Algorithmica
DOI 10.1007/s00453-012-9629-3

Greedy Δ-Approximation Algorithm for Covering with Arbitrary Constraints and Submodular Cost

Christos Koufogiannakis · Neal E. Young

Received: 31 August 2010 / Accepted: 20 February 2012

Abstract This paper describes a simple greedy Δ-approximation algorithm for any covering problem whose objective function is submodular and non-decreasing, and whose feasible region can be expressed as the intersection of arbitrary (closed upwards) covering constraints, each of which constrains at most Δ variables of the problem. (A simple example is VERTEX COVER, with $\Delta = 2$.) The algorithm generalizes previous approximation algorithms for fundamental covering problems and online paging and caching problems.

Keywords Covering · Linear programming · Approximation algorithms · Local ratio · Primal-dual · Vertex cover · Set cover · Integer linear programming · Online algorithms · Competitive analysis · Submodular cost · Paging · Caching

1 Introduction and Summary

The classification of general techniques is an important research program within the field of approximation algorithms. What abstractions are useful for capturing a wide variety of problems and analyses? What are the scopes of, and the relationships between, the various algorithm-design techniques such as the primal-dual method, the local-ratio method [9], and randomized rounding? Which problems admit optimal and fast greedy approximation algorithms [11, 12, 26]? What general techniques are useful for designing online algorithms? What is the role of locality among constraints and variables [9, 46, 53]? We touch on these topics, exploring a simple greedy algorithm for a general class of covering problems. The algorithm has approximation ratio Δ provided each covering constraint in the instance constrains only Δ variables.

Throughout the paper, $\bar{\mathbb{R}}_{\geq 0}$ denotes $\mathbb{R}_{\geq 0} \cup \{\infty\}$ and $\bar{\mathbb{Z}}_{\geq 0}$ denotes $\mathbb{Z}_{\geq 0} \cup \{\infty\}$.

C. Koufogiannakis · N.E. Young
Department of Computer Science and Engineering, University of California, Riverside, Riverside, CA, USA

Published online: 14 March 2012

The conference version of this paper is [44].

Definition 1 (Submodular-Cost Covering) An instance is a triple $(c, \mathcal{C}, U)$, where

- The cost function $c : \bar{\mathbb{R}}^n_{\geq 0} \to \bar{\mathbb{R}}_{\geq 0}$ is submodular,[1] continuous, and non-decreasing.
- The constraint set $\mathcal{C} \subseteq 2^{\bar{\mathbb{R}}^n_{\geq 0}}$ is a collection of covering constraints, where each constraint $S \in \mathcal{C}$ is a subset of $\bar{\mathbb{R}}^n_{\geq 0}$ that is closed upwards[2] and under limit. We stress that each S may be non-convex.
- For each $j \in [n]$, the domain U_j (for variable x_j) is any subset of $\bar{\mathbb{R}}_{\geq 0}$ that is closed under limit.

The problem is to find $x \in \bar{\mathbb{R}}^n_{\geq 0}$, minimizing $c(x)$ subject to $x_j \in U_j$ $(\forall j \in [n])$ and $x \in S$ $(\forall S \in \mathcal{C})$.

The definition assumes the objective function $c(x)$ is defined over all $x \in \bar{\mathbb{R}}^n_{\geq 0}$, even though the solution space is constrained to $x \in U$. This is unnatural, but any c that is properly defined on U extends appropriately[3] to $\bar{\mathbb{R}}^n_{\geq 0}$. In the cases discussed here c extends naturally to $\bar{\mathbb{R}}^n_{\geq 0}$ and this issue does not arise.

We call this problem Submodular-Cost Covering.[4]

For intuition, consider the well-known Facility Location problem. An instance is specified by a collection of customers, a collection of facilities, an opening cost $f_j \geq 0$ for each facility, and an assignment cost $d_{ij} \geq 0$ for each customer i and facility $j \in N(i)$. The problem is to open a subset $\mathcal{F}$ of the facilities so as to minimize the cost to open facilities in $\mathcal{F}$ (that is, $\sum_{j \in \mathcal{F}} f_j$) plus the cost for each customer to reach its nearest open, admissible facility (that is, $\sum_i \min\{d_{ij} \mid j \in \mathcal{F}\}$). This is equivalent to Submodular-Cost Covering instance $(c, \mathcal{C}, U)$, with

- a variable x_{ij} for each customer i and facility j, with domain $U_{ij} = \{0, 1\}$,
- for each customer i, (non-convex) constraint $\max_{j \in N(i)} x_{ij} \geq 1$ (the customer is assigned a facility),
- and (submodular) cost $c(x) = \sum_j f_j \max_i x_{ij} + \sum_{i,j} d_{ij} x_{ij}$ (opening cost plus assignment cost).

[1] Formally, $c(x) + c(y) \geq c(x \wedge y) + c(x \vee y)$, where $x \wedge y$ (and $x \vee y$) are the component-wise minimum (and maximum) of x and y. Intuitively, there is no positive synergy between the variables: let $\partial_j c(x)$ denote the rate at which increasing x_j would increase $c(x)$; then, increasing x_i (for $i \neq j$) does not increase $\partial_j c(x)$. Any separable function $c(x) = \sum_j c_j(x_j)$ is submodular, the product $c(x) = \prod_j x_j$ is not. The maximum $c(x) = \max_j x_j$ is submodular, the minimum $c(x) = \min_j x_j$ is not.

