1)$, as we show now.

The first step is to solve the following LP relaxation of the natural IP formulation of the problem.

$$\begin{aligned} \text{Maximize} \quad & \sum_{u \in V} w(u) x_u \quad \text{subject to} \\ & x_u + x_v \leq 1 \qquad \forall\, (u, v) \in E, \\ & 0 \leq x_u \leq 1 \qquad \forall\, u \in V. \end{aligned}$$

Let x^* be an optimal (fractional) solution. We now run the recursive algorithm described next to obtain an independent set. Note that the algorithm does not

contain problem-size reduction steps, as do other local ratio algorithms. Specifically, it does not delete vertices of non-positive weight. This is due to the nature of the analysis to follow; it is more convenient to simply ignore vertices of non-positive weight than to delete them.

Algorithm FracLRIS(G, w, x^*)

1. Let $V_+ = \{u \mid w(u) > 0\}$ and let $G_+ = (V_+, E_+)$ be the subgraph of G induced by V_+.
2. If $E_+ = \emptyset$, return V_+.
3. Let $v \in V_+$ be a vertex minimizing $\sum_{u \in N[v] \cap V_+} x^*_u$ and let $\epsilon = w(v)$.
4. Define the weight functions $w_1(u) = \begin{cases} \epsilon & u \in N[v] \cap V_+, \\ 0 & \text{otherwise}, \end{cases}$ and $w_2 = w - w_1$.
5. $S' \leftarrow$ **FracLRIS**(G, w_2, x^*).
6. If $S' \cup \{v\}$ is an independent set do:
7. Return $S = S' \cup \{v\}$.
8. Else:
9. Return $S = S'$.

For the analysis, let x denote the incidence vector of the solution S returned by the algorithm. We claim that x is $\frac{1}{2}(\Delta + 1)$-approximate relative to x^* (in other words, that $w \cdot x \geq 2/(\Delta+1)(w \cdot x^*)$), and thus S is $\frac{1}{2}(\Delta+1)$-approximate. As usual, the proof is by induction on the recursion, but to facilitate it we need an additional claim, namely, that $S \subseteq V_+$. We prove both claims simultaneously. In the base case ($E_+ = \emptyset$) we have $S = V_+$ and $w \cdot x = \sum_{u \mid w(u)>0} w(u) \cdot 1 \geq \sum_{u \mid w(u)>0} w(u) \cdot x^*_u \geq \sum_u w(u) \cdot x^*_u = w \cdot x^*$. Since $w \cdot x \geq 0$, we get $w \cdot x \geq 2/(\Delta+1)(w \cdot x^*)$ even if $w \cdot x^*$ is negative. For the inductive step, let x' be the incidence vector of S' (obtained in Line 5), and let $V'_+ = \{u \mid w_2(u) > 0\}$. Then, by the inductive hypothesis and the definition of w_2, $S' \subseteq V'_+ \subseteq V_+$. Since $v \in V_+$, Lines 6–9 ensure that $S \subseteq V_+$. In addition, these lines also ensure that either $v \in S$ or else S contains some $v' \in N[v] \cap S' \subseteq N[v] \cap V_+$. In either case, S contains at least one vertex from $N[v] \cap V_+$ and thus $w \cdot x \geq \epsilon$. The following lemma completes the proof, as it implies that $w \cdot x^* \leq \frac{1}{2}\epsilon(\Delta + 1)$.

LEMMA 19. *The choice of v in Line 3 ensures that $\sum_{u \in N[v] \cap V_+} x^*_u \leq \frac{1}{2}(\Delta + 1)$.*

PROOF. It suffices to show that $\sum_{u \in N[s] \cap V_+} x^*_u \leq \frac{1}{2}(\Delta + 1)$ is satisfied for some vertex $s \in V_+$. We show this for $s = \arg\max_{u \in V_+} x^*_u$. Note that $N[s] \cap V_+$ is the set consisting of s and its neighbors in G_+, and that $|N[s] \cap V_+| \leq \Delta + 1$. If s is an isolated vertex in G_+, it clearly satisfies the condition. Otherwise, if $x^*_s \leq 1/2$, then $x^*_u \leq 1/2$ for all vertices u in G_+, and the condition is again satisfied. Otherwise, $x^*_s > 1/2$ and thus $x^*_u < 1/2$ for all neighbors u of s in G_+. Fix any such neighbor s'. Then $\sum_{u \in N[s] \cap V_+} x^*_u = (x^*_s + x^*_{s'}) + \sum_{u \in (N[v] \cap V_+) \setminus \{s, s'\}} x^*_u < 1 + \frac{1}{2}(\Delta - 2) =$

$\frac{1}{2}(\Delta+1)$. □

8.1.0.1 *Remarks.* The analysis above shows that v can actually be chosen (in Line 3) as *any* vertex satisfying $\sum_{u\in N[v]\cap V_+} x^*_u \leq \frac{1}{2}(\Delta+1)$—not necessarily the one minimizing $\sum_{u\in N[v]\cap V_+} x^*_u$. By the proof of Lemma 19, the simplest such choice is the vertex in $v \in V_+$ maximizing x^*_v.

