Greedy Methods

Samir Khuller* Balaji Raghavachari† Neal E. Young‡

September 9, 2005

1 Introduction

Greedy algorithms can be used to solve many optimization problems exactly and efficiently. Examples include classical problems such as finding minimum spanning trees and scheduling unit length jobs with profits and deadlines. These problems are special cases of finding a maximum- or minimum-weight basis of a *matroid*. This well-studied problem can be solved exactly and efficiently by a simple greedy algorithm [23, 28].

Greedy methods are also useful for designing efficient *approximation* algorithms for intractable (i.e. NP-hard) combinatorial problems. Such algorithms find solutions that may be suboptimal, but still satisfy some performance guarantee. For a minimization problem, an algorithm has *approximation ratio* α, if, for every instance I, the algorithm delivers a solution whose cost is at most $\alpha \times \mathrm{OPT}(I)$, where $\mathrm{OPT}(I)$ is the cost of an optimal solution for instance I. An α-approximation algorithm is a polynomial-time algorithm with an approximation ratio of α.

In this chapter we survey several NP-hard problems that can be approximately solved via greedy algorithms. For a couple of fundamental problems we sketch the proof of the approximation ratio. For most of the other problems that we survey, we give brief descriptions of the algorithms and citations to the articles where these results were reported.

*Department of Computer Science, University of Maryland, College Park, MD 20742. Email:samir@cs.umd.edu

†Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083-0688. Email: rbk@utdallas.edu

‡Department of Computer Science, University of California at Riverside, Riverside, CA 92521. Email: neal@cs.ucr.edu

2 Set Cover

We start with SET COVER, perhaps one of the most elementary of the NP-hard problems. The problem is defined as follows. The input is a set $X = \{x_1, x_2, \ldots, x_n\}$ of elements and a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots S_m\}$ whose union is X. Each set S_i has a weight of $w(S_i)$. A *set cover* is a subset $\mathcal{S}' \subseteq \mathcal{S}$ such that $\bigcup_{S_j \in \mathcal{S}'} S_j = X$. Our goal is to find a set cover $\mathcal{S}' \subseteq \mathcal{S}$ so as to minimize $w(\mathcal{S}') = \sum_{S_i \in \mathcal{S}'} w(S_i)$. In other words, we wish to choose a minimum-weight collection of subsets that covers all the elements.

Intuitively, for a given weight, one prefers to choose a set that covers the most elements. This suggests the following algorithm: start with an empty collection of sets, then repeatedly add sets to the collection, each time adding a set that minimizes the cost per newly covered element (i.e., the set that minimizes the weight of the set divided by the number of its elements that are not yet in any set in the collection.)

2.1 Algorithm for set cover

Next we prove that this algorithm has approximation ratio $H(|S_{\max}|)$, where $S_{\max}$ is the largest set in $\mathcal{S}$ and H is the harmonic function, defined as $H(d) = \sum_{i=1}^{d} 1/i$. For simplicity, assume each set has weight 1.

We use the following charging scheme: *when the algorithm adds a set S to the collection, let u denote the number of not-yet-covered elements in S and charge $1/u$ to each of those elements.* Clearly the weight of the chosen sets is at most the total amount charged. To finish, we observe that the total amount charged is at most $\mathrm{OPT} \times H(|S_{\max}|)$. To see why this is so, let $S^* = \{e_s, e_{s-1}, \ldots, e_1\}$ be any set in OPT. Assume that when the greedy algorithm chooses sets to add to its collection, it covers the elements in S^* in the order given (each e_i is covered by the time e_{i-1} is). When the charge for an element e_i is computed (i.e., when the greedy algorithm chooses a set S containing e_i for the first time) at least i elements $(e_i, e_{i-1}, e_{i-2}, \ldots, e_1)$ in S^* are not yet covered. Since the greedily chosen set S contains at least as many not-yet-covered elements as S^*, the charge to e_i is at most $1/i$. Thus, the total charge to elements in S^* is at most

$$\frac{1}{s} + \frac{1}{s-1} + \cdots + \frac{1}{2} + 1 \;=\; H(s) \;\leq\; H(|S_{\max}|).$$

Thus, the total charge to elements covered by OPT is at most $\mathrm{OPT} \times H(|S_{\max}|)$. Since every element is covered by OPT, this means that the total charge altogether is at most $\mathrm{OPT} \times H(|S_{\max}|)$. This implies that the greedy algorithm is an $H(|S_{\max}|)$-approximation algorithm.

These results were first reported in the mid 1970's [19, 36, 24, 7]. Since then, it has been proven that no polynomial-time approximation algorithm for set cover has a significantly better approximation ratio unless P=NP [33].

The algorithm and approximation ratio extend to a fairly general class of problems called *minimizing a linear function subject to a sub-modular constraint.* This problem generalizes set cover as follows. Instead of asking for a set cover, we ask for a collection of sets C such that some function $f(C) \geq f(X)$. The function $f(C)$ should be increasing as we add sets to C and it should have the following property: if $C \subset C'$, then, for any set S, $f(C' \cup \{S'\}) - f(C') \leq f(C \cup \{S\}) - f(C)$. In terms of the greedy algorithm, this means that adding a set S to the collection now increases f at least as much as adding it later. (For set cover, take $f(C)$ to be the number of elements covered by sets in C.) See [27] for details.

2.2 Shortest superstring problem

We consider an application of the set cover problem, Shortest Superstring problem. Given an alphabet Σ, and a collection of n strings $S = \{s_1, \ldots, s_n\}$, where each s_i is a string from the alphabet Σ, find a shortest string s that contains each s_i as a substring. There are several constant-factor approximation algorithms for this problem [5]; here we simply want to illustrate how to reduce this problem to the set cover problem.

For each $s_i, s_j \in S$ and for each value $0 < k < \min(|s_i|, |s_j|)$, we first check to see if the last k symbols of s_i are identical to the first k symbols of s_j. If so, we define a new string β_{ijk} obtained by concatenating s_i with s_j^k, the string obtained from s_j by deleting the first k characters of s_j. Let $\mathcal{S}$ be the set of strings β_{ijk}. For a string π we define $S(\pi) = \{s \in S | s \text{ is a substring of } \pi\}$. The underlying set of elements of the set cover is S. The specified subsets of S are the sets $S(\pi)$ for each $\pi \in S \cup \mathcal{S}$. The weight of each set $S(\pi)$ is $|\pi|$, the length of the string.

