Sequential and Parallel Algorithms for Mixed Packing and Covering

Neal E. Young
Akamai Technologies
Cambridge, Massachusetts, USA.
neal@acm.org

Abstract

We describe sequential and parallel algorithms that approximately solve linear programs with no negative coefficients (a.k.a. mixed packing and covering problems*).*

For explicitly given problems, our fastest sequential algorithm returns a solution satisfying all constraints within a $1 \pm \epsilon$ factor in $O(md\log(m)/\epsilon^2)$ time, where m is the number of constraints and d is the maximum number of constraints any variable appears in.

Our parallel algorithm runs in time polylogarithmic in the input size times ϵ^{-4} and uses a total number of operations comparable to the sequential algorithm.

The main contribution is that the algorithms solve mixed *packing and covering problems (in contrast to pure packing or pure covering problems, which have only "$\le$" or only "$\ge$" inequalities, but not both) and run in time independent of the so-called* width *of the problem.*

1. Background

Packing and covering problems are problems that can be formulated as linear programs using only non-negative coefficients and non-negative variables. Special cases include pure packing problems, which are of the form $\max\{c \cdot x : Ax \le b\}$ and pure covering problems, which are of the form $\min\{c \cdot x : Ax \ge b\}$.

Lagrangian-relaxation algorithms are based on the following basic idea. Given an optimization problem specified as a collection of constraints, modify the problem by selecting some of the constraints and replacing them by a continuous "penalty" function that, given a partial solution $\mathbf{x}$, measures how close $\mathbf{x}$ is to violating the removed constraints. Construct a solution iteratively in small steps, making each choice to maintain the remaining constraints while minimizing the increase in the penalty function.

While Lagrangian-relaxation algorithms have the disadvantage of producing only approximately optimal (or approximately feasible) solutions, the algorithms have the following potential advantages in comparison to the simplex, interior point, and ellipsoid methods. They can be faster, easier to implement, and easier to parallelize. They can be particularly useful for problems that are sparse, or that have exponentially many variables or constraints (but still have some polynomial-size representation).

Lagrangian relaxation was one of the first methods proposed for solving linear programs — as early as the 1950's, John von Neumann apparently proposed and analyzed an $O(m^2 n \log(mn)/\epsilon^2)$-time Lagrangian-relaxation algorithm for solving two-person zero-sum matrix games (equivalent to pure packing or covering) [18]. The algorithm returned a solution with *additive* error ϵ assuming the matrix was scaled to lie between 0 and 1. In 1950, Brown and von Neumann also proposed a system of differential equations that converged to an optimal solution, with the suggestion that the equations could form the basis of an algorithm [3].

Subsequent examples include a multicommodity flow algorithm by Ford and Fulkerson (1958), Dantzig-Wolfe decomposition (1960), Benders' decomposition (1962), and Held and Karp's lower bound for the traveling salesman problem (1971). In 1990, Shahrokhi and Matula proved polynomial-time convergence rates for a Lagrangian-relaxation algorithm for multicommodity flow. This caught the attention of the theoretical computer science research community, which has since produced a large body of research on the subject. Klein et al. [15] and Leighton et al. [19] (and many others) gave additional multicommodity flow results. Plotkin, Shmoys, and Tardos [21] and Grigoriadis and Khachiyan [10, 11, 8, 9] adapted the techniques to the general class of packing/covering problems, including mixed packing and covering problems. These algorithms' running times depended linearly on the *width* — an unbounded function of the input instance. Relatively complicated techniques were developed to transform problems so as to reduce their width.

From this body of work we adapt and use the following specific techniques: the technique of *variable-size increments* (Garg and Konemann [7, 17]); a way of partitioning the steps of the Garg/Konemann algorithm into *phases* (Fleischer [5]); and the idea of *incrementing multiple variables simultaneously* (Luby and Nisan [20]).

Variable-sized increments yield algorithms whose running times are independent of the width of the problem instance, effectively replacing the width by the number of constraints. Partitioning into phases reduces the time to implement each step. Finally, incrementing multiple variables simultaneously allows fast parallel algorithms.

Previously, as far as we know, these techniques have only been applied to pure packing or covering problems, not to mixed packing and covering problems. We know of no other width-independent or parallel algorithms for *mixed* packing and covering. Our contribution here is to present such algorithms.

Although Luby and Nisan characterize their algorithm as solving "linear programs with non-negative coefficients", in fact it applies only to pure packing or pure covering problems [20].

After presenting and analyzing the algorithms, we conclude with two illustrative examples.

2. Mixed Packing and Covering

We consider problems in the following form:
APPROXIMATE MIXED PACKING AND COVERING: Given non-negative matrices $\mathbf{P}, \mathbf{C}$, vectors $\mathbf{p}, \mathbf{c}$ and $\epsilon \in (0,1)$, find an *approximately feasible* vector $\mathbf{x} \geq \mathbf{0}$ (s.t. $\mathbf{Px} \leq (1+\epsilon)\mathbf{p}$ and $\mathbf{Cx} \geq \mathbf{c}$) or a proof that no vector $\mathbf{x}$ is feasible (i.e., satisfies $\mathbf{x} \geq \mathbf{0}$, $\mathbf{Px} \leq \mathbf{p}$ and $\mathbf{Cx} \geq \mathbf{c}$).

In Section 5 we describe how to reduce the optimization version $\min\{\lambda : \mathbf{Px} \leq \lambda\,\mathbf{p}, \mathbf{Cx} \geq \mathbf{c}\}$ to the above form.

For notational simplicity, we assume throughout that each coordinate of $\mathbf{p}$ and $\mathbf{c}$ is some constant N. This is without loss of generality by the reduction $\mathbf{P}'_{ij} = \mathbf{P}_{ij}N/\mathbf{p}_i$ and $\mathbf{C}'_{ij} = \mathbf{C}_{ij}N/\mathbf{c}_i$ (after removing any constraints of the form $(\mathbf{Px})_i \leq 0$ — which only force to zero each $\mathbf{x}_j$ such that $\mathbf{P}_{ij} > 0$ — or the form $(\mathbf{Cx})_i \geq 0$ — which do not constrain $\mathbf{x}$ at all). A vector $\mathbf{x}$ is *feasible* if $\max \mathbf{Px} \leq N \leq \min \mathbf{Cx}$.

All of the algorithms in this paper are specializations of the generic algorithm in Fig. 1. The algorithm starts with an infeasible vector $\mathbf{x}$ and adds to $\mathbf{x}$ in small increments until $\mathbf{x}$ becomes approximately feasible, that is, until $\min \mathbf{Cx} \geq N$ and $\max \mathbf{Px} \leq (1+O(\epsilon))N$. Instead of working with the max and min functions, the algorithm works to achieve a stronger condition: $\mathsf{lmin}\,\mathbf{Cx} \geq N$ and $\mathsf{lmax}\,\mathbf{Px} \leq (1+O(\epsilon))N$, where lmax and lmin are "smooth" functions that approximate max and min:

Definition 1 *For real values $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_m)$, define*

$$\mathsf{lmax}\ \mathbf{y} = \ln \textstyle\sum_i e^{\mathbf{y}_i} \quad \textit{and} \quad \mathsf{lmin}\ \mathbf{y} = -\ln \textstyle\sum_i e^{-\mathbf{y}_i}.$$

Note $\max \mathbf{y} \leq \mathsf{lmax}\,\mathbf{y}$ *and* $\min \mathbf{y} \geq \mathsf{lmin}\,\mathbf{y}$.

