A Min-Edge Cost Flow Framework for Capacitated Covering Problems

Jessica Chang* Samir Khuller†

Abstract

In this work, we introduce the Cov-MECF framework, a special case of minimum-edge cost flow in which the input graph is bipartite. We observe that several important covering (and multi-covering) problems are captured in this unifying model and introduce a new heuristic *LPO* for any problem in this framework. The essence of *LPO* harnesses as an oracle the fractional solution in deciding how to greedily modify the partial solution. We empirically establish that this heuristic returns solutions that are higher in quality than those of Wolsey's algorithm. We also apply the analogs of Leskovec et. al.'s [25] optimization to *LPO* and introduce a further freezing optimization to both algorithms. We observe that the former optimization generally benefits *LPO* more than Wolsey's algorithm, and that the additional freezing step often corrects suboptimalities while further reducing the number of subroutine calls. We tested these implementations on randomly generated testbeds, several instances from the Second DIMACS Implementation Challenge and a couple networks modeling real-world dynamics.

1 Introduction

Flow networks have played a pivotal role in the design of algorithms, with an impact spanning several areas of computer science and operations research, from network design and computer vision to scheduling and routing [1]. In particular, many important covering problems are special cases of the fundamental minimum-edge cost flow problem (MECF). Several of them have direct connections to applications outside of computer science; from understanding glycoprotein formation [17] to identifying galaxies and quasars via spectroscopic data [27], the effects of good algorithms for covering problems are far-reaching. In this work, we investigate the relationship between MECF and generalizations of well-studied covering problems as well as batch scheduling models motivated by energy efficiency in data centers; in the process, we develop powerful heuristics and optimizations and empirically demonstrate their merits in scheduling contexts and other covering settings.

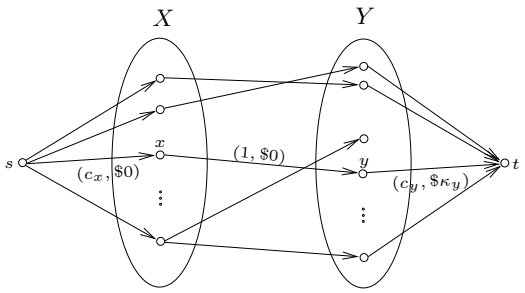
Recall the minimum-edge cost flow problem: G is a directed graph given by $G = (V, A)$ with a specified source s and sink t. Each arc a has a corresponding integral capacity $c(a)$ and price $\kappa(a)$. Given a target flow value f^*, the goal is to find a flow function on G realizing at least f^* units of flow from s to t with minimum cost. The difficulty of MECF is that it adheres to the fixed cost model, under which any positive flow sent across arc a incurs the entire cost $\kappa(a)$ of the arc. In other words, if A' is the subset of arcs along which positive flow is sent, the total cost of the flow is $\sum_{a \in A'} \kappa(a)$.

Our study pertains to the following framework of MECF, henceforth denoted *Cov-MECF*. The graph G is such that $G - \{s, t\}$ is a bipartite graph (X, Y, E), s has arcs only to X and t has arcs only from Y. (All edges in E are oriented from X to Y.) In addition, the capacities $c(a)$ are unit for $a \in E$. The arcs (y, t) also have costs κ_y for all $y \in Y$; the prices of all other arcs are zero. See Figure 1(a). For ease of notation, we will denote the capacities for edges from s to $x \in X$ as c_x and capacities from $y \in Y$ to t as c_y.

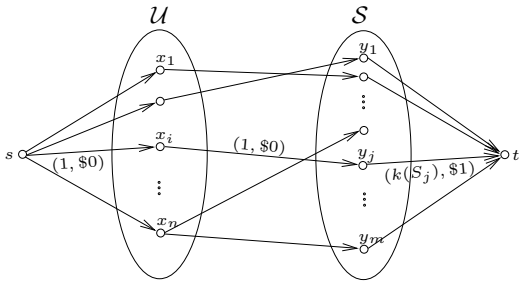
Several covering problems fit into this framework, including the following canonical problem. In Set Cover, there is a ground set $\mathcal{U} = \{u_1, \ldots, u_n\}$ of elements and a collection $\mathcal{S} \subseteq 2^{\mathcal{U}}$ of subsets $S_1, \ldots, S_m$. The goal is to find a minimum cardinality subset $\mathcal{S}' \subseteq \mathcal{S}$ such that $\cup_{i \in \mathcal{S}'} S_i = \mathcal{U}$. An $O(\log n)$-approximation exists for Set Cover [19, 26, 8], that greedily selects the set that minimizes the cost to marginal benefit ratio until all ground elements are covered. Feige [12] provides a matching lower bound. Wolsey extended the $O(\log n)$-approximation to a larger class of covering problems to which all problems considered here belong [32]. The connection between Cov-MECF and a natural generalization of Set Cover called Capacitated Set Cover follows.

*Dept. of Computer Science and Engineering, University of Washington, Seattle WA 98195, email: jschang@cs.washington.edu. Research done while author was visiting the University of Maryland, supported by an NSF Graduate Research Fellowship, an NSF ADVANCE grant and NSF CCF-0937865.

†Dept. of Computer Science, University of Maryland, College Park MD 20742, email: samir@cs.umd.edu. Research supported by NSF CCF-1217890, NSF CCF-0937865 and a Google Research Award.



(a) The Cov-MECF framework.



(b) Capacitated Set Cover.

Figure 1: Each edge is labeled with its capacity, followed by its fixed cost.

Capacitated Set Cover. In Capacitated Set Cover (CapSC), each set S_j has a capacity $k(j)$ and the goal is to find the smallest collection of sets, with the additional requirement that set S_j covers at most $k(j)$ elements. This problem can be cast into the Cov-MECF framework by creating the following graph G' (depicted in Figure 1(b)). There is an "element node" x_i for every ground element $u_i \in \mathcal{U}$, as well as a "set node" y_j for every set $S_j \in \mathcal{S}$. In addition, there is a source s and an edge (s, x_i) for every x_i, with unit capacity and zero cost. Create a sink t, and insert edge (y_j, t) for every y_j, with capacity $k(j)$ and unit cost. Finally, there is an edge (x_i, y_j) if and only if element u_i can be covered by set S_j in the original instance. Identifying a capacity-respecting set cover of minimum size amounts to finding in G' a minimum edge-cost flow of value equal to n.

As this problem generalizes Set Cover, the lower-bounds on the approximation guarantee apply. Wolsey's greedy algorithm is an $O(\log n)$-approximation [32]*. We highlight that unlike its uncapacitated counterpart, computing the marginal benefit of adding a set to a partial cover requires a flow computation.