We can now apply the greedy set cover algorithm to find a collection of sets $S(\pi_i)$ and then simply concatenate the strings π_i to find a superstring. The approximation factor of this algorithm can be shown to be $2H(n)$.

3 Primal-Dual Methods

In this section we study a powerful technique, namely the primal-dual method, for designing approximation algorithms [12]. Duality provides a systematic approach for bounding OPT, a key task in proving any approximation ratio. The approach underlies many approximation algorithms. In this section, we illustrate the basic method via a simple example.

A closely related method, one that we don't explore here, is the "local-ratio" method developed by Bar-Yehuda [3]. It seems that most problems that have been solved by the primal-dual method appear amenable to attack by the local-ratio method as well.

We use as our example another fundamental NP-hard problem, the VERTEX COVER problem. Given a graph $G = (V, E)$ with weights on the vertices given by $w(v)$, we wish to find a minimum-weight vertex cover. A vertex cover is a subset of vertices, $S \subseteq V$, such that for each edge $(u, v) \in E$, either $u \in S$ or $v \in S$ or both. This problem is equivalent to the special of the set cover problem where each element appears in exactly two of the given sets.

We describe a 2-approximation algorithm. First, write an integer linear program (IP) for this problem. For each vertex v in the given graph, the program has a binary variable $x_v \in \{0, 1\}$. Over this space of variables, the problem is to find

$$\min \Big\{ \sum_{v \in V} w(v) x_v \ : \ x_u + x_v \geq 1 \ \ (\forall (u, v) \in E) \Big\}.$$

It is easy to see that an optimal solution to this integer program gives an optimal solution to the original vertex cover problem. Thus, the integer program is NP-hard to solve. Instead of solving it directly, we relax IP to a linear program (LP), which is to optimize the same objective function over the same set of constraints, but with real-valued variables $x_v \in [0, 1]$.

Each linear program has a dual. Let $N(v)$ denote the neighbor set of v. The dual of LP has a variable $y_{(u,v)} \geq 0$ for each edge $(u, v) \in E$. Over this space of variables, the dual of LP is to find

$$\max \Big\{ \sum_{(u,v) \in E} y_{u,v} \ : \ \sum_{u \in N(v)} y_{(u,v)} \leq w(v) \ \ (\forall v \in V) \Big\}.$$

The key properties of these programs are the following:

1. Weak duality: the cost of any feasible solution to the dual is a lower bound on the cost of any feasible solution to LP. Consequently, the cost of any feasible solution to the dual is a lower bound on the cost of any feasible solution to IP.

2. If we can find feasible solutions for IP and the dual, where the cost of our solution to IP is at most α times the cost of our solution to the dual, then our solution to IP has cost at most $\alpha\,\mathrm{OPT}$.

One way to get an approximate solution is to solve the vertex cover LP optimally (e.g., using a network flow algorithm [38]), and then round the obtained fractional solution to an integral solution. Here we describe a different algorithm — a greedy algorithm that computes solutions to both IP and the dual. The solutions are not necessarily optimal, but will have costs within a factor of 2.

The dual solution is obtained by the following simple heuristic: *Initialize all dual variables to 0, then simultaneously and uniformly raise all dual variables, except those dual variables that occur in constraints that are currently tight. Stop when all constraints are tight.* The solution to IP is obtained as follows: *Compute the dual solution above. When the constraint for a vertex v becomes tight, add v to the cover.* (Thus, the vertices in the cover are those whose constraints are tight.)

The constraint for vertex v is tight if $\sum_{u\in N(v)} y_{(u,v)} = w(v)$. When we start to raise the dual variables, the sum increases at a rate equal to the degree of the vertex. Thus, the first vertices to be added are those minimizing $\frac{w(v)}{d(v)}$. These vertices and their edges are effectively deleted from the graph, and the process continues.

The algorithm returns a vertex cover because, in the end, for each edge (u,v) at least one of the two vertex constraints is tight. By weak duality, to see that the cost of the cover is at most 2OPT, it suffices to see that the cost of the cover S is at most twice the cost of the dual solution. This is true because each node's weight can be charged to the dual variables corresponding to the incident edges, and each such dual variable is charged at most twice:

$$\sum_{v\in S} w(v) \;=\; \sum_{v\in S}\sum_{u\in N(v)} y_{(u,v)} \;\le\; 2\sum_{(u,v)\in E} y_{(u,v)}.$$

The equality above follows because $w(v) = \sum_{u\in N(v)} y_{(u,v)}$ for each vertex added to the cover. The inequality follows because each dual variable $y_{(u,v)}$ occurs at most twice in the sum.

To implement the algorithm, it suffices to keep track of the current degree $D(v)$ of each vertex v, as well as the slack $W(v)$ remaining in the constraint for v. In fact, with a little bit of effort the reader can see that the following pseudo-code implements the algorithm described above, without explicitly keeping track of dual variables. This algorithm was first described by Clarkson [8]:

Greedy-Vertex-Cover(G, S)

for all $v \in V$ **do** $W(v) \leftarrow w(v); D(v) \leftarrow deg(v)$
$S \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
    Find $v \in V$ for which $\frac{W(v)}{D(v)}$ is minimized.
    for all $u \in N(v)$ **do**
        $E \leftarrow E \setminus (u, v)$
        $W(u) \leftarrow W(u) - \frac{W(v)}{D(v)}$ and $D(u) \leftarrow D(u) - 1$
    end
    $S \leftarrow S \cup \{v\}$ and $V \leftarrow V \setminus \{v\}$
end

More sophisticated applications of the primal-dual method require more sophisticated proofs. In some cases, the algorithm starts with a greedy phase, but then has a final round in which some previously added elements are discarded. The key idea is to develop the primal solution hand in hand with the dual solution in a way that allows the cost of the primal solution to be "charged" to the cost of the dual.