Recall that, for any continuous function $f(\mathbf{x})$, increasing $\mathbf{x}_j$ by δ increases f by approximately δ times the partial derivative of f with respect to $\mathbf{x}_j$. In lines 2 and 3 of the algorithm, $\mathsf{partial}_j(\mathbf{P}, \mathbf{x})$ and $\mathsf{partial}_j(\mathbf{C}, -\mathbf{x})$ are, respectively, the partial derivatives of $\mathsf{lmax}(\mathbf{Px})$ and $\mathsf{lmin}(\mathbf{Cx})$ with respect to $\mathbf{x}_j$. Thus, the condition in line 7b says that a variable $\mathbf{x}_j$ may be increased *only if doing so increases* $\mathsf{lmax}(\mathbf{Px})$ *by at most* $1+O(\epsilon)$ *times as much as it increases* $\mathsf{lmin}(\mathbf{Cx})$. We say "$1+O(\epsilon)$" instead of $1+\epsilon$ because the partial derivatives only approximate the actual increases, and in fact the condition on line 7c is necessary to ensure that the change in $\mathbf{x}$ is small enough so that the partial derivatives do give good approximations.

Because each step increases $\mathsf{lmax}(\mathbf{Px})$ by at most $1+O(\epsilon)$ times as much as it increases $\mathsf{lmin}(\mathbf{Cx})$, the ratio of the two quantities tends to $1+O(\epsilon)$ (or less). Thus, the algorithm drives $\mathbf{x}$ to approximate feasibility.

Line 5 ensures that there is a j meeting the condition of line 7b. Why is line 5 okay? Briefly, because the gradients of lmax and lmin have 1-norm equal to 1 (that is, the sum of their partial derivatives equals 1), one can show that, for *any* $\mathbf{x}$ and any feasible $\mathbf{x}^*$, the dot product of $\mathbf{x}^*$ with the gradient of $\mathsf{lmax}(\mathbf{Px})$ is at most N, while the dot product of $\mathbf{x}^*$ with the gradient of $\mathsf{lmin}(\mathbf{Cx})$ is at least N. Since $\mathbf{x}^* \geq 0$, this means at least one partial derivative of $\mathsf{lmin}(\mathbf{Cx})$ is as large as the corresponding partial derivative of $\mathsf{lmax}(\mathbf{Px})$.

In the remainder of this section we give the complete analysis of the performance guarantee.

Let $\mathsf{partial}'_i(\mathbf{y}) = e^{\mathbf{y}_i} / \sum_i e^{\mathbf{y}_i}$, so that $\mathsf{partial}'_i(\mathbf{y})$ is the partial derivative of $\mathsf{lmax}(\mathbf{y})$ with respect to $\mathbf{y}_i$ and $\mathsf{partial}'_i(-\mathbf{y})$ is the partial derivative of $\mathsf{lmin}(\mathbf{y})$ with respect to $\mathbf{y}_i$. We will use the following "chain rule": for any $\mathbf{M}, \mathbf{x}, \boldsymbol{\alpha}$,

$$\textstyle\sum_i (\mathbf{M}\boldsymbol{\alpha})_i\, \mathsf{partial}'_i(\mathbf{Mx}) = \sum_j \boldsymbol{\alpha}_j\, \mathsf{partial}_j(\mathbf{M}, \mathbf{x}) \qquad (1)$$

We start with a utility lemma.

Lemma 1 (smoothness of lmin and lmax) *For all $\mathbf{y}, \boldsymbol{\beta} \geq 0$, if $0 \leq \boldsymbol{\beta}_i \leq \epsilon \leq 1$ then*

$$\mathsf{lmax}(\mathbf{y}+\boldsymbol{\beta}) \leq \mathsf{lmax}(\mathbf{y}) + (1+\epsilon)\textstyle\sum_i \boldsymbol{\beta}_i\, \mathsf{partial}'_i(\mathbf{y})$$

and

$$\mathsf{lmin}(\mathbf{y}+\boldsymbol{\beta}) \geq \mathsf{lmin}(\mathbf{y}) + (1-\epsilon/2)\textstyle\sum_i \boldsymbol{\beta}_i\, \mathsf{partial}'_i(-\mathbf{y}).$$

Proof: Using the standard sorts of inequalities that underlie Chernoff bounds, namely $\ln(1+z) \leq z$ (for all z) and $e^{\beta} - 1 \leq (1+\epsilon)\beta$ (for $0 \leq \beta \leq \epsilon \leq 1$):

$$\begin{aligned}
&\mathsf{lmax}(\mathbf{y}+\boldsymbol{\beta}) - \mathsf{lmax}(\mathbf{y}) \\
&\quad = \ln[\textstyle\sum_i e^{\mathbf{y}_i+\boldsymbol{\beta}_i} / \sum_i e^{\mathbf{y}_i}] \\
&\quad = \ln[1 + \textstyle\sum_i (e^{\boldsymbol{\beta}_i} - 1)e^{\mathbf{y}_i} / \sum_i e^{\mathbf{y}_i}] \\
&\quad \leq \textstyle\sum_i (e^{\boldsymbol{\beta}_i} - 1)e^{\mathbf{y}_i} / \sum_i e^{\mathbf{y}_i} \\
&\quad \leq (1+\epsilon)\textstyle\sum_i \boldsymbol{\beta}_i\, \mathsf{partial}'_i(\mathbf{y}).
\end{aligned}$$

in: $\mathbf{P}, \mathbf{C}, \epsilon$, arbitrary $\mathbf{x}$
out: 'infeasible' or $\mathbf{x}$ s.t. $\mathbf{Px} \leq (1+O(\epsilon))N, \mathbf{Cx} \geq N$.
1. Let $N \leftarrow (\max \mathbf{Px} + 2\ln m)/\epsilon$, where m is the number of constraints.
2. Define $\mathsf{partial}_j(\mathbf{M}, \mathbf{x}) = \sum_i \mathbf{M}_{ij} e^{(\mathbf{Mx})_i} / \sum_i e^{(\mathbf{Mx})_i}$. *—partial derivative of* lmax *and* lmin
3. Define $\mathsf{ratio}_j(\mathbf{x}) = \mathsf{partial}_j(\mathbf{P}, \mathbf{x}) / \mathsf{partial}_j(\mathbf{C}, -\mathbf{x})$.
4. While $\min \mathbf{Cx} < N$ do:
5. If $\min_j \mathsf{ratio}_j(\mathbf{x}) > 1$ then return 'infeasible'. *—see Lemma 3*
6. Delete the ith row of $\mathbf{C}$ for each i s.t. $(\mathbf{Cx})_i \geq N$. *—constraint deletion for efficiency*
7a. Choose "increment" vector $\boldsymbol{\alpha} \geq 0$ such that
7b. $(\forall j)\ \boldsymbol{\alpha}_j > 0$ only if $\mathsf{ratio}_j(\mathbf{x}) \leq 1+\epsilon$ — $\partial\,\mathsf{lmax}/\partial \mathbf{x}_j \leq (1+\epsilon)\partial\,\mathsf{lmin}/\partial \mathbf{x}_j$
7c. and $\max\{\max \mathbf{C}\boldsymbol{\alpha}, \max \mathbf{P}\boldsymbol{\alpha}\} = \epsilon$. *—step size*
8. Let $\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\alpha}$. *—do the increment*
9. Return $\mathbf{x}$.