Despite the dire state of affairs at the theoretical level, in the real world, there remains an undeniable need to solve problems of a set cover flavor. One commonly implemented algorithm for doing so is in fact Wolsey's greedy algorithm. Despite its tight $O(\log n)$ guarantee, studies indicate that in practice, it fares much better. For instance, in their widely-cited result, Kempe et. al. [21] examine the influence function in social networks, demonstrating that not only does it exhibit submodularity, Wolsey's greedy algorithm in fact returns solutions that are much closer to the optimal solution than its worst-case bound would imply. The approach has since been applied to a plethora of problems within the social networks literature, ranging from outbreak detection in networks [25] to the spread of information via the blogosphere [18].

The primary disadvantage of Wolsey's algorithm is that each iteration may involve computing the marginal benefit for many candidate sets, requiring a high number of subroutine flow calls. For large instances, this can render Wolsey's algorithm unacceptably slow, limiting the size of the input on which it can be run. This issue has been the subject of considerable investigation within the last decade. Leskovec et. al. [25] show that if sets are considered in a particular order, irrelevant computations can be avoided, thereby reducing the total number of flow calls. Their algorithm CELF maintains the spirit of Wolsey's greedy algorithm and was recently improved to CELF++ by Goyel et. al. [16]. The concern is also a focus of this study.

Interest in variations of the Set Cover problem extends to the Sloan Digital Sky Survey project, an effort toward surveying the history of astronomy. Lupton et. al. [27] studied a special geometric case of Capacitated Set Cover in the context of collecting telescopic snapshots of the universe. The telescope may be pointed to a "disc"-shaped area in the sky to capture a high-resolution image for processing; the restriction is that the number of galaxies per image that can be processed cannot exceed a given constant. The goal is to process each object of interest via the smallest number of snapshots. Finding good solutions to this problem has implications on the order of millions of dollars. For this NP-hard problem, Lupton et. al. propose a fast, greedy heuristic whose solutions are in general about 20 percent better than their near-uniform counterparts. Their method is faster than standard IP approaches in the operations research literature and in practice, yields solutions of impressive quality.†

All of these problems are captured in the Cov-MECF framework. Despite its wide array of applica-

*Wolsey proved, in fact, it is an $O(\log(\max_j k(j))$-approximation.

†We note that significant theoretical attention has been given to a special case of geometric set cover, in which ground elements are points and sets are objects in Euclidean space [28, 31, 5, 11, 3, 4]. Many of the results generalize to multicovering variants, but under a different notion of capacitated sets.

tions, MECF is in general less understood compared to its well-studied min-cost flow counterpart, even when restricted to the special case considered here. MECF is a classic NP-hard problem, listed as [ND32] in [14]. Krumke et. al. [23] showed that unless $NP \subseteq DTIME(n^{O(\log\log n)})$, there exists no $(1-\epsilon)\ln f^*$-approximation for any constant $\epsilon > 0$. Even et. al. [10] established that unless $NP \subseteq DTIME(n^{polylog(n)})$, there does not exist a $2^{\log^{1-\epsilon} n}$-approximation, for any constant $\epsilon > 0$.

Despite the fact that it is a special case of MECF, Cov-MECF still subsumes many central problems. One benefit of this formation is that it captures multi-set multi-cover problems. In other words, one can demand that elements be covered multiple times. This is an imperative aspect of many scheduling problems, in particular, when jobs of non-unit length must be scheduled.

Our concrete contributions are two-fold:

1. We introduce the LPO algorithm for problems within the Cov-MECF framework. While Wolsey's approach builds a cover from the empty set, LPO iteratively trims off sets from the initial feasible cover $\mathcal{C} = \mathcal{S}$. The essence of LPO harnesses the fractional optimal solution to guide the selection of the next set to remove. We apply the analogs of Leskovec et. al.'s [25] optimizations to LPO and empirically compare the performance to that of Wolsey' greedy algorithm. We find that despite its lack of theoretical bounds, on an overwhelming majority of instances, LPO returns solutions that are closer to optimal than Wolsey's algorithm. We also discover that the performance verdict is largely dictated by the specific covering problem. In particular, for instances of Capacitated Vertex Cover and Active Time Scheduling (discussed below), LPO is faster than Wolsey's algorithm. The reverse is true for general Capacitated Set Cover; however we find that on these instances, the optimization of Leskovec et. al. levels their performances rather evenly.

2. We further propose an additional "freezing" step. Before LPO (Wolsey's algorithm, respectively) is run, the LP relaxation of the covering problem is solved. Any set variables that take on integral values in the fractional solution are *frozen* to those values (0 or 1, typically) prior to the invoking of the algorithm proper. Our experiments indicate that more often than not, freezing improves rather than hurts the quality of the solution. More importantly, freezing drastically reduces the number of flow calls, since the majority of the variables attain integral values in the fractional solution.

We note that on smaller-sized input, directly solving for the optimal integral solution may be faster than executing the heuristics mentioned above, given their iterative nature. Thanks to the current technology of mixed integer program solvers, it is not until the instance size is rather large that we truly see the cost of computing the optimal solution via exponential heuristics. Our discussion on performance pertains to approaches that scale and thus focuses on the comparison of the polynomial-time heuristics and their optimizations.

The rest of this paper is outlined as follows. We devote the remainder of this section to the delineation of the covering problems considered in this study. In Section 2, we revisit Wolsey's algorithm, define LPO and describe the optimizations applied. Section 3 addresses the implementation details and data sets. Results, interpretation and discussion follow in Section 4.

Covering Problems. Cov-MECF captures several important covering problems. In addition to Capacitated Set Cover, we consider two other problems in this study, the first of which is a well-known classic special case of Capacitated Set Cover. The second application is motivated by data centers and the growing need for an algorithmic understanding of energy-related issues [15].

Capacitated Vertex Cover (CapVC). In the standard Vertex Cover problem, we are given a graph $G = (V, E)$ with weights $w(v)$ on the vertices. The goal is to find a subset $S \subseteq V$ of minimum weight such that every edge is incident to a vertex in S. This is a special case of Set Cover in the sense that each element can be covered by only two sets.