Because the vertex cover problem is a special case of the set cover problem, it is also possible to solve the problem using the greedy set cover algorithm. This gives an approximation ratio of at most $H(|V|)$, and in fact there are vertex cover instances for which that greedy algorithm produces a solution of cost $\Omega(H(|V|))\,\mathrm{OPT}$. The greedy algorithm described above is almost the same; it differs only in that it modifies the weights of the neighbors of the chosen vertices as it proceeds. This slight modification yields a significantly better approximation ratio.

4 Greedy Algorithms via the Probabilistic Method

In their book on the probalistic method, Alon, Spencer, and Erdős [1] describe probabilistic proofs as follows:

> In order to prove the existence of a combinatorial structure with certain properties, we construct an appropriate probability space and show that a randomly chosen element in the space has the desired properties with positive probability.

The *method of conditional probabilities* is used to convert those proofs into efficient algorithms [30].

For some problems, elementary probabilistic arguments easily prove that good solutions exist. In some cases (especially when the proofs are based on iterated random sampling) the probabilistic proof can be converted into a greedy algorithm. This is a fairly general approach for designing greedy algorithms. In this section we give some examples.

4.1 Max cut

Given a graph $G = (V, E)$, the MAX-CUT problem is to partition the vertices into two sets S and $\overline{S}$ so as to maximize the number of edges "cut" (crossing between the two sets). The problem is NP-hard.

Consider the following randomized algorithm: *For each vertex, choose the vertex to be in S or $\overline{S}$ independently with probability 1/2.* We claim this is a (1/2)-approximation algorithm, in expectation. To see why, note that the probability that any given edge is cut is $1/2$. Thus, by linearity of expectation, in expectation $|E|/2$ edges are cut. Clearly the optimal solution cuts at most twice this many edges.

Next we apply the *method of conditional probabilities* [30, 1] to convert this randomized algorithm into a deterministic one. We replace each random choice made by the algorithm by a deterministic choice that does "as well" in a precise sense. Specifically, we modify the algorithm to to maintain the following invariant:

> *After each step, if we were to take the remaining choices randomly, then the expected number of edges cut in the end would be at least $|E|/2$.*

Suppose decisions have been made for vertices $V_t = \{v_1, v_2, \ldots, v_t\}$, but not yet for vertex v_{t+1}. Let S_t denote the vertices in V_t chosen to be in S. Let $\overline{S}_t = V_t - S_t$ denote the vertices in V_t chosen to be in $\overline{S}$. Given these decisions, the status of each edge in $V_t \times V_t$ is known, while the rest still have a $1/2$ probability of being cut. Let $x_t = |E \cap (S_t \times \overline{S}_t)|$ denote the number of those edges that will definitely cross the cut. Let $e_t = |E - V_t \times V_t|$ denote the number of edges which are not yet determined. Then, given the decisions made so far, the expected number of edges that would be cut if all remaining choices were to be taken randomly would be

$$\phi_t \doteq x_t + e_t/2.$$

The x_t term counts the edges cut so far, while the $e_t/2$ term counts the e_t edges with at least one undecided endpoint: each of those edges will be cut with probability 1/2.

Our goal is to replace the random decisions for the vertices with deterministic decisions that guarantee $\phi_{t+1} \geq \phi_t$ at each step. If we can do this, then we will have $|E|/2 = \phi_0 \leq \phi_1 \leq \cdots \leq \phi_n$, and, since ϕ_n is the number of edges finally cut, this will ensure that at least $|E|/2$ edges are cut.

Consider deciding whether the vertex v_{t+1} goes into S_{t+1} or $\overline{S}_{t+1}$. Let s be the number of v_{t+1}'s neighbors in S_t. Let $\overline{s}$ be the number of v_{t+1}'s neighbors in $\overline{S}_{t+1}$. By calculation,

$$\phi_{t+1} - \phi_t = \begin{cases} s/2 - \overline{s}/2 & \text{if } v_{t+1} \text{ is added to } \overline{S}_{t+1} \\ \overline{s}/2 - s/2 & \text{otherwise.} \end{cases}$$

Thus, the following strategy ensures $\phi_{t+1} \geq \phi_t$: *if $s \leq \overline{s}$, then put v_{t+1} in S_{t+1}; otherwise put v_t in $\overline{S}_{t+1}$.* By doing this at each step, the algorithm guarantees that $\phi_n \geq \phi_{n-1} \geq \cdots \geq |E|/2$.

We have derived the following greedy algorithm: *Start with $S = \overline{S} = \emptyset$. Consider the vertices in turn. For each vertex v, put the vertex v in S or $\overline{S}$, whichever has fewer of v's neighbors.* We know from the derivation that this is a 1/2-approximation algorithm.

4.2 Independent set

Although the application of the method of conditional probabilities is somewhat technical, it is routine, in the sense that it follows a similar form in every case. Here is another example.

The problem of finding a MAXIMUM INDEPENDENT SET in a graph $G = (V, E)$ is one of the most basic problems in graph theory. An independent set is defined as a subset S of vertices such that there are no edges between any pair of vertices in S. The problem is NP-hard. Turan's theorem states the following: *Any graph G with n nodes and average degree d has an independent set I of size at least $n/(d+1)$.* Next we sketch a classic proof of the theorem using the probabilistic method. Then we apply the method of conditional probabilities to derive a greedy algorithm.

Let $\widehat{N}(v) = N(v) \cup \{v\}$ denote the neighbor set of v, including v. Consider this randomized algorithm: *Start with $I = \emptyset$. Consider the vertices in random order. When considering v, add it to I if $\widehat{N}(v) \cap I = \emptyset$.*

For a vertex v to be added to I, it suffices for v to be considered before any of its neighbors. This happens with probability $|\widehat{N}(v)|^{-1}$. Thus, by linearity of expectation, the expected number of vertices added to I is at least

$$\sum_v |\widehat{N}(v)|^{-1}.$$

A standard convexity argument shows this is at least $n/(d+1)$, completing the proof of Turan's theorem.