Figure 1. Generic algorithm. Implementable in $O(m\log(m)/\epsilon^2)$ linear-time iterations.

This proves the first inequality in the statement of the lemma. The second inequality follows by an analogous chain of inequalities, using $1 - e^{-\beta} \geq (1-\epsilon/2)\beta$ (for $0 \leq \beta \leq \epsilon \leq 1$). □

Because of this smoothness, if the increment is small enough (i.e. the"step-size" condition on line 7c of the algorithm is met) the partial derivatives approximate the changes in lmin and lmax well — within a $1 \pm \epsilon$ factor:

Lemma 2 *In each increment, the increase of* $\mathsf{lmax}\,\mathbf{Px}$ *is at most* $\frac{(1+\epsilon)^2}{1-\epsilon/2}$ *times that of* $\mathsf{lmin}\,\mathbf{Cx}$.

Proof: When the generic algorithm increments $\mathbf{x}$ by $\boldsymbol{\alpha}$, the vector $\boldsymbol{\alpha}$ meets the following conditions:

1. $(\forall j)\ \boldsymbol{\alpha}_j > 0 \rightarrow \mathsf{ratio}_j(\mathbf{x}) \leq 1+\epsilon$;
2. $\max\{\max \mathbf{C}\boldsymbol{\alpha}, \max \mathbf{P}\boldsymbol{\alpha}\} \leq \epsilon$.

Adding $\boldsymbol{\alpha}$ to $\mathbf{x}$ adds $\mathbf{P}\boldsymbol{\alpha}$ to $\mathbf{Px}$. From Condition 2 above and Lemma 1, it follows that $\mathsf{lmax}\,\mathbf{Px}$ increases by at most $(1+\epsilon)\sum_i (\mathbf{P}\boldsymbol{\alpha})_i\, \mathsf{partial}'_i(\mathbf{Px})$. By the chain rule (1), this equals $(1+\epsilon)\sum_j \boldsymbol{\alpha}_j\, \mathsf{partial}_j(\mathbf{P}, \mathbf{x})$.

Similarly, $\mathsf{lmin}\,\mathbf{Cx}$ increases by at least $(1-\epsilon/2)\sum_j \boldsymbol{\alpha}_j\, \mathsf{partial}_j(\mathbf{C}, -\mathbf{x})$. Since $\boldsymbol{\alpha}_j > 0$ only if $\mathsf{partial}_j(\mathbf{P}, \mathbf{x}) \leq (1+\epsilon)\,\mathsf{partial}_j(\mathbf{C}, -\mathbf{x})$ (Condition 1 above), Lemma 2 follows. □

Next we show that if the problem instance is feasible there always exists a choice of $\boldsymbol{\alpha}$ meeting the conditions on lines 7b and 7c of the algorithm. This is necessary for the algorithm to be well-defined.

Lemma 3 *If the problem instance is feasible, then* $\forall \mathbf{x}\, \exists j :$ $\mathsf{ratio}_j(\mathbf{x}) \leq 1$.

Proof: Let $\mathbf{x}$ be arbitrary and let $\mathbf{x}^*$ be a feasible solution. By the chain rule (1),

$$\textstyle\sum_j \mathbf{x}^*_j\, \mathsf{partial}_j(\mathbf{P}, \mathbf{x}) \;=\; \sum_i (\mathbf{Px}^*)_i\, \mathsf{partial}'_i(\mathbf{Px}).$$

Since $(\mathbf{Px}^*)_i \leq N$, and $\sum_i \mathsf{partial}'_i(\mathbf{Px}) = 1$, the quantity above is at most N.

Likewise, $\sum_j \mathbf{x}^*_j\, \mathsf{partial}_j(\mathbf{C}, -\mathbf{x}) \geq N$.

Since $\mathbf{x}^* \geq 0$, there must be some j such that $\mathsf{partial}_j(\mathbf{C}, -\mathbf{x}) \geq \mathsf{partial}_j(\mathbf{P}, \mathbf{x})$. □

Lemma 2 means that the condition on line 7b can be met, and clearly by scaling the condition on line 7c can also be met.

From Lemma 2, the basic performance guarantee follows easily.

Lemma 4 *If the problem instance is feasible, the generic algorithm returns an approximately feasible solution. Given an initial* $\mathbf{x}$, *the algorithm makes* $O(m(\max \mathbf{Px} + \log m)/\epsilon^2)$ *increments.*

Proof: First we prove the performance guarantee. Define $\Phi = \mathsf{lmax}\,\mathbf{Px} - \frac{(1+\epsilon)^2}{1-\epsilon/2}\,\mathsf{lmin}\,\mathbf{Cx}$.

Before the first increment $\Phi \leq \ln(m\, e^{\max \mathbf{Px}}) + (1+O(\epsilon))\ln m < O(N\epsilon)$. By Lemma 2, no increment operation increases Φ. Deleting a covering constraint increases $\mathsf{lmin}\,\mathbf{Cx}$ and therefore only decreases Φ. Thus, $\Phi \leq O(\epsilon N)$ throughout the course of the algorithm and, just before the last increment (when $\mathsf{lmin}\,\mathbf{Cx} \leq \min \mathbf{Cx} < N$),

$$\begin{aligned} \mathsf{lmax}\,\mathbf{Px} &\leq O(\epsilon N) + (1+O(\epsilon))\,\mathsf{lmin}\,\mathbf{Cx} \\ &\leq (1+O(\epsilon))N. \end{aligned}$$

With the last increment, $\max \mathbf{Px}$ increases by at most ϵ, so at termination, $\max \mathbf{Px} \leq (1+O(\epsilon))N$ while $\min \mathbf{Cx} \geq N$.