[2] If $y \geq x$ and $x \in S$, then $y \in S$, perhaps the minimal requirement for a constraint to be called a "covering" constraint.

[3] One way to extend c from U to $\bar{\mathbb{R}}_{\geq 0}$: take the cost of $x \in \bar{\mathbb{R}}^n_{\geq 0}$ to be the expected cost of $\tilde{x}$, where $\tilde{x}_j$ is rounded up or down to its nearest elements a, b in U_j such that $a \leq x_j \leq b$: take $\tilde{x}_j = b$ with probability $\frac{b - x_j}{b - a}$, otherwise take $\tilde{x}_j = a$. If a or b doesn't exist, let $\tilde{x}_j$ be the one that does. As long as c is non-decreasing, sub-modular, and (where appropriate) continuous over U, this extension will have these properties over $\bar{\mathbb{R}}^n_{\geq 0}$.

[4] Changed from "Monotone Covering" in the conference version [44] due to name conflicts.

problem	approximation ratio	method	where	comment	
VERTEX COVER	$2 - \ln\ln\widehat{\Delta}/\ln\widehat{\Delta}$	local ratio	[30]	see also [6, 24, 28, 29, 31, 34, 40, 50]	
SET COVER	Δ	LP; greedy [33, 34]; [5]		$\Delta = \max_i \lvert\{j \mid A_{ij} > 0\}\rvert$	⋆
CIP-01 w/$A_{ij} \in \mathbb{Z}_{\geq 0}$	$\max_i \sum_j A_{ij}$	primal-dual	[10, 27]	quadratic time	⋆
CIP-UB	Δ	ellipsoid	[16, 54, 55]	KC-ineq., high-degree-poly time	⋆
SUBMOD-COST COVER	Δ	greedy	[our §2]	$\min\{c(x) \mid x \in S\ (\forall S \in \mathcal{C})\}$	new
SET/VERTEX COVER	Δ	greedy	[our §7.1]	linear time	
FACILITY LOCATION	Δ	greedy	[our §7.1]	linear time	new
CMIP-UB	Δ	greedy	[our §7.2]	near-linear time	new
2-STAGE CMIP-UB	Δ	greedy	[our §7.3]	quadratic time	new

Fig. 1 Some Δ-approximation algorithms for covering problems. "⋆" = generalized or strengthened here

A Greedy Algorithm for Submodular-Cost Covering (Sect. 2) The core contribution of the paper is a greedy Δ-approximation algorithm for the problem, where Δ is the maximum number of variables that any individual covering constraint S in $\mathcal{C}$ constrains.

For $S \in \mathcal{C}$, let $\mathsf{vars}(S)$ contain the indices of variables that S constrains (i.e., $j \in \mathsf{vars}(S)$ if membership of x in S depends on x_j). The algorithm is roughly as follows.

> *Start with an all-zero solution x, then repeat the following step until all constraints are satisfied: Choose any not-yet-satisfied constraint S. To satisfy S, raise each x_j for $j \in \mathsf{vars}(S)$ (i.e., raise the variables that S constrains), so that each raised variable's increase contributes the same amount to the increase in the cost.*

Section 2 gives the full algorithm and its analysis.

Fast Implementations (Sect. 7) One important special case of SUBMODULAR-COST COVERING is COVERING INTEGER LINEAR PROGRAMS with upper bounds on the variables (CIP-UB), that is, problems of the form $\min\{c \cdot x \mid x \in \mathbb{Z}^n_{\geq 0}; Ax \geq b; x \leq u\}$ where each c_j, b_i, and A_{ij} is non-negative. This is a SUBMODULAR-COST COVERING instance $(c, U, \mathcal{C})$ with variable domain $U_j = \{0, 1, \ldots, u_j\}$ for each j and a covering constraint $A_i x \geq b_i$ for each i, and Δ is the maximum number of non-zeros in any row of A.

Section 7 describes a nearly linear-time implementation for a generalization of this problem: COVERING MIXED INTEGER LINEAR PROGRAMS with upper bounds on the variables (CMIP-UB). As summarized in the bottom half of Fig. 1, Sect. 7 also describes fast implementations for other special cases: VERTEX COVER, SET COVER, FACILITY LOCATION (linear time); and two-stage probabilistic CMIP-UB (quadratic time).

Related Work: Δ-Approximation Algorithms for Classical Covering Problems (Top Half of Fig. 1) See e.g. [35, 62] for an introduction to classical covering problems. For VERTEX COVER[5] and SET COVER in the early 1980's, Hochbaum gave a Δ-approximation algorithm based on rounding an LP relaxation [33]; Bar-Yehuda and Even gave a linear-time greedy algorithm (a special case of the algorithms here) [5]. A few years later Hall and Hochbaum gave a quadratic-time primal-dual algorithm

[5] SET MULTICOVER is CIP-UB restricted to $A_{ij} \in \{0, 1\}$; SET COVER is SET MULTICOVER restricted to $b_i = 1$; VERTEX COVER is SET COVER restricted to $\Delta = 2$.