In the capacitated version, vertex v has capacity $k(v)$, i.e. an upperbound on the number of edges that can be assigned to it. The goal is to find a minimize size cover that respects the capacities. When the capacities are soft, then multiple copies of a vertex v may be selected; each copy is permitted to cover up to $k(v)$ edges, and the cost $w(v)$ is incurred for each copy of v in the vertex cover. This problem is NP-hard. Guha et. al. [17] give a primal-dual 2-approximation for the soft capacities case. For the unweighted hard capacities version, a 2-approximation was given by Gandhi et. al [13], improving upon the initial bound of three given by Chuzhoy and Naor [7]; the weighted case is Set Cover-hard. Extensions to hypergraphs were made by Khuller and Saha [30]. As far as we know, there is no experimental research suggesting heuristics with performance better than twice the optimal solution.

Active Time Scheduling (ATS). Classical scheduling theory has traditionally focused on objectives that capture interests of the scheduler (e.g. makespan) or of individual jobs (e.g. max tardiness, flow time). However, more and more commonly, the most monetarily expensive computations are executed over massive corpora residing at data centers. Between facility maintenance, cooling requirements and powering hardware, the energy consumption of a data center translates into a budget on the order of millions of dollars. Understanding the extent to which this cost (as well as the carbon footprint) can be mitigated is an area of ongoing research within the systems community.

The considerable bulk of the power consumed by a data center is shared between cooling components and the energy requirements of the memory storage units; the power consumed by the processors of a data center accounts for less than two percent of the total power cost [2]. However, changes in scheduling policies at the processor level "have a dramatic impact in data center efficiency defined as the amount of 'useful work' done, or performance delivered per watt consumed" [15]. As highlighted in the following quote, this calls for scheduling algorithms expressly tailored toward energy efficiency, in contrast to the classical objectives studied in the past.

> "Potential benefits include increased data center capacity and reduced capital expenditures as well as reduced power and cooling costs with power-aware job scheduling ... However, current batch job scheduling algorithms and configurations are tuned only to optimize performance; energy efficiency has been ignored" [15].

In an effort to address this concern, the following scheduling model was introduced by Chang et. al. [6]. Given is a batch machine that can work on up to B different jobs simultaneously as well as a memory storage unit holding the data that each job must access. Each job J_i has a set of time intervals T_i during which the processor may work on it. T_i being a single interval corresponds to the standard setting in which the job has a release time and deadline; in this case, we denote its feasible region by its release time and deadline. (In practice, it is often the case that jobs are periodic in nature; there is a sizeable research effort in periodic scheduling. Moreover, a job's feasible region may depend on irregular constraints, e.g. the availability of an external resource or a user's schedule. The model proposed is general enough to capture jobs of this nature.) The processor must devote p_i slots to J_i, during which time, the memory storage unit should be "on". The *active time* is the total time that memory is on. The notion of active time hinges on the assumption that whenever the storage unit is on, a constant power cost is incurred, whether the processor is working on B jobs or a single job, i.e. that the power consumed by the processor is negligible to that of the storage unit, as commonly observed in data centers [2, 15].

The goal is to find a schedule that minimizes active time subject to all jobs being completed by their deadlines and the processor's batch capacity respected. Within the scope of this paper, jobs have a single feasible interval (i.e. a release time and deadline) and preemption is allowed only at integer time points. We call this problem *active time scheduling*. In Graham's notational convention, the problem is $P|r_j, d_j, p_j, pmtn^+|\sum_t a_t$, where a_t is 1 whenever memory storage unit is on at time unit t, and 0 otherwise. This problem fits into the Cov-MECF framework in the following way: the left side $X = J$ consists of job nodes which must be "covered" multiple times, according to their lengths. The right side $Y = T$ is the set of time slots, each of which can be assigned at most B jobs. The parameter f^* is $\sum_i p_i$. Notice that subgraph (J, T) is convex[‡].

In general, classic scheduling algorithms, e.g. Earliest Deadline First policies, do not immediately minimize active time. When jobs have unit length and a single release time and deadline, minimum active time can be achieved via a 2-pass linear time algorithm. For arbitrary feasible regions and $B = 2$, there is also an exact algorithm based on identifying degree-constrained subgraphs[6]. For arbitrary B, minimizing active time becomes trivially NP-complete via a reduction from Vertex Cover. On the other hand, the complexity of minimizing active time for arbitrary length jobs that have a single feasible interval (i.e., a release time and deadline) remains unclear. All of these problems fit within the Cov-MECF framework.

Demaine and Zadimoghaddam [9] recently considered a different scheduling problem of minimizing cumulative energy consumption for unit-length jobs over multiple processors, demonstrating that the coverage function is submodular, i.e. that the greedy algorithm yields an $O(\log n)$-approximation.

A more general scheduling model is considered by Khuller et. al. [22], in which jobs must be assigned to machines, each machine has an associated cost and capacity, and subject to staying within a budget, the

[‡]A bipartite graph (X, Y) is *convex* means that there is an order $y_1, y_2, \ldots$ of the elements in Y such that for any node $x \in X$, if x is adjacent to y_i and y_j, then x is also adjacent to y_k for all $k \in [i, j]$.

goal is to purchase a set of machines and assign jobs to them to minimize the maximum load on any machine. This problem is clearly Set Cover-hard. Khuller et. al. give a greedy algorithm achieving twice the optimal makespan, when it is permitted to violate the budget by a factor of $O(\ln n)$.

2 Algorithms for Covering

We begin this section with a review of Wolsey's greedy algorithm. Then we introduce the *LPO* heuristic. Finally, we describe the optimizations applied to both algorithms. Even though for simplicity's sake, the discussion is given in the context of Capacitated Set Cover, the algorithms and optimizations can be applied to other covering problems within this framework.

2.1 Wolsey's Algorithm Wolsey's greedy algorithm begins with the empty cover $\mathcal{C}$ and iteratively adds sets to it until all ground elements can be covered. In each iteration, it selects the set that minimizes the cost to marginal benefit ratio. More formally, it adds to $\mathcal{C}$ the set S that minimizes $w(\mathcal{C}, S)$, with

$$w(\mathcal{C}, S) = \frac{c(S)}{f(\mathcal{C} \cup \{S\}) - f(\mathcal{C})}$$

for $f(\mathcal{C} \cup \{S\}) - f(\mathcal{C}) > 0$ and unbounded otherwise, and where $c(S)$ is the cost of S and $f(\mathcal{C}')$ is the maximum number of elements that can be covered by $\mathcal{C}'$.

2.2 LPO While Wolsey's algorithm starts with the empty solution and greedily adds sets to it until it becomes feasible, the LP oracle algorithm (*LPO*) begins with the feasible solution $\mathcal{C}' = \mathcal{S}$ and greedily removes sets until $\mathcal{C}'$ is a minimal cover, i.e. no more sets can be removed without violating feasibility.