in: $\mathbf{P}, \mathbf{C}, \epsilon$, arbitrary $\mathbf{x}$
out: 'infeasible' or $\mathbf{x}$ s.t. $\mathbf{Px} \le (1+O(\epsilon))N, \mathbf{Cx} \ge N$.
1. Let $N \leftarrow (\max \mathbf{Px} + 2\ln m)/\epsilon$, where m is the number of constraints.
2. Define $\mathsf{local}_j(\mathbf{x}) = \sum_i \mathbf{P}_{ij} e^{(\mathbf{Px})_i} / \sum_i \mathbf{C}_{ij} e^{-(\mathbf{Cx})_i}$. *—terms of* $\mathsf{ratio}_j(\mathbf{x})$ *that depend on* j
3. Define $\mathsf{global}(\mathbf{x}) = \sum_i e^{(\mathbf{Px})_i} / \sum_i e^{-(\mathbf{Cx})_i}$. *—so* $\mathsf{ratio}_j(\mathbf{x}) = \mathsf{local}_j(\mathbf{x}) / \mathsf{global}(\mathbf{x})$
4. While $\min \mathbf{Cx} < N$ do:
5a. If g is not yet set or $\min_j \mathsf{local}_j(\mathbf{x})/g > 1+\epsilon$ then
5b. let $g \leftarrow \mathsf{global}(\mathbf{x})$, and *—start new phase*
5c. if $\min_j \mathsf{local}_j(\mathbf{x})/g > 1$ then return 'infeasible'.
6. Delete the ith row of $\mathbf{C}$ for each i s.t. $(\mathbf{Cx})_i \ge N$.
7a. Choose "increment" vector $\boldsymbol{\alpha} \ge 0$ such that
7b. $(\forall j)$ $\boldsymbol{\alpha}_j > 0$ only if $\mathsf{local}_j(\mathbf{x})/g \le 1+\epsilon$ *—stronger cond'n than in generic alg.*
7c. and $\max\{\max \mathbf{C}\boldsymbol{\alpha}, \max \mathbf{P}\boldsymbol{\alpha}\} = \epsilon$.
8. Let $\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\alpha}$.
9. Return $\mathbf{x}$.

Figure 2. Algorithm with phases. Implementable in $O(md\log(m)/\epsilon^2)$ operations, where d is the maximum number of constraints any variable appears in.

Next we bound the number of increments. Let m_c and m_p be the number of rows of $\mathbf{C}$ and $\mathbf{P}$, respectively, so that $m_c + m_p = m$. Define $\Psi = \sum_i (\mathbf{Px})_i + \sum_i((\mathbf{Cx})_i - N - \epsilon)$. It is initially at least $-m_c(N+\epsilon)$, and finally at most $m_p(N+\epsilon)$. By the "step-size" condition in line 7c, each increment increases Ψ by at least ϵ. Because of the constraint-deletion operations in line 6, $(\mathbf{Cx})_i < N$ before each increment and so $(\mathbf{Cx})_i < N + \epsilon$ after each increment (for each row i remaining in $\mathbf{C}$). Thus, each constraint deletion increases Ψ. Thus, the number of increments is at most $m(N+\epsilon)/\epsilon$, which gives the desired bound by the definition of N. □

To specify a particular implementation of the algorithm, we need to specify how the initial $\mathbf{x}$ is chosen and how the increment $\boldsymbol{\alpha}$ is chosen in each iteration. Here is one straightforward implementation: initialize $\mathbf{x}$ to $\mathbf{0}$, and with each increment choose $\boldsymbol{\alpha}$ to be a vector where $\boldsymbol{\alpha}_j = 0$ for all j except for a single j such that $\mathsf{partial}_j(\mathbf{P}, \mathbf{x}) \le (1+\epsilon)\,\mathsf{partial}_j(\mathbf{C}, -\mathbf{x})$. The value of that $\boldsymbol{\alpha}_j$ is determined by the step-size condition. This still leaves some flexibility. For example, one can choose j so as to minimize $\mathsf{partial}_j(\mathbf{P}, \mathbf{x}) - \mathsf{partial}_j(\mathbf{C}, -\mathbf{x})$ or to minimize (within $1+\epsilon$) $\mathsf{partial}_j(\mathbf{P}, \mathbf{x}) / \mathsf{partial}_j(\mathbf{C}, -\mathbf{x})$. Clearly, if the problem instance is given explicitly, then either of these choices can be implemented in time linear in the number of non-zero entries in the matrices. This gives the following corollary:

Corollary 1 *The generic algorithm can implemented to approximately solve any explicitly given problem instance in $O(m\log(m)/\epsilon^2)$ linear-time iterations, where m is the number of constraints.*

3. Algorithm with Phases

This algorithm specializes the generic algorithm. In order to speed the computation of the key function $\mathsf{ratio}_j(\mathbf{x})$, we break it into two components as $\mathsf{ratio}_j(\mathbf{x}) = \mathsf{local}_j(\mathbf{x}) / \mathsf{global}(\mathbf{x})$, where local_j captures the terms that depend on j. As $\mathbf{x}$ changes, we recompute $\mathsf{global}(\mathbf{x})$ only occasionally — at the start of each phase (i.e., iteration of the outer loop). The algorithm is shown in Fig. 2. First we discuss how this is a particular implementation of the generic algorithm, and then we prove a stronger time bound.

Lemma 5 *The algorithm with phases is a specialization of the generic algorithm.*

Proof: We argue that any increment the algorithm does is also an allowable increment for the generic algorithm. Since $\mathsf{local}_j(\mathbf{x}) / \mathsf{global}(\mathbf{x}) = \mathsf{ratio}_j(\mathbf{x})$ and $\mathsf{local}_j(\mathbf{x})$ and $\mathsf{global}(\mathbf{x})$ only increase as the algorithm proceeds, at all times $\mathsf{local}_j(\mathbf{x})/g \ge \mathsf{ratio}_j(\mathbf{x})$. Thus, any $\boldsymbol{\alpha}$ meeting the conditions of the algorithm with phases also meets the conditions of the generic algorithm. □

This and Lemma 4 imply the performance guarantee:

Corollary 2 *The algorithm with phases returns an approximately feasible solution. Given an initial $\mathbf{x}$, the algorithm makes at most $O(m(\max \mathbf{Px} + \log m)/\epsilon^2)$ increments.*

Now here is the stronger time bound:

in: $\mathbf{P}, \mathbf{C}, \epsilon$
out: 'infeasible' or $\mathbf{x}$ s.t. $\mathbf{Px} \le (1 + O(\epsilon))N, \mathbf{Cx} \ge N$.
0. Let $\mathbf{x}_j = \min_i 1/(n\mathbf{P}_{ij})$ for each j, where n is the # of var's. —$\mathbf{x}$ *initialized, not given*
1. Let $N \leftarrow (\max \mathbf{Px} + 2 \ln m)/\epsilon$, where m is the # of constraints.
2. Define $\mathsf{local}_j(\mathbf{x}) = \sum_i \mathbf{P}_{ij} e^{(\mathbf{Px})_i} / \sum_i \mathbf{C}_{ij} e^{-(\mathbf{Cx})_i}$.
3. Define $\mathsf{global}(\mathbf{x}) = \sum_i e^{(\mathbf{Px})_i} / \sum_i e^{-(\mathbf{Cx})_i}$.
4. While $\min \mathbf{Cx} < N$ do:
5a. If g is not yet set or $\min_j \mathsf{local}_j(\mathbf{x})/g > 1 + \epsilon$ then
5b. let $g \leftarrow \mathsf{global}(\mathbf{x})$, and
5c. if $\min_j \mathsf{local}_j(\mathbf{x})/g > 1$ then return 'infeasible'.
6. Delete the ith row of $\mathbf{C}$ for each i s.t. $(\mathbf{Cx})_i \ge N$.
7a. Choose "increment" vector $\boldsymbol{\alpha} \ge 0$ such that for some $\delta > 0$ — *incr.* all *allowed* $\mathbf{x}_j$*'s,*
7b. $(\forall j)$ $\boldsymbol{\alpha}_j = \mathbf{x}_j/\delta$ if $\mathsf{local}_j(\mathbf{x})/g \le 1 + \epsilon$, else $\boldsymbol{\alpha}_j = 0$, — *prop. to current value*
7c. and $\max\{\max \mathbf{C}\boldsymbol{\alpha}, \max \mathbf{P}\boldsymbol{\alpha}\} = \epsilon$.
8. Let $\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\alpha}$.
9. Return $\mathbf{x}$.