Now we apply the method of conditional probabilities. Suppose the first t vertices $V_t = \{v_1, v_2, \ldots, v_t\}$ have been considered. Let $I_t = V_t \cap I$ denote those that have been added to I. Let $R_t = V \setminus (V_t \cup \widehat{N}(I_t))$ denote the remaining vertices that might still be added to I and let $\widehat{N}_t(v) = \widehat{N}(v) \cap R_t$ denote the neighbors of v that might still be added. If the remaining vertices were to be chosen in random order, the expected number of vertices in I by the end would be at least

$$\phi_t \doteq |I_t| + \sum_{v \in R_t} |\widehat{N}_t(v)|^{-1}.$$

We want the algorithm to choose vertex v_{t+1} to ensure $\phi_{t+1} \geq \phi_t$. To do this, it suffices to choose the vertex $w \in R_t$ minimizing $|\widehat{N}_t(w)|$, for then

$$\phi_{t+1} - \phi_t \geq 1 - \sum_{v \in \widehat{N}_t(w)} |\widehat{N}_t(v)|^{-1} \geq 1 - \sum_{v \in \widehat{N}_t(w)} |\widehat{N}_t(w)|^{-1} = 0.$$

This gives us the following greedy algorithm: *Start with $I = \emptyset$. Repeat until no vertices remain: choose a vertex v of minimum degree in the remaining graph; add v to I and delete v and all of its neighbors from the graph. Finally, return I.* It follows from the derivation that this algorithm ensures $n/(d+1) \leq \phi_0 \leq \phi_1 \leq \cdots \leq \phi_n$, so that the algorithm returns an independent set of size at least $n/(d+1)$, where d is the average degree of the graph.

As an exercise, the reader can give a different derivation leading to the following greedy algorithm (with the same performance guarantee): *Order the vertices by increasing degree, breaking ties arbitrarily. Let I consist of those vertices that precede all their neighbors in the ordering.*

4.3 Unweighted set cover

Next we illustrate the method on the set cover problem.

We start with a randomized rounding scheme that uses iterated random sampling to round a fractional set cover (a solution to the relaxed problem) to a true set cover. We prove an approximation ratio for the randomized algorithm, then apply the method of conditional probabilities to derive a deterministic greedy algorithm.

We emphasize that, in applying the method of conditional probabilities, we remove the explicit dependence of the algorithm on the fractional set solution. Thus, the final algorithm does not in fact require first solving the relaxed problem.

Recall the definition of the set cover problem from the beginning of the chapter. For this section, we will assume all weights $w(S_i)$ are 1.

Consider the following relaxation of the problem: assign a value $z_i \in [0,1]$ to each set S_i so as to minimize $\sum_i z_i$ subject to the constraint that, for every element x_j, $\sum_{i:x_j \in S_i} z_i \geq 1$. We call a z meeting these constraints a *fractional set cover.*

The optimal set cover gives one possible solution to the relaxed problem, but there may be other fractional set covers that give a smaller objective function value. However, not too much smaller. We claim the following: *Let z be any fractional set cover. Then there exists an actual set cover C of size at most $T = \lceil \ln(n)|z| \rceil$, where $|z| = \sum_i z_i$.*

To prove this, consider the following randomized algorithm: given z, *draw T sets at random from the distribution p defined by $p(S_i) = z_i/|z|$.* With non-zero probability, this random experiment yields a set cover. Here is why. A calculation shows that, with each draw, the chance that any given element e is covered is at least $1/|z|$. Thus, the expected number of elements left uncovered after T draws is at most

$$n(1 - 1/|z|)^T \ < \ n\exp(-T/|z|) \ \leq \ 1.$$

Since on average less than 1 element is left uncovered, it must be that some outcome of the random experiment covers all elements.

Next we apply the method of conditional probabilities. Suppose that t sets have been chosen so far, and let n_t denote the number of elements not yet covered. Then the conditional expectation of the number of elements left uncovered at the end is at most

$$\phi_t \doteq n_t(1 - 1/|z|)^{T-t}.$$

We want the algorithm to choose each set to ensure $\phi_t \le \phi_{t-1}$, so that in the end $\phi_T \le \phi_0 < 1$ and the chosen sets form a cover.

Suppose the first t sets have been chosen, so that ϕ_t is known. A calculation shows that, if the next set is chosen at random according to the distribution p, then $E[\phi_{t+1}] \le \phi_t$. Thus, choosing the next set to *minimize* ϕ_{t+1} will ensure $\phi_{t+1} \le \phi_t$. By inspection, choosing the set to minimize ϕ_{t+1} is the same as choosing the set to minimize n_{t+1}.

We have derived the following greedy algorithm: *Repeat T times: add a set to the collection so as to minimize the number of elements remaining uncovered.* In fact, it suffices to do the following: *Repeat until all elements are covered: add a set to the collection so as to minimize the number of elements remaining uncovered.* (This suffices because we know from the derivation that a cover will be found within T rounds.)

We have proven the following fact: *The above greedy algorithm returns a cover of size at most* $\min_z \lceil \ln(n)|z| \rceil$, *where z ranges over all fractional set covers.* Since the minimum-size set cover OPT corresponds to a z with $|z| = |\mathrm{OPT}|$, we have the following corollary: *The above greedy algorithm returns a cover of size at most* $\lceil \ln(n)\mathrm{OPT} \rceil$.

This algorithm can be generalized to weighted set cover, and slightly stronger performance guarantees can be shown [19, 36, 24, 7]. This particular greedy approach applies to a general class of problems called "minimizing a linear function subject to a submodular constraint" [27].

Comment: In many cases, applying the method of conditional probabilities will not yield a greedy algorithm, because the conditional expectation ϕ_t will depend on the fractional solution in a non-trivial way. In that case, the derandomized algorithm will first have to compute the fractional solution (typically by solving a linear program). That is Raghavan and Thompson's standard method of *randomized rounding* [31]. The variant we see here was first observed in [39]. Roughly, to get a greedy algorithm, we should apply the method of conditional probabilities to a probabilistic proof based on *repeated random sampling* from the distribution defined by the fractional optimum.