8.2 Maximum Weight Independent Set in a t-interval Graph

Consider a teaching laboratory shared by several different academic courses, where each course has a schedule during which it requires the use of the laboratory for t (or less) sessions of varying duration. The problem is to maximize the utilization of the laboratory by admitting a subset of non-conflicting courses. This type of problem has a simple abstraction as the problem of finding a maximum weight independent set in a vertex-weighted *t-interval* graph. To define what a t-interval graph is, we need the following auxiliary definitions. A *t-interval system* is a collection $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_n\}$ of nonempty sets of real intervals, where each $\mathcal{I}_i$ consists of t or less disjoint intervals. Given a t-interval system, the corresponding *intersection* graph is defined as follows. Each set $\mathcal{I}_i$, is represented by a vertex v_i, and there is an edge between two distinct vertices v_i and v_j if some interval in $\mathcal{I}_i$ intersects some interval in $\mathcal{I}_j$. A *t-interval* graph is the intersection graph of some t-interval system. The set system is called a *realization* of the graph. (Note that a given t-interval graph may have many different realizations.) Figure 7 shows a 2-interval graph and one of its realizations.

In this section we present a $2t$-approximation algorithm for the problem of finding a maximum weight independent set in a vertex-weighted t-interval graph. Since even deciding whether a given graph is t-interval is NP-hard (for $t \geq 2$) [West and Shmoys 1984], we shall assume that a realization of the graph is given as part of the input. Given a realization of a t-interval graph, we identify each vertex with the corresponding set of intervals, and furthermore, we adopt the point of view that the intervals are not merely pairs of endpoints, but rather are objects in their own right. What we mean by this is that if a certain interval $[a, b]$ belongs to two vertices, we do not view $[a, b]$ as a single interval shared by two vertices, but rather as two distinct intervals that happen to have identical endpoints. Thus, when viewed as sets of intervals, the graph vertices are always pairwise disjoint. In addition, for a given point p and vertex v, we denote by $p \in\in v$ the statement *p is contained in some interval belonging to v*. Finally, we assume for simplicity that the intervals are closed.

The algorithm is nearly identical to the algorithm for *maximum weight independent set* in general graphs presented in the previous section; the only difference is that in the first step we solve the following LP relaxation, rather than the one used there. Let R denote the set of right endpoints of intervals in the system. The linear program is:

$$\begin{aligned} \text{Maximize} \quad & \sum w(u)x_u \quad \text{subject to} \\ & \sum_{u \mid p\in\in u} x_u \leq 1 \qquad \forall p \in R, \\ & 0 \leq x_u \leq 1 \qquad \forall u \in V. \end{aligned}$$

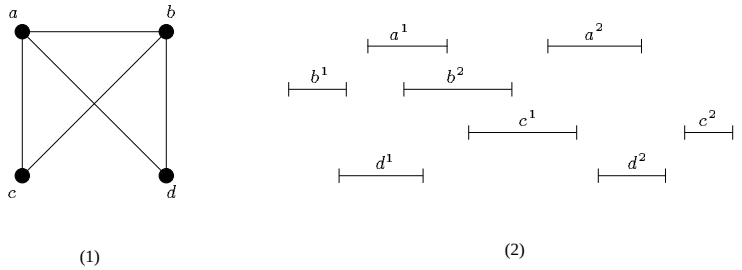

Fig. 7. (1) A 2-interval graph; (2) one of its realizations.

In the second step we run Algorithm **FracLRIS** on the optimal (fractional) solution found, as before. The analysis is similar, except that Lemma 19 is replaced by the following lemma, which provides an approximation guarantee of $2t$.

LEMMA 20. *The choice of v in Line 3 ensures that* $\sum_{u \in N[v] \cap V_+} x^*_u \leq 2t$.

PROOF. To reduce clutter, we restrict the discussion to the graph G_+. Thus, *vertex* means *vertex in* V_+; $\sum_u$ stands for $\sum_{u \in V_+}$; $N[v]$ stands for $N[v] \cap V_+$; etc. Also, If I is an interval belonging to vertex v, we define $x^*_I = x^*_v$. In general, we will use I and J to denote intervals belonging to vertices.