Figure 3. Parallel algorithm. Implementable in parallel time polylogarithmic in input size times ϵ^{-4}.

Lemma 6 *Given an initial* $\mathbf{x}$*, the algorithm with phases uses* $O((\max \mathbf{Px} + \log m)/\epsilon^2)$ *phases.*

Proof: We claim that $\mathsf{global}(\mathbf{x})$ increases by at least a $1+\epsilon$ factor each phase. By inspection, $\mathsf{global}(\mathbf{x})$ is initially at least $1/m$ and finally at most $me^{O(N)}/e^{-O(N)}$, so the result will follow.

To see the claim, note that at the end of a phase, $\mathsf{local}_j(\mathbf{x})/g > 1 + \epsilon$ for all j, i.e. $g < \min_j \mathsf{local}_j(\mathbf{x})/(1 + \epsilon)$. But, by Lemma 3, at the start of the next phase, the same $\mathbf{x}$ and the next g satisfy $\mathsf{local}_j(\mathbf{x})/g = \mathsf{ratio}_j(\mathbf{x}) \le 1$ for some j, i.e. $g \ge \min_j \mathsf{local}_j(\mathbf{x})$. □

Consider the following "round-robin" implementation of the algorithm. Start with $\mathbf{x} = \mathbf{0}$. Implement each phase by cycling through the indices j once. For each j, as long as $\mathsf{local}_j(\mathbf{x})/g \le 1 + \epsilon$, repeatedly increment $\mathbf{x}$ by the vector $\boldsymbol{\alpha}$ that has all coordinates 0 except $\boldsymbol{\alpha}_j$, whose value is determined by the step size.

Maintain the values $(\mathbf{Px})_i$ and $(\mathbf{Cx})_i$ for every i. After a variable $\mathbf{x}_j$ is incremented, the only values that change are those where $\mathbf{P}_{ij}$ or $\mathbf{C}_{ij}$ are non-zero. So maintaining these values requires $O(d)$ time, where $d \le m$ is the maximum number constraints any variable appears in.

With these values in hand, the condition $\mathsf{local}_j(\mathbf{x})/g \le 1 + \epsilon$ can be checked in $O(d)$ time. Since the number of increments is $O(m \log(m)/\epsilon^2)$, the total time to do increments is $O(md \log(m)/\epsilon^2)$. Other than increments, each of the $O(\log(m)/\epsilon^2)$ phases requires $O(md)$ time, so we have the following corollary:

Corollary 3 *The algorithm with phases can be implemented to approximately solve any explicitly given problem instance using* $O(md \log(m)/\epsilon^2)$ *operations, where d is the maximum number of constraints any variable appears in.*

Note that this is an improvement on Corollary 1 by a factor equal to the number of non-zero entries in the matrix, divided by d (this factor can be as large as the number of variables).

4. Parallel Algorithm

Next we further specialize the algorithm to achieve an efficient parallel implementation. The algorithm is shown in Fig. 3. The idea is that we start with each variable having a small but positive value. Then, in each increment step, we increase *all* allowed variables (line 7b), each proportionally to its current value. This method allows us to give a polylogarithmic bound on the number of iterations per phase.

Lemma 7 *The parallel algorithm is a specialization of the algorithm with phases, starting with* $\max \mathbf{Px} \le 1$. *The parallel algorithm makes* $O(\log(m) \log(n \log(m)/\epsilon)/\epsilon^2)$ *increments per phase.*

Proof: The first claim is true by inspection and the initial choice of $\mathbf{x}$.

It remains to bound the number of increments per phase. First, we claim that in each increment $\delta = \Omega(N/\epsilon)$. This is simply because by the choice of $\boldsymbol{\alpha}$, for some i, $(\mathbf{Cx})_i/\delta$ or $(\mathbf{Px})_i/\delta$ is at least ϵ, but $(\mathbf{Cx})_i$ and $(\mathbf{Px})_i$ are $O(N)$ throughout the algorithm. Thus, for each j, each increment that increases $\mathbf{x}_j$ increases it by at least a $1 + \Omega(\epsilon/N)$ factor. Since $\mathbf{x}_j$ is initially $\min_i 1/n\mathbf{P}_{ij}$ and finally at most

$\min_i N/\mathbf{P}_{ij}$, it follows that at most $O(N\log(Nn)/\epsilon)$ increments increase $\mathbf{x}_j$.

Finally, in each phase, the last increment of the phase increases some $\mathbf{x}_j$. In fact *each* increment in the phase must have increased that $\mathbf{x}_j$, since $\mathsf{local}_j(\mathbf{x})$ only increased during the phase. Thus, the number of increments in the phase is $O(N\log(Nn)/\epsilon)$. □

Since each increment can be implemented in parallel in polylogarithmic time, and the number of increments is bounded by the number of phases times the number of increments per phase. We have the following corollary.

Corollary 4 *The parallel algorithm can be implemented to approximately solve any explicitly given problem instance in parallel time polylogarithmic in the input size times* $1/\epsilon^4$.

5. Reducing Optimization to Feasibility

Given a problem instance $\mathbf{P}, \mathbf{p}, \mathbf{C}, \mathbf{c}$ and $\epsilon > 0$, let $\lambda^* = \min\{\lambda : (\exists \mathbf{x})\ \mathbf{P}\mathbf{x} \leq \lambda\mathbf{p}, \mathbf{C}\mathbf{x} \geq \mathbf{c}\}$. In this section we describe how to use the algorithms in this paper to approximately solve this optimization problem — that is, to compute a feasible solution $(\lambda, \mathbf{x})$ such that $\lambda^* \leq \lambda \leq (1+\epsilon)\lambda^*$.

We reduce the optimization problem to a sequence of approximate feasibility subproblems. Each subproblem requires ϵ'-approximately solving $\exists ?\mathbf{x} : \mathbf{P}\mathbf{x} \leq \lambda' \mathbf{p}, \mathbf{C}\mathbf{x} \geq \mathbf{c}$ for a particular ϵ' and λ'. We can solve such a subproblem using any of the algorithms in this paper.