online problem	competitive ratio	deterministic online		comment	
SKI RENTAL	2; $\frac{e}{e-1}$	det.; random	[39, 47]		⋆
PAGING	$k = \Delta$	potential function	[56, 59]	e.g. LRU, FIFO, FWF, Harmonic	⋆
CONNECTION CACHING	$O(k)$	reduction to paging	[1, 20]		⋆
WEIGHTED CACHING	k	primal-dual	[56, 63, 64]	e.g. Harmonic, Greedy-Dual	⋆
FILE CACHING	k	primal-dual	[15, 65, 66]	e.g. Greedy-Dual-Size, Landlord	⋆
UNW. SET COVER	$O(\log \Delta \log \frac{n}{\text{opt}})$	primal-dual	[13, 14]	unweighted	
CLP	$O(\log n)$	fractional	[13, 14]	$\min\{c \cdot x \mid Ax \geq b; x \leq u\}$,	
SUBMOD-COST COVER	Δ	potential function	[our §2]	includes the above, CMIP-UB	new
UPGRADABLE CACHING	$d + k$	by reduction	[our §3]	d components, k files in cache	new

Fig. 2 Δ-competitive online paging and caching. "⋆" = generalized or strengthened here

for SET MULTICOVER [27]. In the late 1990's, Bertsimas and Vohra further generalized that result with a quadratic-time primal-dual algorithm for COVERING INTEGER PROGRAMS with $\{0, 1\}$-variables (CIP-01), but restricted to integer constraint matrix A and with approximation ratio $\max_i \sum_j A_{ij} \geq \Delta$ [10]. In 2000, Carr et al. gave the first Δ-approximation for CIP-01 [16]. In 2009 (independently of our work), Pritchard extended that result to CIP-UB [54, 55]. Both [16] and [54, 55] use the (exponentially many) Knapsack-Cover (KC) inequalities to obtain integrality gap[6] Δ, and their algorithms use the ellipsoid method, so have high-degree-polynomial running time.

As far as we know, SET COVER is the most general special case of SUBMODULAR-COST COVERING for which any nearly linear time Δ-approximation algorithm was previously known, while CIP-UB is the most general special case for which any polynomial-time Δ-approximation algorithm was previously known.

Independently of this paper, Iwata and Nagano give Δ-approximation algorithms for variants of VERTEX COVER, SET COVER, and EDGE COVER with submodular (and possibly decreasing!) cost [36].

Online Covering, Paging, and Caching (Sect. 3) In *online* covering (following, e.g. [2, 13, 14]), the covering constraints are revealed one at a time in any order. An online algorithm must choose an initial solution x, then, as each constraint "$x \in S$" is revealed, must increase variables in x to satisfy the constraint, without knowing future constraints and without decreasing any variable. The algorithm has *competitive ratio* Δ if the cost of its final solution is at most Δ times the optimal (offline) cost (plus a constant that is independent of the input sequence). The algorithm is said to be Δ-*competitive*.

The greedy algorithm here is a Δ-competitive online algorithm for SUBMODULAR-COST COVERING.

As recently observed in [2, 13, 14], many classical online paging and caching problems reduce to online covering (usually online SET COVER). Via this reduction, the algorithm here generalizes many classical deterministic online paging and caching algorithms. These include LRU and FWF for PAGING [59], BALANCE and GREEDY DUAL for WEIGHTED CACHING [17, 63, 64], LANDLORD [65, 66] (a.k.a. GREEDY DUAL SIZE) [15], for

[6] The standard LP relaxation has arbitrarily large gap (e.g. $\min\{x_1 \mid 10x_1 + 10x_2 \geq 11; x_2 \leq 1\}$ has gap 10).

Carr et al. [16] state (without details) that their CIP-01 result extends CIP-UP, but it is not clear how (see [54, 55]).

File Caching, and algorithms for Connection Caching [1, 20–22] (all results marked with "⋆" in Fig. 2).

As usual, the competitive ratio Δ is the cache size, commonly denoted k, or, in the case of File Caching, the maximum number of files ever held in cache (which is at most the cache size).

Section 3 illustrates this connection using Connection Caching as an example.

Section 3 also illustrates the generality of online Submodular-Cost Covering by describing a $(d + k)$-competitive algorithm for a new class of *upgradable* caching problems, in which the online algorithm chooses not only which pages to evict, but also how to pay to upgrade d hardware parameters (e.g. cache size, CPU, bus, network speeds, etc.) to reduce later costs and constraints (somewhat like Ski Rental [39] and multi-slope Ski Rental [47]—special cases of online Submodular-Cost Covering with $\Delta = 2$).

Section 4 describes a natural randomized generalization of the greedy algorithm (Algorithm 2), with even more flexibility in incrementing the variables. This yields a *stateless* Δ-competitive online algorithm for Submodular-Cost Covering, generalizing Pitt's Vertex Cover algorithm [4] and the Harmonic k-server algorithm as it specializes for Paging and Weighted Caching [56].

Related Work: Randomized Online Algorithms For most online problems here, no deterministic online algorithm can be better than Δ-competitive (where $\Delta = k$), but better-than-Δ-competitive *randomized* online algorithms are known. Examples include Ski Rental [39, 47], Paging [25, 49], Weighted Caching [2, 15], Connection Caching [20], and File Caching [3]. Some cases of online Submodular-Cost Covering (e.g. Vertex Cover) are unlikely to have better-than-Δ-competitive randomized algorithms. It would be interesting to classify which cases admit better-than-Δ-competitive randomized online algorithms.