4.4 Lagrangian relaxation for fractional set cover

The algorithms described above fall naturally into a larger and technically more complicated class of algorithms called *Lagrangian relaxation algorithms.* Typically, such an algorithm is used to find a structure meeting a given a set of constraints. The algorithm constructs a solution in small steps. Each step is made so as to minimize (or keep from increasing) a *penalty function* that approximates some of the underlying constraints. Finally, the algorithm returns a solution that approximately meets the underlying constraints.

These algorithms typically have a greedy outer loop. In each iteration, they solve a subproblem that is simpler than the original problem. For example, a multicommodity flow algorithm may solve a sequence of shortest-path subproblems, routing small amounts of flow along paths chosen to minimize the sum of edge penalties that grow exponentially with the current flow on the edge.

Historical examples include algorithms by von Neumann, Ford and Fulkerson, Dantzig-Wolfe decomposition, Benders' decomposition, and Held and Karp. In 1990, Shahrokhi and Matula proved a polynomial time bound for such an algorithm for multicommodity flow. This sparked a long line of work generalizing and strengthening this result (e.g. [29, 14, 40]). See the recent text by Bienstock [4]. These works focus mainly on *packing and covering problems* — linear programs and integer linear programs with non-negative coefficients.

As a rule, the problems in question can also be solved by standard linear programming algorithms such as the simplex algorithm, the ellipsoid algorithm, or interior-point algorithms. The primary motivation for studying Lagrangian relaxation algorithms has been that, like other greedy algorithms, they can often be implemented without explicitly constructing the full underlying problem. This can make them substantially faster.

As an example, here is a Lagrangian relaxation algorithm for fractional set cover (given an instance of the set cover problem, find a fractional set cover z of minimum size $|z| = \sum_i z_i$; see the previous subsection for definitions). Given a set cover instance and $\varepsilon \in [0, 1/2]$, the algorithm returns a fractional set cover of size at most $1 + O(\varepsilon)$ times the optimum:

1. Let $N = 2\ln(n)/\varepsilon^2$, where n is the number of elements.
2. Repeat until all elements are sufficiently covered ($\min_j c(j) \geq N$):
3. Choose a set S_i maximizing $\sum_{x_j \in S_i} (1-\varepsilon)^{c(j)}$, where $c(j)$ denotes the number of times any set containing element x_j has been chosen so far.
4. Return z, where z_i is the number of times S_i was chosen divided by N.

The naive implementation of this algorithm runs in $O(nM\log(n)/\varepsilon^2)$ time, where $M = \sum_i |S_i|$ is the size of the input. With appropriate modifications, the algorithm can be implemented to run in $O(M\log(n)/\varepsilon^2)$ time.

For readers who are interested, we sketch how this algorithm may be derived using the probabilistic framework. To begin, we imagine that we have in hand any fractional set cover z^*, to which we apply the following randomized algorithm: *Define probability distribution p on the sets by $p(S_i) = z_i^*/|z^*|$. Draw sets randomly according to p until every element has been covered (in a drawn set) at least $N = 2\ln(n)/\varepsilon^2$ times. Return z, where z_i is the number of times set S_i was drawn, divided by N.* (The reader should keep in mind that the dependence on z^* will be removed when we apply the method of conditional probabilities.)

Claim: *With non-zero probability the algorithm returns a fractional set cover of size at most* $(1+O(\varepsilon))|z^*|$. Next we prove the claim. Let $T = |z^*|N/(1-\varepsilon)$. We will prove that, with non-zero probability, within T draws each set will be covered at least N times. This will prove the claim because then the size of z is at most $T/N = |z^*|/(1-\varepsilon)$.

Fix a given element x_j. With each draw, the chance that x_j is covered is at least $1/|z^*|$. Thus, the expected number of times x_j is covered in T draws is at least $T/|z^*| = N/(1-\varepsilon)$. By a standard Chernoff bound, the probability that x_j is covered less than N times in T rounds is at most $\exp(-\varepsilon^2 N/2(1-\varepsilon)) < 1/n$.

By linearity of expectation, the expected number of elements that are covered less than N times in T rounds is less than 1. Thus, with non-zero probability, all elements are covered at least N times in T rounds. This proves the claim. Next we apply the method of conditional probabilities to derive a greedy algorithm.

Let X_{jt} be an indicator variable for the event that x_j is covered in round t, so that for any j the X_{jt}'s are independent with $E[X_{jt}] \geq 1/|z^*|$. Let $\mu = N/(1-\varepsilon)$. The proof of the Chernoff bound bounds

$\Pr[\sum_t X_{jt} \le (1-\varepsilon)\mu]$ by the expectation of the following quantity:

$$\frac{(1-\varepsilon)^{\sum_t X_{jt}}}{(1-\varepsilon)^{(1-\varepsilon)\mu}} = \frac{(1-\varepsilon)^{\sum_t X_{jt}}}{(1-\varepsilon)^N}.$$

Thus, the proof of our claim above implicitly bounds the probability of failure by the expectation of

$$\phi = \sum_j \frac{(1-\varepsilon)^{\sum_t X_{jt}}}{(1-\varepsilon)^N}.$$

Furthermore, the proof shows that the expectation of this quantity is less than 1.

To apply the method of conditional probabilities, we will *choose each set to keep the conditional expectation of the above quantity ϕ below 1.*

After the first t sets have been drawn, the random variables X_{js} for $s \le t$ are determined, while X_{js} for $s > t$ are not yet determined. Using the inequalities from the proof of the Chernoff bound, the conditional expectation of ϕ given the choices for the first t sets is at most

$$\phi_t \doteq \sum_j \frac{\prod_{s\le t}(1-\varepsilon)^{X_{js}} \times \prod_{s>t}(1-\varepsilon/|z^*|)}{(1-\varepsilon)^N}.$$

This quantity is initially less than 1, so it suffices to *choose each set to ensure* $\phi_{t+1} \le \phi_t$. If the $t+1$'st set is chosen randomly according to p, then $E[\phi_{t+1}] \le \phi_t$. Thus, to ensure $\phi_{t+1} \le \phi_t$, it suffices to choose the set to minimize ϕ_{t+1}. By a straightforward calculation, this is the same as choosing the set S_i to maximize $\sum_{x_j \in S_i}(1-\varepsilon)^{\sum_{s\le t} X_{jt}}$. This gives us the algorithm in question (at the top of this section). From the derivation, we know the following fact: *The algorithm above returns a fractional set cover of size at most $(1+O(\varepsilon))\min_{z^*}|z^*|$, where z^* ranges over all the fractional set covers.*