Lemma 8 *The approximate optimization problem reduces to a sequence of approximate feasibility subproblems:* $O(\log\log m)$ *subproblems with* $\epsilon' = 1/2$ *and* $O(\log 1/\epsilon)$ *subproblems where the* i*th-to-last subproblem has* $\epsilon' = \Omega(\epsilon(\frac{4}{3})^i)$.

Note that if we solve the subproblems using any algorithm whose time depends at least linearly on $1/\epsilon$ (such as the ones in this paper), then the total time to solve the second set of subproblems (those with $\epsilon' < 1/2$) dominated by the time used to solve the last such subproblem.

The remainder of this section contains the proof. The basic idea is to use binary search for λ^*, solving a feasibility problem at each step to bound λ^*. This lemma gives starting upper and lower bounds:

Lemma 9 *Let* $\lambda = \sum_i \min_j \sum_{i'} (\mathbf{P}_{i'j}/\mathbf{p}_{i'})/(\mathbf{C}_{ij}/\mathbf{c}_i)$. *Then* $\lambda^* \leq \lambda \leq m^2\lambda^*$.

Proof: Recall $\lambda^* = \min\{\lambda : (\exists \mathbf{x})\ \mathbf{P}\mathbf{x} \leq \lambda\mathbf{p}, \mathbf{C}\mathbf{x} \geq \mathbf{c}\}$. For each $i = 1, \ldots, m_c$, consider the following relaxation:

$$\lambda_i^* = \min\{\lambda : (\exists \mathbf{x})\ \textstyle\sum_{i'} (\mathbf{P}\mathbf{x})_{i'}/\mathbf{p}_{i'} \leq m_p\lambda, (\mathbf{C}\mathbf{x})_i \geq \mathbf{c}_i\}.$$

That is, only *one* specified covering constraint, and the *sum* of the packing constraints, need to hold. Clearly $\lambda_i^* \leq \lambda^*$. Furthermore the optimal solution $\mathbf{z}(i)$ to the ith relaxed problem is given by finding j that minimizes $\sum_{i'} (\mathbf{P}_{ij'}/\mathbf{p}_{i'})/(\mathbf{C}_{ij}/\mathbf{c}_i)$ and setting all coordinates of $\mathbf{z}(i)$ to zero except $\mathbf{z}_j(i) = 1/(\mathbf{C}_{ij}/\mathbf{c}_i)$. Correspondingly $\lambda_i^* = \min_j \sum_{i'} (\mathbf{P}_{ij'}/\mathbf{p}_{i'})/(\mathbf{C}_{ij}/\mathbf{c}_i)$.

To get an m^2-approximate solution to the original problem, take $\mathbf{x} = \sum_i \mathbf{z}(i)$ and $\lambda = \sum_i m_p\lambda_i^*$. The pair $(\mathbf{x}, \lambda)$ is a feasible solution to the original problem (each covering constraint is met the contribution of the corresponding $z(i)$, while each packing constraint is met because $\sum_{i'} (\mathbf{P}\mathbf{x})_{i'}/\mathbf{p}_{i'} \leq \lambda$). Finally, $\lambda \leq m^2\lambda^*$ because each $\lambda_i^* \leq \lambda^*$. □

Take $\lambda_0 = \lambda/m^2$ (for λ as in the lemma) so $1 \leq \lambda^*/\lambda_0 \leq m^2$. Next use binary search to find an integer j such that $2^j \leq \lambda^*/\lambda_0 < 2^{j+1}$. Given an arbitrary i, to decide whether $i \leq j$, solve the feasibility subproblem taking $\lambda' = \lambda_0 2^i$ and $\epsilon' = 1/2$. If there is an approximate solution then $\lambda^* \leq (1+\epsilon')\lambda' < \lambda_0 2^{i+1}$ and hence $i \leq j$. Otherwise the problem is infeasible so $\lambda^* > \lambda' = \lambda_0 2^i$ and hence $i > j$. Since there are $O(\log m)$ possible values of i, the binary search takes $O(\log\log m)$ subproblems each with $\epsilon' = 1/2$.

We have now computed $\lambda_1 = \lambda_0 2^j$ such that $1 \leq \lambda^*/\lambda_1 < 2$. Next we increase the precision. We start the ith step with λ_i such that $\lambda^*/\lambda_i \in [1, 1+\delta_i]$ for some $\delta_i > 0$, solve the feasibility problem taking $\lambda' = \lambda_i(1+\delta_i/4)$ and $\epsilon' = \delta_i/4$. If there is an approximate solution then $\lambda^*/\lambda_i \in [1, (1+\delta_i/4)^2]$, so take $\lambda_{i+1} = \lambda_i$. Otherwise the problem is infeasible, implying $\lambda^*/\lambda_i \in [1+\delta_i/4, 1+\delta_i]$, so take $\lambda_{i+1} = \lambda_i(1+\delta_i/4)$. In either case a calculation shows that $\lambda^*/\lambda_{i+1} \in [1, 1+\delta_{i+1}]$ for $\delta_{i+1} = (3/4)\delta_i$. Before $O(\log 1/\epsilon)$ steps, $\delta_i \leq \epsilon$, at which point the most recent solution produced will be ϵ-optimal.

The second phase (increasing the precision) requires solving $O(\log 1/\epsilon)$ subproblems, but, because ϵ' decreases geometrically in each step, the time to solve the subproblems is dominated by the time to solve the final subproblem (with $\delta = \Omega(\epsilon)$). Thus, the entire computation time is $O(\log\log m)$ times the time to solve a feasibility problem with $\epsilon' = 1/2$ plus the time to solve a single feasibility problem with $\epsilon' = \epsilon$.

6. Examples

6.1. Min-Cost Concurrent Multicommodity Flow

This section illustrates how to handle problems with exponentially many variables.

An instance of the min-cost concurrent multicommodity flow problem is defined by a weighted, capacitated, directed graph G, a collection of *commodities* $C_1, C_2, ..., C_k$, and a demand $d_i \geq 0$ for each commodity. Each commodity C_i is

in: weighted, capacitated digraph G, commodities, $\{(s_i, t_i, d_i)\}$, budget W.
out: 'infeasible' or f s.t. $f(e) \le \mu_e(1+O(\epsilon)), f(s_i,t_i) \ge d_i, w \cdot f \le (1+O(\epsilon))W$
0. Initialize $f(p) \leftarrow 0$ for all p.
1. Let $N \leftarrow 2\ln(m)/\epsilon$, where $m = 1 + \#\text{edges} + \#\text{commodities}$.
2. Define $\mathsf{local}_p(f) = [w(p)e^{w\cdot f/W}/W + \sum_{e\in p} e^{f(e)/\mu_e}/\mu_e]/[e^{-f(s_i,t_i)/d_i}/d_i]$.
3. Define $\mathsf{global}(f) = [e^{w\cdot f/W} + \sum_e e^{f(e)/\mu_e}]/[\sum_i e^{-f(s_i,t_i)/d_i}]$.
4. Until each commodity's demand is exceeded by a factor of N do:
5a. If g is not yet set or $\min_p \mathsf{local}_p(f)/g > 1+\epsilon$, then
5b. let $g \leftarrow \mathsf{global}(f)$, and
5c. if $\min_p \mathsf{local}_p(f)/g > 1$ then return 'infeasible'.
6. Delete any commodity whose demand is exceeded by a factor of N.
7. Choose any commodity i and path p for it s.t. $\mathsf{local}_p(f)/g \le 1+\epsilon$.
8. Set $f(p) \leftarrow f(p) + \delta$, where $\delta = \epsilon \min\{d_i, W/w(p), \min_{e\in p} \mu_e\}$.
9. Return f/N.