At the heart of *LPO* is an LP solver that provides "hints" as to which sets should be closed. Over the course of the algorithm, *LPO* maintains a cover $\mathcal{C}$ of sets that have not yet been closed, i.e. removed. In iteration i, it determines the set $\mathcal{S}_i$ of sets where $\mathcal{S}_i = \{S \in \mathcal{C} : \mathcal{C} \backslash \{S\} \text{ is feasible }\}$. In other words, $\mathcal{S}_i$ is the sets that are candidates for being removed from $\mathcal{C}$. *LPO* closes the set S' such that

$$S' \in \arg\min_{S \in \mathcal{S}_i} \ell(\mathcal{C} \backslash \{S\})$$

where $\ell(\mathcal{C}')$ is the fractional cost of the instance in which only sets in $\mathcal{C}'$ can be open (even partially open). The value $\ell(\mathcal{C} \backslash \{S\})$ can be interpreted as the fractional cost of removing set S from $\mathcal{C}$. Intuitively, *LPO* performs well when $\ell(\mathcal{C} \backslash \{S\})$ corresponds to the effect of closing S on the integral solution. *LPO* repeats these steps until there are no more open sets that can be closed. The intuition behind this is that the LP oracle gives warning when closing a set might impose a heavy cost in later iterations. In some sense, it gives an idea of which sets are more important than others.

As an example, consider the following instance of active time scheduling of unit jobs with batch parameter $B = 5$ (Figure 2). There is a block of four rigid jobs, each of length four, which must be scheduled in slots six through nine. These slots each have room for one more job unit. What does the rest of the optimal solution look like? If the last slot is open, then one can achieve an active time of 6 (or $B+1$ in general). However, if the last slot is closed, then the unit jobs are forced to occupy the remainder of the otherwise rigid block, which will then force the long chain of 5 (or B) to be scheduled earlier, for a total active time of 9 (or $2B - 1$). This is suboptimal, and *LPO* can detect this suboptimality. It cannot feasibly close slots six through nine. If it closes the last slot, the cost of the scheduling will necessarily go up; the fractional solution subject to slot 10 being closed reflects this.

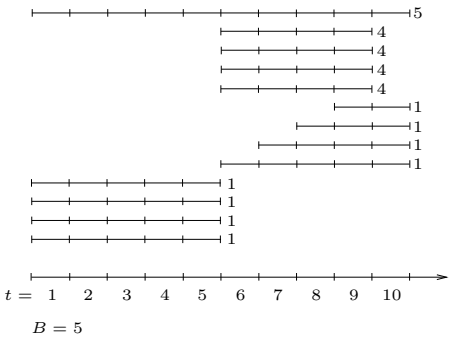


Figure 2: An example of Active Time Scheduling, highlighting the strength of *LPO*.

One can identify instances on which *LPO* performs poorly. Figure 3 is such an instance. The optimal cover is $\{S_1, S_2, S_5\}$ and has a total cost of seven: S_1 covers $\{e_1, e_4, e_5\}$, S_2 covers $\{e_3\}$, and S_5 covers $\{e_2, e_6\}$. However, *LPO* can return a cover of cost eight: in its first iteration, it computes the values $\ell = (7.0, 8.0, 7.0, 7.0, \infty)$, where ℓ_i is the fractional cost subject to the removal of set S_i. At this point, *LPO* has a choice between removing sets S_1, S_3 and S_4. Removal of set S_3 or S_4 will result in an optimal solution. On the other hand, removing set S_1 is a mistake, leading to the cover $\{S_3, S_4, S_5\}$ of cost eight. Unfortunately, *LPO* can select any set S_i of minimum ℓ_i value, breaking ties arbitrarily. For example, as in our implementation, it

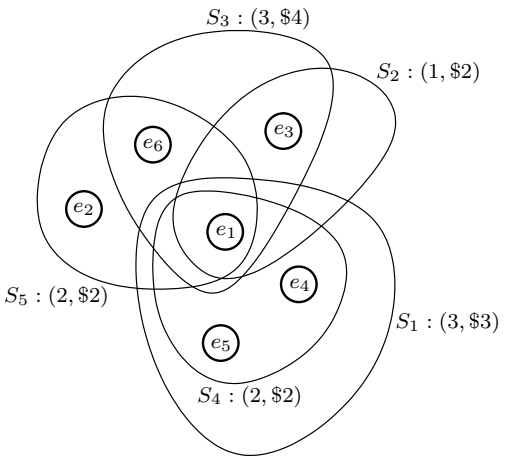


Figure 3: An example of Capacitated Set Cover on which *LPO* can perform suboptimally. Sets are labeled by their capacities, followed by their costs.

may favor sets of lower index. The issue extends beyond tie-breaking conventions. Even though *LPO* views sets S_1, S_3 and S_4 as equally good candidates for removal, ℓ_3 and ℓ_4 can be achieved via integral assignments, while no such assignment exists for ℓ_1. In this regard, the values ℓ_i can misguide *LPO*; they are not always good estimates of the cost to the integral solution. Despite this, our results empirically demonstrate that *LPO* performs very well in general, frequently beating out Wolsey's algorithm and often achieving optimality itself.

2.3 Optimizations The primary disadvantage of Wolsey's algorithm is that it can require $\Theta(m^2)$ computations of $w(\mathcal{C}, S)$, each being quite costly. In particular, for problems involving hard capacities, a maximum flow computation is necessary to determine $w(\mathcal{C}, S)$; it may be that the elements in S can be covered by sets already in $\mathcal{C}$, but that due to the capacity constraints, the addition of S to $\mathcal{C}$ would still increase the total number of elements that can be covered. *LPO* is plagued by a similar affliction, in that it can necessitate several computations of $\ell(\mathcal{C}')$, especially on instances for which there are many feasible covers of small cardinality. The following optimizations were designed to reduce the number of such flow computations without compromising solution quality.