5 Steiner Trees

The STEINER TREE problem is defined as follows. Given an edge-weighted graph $G = (V, E)$ and a set of terminals $S \subset V$, find a minimum-weight tree that includes all the nodes in S. (When $S = V$, then this is the problem of finding a minimum-weight *spanning* tree. There are several very fast greedy algorithms that can be used to solve this problem optimally.) The Steiner tree problem is NP-hard and several greedy algorithms have been designed that give a factor 2 approximation [22, 37]. We briefly describe the idea behind one of the methods. Let $T_1 = \{s_1\}$ (an arbitrarily chosen terminal from S). At each step T_{i+1} is computed from T_i as follows: attach the vertex from $S - T_i$ that is the "closest" to T_i by a path to T_i and call the newly added special vertex s_{i+1}. Thus T_i always contains the vertices $s_1, s_2, \ldots, s_i$. It is clear that the solution produces a Steiner tree. It is possible to prove that the weight of this tree is at most twice the weight of an optimal Steiner tree.

Zelikovsky [41] developed a greedy algorithm with an approximation ratio of $11/6$. This bound has been further improved subsequently, but by using more complex methods.

A generalization of Steiner trees called NODE-WEIGHTED STEINER TREES is defined as follows. Given a node weighted graph $G = (V, E)$ and a set of terminals $S \subset V$, find a minimum-weight tree that includes all the nodes in S. Here, the weight of a tree is the sum of the weights of its nodes. It can be shown that this problem is at least as hard as the Set-Cover problem to approximate [20]. Interestingly, this problem is solved via a greedy algorithm similar to the one for the Set Cover problem with costs. We define a "spider" as a tree on ℓ terminals where there is at most one vertex with degree more than 2. Each leaf in the tree corresponds to a terminal. The weight of the spider is simply the weight of the nodes in the spider. The algorithm at each step greedily picks a spider with minimum ratio of weight to number of terminals in it. It collapses all the terminals spanned by the spider into a single vertex, makes this new vertex a terminal and repeats until one one terminal remains. The approximation guarantee of this algorithm is $2 \ln |S|$. Further improvements appear in [16]. For more on the Steiner tree problem, see the book by Hwang et al. [18].

6 K-Centers

The K-CENTER problem is a fundamental facility location problem and is defined as follows: given an edge-weighted graph $G = (V, E)$ find a subset $S \subseteq V$ of size at most K such that each vertex in V is close to some vertex in S. More formally, the objective function is defined as follows:

$$\min_{S \subseteq V} \max_{u \in V} \min_{v \in S} d(u, v)$$

where d is the distance function. For example, one may wish to install K fire stations and minimize the maximum distance (response time) from a location to its closest fire station.

Gonzalez [13] describes a very simple greedy algorithm for the basic K-center problem and proves that it gives an approximation factor of 2. The algorithm works as follows. Initially pick any node v_0 as a center and add it to the set C. Then for $i = 1$ to K do the following: in iteration i, for every node $v \in V$, compute its distance $d^i(v, C) = \min_{c \in C} d(v, c)$ to the set C. Let v_i be a node that is farthest away from C, i.e., a node for which $d^i(v_i, C) = \max_{v \in V} d(v, C)$. Add v_i to C. Return the nodes $v_0, v_1, \ldots, v_{K-1}$ as the solution.

The above greedy algorithm is a 2-approximation for the K-center problem. First note that the radius of our solution is $d^K(v_K, C)$, since by definition v_K is the node that is farthest away from our set of centers. Now consider the set of nodes $v_0, v_1, \ldots, v_K$. Since this set has cardinality $K + 1$, at least two of these nodes, say v_i and v_j, must be covered by the same center c in the optimal solution. Assume without loss of generality that $i < j$. Let R^* denote the radius of the optimal solution. Observe that the distance from each node to the set C does not increase as the algorithm progresses. Therefore $d^K(v_K, C) \leq d^j(v_K, C)$. Also we must have $d^j(v_K, C) \leq d^j(v_j, C)$ otherwise we would not have selected node v_j in iteration j. Therefore

$$d(c, v_i) + d(c, v_j) \geq d(v_i, v_j) \geq d^j(v_j, C) \geq d^K(v_K, C)$$

by the triangle inequality and the fact that v_i is in the set C at iteration j. But since $d(c, v_i)$ and $d(c, v_j)$ are both at most R^*, we have the radius of our solution $= d^K(v_K, C) \leq 2R^*$.

7 Connected Dominating Sets

The connected dominating set (CDS) problem is defined as follows. Given a graph $G = (V, E)$, find a minimum size subset S of vertices, such that the subgraph induced by S is connected and S forms a dominating set in G. This problem is known to be NP-hard. Recall that a dominating set is one in which each vertex is either in the dominating set, or adjacent to some vertex in the dominating set.

We describe a greedy algorithm for this problem [15]. The algorithm runs in two phases. At the start of the first phase all nodes are colored white. Each time we include a vertex in the dominating set, we color it black. Nodes that are dominated are colored gray (once they are adjacent to a black node). In the first phase the algorithm picks a node at each step and colors it black, coloring all adjacent white nodes gray. A *piece* is defined as a white node or a black connected component. *At each step we pick a node to color black that gives the maximum (non-zero) reduction in the number of pieces.*

It is easy to show that at the end of this phase if no vertex gives a non-zero reduction to the number of pieces, then there are no white nodes left.

In the second phase, we have a collection of black connected components that we need to connect. Recursively connect pairs of black components by choosing a chain of vertices, until there is one black connected component. Our final solution is the set of black vertices that form the connected component.

Key Property: At the end of the first phase if there is more than one black component, then there is always a pair of black components that can be connected by choosing a chain of two vertices.

It can be shown that the connected dominating set found by the algorithm is of size at most $(\ln \Delta + 3) \cdot |OPT_{CDS}|$, where Δ is the maximum degree of a node.