Relation to Local-Ratio and Primal-Dual Methods (Sect. 6) Sect. 6 describes how the analyses here can be recast (perhaps at some expense in intuition) in either the local-ratio framework or (at least for linear costs) the primal-dual framework. Local ratio is usually applied to problems with variables in $\{0, 1\}$; the section introduces an interpretation of local ratio for more general domains, based on residual costs.

Similarly, the Knapsack Cover (KC) inequalities are most commonly used for problems with variables in $\{0, 1\}$, and it is not clear how to extend the KC inequalities to more general domains (e.g. from CMIP-01 to CMIP-UB). (The standard KC inequalities suffice for $O(\log(\widehat{\Delta}))$-approximation of CMIP-UB [41], but may require some modification to give Δ-approximation of CMIP-UB [54, 55].) The primal-dual analysis in Sect. 6 uses a new linear program (LP) relaxation for Linear-Cost Covering that may help better understand how to extend the KC inequalities.

Section 6 also discusses how the analyses here can be interpreted via a certain class of valid linear inequalities, namely inequalities that are "local" in that they can be proven valid by looking only at each single constraint $S \in \mathcal{C}$ in isolation.

Related Work: Hardness Results, Log-Approximation Algorithms Even for simple covering problems such as Set Cover, no polynomial-time $(\Delta - \varepsilon)$-approximation

algorithms (for any constant $\varepsilon > 0$) are currently known for small (e.g. constant) Δ. A particularly well studied special case, with $\Delta = 2$, is Vertex Cover, for which some complexity-theoretic evidence suggests that such an algorithm may not exist [6, 24, 28–31, 34, 40, 50].

For instances where Δ is large, $O(\log \widehat{\Delta})$-approximation algorithms may be more appropriate, where $\widehat{\Delta}$ is the maximum number of constraints in which any variable occurs. Such algorithms exist for set cover [19, 37, 38, 48, (greedy, 1975)] for CIP [60, 61, (ellipsoid, 2000)] and CIP-UB [41, (ellipsoid/KC inequalities, 2005)]. It is an open question whether there is a fast *greedy* $O(\log \widehat{\Delta})$-approximation algorithm handling all of these problems (via, say, CIP-UB).

Recent works with log-approximations for submodular-cost covering problems include [18, 32, 57, 58]. Most of these have high-degree-polynomial run time. For example, the $(\ln n)$-approximation algorithm for two-stage probabilistic Set-Cover [32] requires solving instances of Submodular Function Minimization [51, 52], which requires high-degree-polynomial run time. ([32] also claims a related 2-approximation for two-stage probabilistic Vertex Cover without details.)

Related Work: Distributed and Parallel Algorithms Distributed and parallel approximation algorithms for covering problems are an active area of study. The simple form of the greedy algorithm here makes it particularly amenable for distributed and/or parallel implementation. In fact, it admits poly-log-time distributed and parallel implementations, giving (for example) the first poly-log-time 2-approximation algorithms for the well-studied (weighted) Vertex Cover and Maximum Weight Matching problems. See [42, 43, 45] for details and related results.

Organization Section 2 gives the greedy algorithm for Submodular-Cost Covering (Algorithm 2) and proves that is has approximation ratio Δ. Section 3 describes applications to online problems. Section 4 describes randomized generalizations of the greedy algorithm, including a stateless online algorithm. Sections 5 and 6 explain how to view the analysis via the local-ratio and primal-dual methods. Section 7 details fast implementations for some special cases. After Sect. 2, each section may be read independently of the others.

2 Greedy Algorithm for Submodular-Cost Covering (Algorithm 2)

This section gives the full greedy algorithm for Submodular-Cost Covering (Algorithm 2) and the analysis of its approximation ratio. We assume Submodular-Cost Covering instances are given in *canonical form*:

Definition 2 (Canonical form) An instance $(c, U, \mathcal{C})$ is in *canonical form* if each variable domain is unrestricted (each $U_j = \bar{\mathbb{R}}_{\geq 0}$). Such an instance is specified by just the pair $(c, \mathcal{C})$.

This assumption is without loss of generality by the following reduction:

Observation 1 *For any* Submodular-Cost Covering *instance* $(c, U, \mathcal{C})$, *there is an equivalent canonical form instance* $(c, \mathcal{C}')$. *By "equivalent", we mean that any* x *that is feasible in* $(c, U, \mathcal{C})$ *is also feasible in* $(c, \mathcal{C}')$, *and that any* x' *that is minimally feasible in* $(c, \mathcal{C}')$ *is also feasible in* $(c, U, \mathcal{C})$.

Given any feasible solution x' to $(c, \mathcal{C}')$, one can compute a feasible solution x to $(c, U, \mathcal{C})$ with $c(x) \le c(x')$ by taking each $x_j = \max\{\alpha \in U_j \mid \alpha \le x_j\}$.

The reduction is straightforward and is given in the Appendix. The idea is to incorporate the variable-domain restrictions "$x_j \in U_j$" directly into each covering constraint $S \in \mathcal{C}$, replacing each occurrence of x_j in each S by $\max\{\alpha \in U_j \mid \alpha \le x_j\}$. For example, applied to a CIP-UB instance $(c, U, \mathcal{C})$ as described in the introduction, the reduction produces the canonical instance $(c, \mathcal{C}')$ in which each covering constraint $A_i x \ge b_i$ in $\mathcal{C}$ is replaced in $\mathcal{C}'$ by the stronger non-convex covering constraint

$$\sum_j A_{ij} \lfloor \min(x_j, u_j) \rfloor \ge b_i.$$

To satisfy these constraints, it doesn't help to assign any x_j a value outside of $U_j = \{0, 1, \ldots, u_j\}$: any minimal x satisfying the constraints in $\mathcal{C}'$ will also satisfy $x_j \in \{0, 1, \ldots, u_j\}$ for each j.