Optimization: Ordering sets [25]. The following observation can reduce Wolsey's number of computations, without affecting the cost of the solution, up to tiebreaking. Due to the submodularity of f, the value of $w(\mathcal{C}, S)$ for a given set S can only increase as $\mathcal{C}$ grows. Therefore one can delay recomputing $w(\mathcal{C}, S)$ for sets S that already had high values in previous iterations, since they are guaranteed not to be selected in the current iteration. More formally, let $\mathcal{C}_i$ be the partial cover at the beginning of the ith iteration. Suppose that $S_1, S_2, \ldots, S_m$ is the order by which the algorithm computes $w(\mathcal{C}_i, \cdot)$. Given a particular set S_j, let B_j denote the minimum $w(\mathcal{C}, \cdot)$ value among the first j sets, i.e.

$$B_j = \min_{S \in \{S_1, \ldots, S_j\}} w(\mathcal{C}_i, S)$$

If, for set S_{j+1} and some earlier iteration $i' < i$, $B_j < w(\mathcal{C}_{i'}, S_{j+1})$, then it follows that $B_j < w(\mathcal{C}_i, S_{j+1})$ as well, since $w(\mathcal{C}_{i'}, S_{j+1}) \leq w(\mathcal{C}_i, S_{j+1})$. Thus we know that S_{j+1} will not be the next set added to $\mathcal{C}_i$, and there is no need to compute $w(\mathcal{C}_i, S_{j+1})$ explicitly. This observation holds regardless of the order by which the algorithm iterates over sets. Thus, if it considers sets in increasing order of their most recently computed $w(\cdot, S_j)$ value, the number of computations can only further decrease while the solution cost remains the same (again, up to tiebreaking).

As observed in [25], applying this optimization to Wolsey's algorithm yields a significant improvement in performance; we demonstrate this independently in our results. Since the the spirit of the original algorithm is preserved, the $O(\log n)$ guarantee still holds. However, as before, this bound is somewhat irrelevant since, as with the naive implementation, the solution quality is often much better.

One can apply the analogous optimization to *LPO*, since for any set S, $\ell(\mathcal{C} \backslash \{S\})$ can only increase as $\mathcal{C}$ gets smaller with each passing iteration. We omit the details since they are a straightforward modification of the optimization as applied to Wolsey's algorithm.

Optimization: Freezing set variables. We say a set is *closed* if it is not in the cover and *open* otherwise. Simply stated, the following preprocessing step is done prior to the execution of the (original or otherwise optimized) algorithm: first solve the LP relaxation (see Appendix B). Any sets having values 0 or 1 in the fractional optimal solution are *frozen* to being closed or open, respectively. Then run Wolsey's algorithm (*LPO*, respectively) on the remaining *unfrozen* sets, until a feasible solution is attained (until no more unfrozen sets can be closed, respectively). Note that for this optimization applied to Wolsey's algorithm, the theoretical bounds no longer hold. In spite of this, we empirically demonstrate that Wolsey and *LPO* with both modifications return solutions that are still quite close to optimal. In particular, not only does the freezing optimization frequently

correct suboptimal solutions, it also severely reduces the number of flow evaluations that would have been necessary otherwise.

3 Implementation

All implementations were developed in Java, invoking CPLEX (version 12.4) wherever a flow or LP call was needed. Experiments were performed on a shared machine with eight 2.29gHz quad-core processors and 256gb RAM, the unoptimized heuristics taking anywhere from several hours to days to compute over the heavier data sets.

3.1 Data Sets We generated several structured random instances of CapSC and ATS. For notational convenience, we describe CapSC in the context of jobs and slots, rather than ground elements and sets. Given parameters (N_1, N_2, M_1, M_2, B), instances of n jobs for $N_1 \leq n \leq N_2$ random and T slots for $M_1 \leq T \leq M_2$ random were created. For a CapSC instance, capacities bounded by B were independently and randomly chosen for each slot, and each job randomly selected the slots that could cover it. If the instance was that of ATS, a single capacity $B' < B$ was randomly selected, and each job randomly chose its release time, a deadline after its release time, and a job length no more than the length of its feasible window. Infeasible instances were dropped and replaced with feasible ones.

Fifteen CapSC instances and fifteen ATS instances were generated with parameters $(25, 50, 30, 60, 6)$. These data sets are denoted "small" sets. The "medium" data set consisted of five CapSC instances and five ATS instances, generated with parameters $(50, 100, 75, 180, 7)$. For the CapVC testbed and parameters (N_1, N_2, P_1, P_2), a random graph $G(n, p)$ was created where n was uniformly selected at random from $[N_1, N_2]$ and with p similarly selected from the range $[P_1, P_2]$. Then, capacity $k(v)$ was computed as $k(v) \sim \mathcal{U}(0, d(v))$. Infeasible instances were dropped and replaced with feasible ones. Twenty instances were generated from each of the following parameter vectors: $(15, 50, 0.2, 0.95)$ and $(80, 130, 0.2, 0.6)$.

Our heuristics were also tested on modified graph instances taken from the public testbed given by the DIMACS Implementation Challenge [20]. Some graphs were originally intended to test Clique algorithms and are quite dense. Others were intended to test heuristics for the k-coloring problem and are less so. Finally, we evaluate our heuristics on two instances that model large-scale social interaction and connectivity structure, respectively. Newman's NetSci graph is a collaboration graph between authors publishing in Network Science [29]. It contains 1589 vertices and 2,742 edges. We also consider a network of peer-to-peer file sharing connections; it contains 6,301 vertices, one per Gnutella host, and 20,777 edges denoting observed connection between two hosts [24].

We note that for instances $G = (V, E)$ of CapVC, the X side of the Cov-MECF framework has $|X| = |E|$, while the Y side has $|Y| = |V|$. The out-degree of each node $x \in X$ is two, since each edge can be covered only by its two incident nodes.

3.2 Inducing Capacities The DIMACS graphs and the real world graphs are instances of standard Vertex Cover; we aggressively imposed capacities that were as low as possible, without rendering the instance infeasible in the following way. For a given instance, assign to vertex v a capacity that is selected independently and uniformly from the range $[\lceil \lambda d(v) \rceil, d(v)]$, where $d(v)$ is the degree of vertex v and $0 < \lambda < 1$ is a constant. The subsequent procedure was followed to ensure that feasibility was maintained: for each candidate λ, assign capacities until feasibility is achieved or the fifth attempt has been made. If the instance is infeasible, increase λ by 0.05 and repeat. This process is guaranteed to return feasible capacity assignments as long as the uncapacitated version is feasible. In our case, λ was initialized to 0.8.

3.3 Additional Perspective Two LP rounding algorithms, SD and SR, were implemented to provide additional perspective to the solution qualities returned by *LPO* and Wolsey's algorithm. Let u_y denote the value of the indicator variable for the set S_y corresponding to $y \in Y$ in the fractional solution. SD adds S_y to the cover if $u_y \geq \frac{1}{2}$. If the cover is still not feasible, SD greedily adds the remaining sets in decreasing order of their u_y's. SR adds each set S_y to the cover independently with probability $p_y = \min\{1, 2u_y\}$. If the cover is feasible, it returns; otherwise, it repeats.