Figure 4. Algorithm with phases, applied to min-cost concurrent multicommodity flow. Implementable in time bounded by time to solve $O(m\log(m)/\epsilon^2)$ shortest-path subproblems, where m is the number of edges plus the number of commodities. Note: $f(s_i,t_i)$ denotes the flow shipped for commodity i.

the set of paths from some source vertex s_i to a sink vertex t_i. We also assume we are given a budget $W \ge 0$.

A solution is a multicommodity flow f, consisting of a network flow f_i for each commodity C_i. We think of f_i as specifying a flow $f_i(p) \ge 0$ for each path $p \in C_i$, but it also induces a flow $f_i(e) = \sum_{p\ni e} f_i(p)$ on each edge e. Without loss of generality, no path is in two commodities, so we drop the subscript i from $f_i(p)$.

Let w_e and μ_e denote the weight and capacity of edge e, respectively. Define the weight of path p to be $w_p = \sum_{e\in p} w_e$, and the weight of flow f to be $w \cdot f = \sum_p w_p f(p)$. For the total flow on an edge e or path p, we use $f(e) = \sum_{p\ni e} f(p)$. The amount of commodity i shipped is $\sum_{p\in C_i} f(p)$.

A solution f is *feasible* if: the amount of commodity i shipped is at least d_i, the flow on each edge is within the capacity ($f(e) \le \mu_e$), and the weight of the flow is within the budget ($w \cdot f \le W$). An approximate solution, given $\epsilon > 0$, is one where $|f_i| \ge (1-\epsilon)d_i$, $f(e) \le (1+\epsilon)\mu_e$, and $w \cdot f \le (1+\epsilon)W$.

As described, the problem is naturally a mixed packing/covering problem with a variable $f(p)$ for each path p and with the following constraints: $(\forall i) \sum_{p\in C_i} f(p) \ge d_i$, $(\forall e) \sum_{p\ni e} f(p) \le \mu_e$, $\sum_p f(p)w_p \le W$.

The simple implementation of the algorithm with phases reduces in this case to the algorithm in Fig. 4. As presented in the figure, the algorithm uses exponentially many variables (one for each path). However, to implement the algorithm it suffices to maintain only the flow for each edge and commodity and the total cost. To implement the inner loop, do the following for each commodity i: repeatedly find the shortest path p from s_i to t_i in the graph with edge weights given by $\ell(e) = w_e e^{w\cdot f/W} + e^{f(e)/\mu_e}/\mu_e$. If the length of p is at most $(1+\epsilon)g\, e^{-f(s_i,t_i)/d_i}/d_i$, then $\mathsf{local}_p(f)/g \le (1+\epsilon)$, so augment flow on p as described in the figure, otherwise, move on to the next commodity.

The time the algorithm takes is bounded by the shortest path computations. The number of these is equal to the number of increments plus at most one per commodity per phase. The number of increments is $O(m\log(m)/\epsilon^2)$, while there are $O(\log(m)/\epsilon^2)$ phases and $O(m)$ commodities. Thus there are $O(m\log(m)/\epsilon^2)$ shortest path computations.

6.2. X-Ray Tomography / Linear Equations

Computer tomography (a.k.a. x-ray tomography or Radon transform) is a special case of mixed packing and covering. Briefly, x-rays are taken of an object from many directions, and the internal structure (density at each point) of the object is reconstructed from the results.

For illustration, consider the following simple case. Assume an object resides within an $n \times n \times n$ cube. Discretize the cube by partitioning it into n^3 $1 \times 1 \times 1$ subcubes in the obvious way. Enumerate the subcubes in some order and introduce a variable $\mathbf{x}_j$ representing the density of the jth subcube.

Take d x-ray snapshots of the object from different directions. With current techniques, n is typically a few hundred and $d = \Theta(n^2)$ so that enough information is gathered to reconstruct a single solution. Even two-dimensional reconstruction problems are useful, as a volume can be recon-

in: $\mathbf{A}, \epsilon$
out: 'infeasible' or $\mathbf{x}$ s.t. $1 \le \mathbf{Ax} \le 1 + O(\epsilon)$.
0. Let $\mathbf{x}_j = \min_i 1/(n\mathbf{A}_{ij})$, where n is the number of variables.
1. Let $N \leftarrow (1 + 2\ln m)/\epsilon$, where m is the number of constraints.
2. Define $\mathsf{local}_j(\mathbf{x}) = \sum_i \mathbf{A}_{ij} e^{(\mathbf{Ax})_i} / \sum_i \mathbf{A}_{ij} e^{-(\mathbf{Ax})_i}$.
3. Define $\mathsf{global}(\mathbf{x}) = \sum_i e^{(\mathbf{Ax})_i} / \sum_i e^{-(\mathbf{Ax})_i}$.
4. While $\min \mathbf{Ax} < N$ do:
5a. If g is not yet set or $\min_j \mathsf{local}_j(\mathbf{x})/g > 1 + \epsilon$, then
5b. let $g \leftarrow \mathsf{global}(\mathbf{x})$, and
5c. if $\min_j \mathsf{local}_j(\mathbf{x})/g > 1$ then return 'infeasible'.
6. Delete the ith row of $\mathbf{A}$ for each i s.t. $(\mathbf{Ax})_i \ge N$. *—unnecessary, as* $\mathbf{Ax} \le O(N)$
7a. Compute $\boldsymbol{\alpha}$ and then δ such that
7b. $\boldsymbol{\alpha}_j = \mathbf{x}_j$ if $\mathsf{local}_j(\mathbf{x})/g \le 1 + \epsilon$ and $\boldsymbol{\alpha}_j = 0$ otherwise,
7b. and $\delta = \max_i(\mathbf{A}\boldsymbol{\alpha})_i$.
8. Set $\mathbf{x} \leftarrow \mathbf{x} + \epsilon\, \boldsymbol{\alpha}/\delta$.
9. Return $\mathbf{x}/N$.

Figure 5. Parallel algorithm as it specializes to approximately solve $\mathbf{Ax} = \mathbf{b}$ (in normalized form $\mathbf{Ax} = 1$). Used in the x-ray tomography example. Runs in $O(\log(m)/\epsilon^2)$ phases, makes $O(m\log(m)/\epsilon^2)$ increments, and runs in time $1/\epsilon^4$ times polylogarithmic in n and m.

structed in slices.