In the rest of the paper, we assume all instances are given in canonical form. To handle an instance $(c, U, \mathcal{C})$ that is not in canonical form, apply the above reduction to obtain canonical instance $(c, \mathcal{C}')$, use one of the algorithms here to compute a Δ-approximate solution x for $(c, \mathcal{C}')$, then compute vector x' as described after Observation 1.

Definition 3 For any covering constraint S and $x \in \mathbb{R}^n_{\ge 0}$, let "$x \le_S y$", "$x >_S y$", etc., mean that the operator holds coordinate-wise for coordinates in $\mathsf{vars}(S)$. E.g. $x \le_S y$ if $x_j \le y_j$ for all $j \in \mathsf{vars}(S)$.

Observation 2 *If* $x \in S$ *and* $y \ge_S x$, *then* $y \in S$.

The observation is true simply because S is closed upwards, and membership of y in S depends only on y_j for $j \in \mathsf{vars}(S)$. We use this observation throughout the paper.

To warm up the intuition for Algorithm 2, we first introduce and analyze a simpler version, Algorithm 1, that works only for linear costs. The algorithm starts with $x \leftarrow \mathbf{0}$, then repeatedly chooses any unmet constraint S, and, to satisfy S, raises all variables x_j with $j \in \mathsf{vars}(S)$ at rate $1/c_j$, until x satisfies S:

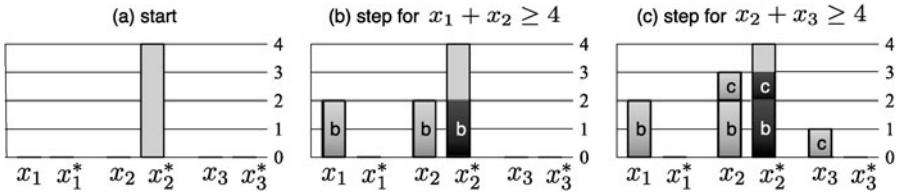

Fig. 3 Two steps of Algorithm 1, where $x^* = (0, 2, 0)$. Dark portions of x_2^* have been charged

Greedy algorithm for Linear-Cost Covering ALG. 1

Input: (linear) cost vector $c \in \mathbb{R}^n_{\geq 0}$, canonical constraints $\mathcal{C}$

Output: Δ-approximate solution x

1. Recall that $\mathsf{vars}(S)$ contains the indices of variables that S constrains.
2. Start with $x \leftarrow \mathbf{0}$, then, for each of the constraints $S \in \mathcal{C}$, in any order:
3. Just until $x \in S$, do:
4. for all $j \in \mathsf{vars}(S)$ simultaneously, raise x_j continuously at rate $1/c_j$.
5. Return x.

As the variables increase in Line 4, the cost of x increases at rate $|\mathsf{vars}(S)| \leq \Delta$ (each variable contributes to the cost increase at unit rate).[7] The proof of the approximation ratio relies on the following observation:

Observation 3 *Let y be any feasible solution. Consider an iteration for a constraint S. Unless the current solution x already satisfies S at the start of the iteration, at the end of the iteration, x has some variable x_k with $k \in \mathsf{vars}(S)$ such that $x_k \leq y_k$. (That is, $x \not\succ_S y$.)*

Proof At the start of the iteration, since y but not x satisfies S, Observation 2 implies that $x \not\geq_S y$. During the iteration, while Line 4 is raising the x_j for $j \in \mathsf{vars}(S)$, if at some moment $x \geq_S y$, then, since $y \in S$, it must be (by Observation 2) that $x \in S$ also, so at that moment Line 4 stops raising the variables, before $x >_S y$. □

As the variables increase in Line 4, Observation 3 implies that, for some x_k, the growing interval $[0, x_k]$ covers (at rate $1/c_k$) a larger and larger fraction of the corresponding interval $[0, x_k^*]$ in the optimal solution x^*. This allows the Δ-rate increase in the cost of x to be charged at unit rate to the cost of x^*, proving that the final cost of x is at most Δ times the cost of x^*.

For example, consider two iterations of Algorithm 1 on input $\min\{x_1 + x_2 + x_3 \mid x_1 + x_2 \geq 4; x_2 + x_3 \geq 4\}$ with optimal solution $x^* = (0, 4, 0)$, as shown in Fig. 3. The first iteration does a step for the first constraint, raising x_1 and x_2 by 2, and charging the cost increase of 4 to the $[0, 2]$ portion of x_2^*. The second iteration does a step for the second constraint, raising x_2 and x_3 both by 1, and charging the cost increase of 2 to the $[2, 3]$ portion of x_2^*.

[7] If some $c_j = 0$, then x_j is raised instantaneously to ∞ at cost 0, after which the cost of x increases at rate less than $|\mathsf{vars}(S)|$.