4 Results and Discussion

In this section, we discuss the results of the various implementations of Wolsey's algorithm and *LPO*. We evaluate them on solution quality and efficiency of computation perspectives. For the latter, the primary metric considered is the total number of flow (or LP) evaluations made. We draw no conclusions based on timing constructs since the experiments were executed on a

[§]One instance failed to complete computation. Its contribution is omitted from the relevant statistics.

	Num. Inst.	*LPO*	*LPO*/OO	*LPO*/OOF	Wol	Wol/OO	Wol/OOF	SD	SR
ATS	20	0	0	0	3	5	0	0	0
CapVC	23	0	1	1	7[§]	6	1	6	11
CapSC	20	1	1	1	5	6	5	12	18

Table 1: Number of *suboptimal* instances for all implementations of *LPO* and Wolsey's algorithm on random testbeds. The total number of instances in each testbed is also given in the first column.

	LPO	*LPO*/OO	*LPO*/OOF	Wol	Wol/OO	Wol/OOF	SD	SR
ATS	1.000	1.000	1.000	1.003	1.005	1.000	1.000	1.000
CapVC	1.000	1.001	1.001	1.012[§]	1.009	1.001	1.027	1.045
CapSC	1.013	1.013	1.013	1.073	1.085	1.068	1.250	1.760

Table 2: Average *ALG*:*OPT* ratio for all implementations of *LPO* and Wolsey's algorithms on the random testbeds.

shared machine. However, we note that Wolsey's algorithm involves max flow computations while *LPO* calls the relaxation of min-edge cost flow computations. On some of the larger graphs, the vanilla implementations of Wolsey and *LPO* did not complete, and only the results of their optimized versions are reported. For the sake of notational convenience, ALG/OO will refer to the algorithm ALG with optimization by order. ALG/OOF refers to ALG with both optimization by order and optimization via freezing.

4.1 Solution Quality On the scheduling instances (line 1 of Table 1), *LPO*, SD and SR returned optimal solutions every time. On the other hand, Wolsey's algorithm was suboptimal on several instances, even with intelligent ordering of the sets. In some cases, intelligent ordering even hurt Wolsey's solution due to the change in tie-breaking. On the bright side, all of Wolsey's suboptimalities were corrected by freezing. On CapVC instances, *LPO* underperformed on one instance while Wolsey's algorithm was suboptimal on seven (six of which were corrected with freezing). While this is comparable to the number of instances on which SD and SR were suboptimal, Table 2 demonstrates that Wolsey's solutions were on average significantly closer to optimal than those of SD and SR: Wolsey's algorithm attained an average 1.009-approximation (1.001 with freezing), while SD and SR were on average within 1.027 and 1.045 of optimal. *LPO* and Wolsey's algorithms maintained similar performance on the CapSC testbed, though the freezing optimization did little to improve the solution.

For the DIMACS and real world graphs, the optimized version of *LPO* achieved optimality on all but one of them while Wolsey's algorithm underachieved on at least two graphs, failing to complete on one other. Nevertheless, the performance is reasonable, relative to the performance of SD and SR on those graphs. See Tables 8, 9 and 10 in Appendix A for more.

It is clear that though there exist no worst-case guarantees for *LPO*, it returns optimal solutions significantly more often than Wolsey's implementations. On all implementations, whenever any of the implementations were suboptimal, they were never so by much; this is in stark contrast to SD and SR solutions (see Table 2) and only confirms what has already been long accepted in practice: Wolsey's algorithm on the whole performs much better than its theoretical bound would suggest.

We point out that the freezing optimization did not hurt the quality of solution in any of the random instances. At least for *LPO*, this is not surprising. The *LPO*'s subroutine involves solving the LP relaxation of the problem; this is the exact LP relaxation that the freezing step solves, and the variables that are frozen to zero are in fact the sets that *LPO* would have discarded first anyway.

4.2 Computational Efficiency On the ATS, CapVC and DIMACS instances, the vanilla implementation of *LPO* required significantly fewer subroutine calls than its Wolsey counterpart; the reverse is true for the CapSC instances. This is to be expected, since by design, *LPO* and Wolsey's algorithm are complementary in their approaches. On instances for which *LPO*'s final solution consists of a small number of sets, *LPO* will make many calls. Similarly, on instances where feasible covers tend to be large, Wolsey's algorithm will involve many more iterations. The multicover property of the ATS instances results in higher cardinality solutions. The sparsity of the CapVC instances, i.e. that each edge can be covered by at most two vertices, leads to a similar effect.

See Appendix A for tables on the actual average number of subroutines required by each implementation for each random testbed. It also contains statistics on the DIMACS graphs, the NetSci graph and the Gnutella graph, specifically, the actual solution values acquired as well as the number of subroutine calls required by each of the implementations.

4.2.1 Optimization: Ordering sets This particular optimization yields significant improvement in the number of calls, though the extent of its effect varies from testbed to testbed. On instances where Wolsey's vanilla algorithm performed better than *LPO*'s, intelligently ordering sets leveled the playing field. The reverse is not true; on instances where vanilla *LPO* already involved fewer calls than vanilla Wolsey, *LPO*/OO continued to excel.

Table 3 contains the average percentage reduction per implemention, per generated testbed. On ATS instances, *LPO* enjoyed greater improvement than Wolsey. For example, on the medium set, the optimization reduced *LPO*'s average number of calls from 3700 to 271, while it only reduced Wolsey's average number of calls from 9699 to 1110. We point out that Wolsey's vanilla implementation did not finish on one instance of the medium CapVC set, so its percentage reductions are not reported. On all CapSC instances, this optimization in some sense leveled the playing field for *LPO*. *LPO* observed an average 91 percent reduction in the number of calls while Wolsey witnessed only a 34 percent reduction. However, this reduction is misleading; it is only because vanilla *LPO* involved so many more calls than vanilla Wolsey. For example, on the medium data set, the optimization reduced *LPO*'s average number of calls from 7880 calls to 314 calls and Wolsey's from 475 calls to 331. See Tables 5 and 6 in Appendix A for more.