To prove the lemma it suffices to show that there exists a vertex s such that $\sum_{u \in N[s]} x^*_u \leq 2t$. Clearly, every isolated vertex satisfies this condition, so we assume that G_+ contains no isolated vertices. Thus, an equivalent claim to the above is that there exists a vertex s satisfying $x^*_s \sum_{u \in N[s]} x^*_u \leq 2t x^*_s$ (note that $x^*_s > 0$ since s is a vertex in G_+), and to prove this claim it suffices to show that $\sum_s x^*_s \sum_{u \in N[s]} x^*_u \leq 2t \sum_s x^*_s$.

Let us denote by $I \backsim J$ the statement *interval I and interval J intersect*, and by $J \backsim_R I$ the statement *interval J contains the right endpoint of interval I*. (Note that $I \backsim I$ and $I \backsim_R I$ for all I.) Then,

$$\begin{aligned}
\sum_s x^*_s \sum_{u \in N[s]} x^*_u &\leq \sum_{I \backsim J} x^*_I x^*_J \\
&\leq 2 \sum_{J \backsim_R I} x^*_I x^*_J \\
&= 2 \sum_s \sum_{I \in s} \sum_{J \backsim_R I} x^*_I x^*_J \\
&= 2 \sum_s x^*_s \sum_{I \in s} \sum_{J \backsim_R I} x^*_J .
\end{aligned}$$

(In right-hand-side of the first inequality, I and J are ordered, so every pair of distinct intersecting intervals appears twice.) The second inequality follows from the fact that if two intervals intersect, one of them necessarily contains the right endpoint of the other. To complete the proof, note that the feasibility constraints ensure that $\sum_{J \backsim_R I} x^*_J \leq 1$ for all intervals I, and thus $\sum_{I \in s} \sum_{J \backsim_R I} x^*_J \leq t$ for all s (as every vertex contains at most t intervals). □

8.3 Background

The algorithm for maximum weight independent set in t-interval graphs (Algorithm **FracLRIS**) is taken from Bar-Yehuda et al. [2002], where it is also shown that the problem is APX-hard for all $t \geq 2$, even for highly restricted instances. (Note that a 1-interval graph is an ordinary interval graph, in which finding a maximum weight independent set is a polynomial-time solvable problem.) Previously, the problem was considered only on proper union graphs [Bafna et al. 1996]—a restricted subclass of t-interval graphs—and the approximation factor achieved was $(2^t - 1 + 1/2^t)$. The approximation results of Bar-Yehuda et al. [2002] can be generalized to $k \geq 2$ machines, in which case the approximation guarantee increases to $2t + 1$.

ACKNOWLEDGMENT

We thank Guy Even for suggestions on the presentation.

REFERENCES

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

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

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

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

Bafna, V., Narayanan, B., and Ravi, R. 1996. Nonoverlapping local alignments (weighted independent sets of axis parallel rectangles). *Discreate Applied Mathematics 71*, 41–53. Special issue on Computational Molecular Biolagy.

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

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

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

Bar-Yehuda. 2001. Using homogeneous weights for approximating the partial cover problem. *Journal of Algorithms 39*, 137–144.

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

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

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

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

Bar-Yehuda, R., Halldórsson, M. M., Naor, J., Shachnai, H., and Shapira, I. 2002. Scheduling split intervals. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms.* 732–741.

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

Bar-Yehuda, R. and Rawitz, D. 2002. Approximating element-weighted vertex deletion problems for the complete k-partite property. *Journal of Algorithms 42,* 1, 20–40.

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

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

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

Bhatia, R., Chuzhoy, J., Freund, A., and Naor, J. 2003. Algorithmic aspects of bandwidth trading. Submitted.

Bshouty, N. H. and Burroughs, L. 1998. 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*. LNCS, vol. 1373. Springer, 298–308.

Cai, M., Deng, X., and Zang, W. 2001. An approximation algorithm for feedback vertex sets in tournaments. *SIAM Journal on Computing 30,* 6, 1993–2007.

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

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

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

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. The MIT Press.

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

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

Even, S. 1979. *Graph Algorithms*. Computer Science Press.

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

Ford, L. R. and Fulkerson, D. R. 1956. Maximal flow through a network. *Canadian Journal of Mathematics 8*, 399–404.

Freund, A. and Rawitz, D. 2002. Approximating certain graph editing problems. Unpublished manuscript.

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

Gabow, H. N., Goemans, M. X., and Williamson, D. P. 1993. An efficient approximation algorithm for the survivable network design problem. In *3rd MPS Conference on Integer Programming and Combinatorial Optimization*.