We generalize this charging argument by defining the *residual problem for the current x*, which is the problem of finding a minimum-cost augmentation of the current x to make it feasible. For example, after the first iteration of Algorithm 1 in the example above, the residual problem for $x = (2, 2, 0)$ is equivalent to $\min\{y_1 + y_2 + y_3 \mid y_1 + y_2 \geq 0;\ y_2 + y_3 \geq 2\}$. For notational simplicity, in the definition of the residual problem, instead of shifting each constraint, we (equivalently, but perhaps less intuitively) leave the constraints alone but modify the cost function (we charge y only for the part of y that exceeds x):[8]

Definition 4 (Residual problem) Given any Submodular-Cost Covering instance $(c, \mathcal{C})$, and any $x \in \bar{\mathbb{R}}^n_{\geq 0}$, define the *residual problem for x* to be the instance $(\tilde{c}_x, \mathcal{C})$ with cost function $\tilde{c}_x(y) = c(x \vee y) - c(x)$.

For $Q \subseteq \mathbb{R}^n_{\geq 0}$, define the *cost of Q in the residual problem for x* to be $\tilde{c}_x(Q) = \min_{y \in Q} \tilde{c}_x(y)$.

If Q is closed upwards, then $\tilde{c}_x(Q)$ equals $\min\{c(y) - c(x) \mid y \geq x,\ y \in Q\}$.

In all cases here Q is closed upwards, and we interpret $\tilde{c}_x(Q)$ as *the minimum increase in $c(x)$ necessary to raise coordinates of x to bring x into Q*. The residual problem $(\tilde{c}_x, \mathcal{C})$ has optimal cost $\tilde{c}_x(\bigcap_{S \in \mathcal{C}} S)$.

Here is the formal proof of the approximation ratio, as it specializes for Algorithm 1.

Lemma 1 (Correctness of Algorithm 1) *Algorithm 1 is a Δ-approximation algorithm for* Linear-Cost Covering.

Proof First consider the case when every cost c_j is non-zero. Consider an iteration for a constraint $S \in \mathcal{C}$.

Fix any feasible y. The cost $\tilde{c}_x(y)$ of y in the residual problem for x is the sum $\sum_j c_j \max(y_j - x_j, 0)$. As Line 4 raises each variable x_j for $j \in \mathsf{vars}(S)$ at rate $1/c_j$, by Observation 3, one of the variables being raised is an x_k such that $x_k < y_k$. For this k, the term $c_k \max(y_k - x_k, 0)$ in the sum is decreasing at rate 1. No terms in the sum increase. Thus, $\tilde{c}_x(y)$ decreases at rate at least 1.

Meanwhile, the cost $c(x)$ of x increases at rate $|\mathsf{vars}(S)| \leq \Delta$. Thus, the algorithm maintains the invariant $c(x)/\Delta + \tilde{c}_x(y) \leq c(y)$ (true initially because $c(\mathbf{0}) = 0$ and $\tilde{c}_{\mathbf{0}}(y) = c(y)$). Since $\tilde{c}_x(y) \geq 0$, this implies that $c(x) \leq \Delta c(y)$ at all times.

In the case that some $c_j = 0$ during an iteration, the corresponding x_j's are set instantaneously to ∞. This increases neither $c(x)$ nor $\tilde{c}_x(y)$, so the above invariant is still maintained and the conclusion still holds. □

The main algorithm (Algorithm 2, next) generalizes Algorithm 1 in two ways: First, the algorithm works with any submodular (not just linear) cost function. (This generalization is more complicated but technically straightforward.) Second, in each iteration, instead of increasing variables just until the constraint is satisfied, it chooses a *step size* $\beta \geq 0$ explicitly (we will see that this will allow a *larger* step than in

[8] Readers may recognize a similarity to the local-ratio method. This is explored in Sect. 5.

Algorithm 1). Then, for each $j \in \mathsf{vars}(S)$, it increases x_j maximally so that the cost $c(x)$ of x increases by (at most) β.

Greedy algorithm for Submodular-Cost Covering ALG. 2

Input: objective c, canonical constraints $\mathcal{C}$

Output: Δ-approximate solution x (provided conditions of Theorem 1 are met).

1. Let $x \leftarrow \mathbf{0}$.
2. While $\exists\ S \in \mathcal{C}$ such that $x \notin S$ do:
3. Choose any S such that $x \notin S$ and do $\mathbf{step}_c(x, S)$ (defined below).
4. Return x.

Subroutine $\mathbf{step}_c$ *... makes progress towards satisfying $x \in S$.*

Input: current solution x, unsatisfied $S \in \mathcal{C}$

1. Choose any *step size* $\beta \in [0, \tilde{c}_x(S)]$. *... discussed before Theorem 1.*
2. For each $j \in \mathsf{vars}(S)$, let $x'_j \in \bar{\mathbb{R}}_{\geq 0}$ be the maximum such that raising x_j to x'_j would increase $c(x)$ by at most β. *... recall $c(x)$ is continuous*
3. For $j \in \mathsf{vars}(S)$, let $x_j \leftarrow x'_j$.

Choosing the Step Size β In an iteration for a constraint S, the algorithm can choose any step size $\beta \geq 0$ subject to the restriction $\beta \leq \tilde{c}_x(S) = \min\{c(y) - c(x) \mid y \in S, y \geq x\}$. That is, β is at most the minimum cost that would be necessary to increase variables in x to bring x into S. To understand ways in which Algorithm 2 can choose β, consider the following.