On each DIMACS instances on which both vanilla implementations completed, the number of calls invoked by *LPO* was at least fifty percent less than the number of calls made by Wolsey. Intelligently ordering the sets yielded an average 58 percent reduction in *LPO*'s number of calls and a 67 percent reduction in that of Wolsey's. Because each edge can be covered by at most two vertices, the final solution for both *LPO* and Wolsey tended to be of higher cardinality, relative to the total number of vertices. This explains why *LPO* on average made fewer calls than Wolsey, even with optimizations applied. On the other hand, the number of calls made by Wolsey is significantly less on the NetSci graph than that of *LPO*, at least on their optimized implementations.

4.2.2 Optimization: Freezing set variables By and large, the freezing optimization further improved the performance of *LPO* and Wolsey, giving *LPO* the edge over Wolsey. In fact, for most of the random testbeds, *LPO*/OOF involved on average fewer subroutine calls than Wolsey/OOF (see Tables 5 and 6 in Appendix A). The discussion that follows pertains to the percentage reduction in number of calls.

For each ATS instance, freezing yielded a feasible integral solution, resulting in identical performances by *LPO*/OOF and Wolsey/OOF. Though there exist scheduling instances for which the fractional solution is not integral, these examples may be few in number; none of them were present in the random testbed. On the CapVC testbed, freezing witnessed an additional 44 percent reduction in *LPO*'s performance, for a combined 93 percent reduction in tandem with the ordering optimization. Wolsey's algorithm saw a total 97 percent reduction in the number of calls, 22 percent attributed to the freezing optimization. For the CapSC instances, the freezing step further reduced *LPO*'s number of calls to 98 percent that of vanilla *LPO*. While intelligently ordering the sets reduced Wolsey's number of subroutine calls by about 34 percent, the freezing step additionally reduced it by another 45, for a combined 79 percent reduction in the number of calls. This suggests that the impact of freezing on Wolsey's algorithm, lack of theoretical guarantee notwithstanding, is comparable to that of imposing an intelligent orders on sets, at least for Capacitated Set Cover.

Freezing had negligible effect on most of the DIMACS instances, since the fractional solutions had few integral values. However, the opposite was true for both the NetSci graph and the Gnutella peer-to-peer network, the fraction solution of the latter achieving integrality. Thus, frozen implementations of both *LPO* and Wolsey's algorithm were impressive, in contrast to their less-optimized counterparts, all of which timed out after several hours. See Tables 9 through 12 in Appendix A.

5 Conclusions

In this work, we introduce the Cov-MECF framework and study several important covering problems in the context of this model. We introduce the heuristic *LPO*, which queries the fractional solution to guide the iterative removal of sets from a working cover. We empirically demonstrate that *LPO* returns solutions that are closer to optimal than those of Wolsey's algorithm, while the required number of subroutine calls is largely dictated by the nature of the input. However, with the freezing optimization, *LPO* almost unilaterally requires fewer subroutine calls than Wolsey's algorithm,

	LPO/OO	*LPO*/OOF	Wol/OO	Wol/OOF
ATS	0.812	0.996	0.777	0.998
CapVC	0.492	0.932	0.747[§]	0.967
CapSC	0.912	0.977	0.338	0.792

Table 3: Average percentage reduction in the number of subroutine calls.

though the latter still enjoys heavy benefits from freezing. These results solidly establish *LPO* as a competitive (if not superior) alternative to Wolsey's greedy algorithm. In general, we find that freezing yields a significant reduction in the number of calls, while very rarely harming the quality of the solution.

It remains to further explore the value of *LPO* and the freezing optimizations for other covering problems, for example the unrelated machine activation problem introduced by Khuller et. al. [22]. It is also unclear whether these contributions extend beyond covering problems, e.g. to probabilistic models which exhibit similar graph structure to Cov-MECF.

References

[1] Ravindra K. Ahuja, Magnanti, Thomas L., and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[2] Phil Ainsworth, Miguel Echenique, Bob Padzieski, Claudio Villalobos, Paul Walters, and Debbie Landon. Going green with IBM systems director active energy manager. *IBM Red Paper*, September 2008.

[3] Nikhil Bansal and Kirk Pruhs. The geometry of scheduling. In *Proceedings of 51st Annual IEEE Symposium on Foundations of Computer Science*, pages 407–414, 2010.

[4] Nikhil Bansal and Kirk Pruhs. Weighted geometric set multi-cover via quasi-uniform sampling. In *Proceedings of the 20th Annual European Symposium of Algorithms*, pages 145–156, 2012.

[5] Hervé Brönnimann and Michael T. Goodrich. Almost optimal set covers in finite VC-dimension. *Discrete and Computational Geometry*, 14(1):463–479, 1995.

[6] Jessica Chang, Harold N. Gabow, and Samir Khuller. A model for minimizing active processor time. In *Proceedings of the 20th Annual European Symposium on Algorithms*, pages 289–300, 2012.

[7] Julia Chuzhoy and Joseph (Seffi) Naor. Covering problems with hard capacities. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 481 – 489, 2002.

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

[9] Erik D. Demaine and Morteza Zadimoghaddam. Scheduling to minimize power consumption using submodular functions. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–29, 2010.

[10] Guy Even, Guy Kortsarz, and Wolfgang Slany. On network design problems: fixed cost flows and the covering steiner problem. *ACM Transactions on Algorithms*, 1(1):74–101, July 2005.

[11] Guy Even, Dror Rawitz, and Shimon (Moni) Shahar. Hitting sets when the VC-dimension is small. *Information Processing Letters*, pages 358–362, 2005.

[12] Uriel Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, July 1998.

[13] Rajiv Gandhi, Eran Halperin, Samir Khuller, Guy Kortsarz, and Aravind Srinivasan. An improved approximation algorithm for vertex cover with hard capacities. *Journal of Computer and System Sciences*, 72(1):16 – 33, 2006.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[15] Ravi A. Giri and Anand Vanchi. Increasing data center efficiency with server power measurements. *Intel Technical Report*, January 2010.

[16] Amit Goyel, Wei Lu, and Laks V.S. Lakshmanan. CELF++: Optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 19th International World Wide Web Conference*, Hyderabad, India, 2011.

[17] Sudipto Guha, Refael Hassin, Samir Khuller, and Einat Or. Capacitated vertex covering with applications. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 858–865, 2002.

[18] Akshay Java, Pranam Kolari, Tim Finin, and Tim Oates. Modeling the spread of influence on the blogosphere. In *Proceedings of the 14th International World Wide Web Conference*, 2006.

[19] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[20] David S. Johnson and Michael A. Trick (eds.). Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. `www.dimacs.rutgers.edu/Challenges`, 1991.

[21] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

[22] Samir Khuller, Jian Li, and Barna Saha. Energy efficient scheduling via partial shutdown. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1360–1372, 2010.