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

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

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

Goemans, M. X., Goldberg, A., S.Plotkin, Shmoys, D., Tardos, E., and Williamson, D. P. 1994. Improved approximation algorithms for network design problems. In *5th Annual ACM-SIAM Symposium on Discrete Algorithms*. 223–232.

GOEMANS, M. X. AND WILLIAMSON, D. P. 1995. A general approximation technique for constrained forest problems. *SIAM Journal on Computing 24,* 2, 296–317.

GOEMANS, M. X. AND WILLIAMSON, D. P. 1997. The primal-dual method for approximation algorithms and its application to network design problems. See Hochbaum [1997a], Chapter 4, 144–191.

GOLUMBIC, M. C. 1980. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press.

GRÖTSCHEL, M., LOVÁSZ, L., AND SCHRIJVER, A. 1988. Geometric algorithms and combinatorial optimization. Springer-Verlag, Berlin.

GUHA, S., HASSIN, R., KHULLER, S., AND OR, E. 2002. Capacitated vertex covering with applications. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms.* 858–865.

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

HÅSTAD, J. 1997. Some optimal inapproximability results. In *29th Annual ACM Symposium on the Theory of Computing.* 1–10.

HOCHBAUM, D. S. 1982. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on Computing 11,* 3, 555–556.

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

HOCHBAUM, D. S., Ed. 1997a. *Approximation Algorithms for NP-Hard Problem.* PWS Publishing Company.

HOCHBAUM, D. S. 1997b. A framework of half integrality and good approximations. Manuscript, UC berkeley.

HOCHBAUM, D. S. 1998. Approximating clique and biclique problems. *Journal of Algorithms 29,* 1, 174–200.

IMIELIŃSKA, C., KALANTARI, B., AND KHACHIYAN, L. 1993. A greedy heuristic for a minimum-weight forest problem. *Operations Research Letters 14,* 65–71.

JAIN, K. 1998. A factor 2 approximation algorithm for the generalized steiner network problem. In *39th IEEE Symposium on Foundations of Computer Science.* 448–457.

JAIN, K., MAHDIAN, M., AND SABERI, A. 2002. A new greedy approach for facility location problems. In *34th ACM Symposium on the Theory of Computing.* 731–740.

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

JOHNSON, D. S. 1974. Approximation algorithms for combinatorial problems. *J. Comput. System Sci. 9,* 256–278.

KARP, R. M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, Eds. Plenum Press, New York, 85–103.

KEARNS, M. 1990. *The Computational Complexity of Machine Learning.* M.I.T. Press.

KOLEN, A. W. J. AND KROON, L. G. 1991. On the computational complexity of (maximum) class scheduling. *European Journal of Operational Research 54,* 23–38.

KOLEN, A. W. J. AND KROON, L. G. 1994. An analysis of shift class design problems. *European Journal of Operational Research 79,* 417–430.

KRUSKAL, J. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. In *The American Mathematical Society.* Vol. 7. 48–50.

LAPORTE, G. 1988. Location-routing problems. In *Vehicle routing: methods and studies,* B. L. Golden and A. A. Assad, Eds. 163–197.

LAPORTE, G., NOBERT, Y., AND PELLETIER, P. 1983. Hamiltonian location problems. *European Journal of Operation Research 12,* 82–89.

LEWIS, J. M. AND YANNAKAKIS, M. 1980. The node-deletion problem for hereditary problems is NP-complete. *Journal of Computer and System Sciences 20,* 219–230.

LOVÁSZ, L. 1975. On the ratio of optimal integral and fractional covers. *Discreate Mathematics 13,* 383–390.

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

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

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

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

Nicholson, T. T. J. 1966. Finding the shortest route between two points in a network. *Computer Journal 9*, 275–280.

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

Slavík, P. 1997. Improved performance of the greedy algorithm for partial cover. *Information Processing Letters 64,* 5, 251–254.

Spieksma, F. C. R. 1999. On the approximability of an interval scheduling problem. *Journal of Scheduling 2,* 5, 215–227.

West, D. and Shmoys, D. 1984. Recognizing graphs with fixed interval number is NP-complete. *Discrete Applied Mathematics 8*, 295–305.

Williamson, D. P. 2001. The primal dual method for approximation algorithms. *Mathematical Programming (Series B) 91,* 3, 447–478.

Williamson, D. P., Goemans, M. X., Mihail, M., and Vazirani, V. V. 1995. A primal-dual approximation algorithm for generalized Steiner network problems. *Combinatorica 15*, 435–454.

Received ; revised; accepted