Assume each x-ray produces an $n \times n$ image. Discretize the image into its n^2 squares in the obvious way. Enumerate all dn^2 squares of all snapshots in some order. For the ith square, compute from the darkness of the square the total mass μ_i of the matter that the x-rays aimed at that square passed through. Add a constraint of the form $\sum_j \mathbf{x}_j \mathbf{A}_{ij} = 1$ where $\mathbf{A}_{ij}$ is the volume of the intersection of cube j and the cylinder of x-rays aimed at square i, divided by μ_i. (If $\mu_i = 0$, delete all variables $\mathbf{x}_j$ such that the $\mathbf{A}_{ij} > 0$.)

The reconstruction problem is to find $\mathbf{x} \ge 0$ such that $\mathbf{Ax} = 1$. The approximate version is to find $\mathbf{x}$ such that $\mathbf{Ax} \ge 1$ and $\mathbf{Ax} \le \mathbf{1} + O(\epsilon)$. The problem has $O(dn^2)$ constraints, and each variable occurs in d constraints.

Variables for cubes known to be outside the object can be deleted. If appropriate, additional constraints such as $\mathbf{x}_j \le 1$ (to constrain the maximum density) can be added.

Without these additional constraints, the problem is a special case of approximately solving a system of equations with non-negative coefficients: given $\mathbf{A}$, finding $\mathbf{x} \ge 0$ such that $\mathbf{Ax} = 1$. The parallel algorithm, as it specializes for this problem, is shown in Fig. 5. Note that deletion of satisfied covering constraints can be omitted and the analysis of the algorithm will still hold, because the packing constraints ensure that no covering constraint exceeds its upper bound by more than an $O(1)$ factor.

The total work done by this algorithm is more than with traditional methods for x-ray tomography (filtered back-projection and Fourier reconstruction). However, this method may be easier to parallelize. It is more flexible, in that additional constraints can be added. In some cases, for example when directions from which the snapshots can be taken are constrained, traditional methods suffer from ill-conditioning, whereas this approach may not.

7. Final Remarks

Open problem: find an efficient width-independent Lagrangian-relaxation algorithm for the abstract mixed-packing covering problem:

$$\text{Find } \mathbf{x} : \mathbf{Px} \le (1+\epsilon)\mathbf{p}, \mathbf{Cx} \ge (1-\epsilon)\mathbf{c}, \mathbf{x} \in \mathcal{P}$$

where $\mathcal{P}$ is a polytope that can be queried by an optimization oracle (given $\mathbf{c}$, return $\mathbf{x} \in \mathcal{P}$ minimizing $\mathbf{c} \cdot \mathbf{x}$) or some other suitable oracle. The min-cost multicommodity flow example earlier in the paper is a special case. Although that example illustrates how to deal with exponentially many variables, the polytope in that example is the degenerate one $\{\mathbf{x} : \mathbf{x} \ge 0\}$. A polytope that illustrates the difficulty of the general case is $\mathcal{P} = \{\mathbf{x} : \sum_j \mathbf{x}_j = 1\}$. The difficulty seems to be using variable-size increments with *three* constraints: the packing constraints, the covering constraints, and the constraint of staying in the polytope.

Find a parallel algorithm whose number of iterations is polylogarithmic in the number of constraints, even if the number of variables is exponential. Find a parallel algorithm whose running time has an ϵ^{-2} or ϵ^{-3} term instead of the ϵ^{-4} term.

The algorithms in this paper handle any pure packing or covering problem as a special case. In this case the algorithms simplify somewhat, so that they can handle the optimization versions of the problems directly.

The algorithms can be viewed as derandomizations (using the method of conditional probabilities) of natural randomized rounding schemes (see [23, 24] for this approach). Lower bounds on the number of iterations required by Lagrangian-relaxation algorithms are presented in [16].

Thanks to Lisa Fleischer for useful discussions.

References

[1] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[2] D. Bienstock. Approximately solving large-scale linear programs. i. Strengthening lower bounds and accelerating convergence. Technical Report 1999-1, CORC, Columbia Univeristy, June 1999.

[3] G. W. Brown and J. von Neumann. Solutions of games by differential equations. In *John von Neumann: Collected Works*, volume VI, pages 38–43. Pergamon Press, 1963. Reprinted from Ann. Math Studies No. 24 pp. 73-79 (1950).

[4] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Res.*, 8:101–111, 1960.

[5] L. K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, Nov. 2000.

[6] L. R. Ford Jr. and D. R. Fulkerson. A suggested computation for maximal multicommodity network flow. *Management Sci.*, 5:97–101, 1958.

[7] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *39th Annual Symposium on Foundations of Computer Science*. IEEE, 1998.

[8] M. Grigoriadis and L. Khachiyan. Approximate minimum-cost multicommodity flows in $o(\epsilon^{-2}knm)$ time. *Math. Programming*, 75:477–482, 1996.

[9] M. Grigoriadis and L. G. Khachiyan. A sublinear-time randomized approximation algorithm for matrix games. Operations Research Letters, 18(2):53–58, Sep. 1995.

[10] M. Grigoriadis and L. G. Khachiyan. An exponential-function reduction method for block-angular convex programs, 1995. Networks 26(1.2), p. 59-68.

[11] M. D. Grigoriadis and L. G. Khachiyan. Coordination complexity of parallel price-directive decomposition. *Mathematics of Operations Research*, 21:321–340, 1996.

[12] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1971.

[13] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1:6–25, 1971.

[14] D. Karger and S. A. Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *ACM Symposium on Theory of Computing*, pages 18–25, 1995.

[15] P. Klein, S. Plotkin, C. Stein, and E. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM J. Comput.*, 23(3):466–487, June 1994.

[16] P. Klein and N. E. Young. On the number of iterations for Dantzig-Wolfe optimization and packing-covering approximation algorithms. In *Lecture Notes in Computer Science*, number 1610, pages 320–327, 1999. IPCO ’99.

[17] J. Könemann. *Fast Combinatorial Algorithms for Packing and COvering Problems*. PhD thesis, Max-Planck-Institute for Informatik, 2000.

[18] H. W. Kuhn and A. W. Tucker. Review of ‘A numerical method for determination of the value and the best strategies of a zero-sum two-person game with large numbers of strategies’, by John von Neumann. In *John von Neumann: Collected Works*, volume VI, pages 96–97. Pergamon Press, 1963.

[19] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *J. Comput. Syst. Sci.*, 50(2):228–243, Apr. 1995.

[20] M. Luby and N. Nisan. A parallel approximation algorithm for positive linear programming. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 448–457, San Diego, California, 16–18 May 1993.

[21] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20(2):257–301, 1995.

[22] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *JACM*, 37:318–334, 1990.

[23] N. 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 Jan. 1995.

[24] N. E. Young. K-medians, facility location, and the Chernoff-Wald bound. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 86–95, January 2000.