- In all cases, Algorithm 2 can take β as Algorithm 1 does: just large enough to ensure $x \in S$ after the iteration. By an argument similar to the proof of Lemma 1, this particular β is guaranteed to satisfy the restriction $\beta \leq \tilde{c}_x(S)$. (Of course another option is to take any β smaller than this β.)
- In some cases, Algorithm 2 can take β larger than Algorithm 1 does. For example, consider a linear constraint $x_u + x_w \geq 1$ with linear cost $c(x) = x_u + x_w$. Consider an iteration for this constraint, starting with $x_u = x_w = 0$. Algorithm 1 would take $\beta = 1/2$ and $x_u = x_w = 1/2$, satisfying the constraint. But $\tilde{c}_x(S) = 1$ (to bring x into S would require raising $x_u + x_w$ to 1), so Algorithm 2 can take $\beta = 1$ and $x_u = x_w = 1$, "over-satisfying" the constraint.
- It would be natural to set β to its maximum allowed value $\tilde{c}_x(S)$, but this value can be hard to compute. Consider a single constraint S: $\sum_j c_j \min(1, \lfloor x_j \rfloor) \geq 1$, with cost function $c(x) = \sum_j c_j x_j$. Then $\tilde{c}_{\mathbf{0}}(S) = 1$ if and only if there is a subset Q such that $\sum_{j \in Q} c_j = 1$. Determining this for arbitrary c is SUBSET SUM, which is NP-hard. Still, determining a "good enough" β is not hard: take, e.g. $\beta = \min\{c_j(1 - x_j) \mid x_j < 1\}$. If $x \notin S$, then this is at most $\tilde{c}_x(S)$ because to bring x into S would require raising at least one $x_j < 1$ to 1. This choice of β is easy to compute, and with it Algorithm 2 will satisfy S within Δ iterations.

In short, computing $\tilde{c}_x(S)$ can be hard, but finding a "good" $\beta \leq \tilde{c}_x(S)$ is not hard. A generic choice is to take β just large enough to bring x into S after the iteration, as Algorithm 1 does, but in some cases (especially in online, distributed, and parallel settings where the algorithm is restricted) other choices may be easier to implement

or lead to fewer iterations. For a few examples, see the specializations of Algorithm 2 in Sect. 7.

The proof of the approximation ratio for Algorithm 2 generalizes the proof of Lemma 1 in two ways: the proof has a second case to handle step sizes β larger than Algorithm 1 would take, and the proof handles the more general (submodular) cost function (the generality of which makes this proof unfortunately more abstract).

Theorem 1 (Correctness of Algorithm 2) *For* Submodular-Cost Covering, *if Algorithm 2 terminates, it returns a Δ-approximate solution.*

Proof Consider an iteration for a constraint $S \in \mathcal{C}$. By the submodularity of c, the iteration increases the cost $c(x)$ of x by at most $\beta|\mathsf{vars}(S)|$.[9] We show that, for any feasible y, the cost $\tilde{c}_x(y)$ of y in the residual problem for x decreases by at least β. Thus, the invariant $c(x)/\Delta + \tilde{c}_x(y) \le c(y)$, and the theorem, hold.

Recall that $x \wedge y$ (resp. $x \vee y$) denotes the coordinate-wise minimum (resp. maximum) of x and y.

Let x and x' denote the vector x before and after the iteration, respectively. Fix any feasible y.

First consider the case when $y \ge x$ (the general case will follow from this one). The submodularity of c implies $c(x') + c(y) \ge c(x' \vee y) + c(x' \wedge y)$. Subtracting $c(x)$ from both sides and rearranging terms gives (with equality if c is separable, e.g. linear)

$$\big[c(y) - c(x)\big] - \big[c\big(y \vee x'\big) - c\big(x'\big)\big] \ge c\big(x' \wedge y\big) - c(x).$$

The first bracketed term is $c(y \vee x) - c(x) = \tilde{c}_x(y)$ (using here that $y \ge x$ so $y \vee x = y$). The second bracketed term is $\tilde{c}_{x'}(y)$. Substituting $\tilde{c}_x(y)$ and $\tilde{c}_{x'}(y)$ for the two bracketed terms, respectively, we have

$$\tilde{c}_x(y) - \tilde{c}_{x'}(y) \ge c\big(x' \wedge y\big) - c(x). \tag{1}$$

Note that the left-hand side is the decrease in the residual cost for y in this iteration, which we want to show is at least β. The right-hand side is the cost increase when x is raised to $x' \wedge y$ (i.e., each x_j for $j \in \mathsf{vars}(S)$ is raised to $\min(x'_j, y_j)$).

To complete the proof for the case $y \ge x$, we show that the right-hand side is at least β.

Recall that if y is feasible, then there must be at least one x_k with $k \in \mathsf{vars}(S)$ and $x_k < y_k$.

Subcase 1. When also $x'_k < y_k$ for some $k \in \mathsf{vars}(S)$. The intuition in this case is that raising x to $x' \wedge y$ raises x_k to x'_k, which alone costs β (by Algorithm 2). Formally, let z be x with just x_k raised to x'_k. Then:

(i) Algorithm 2 chooses x'_k maximally such that $c(z) \le c(x) + \beta$.