Let a_i be the number of pieces left after the i^{th} iteration, and $a_0 = n$. Since a node can connect up to Δ pieces, $|OPT_{CDS}| \geq \frac{a_0}{\Delta}$. (This is true if the optimal solution has at least two nodes.) Consider the $i+1^{st}$ iteration. The optimal solution can connect a_i pieces. Hence the greedy procedure is guaranteed to pick a node which connects at least $\lceil \frac{a_i}{|OPT_{CDS}|} \rceil$ pieces. Thus the number of pieces will reduce by at least $\left\lceil \frac{a_i}{|OPT_{CDS}|} \right\rceil - 1$. This gives us the recurrence relation,

$$a_{i+1} \leq a_i - \left\lceil \frac{a_i}{|OPT_{CDS}|} \right\rceil + 1 \leq a_i(1 - \frac{1}{|OPT_{CDS}|}) + 1 \ .$$

Its solution is,

$$a_{i+1} \leq a_0(1 - \frac{1}{|OPT_{CDS}|})^i + \sum_{j=0}^{i-1}(1 - \frac{1}{|OPT_{CDS}|})^j \ .$$

Notice after $|OPT_{CDS}| \cdot \ln \frac{a_0}{|OPT_{CDS}|}$ iterations, the number of pieces left is less than $2 \cdot |OPT_{CDS}|$. After this, for each node we choose, we will decrease the number of pieces by at least one until the number of black components is at most $|OPT_{CDS}|$, thus at most $|OPT_{CDS}|$ more vertices are picked. So after $|OPT_{CDS}| \cdot \ln \frac{a_0}{|OPT_{CDS}|} + |OPT_{CDS}|$ iterations at most $|OPT_{CDS}|$ pieces are left to connect. We connect the remaining pieces choosing chains of at most two vertices in the second phase. The total number of nodes chosen is at most $|OPT_{CDS}| \cdot \ln \frac{a_0}{|OPT_{CDS}|} + |OPT_{CDS}| + 2|OPT_{CDS}|$, and since $\Delta \geq \frac{a_0}{|OPT_{CDS}|}$, the solution found has at most $|OPT_{CDS}| \cdot (\ln \Delta + 3)$ nodes.

8 Scheduling

We consider the following simple scheduling problem [2]. There are k identical machines. We are given a collection of n jobs. Job J_i is specified by the following vector: (r_i, d_i, p_i, w_i). The job has a *release* time of r_i, a *deadline* of d_i and a *processing time* of p_i. The weight of the job is w_i. Our goal is to schedule a subset of the jobs such that each job starts after its release time and is completed by its deadline. If S is a subset of jobs that are scheduled, then the total profit due to set S is $\sum_{J_i \in S} w_i$. We do not get any profit if the job is not completed by its deadline. Our objective is to find a maximum-profit subset of jobs that can be scheduled on the k machines. The jobs are scheduled on one machine, with no pre-emption. In other words if job J_i starts on machine j at time s_i, then $r_i \leq s_i$ and $s_i + p + i \leq d_i$. Moreover, each machine can be executing at most one job at any point of time.

A number of algorithms for the problem are based on LP rounding [2]. A special case of interest is when all jobs have unit weight (or identical weight). In this case, we simply wish to maximize the number of scheduled jobs. The following greedy algorithm has the property that it schedules a set of jobs such that the total number of scheduled jobs is at least ρ_k times the number of jobs in an optimal schedule. Here $\rho_k = 1 - \frac{1}{(1+\frac{1}{k})^k}$. Observe that when $k = 1$, then $\rho_k = \frac{1}{2}$ and this bound is tight for the greedy algorithm.

The algorithm considers each machine in turn and finds a maximal set of jobs to schedule for the machine; it removes these jobs from the collection of remaining jobs, then recurses on the remaining set of jobs. Now we discuss how a maximal set of jobs is chosen for a single machine. The idea is to pick a job that can be finished as quickly as possible. After we pick this job, we schedule it, starting it at the earliest possible time. Making this choice might force us to reject several other jobs. We then consider starting a job after the end of the last scheduled job, and again pick one that we can finish at the earliest possible time. In this way, we construct the schedule for a single machine.

9 Minimum-degree spanning trees (MDST)

In this problem, the input is a graph $G = (V, E)$, with non-negative weights $w : E \mapsto R^+$ on its edges. We are also given an integer $d > 1$. The objective of the problem is to find a minimum-weight spanning tree of

G in which the degreee of every node is at most d. It is a generalization of the Hamiltonian path problem, and is therefore NP-hard. It is known that the problem is not approximable to any ratio unless P=NP or the approximation algorithm is allowed to output a tree whose degree is larger than d. Approximation algorithms try to find a tree whose degree is as close to d as possible, but whose weight is not much more than an optimal degree-d tree.

Greedy algorithms usually select one edge at a time, and once an edge is chosen, that decision is never revoked and the edge is part of the output. Here we add a subset S of edges at a time (e.g. a spanning forest) where S is chosen to minimize a relaxed version of the objective function. We get an iterative solution and the output is a union of the edges selected in each of the steps. This approach typically provides a logarithmic approximation. For MDST, the algorithm finds a tree of degree $O(d \log n)$, whose weight is within $O(\log n)$ of an optimal degree-d tree, where the graph has n vertices. The ideas have appeared in [11, 32]. Such algorithms in which two objectives (degree and weight) are approximated are called *bicriteria* approximation algorithms.

A minimum-weight subgraph in which each node has degree at most d and at least 1 can be computed using algorithms for matching. Except for possibly being disconnected, this subgraph satisfies the other properties of an MDST: degree constraints and weight at most OPT. A greedy algorithm for MDST works by repeatedly finding d-forests, where each d-forest is chosen to connect the connected components left from the previous stages. The number of components decreases by a constant factor in each stage, and, in $O(\log n)$ stages, we get a tree of degree at most $d \log n$.