[23] Sven O. Krumke, Hartmut Noltemeier, S. Schwarz, Hans-Christoph Wirth, and R. Ravi. Flow improvement and network flows with fixed costs. In *Proceedings of the International Conference of Operations Research (OR'98)*, Zürich, 1998.

[24] Jure Leskovec. SNAP library. `http://snap.stanford.edu/snap/license.html`.

[25] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–429, 2007.

[26] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975.

[27] Robert Lupton, Miller Maley, and Neal E. Young. Data collection for the Sloan Digital Sky Survey: A network-flow heuristic. *Journal of Algorithms*, 27(2):339–356, 1998.

[28] Nabil H. Mustafa and Saurabh Ray. PTAS for geometric hitting set problems via local search. In *Proceedings of the 25th Annual Symposium on Computational Geometry*, pages 17–22, 2009.

[29] Mark E. Newman. Coauthorships in network science. *Physical Review E*, 74:036104, Sep 2006.

[30] Barna Saha and Samir Khuller. Set cover revisited: Hypergraph cover with hard capacities. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming*, pages 762–773, 2012.

[31] Kasturi Varadarajan. Weighted geometric set cover via quasi-uniform sampling. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, pages 641–648, 2010.

[32] Laurence Wolsey. An analysis of the greedy algorithm for the submodular set covering problem. *Combinatorica*, 2:385–393, 1982.

A Appendix: Additional Graphs and Charts

This section contains statistics regarding solution quality and percentage reductions in the number of flow evaluations. What follows is a description of the size of the randomly generated testbeds. Tables 5 and 6 contain the actual average number of calls (as opposed to the percentage reductions that were reported in Table 3 in Section 4). Following that are statistics regarding the larger graphs. The first five instances are taken from the DIMACS 2nd Implementation Challenge. The sixth graph is one of author collaborations for Network Science [29], and the last is one of connections in a peer-to-peer graph [24].

		num instances	avg n	avg m
ATS	small	15	39	43
	medium	5	77	136
CapVC	small	20	229	30
	medium	3	8492	191
CapSC	small	15	36	47
	medium	5	74	121

Table 4: Descriptions of the generated testbeds.

		LPO	*LPO*/OO	*LPO*/OOF
ATS	small	440.07	85.40	2.00
	medium	3700.80	271.00	2.00
CapVC	small	118.20	47.20	8.53
	medium	617.33	206.00	14.25
CapSC	small	1141.53	108.73	26.13
	medium	7880.20	313.60	84.40

Table 5: *LPO*'s average number of calls on randomly generated testbeds.

		Wol	Wol/OO	Wol/OOF
ATS	small	968.73	243.00	2.00
	medium	9699.20	1109.80	2.00
CapVC	small	534.73	120.60	13.60
	medium	6366.00	568.00	24.25
CapSC	small	182.80	118.80	39.07
	medium	474.60	330.60	82.60

Table 6: Wolsey's average number of calls for randomly generated testbeds.

[¶] Computing the optimal solution timed out. Freezing revealed that the fractional solution was integral.

	n	m	Num Frozen
johnson8-2-4	210	28	0
hamming6-4	704	64	0
hamming6-2	1824	64	0
mulsoli4	3946	185	11
school1	19095	385	23
NetSci	2742	1589	1300
Gnutella	20777	6301	6301

Table 7: Descriptions of DIMACS, NetSci and Gnutella instances. The right column gives the number of variables that would be frozen to 0 or 1 if the freezing optimization were applied.

	OPT	SD	SR
johnson8-2-4	21	28	28
hamming6-4	52	64	64
hamming6-2	62	64	64
mulsoli4	99	111	161
school1	344	376	379
NetSci	944	1022	1020
Gnutella	4434¶	4434	4434

Table 8: Optimal solutions and performance of scaling algorithms for DIMACS, NetSci and Gnutella graphs. The scaling algorithms provide additional perspective on the solution quality of *LPO* and Wolsey.

B Cov-MECF Program

The integer program capturing Cov-MECF is straightforward and is given below.

$$
\begin{array}{lrcll}
\text{min.} & \sum_{y\in Y} \kappa_y u_y & & & \\
\text{s.t.} & \sum_{y\in\delta(x)} f_{x,y} & = & f_{s,x} & \forall\ x \in X \\
& \sum_{x\in\delta(y)} f_{y,t} & = & f_{y,t} & \forall\ y \in Y \\
& f_{s,x} & \leq & c_x & \forall\ x \in X \\
& f_{y,t} & \leq & c_y u_y & \forall\ y \in Y \\
& f_{x,y} & \leq & 1 & \forall\ (x,y) \in E \\
& \sum_x f_{s,x} & \geq & f^* & \\
& f_{x,y} & \leq & u_y & \forall\ (x,y) \in E \\
& u_y & \in & \{0,1\} & \forall\ y \in Y \\
& u_y, f_{s,x}, f_{x,y}, f_{y,t} & \geq & 0 & \forall\ x \in X, y \in Y
\end{array}
$$

	LPO	*LPO*/OO	*LPO*/OOF
johnson8-2-4	21	21	21
hamming6-4	52	52	52
hamming6-2	62	62	62
mulsoli4	99	99	99
school1	–	–	347
NetSci	–	944	944
Gnutella	–	–	4434

Table 9: Value of *LPO*'s solutions on DIMACS, NetSci and Gnutella graphs.

	Wol	Wol/OO	Wol/OOF
johnson8-2-4	23	24	24
hamming6-4	56	56	56
hamming6-2	62	62	62
mulsoli4	–	99	99
school1	–	–	–
NetSci	–	947	944
Gnutella	–	–	4434

Table 10: Value of solutions returned by Wolsey's algorithm on DIMACS, NetSci and Gnutella graphs.

	LPO	*LPO*/OO	*LPO*/OOF
johnson8-2-4	196	56	58
hamming6-4	754	200	202
hamming6-2	189	132	134
mulsoli4	12354	452	430
school1	–	–	1603
NetSci	–	13178	8697
Gnutella	–	–	2

Table 11: Number of flow computations made by *LPO* on DIMACS, NetSci and Gnutella graphs.

	Wol	Wol/OO	Wol/OOF
johnson8-2-4	437	179	181
hamming6-4	2156	399	401
hamming6-2	2201	902	904
mulsoli4	–	754	742
school1	–	–	–
NetSci	–	5508	1295
Gnutella	–	–	2

Table 12: Number of flow computations made by Wosley's algorithm on DIMACS, NetSci and Gnutella graphs.