10 Maximum-weight b-matchings

In this problem, we are interested in computing a maximum-weight subgraph of a given graph G in which each node has degree at most b. The classical matching problem is a b-matching with $b = 1$. This problem can be solved optimally in polynomial time, but the algorithms take about $O(n^3)$ time. We discuss a 1/2-approximation algorithm that runs in $O(bE + E \log E)$ time. The edges are sorted by weight, with the heaviest edges considered first. Start with an empty forest as the initial solution. When an edge is considered, we see if adding it to the solution violates the degree bound of its end vertices. If not, we add it to our solution. Intuitively, each edge of our solution can displace at most 2 edges of an optimal solution, one incident to each of its end vertices, but of smaller weight.

11 Conclusions

In this chapter we surveyed a collection of problems and described simple greedy algorithms for several of these problems. In several cases, the greedy algorithms described do not represent the state of the art for these problems. The reader is referred to other chapters in this handbook to read in more detail about the specific problems and the techniques that yield the best worst-case approximation guarantees. In many instances, the performance of greedy algorithms may be better than their worst-case bounds suggest. This and their simplicity make them important in practice.

For some problems (e.g. set cover) it is known that a greedy algorithm gives the best possible approximation ratio unless $NP \subset \mathrm{DTIME}(n^{\log \log n})$. But for some problems no such intractability results are yet known. In these cases, instead of proving hardness of approximation for all polynomial-time algorithms, one may try something easier: to prove that no *greedy* algorithm gives a good approximation. Of course this requires a formal definition of the class of algorithms. (A similar approach has been fruitful in competitive analysis of online algorithms.) Such a formal study of greedy algorithms with an eye towards lower bound results has been the subject of several recent papers [6].

For additional information on Combinatorial Optimization, the reader is referred to books by Papadimitriou and Steiglitz [28], Cook et al. [9], and a series of three books by Schrijver [34]. For more on approximation algorithms, there is a book by Vazirani [38], lecture notes by Motwani [25], and a book edited by Hochbaum [17]. There is a chapter on greedy algorithms in several textbooks, such as Kleinberg and Tardos [21], and Cormen et al. [10]. More on randomized algorithms can be found in a book by Motwani and Raghavan [26], and a survey by Shmoys [35].

References

[1] Noga Alon, Joel H. Spencer, and Paul Erdős. *The probabilistic method.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester-New York-Brisbane-Toronto-Singapore, 1992.

[2] Amotz Bar-Noy, Sudipto Guha, Joseph Naor, and Baruch Schieber. Approximating the throughput of multiple machines in real-time scheduling. *SIAM J. Comput.*, 31(2):331–352, 2001.

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

[4] Daniel Bienstock. *Potential Function Methods for Approximately Solving Linear Programming Problems: Theory and Practice.* Kluwer Academic Publishers, Boston, MA, 2002.

[5] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, 1994.

[6] A. Borodin, M. Nielsen, and C. Rackoff. (Incremental) priority algorithms. *Algorithmica*, 37:295–326, 2003.

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

[8] K. Clarkson. A modification of the greedy algorithm for vertex cover. *Information Processing Letters*, 16:23–25, 1983.

[9] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization.* John Wiley, New York, 1997.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[11] Martin Fürer and Balaji Raghavachari. An NC approximation algorithm for the minimum-degree spanning tree problem. In *Proc 28th Ann Allerton Conference on Communication, Control and Computing*, pages 274–281, 1990.

[12] Michel X. Goemans and David P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995.

[13] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.*, 38:293–306, 1985.

[14] M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. Technical Report DCS-TR-273, Rutgers University Computer Science Department, New Brunswick, NJ, 1991.

[15] Sudipto Guha and Samir Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998.

[16] Sudipto Guha and Samir Khuller. Improved methods for approximating node weighted Steiner trees and connected dominating sets. *Inf. Comput.*, 150(1):57–74, 1999.

[17] Dorit Hochbaum, editor. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, 1997.

[18] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem.* Number 53 in Annals of Discrete Mathematics. Elsevier Science Publishers B. V., Amsterdam, 1992.

[19] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[20] Philip N. Klein and R. Ravi. A nearly best-possible approximation algorithm for node-weighted Steiner trees. *J. Algorithms*, 19(1):104–115, 1995.

[21] Jon Kleinberg and Éva Tardos. *Algorithm Design.* Addison-Wesley, 2005.

[22] G. Markowsky L. Kou and L. Berman. An fast algorithm for Steiner trees. *Acta. Informatica*, 15:141–145, 1981.

[23] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Wilson, 1976.

[24] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[25] R. Motwani. Lecture notes on approximation algorithms. Technical report, Stanford University, 1992.

[26] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1997.

[27] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, New York, 1988.

[28] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization.* Prentice-Hall, 1982.

[29] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20(2):257–301, 1995.

[30] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms approximating packing integer programs. *Journal of Computer and System Sciences*, 37(2):130–143, October 1988.

[31] Prabhakar Raghavan and C. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.

[32] R. Ravi, Madhav V. Marathe, S. S. Ravi, Daniel J. Rosenkrantz, and Harry B. Hunt III. Approximation algorithms for degree-constrained minimum-cost network-design problems. *Algorithmica*, 31(1):58–78, 2001.

[33] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *STOC*, pages 475–484, 1997.

[34] Alexander Schrijver. *Combinatorial Optimization — Polyhedra and Efficiency, Volume A: Paths, Flows, Matchings, Volume B: Matroids, Trees, Stable Sets, Volume C: Disjoint Paths, Hypergraphs*, volume 24

of *Algorithms and Combinatorics*. Springer-Verlag, Berlin-Heidelberg-New York-Hong Kong-London-Milan-Paris-Tokyo, 2003.

[35] David B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems, 1995.

[36] S. K. Stein. Two combinatorial covering theorems. *J. Comb. Theory A*, 16:391–397, 1974.

[37] H. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Japonica*, 24(6):573–577, 1980.

[38] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[39] Neal E. Young. Randomized rounding without solving the linear program. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 170–178, San Francisco, California, 22–24 January 1995.

[40] Neal E. Young. Sequential and parallel algorithms for mixed packing and covering. In *IEEE Symposium on Foundations of Computer Science*, pages 538–546, 2001.

[41] Alexander Zelikovsky. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica*, 9(5):463–470, 1993.