[9] To see this, consider the variables x_j for $j \in \mathsf{vars}(S)$ one at a time, in at most Δ steps; by submodularity of c, in a step that increases a given x_j, the increase in $c(x)$ is at most what it would have been if x_j had been increased first, i.e., at most β.

(ii) $c(z) = c(x) + \beta$ because (i) holds, c is continuous, and $x'_k < \infty$.
(iii) $z \le x' \wedge y$ because $z \le x'$ and (using $x \le y$ and $x'_k < y_k$) $z \le y$.
(iv) $c(z) \le c(x' \wedge y)$ because c is non-decreasing, and (iii) holds.

Substituting (ii) into (iv) gives $c(x) + \beta \le c(x' \wedge y)$, that is, $c(x' \wedge y) - c(x) \ge \beta$.

Subcase 2. Otherwise $x' \ge_S y$. The intuition in this case is that $x' \wedge y =_S y$, so that raising x to $x' \wedge y$ is enough to bring x into S. And, by the assumption on β in Algorithm 2, it costs at least β to bring x into S.
Here is the formal argument. Let $z = x' \wedge y$. Then:

(a) $z =_S y$ by the definition of z and $x' \ge_S y$.
(b) $z \in S$ by (a), $y \in S$, and Observation 2.
(c) $z \ge x$ by the definition of z and $x' \ge x$ and $y \ge x$.
(d) $\tilde{c}_x(S) \le c(z) - c(x)$ by (b), (c), and the definition of $\tilde{c}_x(S)$.
(e) $\beta \le \tilde{c}_x(S)$ by the definition of Algorithm 2.

By transitivity, (d) and (e) imply $\beta \le c(z) - c(x)$, that is, $c(x' \wedge y) - c(x) \ge \beta$.

For the remaining case (when $y \not\ge x$), we show that the case $y \ge x$ implies this case. The intuition is that if $y_j < x_j$, then $\tilde{c}_x(y)$ is unchanged by raising y_j to x_j, so we may as well assume $y \ge x$. Formally, define $\hat{y} = y \vee x \ge y$. Then $\hat{y} \ge x$ and $\hat{y}$ is feasible.

By calculation, $\tilde{c}_x(y) = c(x \vee y) - c(x) = c(x \vee (y \vee x)) - c(x) = \tilde{c}_x(\hat{y})$.
By calculation, $\tilde{c}_{x'}(y) = c(x' \vee y) - c(x') = c(x' \vee (y \vee x)) - c(x') = \tilde{c}_{x'}(\hat{y})$.
Thus, $\tilde{c}_x(y) - \tilde{c}_{x'}(y)$ equals $\tilde{c}_x(\hat{y}) - \tilde{c}_{x'}(\hat{y})$, which by the case already considered is at least β. □

3 Online Covering, Paging, and Caching

Recall that in online Submodular-Cost Covering, each constraint $S \in \mathcal{C}$ is revealed one at a time; an online algorithm must raise variables in x to bring x into the given S, without knowing the remaining constraints. Algorithm 1 or Algorithm 2 can do this, so by Theorem 1 they yield Δ-competitive online algorithms.[10]

Corollary 1 *Algorithm 1 and Algorithm 2 give Δ-competitive deterministic online algorithms for* Submodular-Cost Covering.

Using simple variants of the reduction of Weighted Caching to online Set Cover from [2], Corollary 1 naturally generalizes a number of known results for Paging, Weighted Caching, File Caching, Connection Caching, etc. as described in the introduction. To illustrate such a reduction, consider the following Connection Caching problem. A request sequence r is given online. Each request r_t is a subset of the nodes in a network. In response to each request r_t, a connection is activated (if not

[10] If the cost function is linear, in responding to S this algorithm needs to know only S and the values of variables in S and their cost coefficients. In general, the algorithm needs to know S, the entire cost function, and all variables' values.

already activated) between all nodes in r_t. Then, if any node in r_t has more than k active connections, some of the connections (other than r_t) must be deactivated (paying $\mathsf{cost}(r_s)$ to deactivate connection r_s) to leave each node with at most k active connections.

Reduce this problem to online Set Cover as follows. Let variable x_t indicate whether connection r_t is closed before the next request to r_t after time t, so the total cost is $\sum_t \mathsf{cost}(r_t)x_t$. For each node u and each time t, for any $(k+1)$-subset $Q \subseteq \{r_s \mid s \le t; u \in r_s\}$, at least one connection $r_s \in Q - \{r_t\}$ (where s is the time of the most recent request to r_s) must have been deactivated, so the following constraint[11] is met: $\max_{r_s \in Q-\{r_t\}} x_s \ge 1$.

This is an instance of online Set Cover, with a set for each time t (corresponding to x_t) and an element for each triple (u, t, Q) (corresponding to the constraint for that triple as described above).

Algorithm 1 (via Corollary 1) gives the following k-competitive algorithm. In response to a connection request r_t, the connection is activated and x_t is set to 0. Then, as long as any node, say u, has $k+1$ active connections, the current x violates the constraint for the triple (u, Q, t), where Q contains u's active connections. Node u implements an iteration of Algorithm 1 for the violated constraint: for all connections $r_s \in Q - \{r_t\}$, it simultaneously raises x_s at rate $1/\mathsf{cost}(r_s)$, until some x_s reaches 1. Node u then arbitrarily deactivates connections $r_s \in Q$ with $x_s = 1$ so that at most k of u's connections remain active.

For a more involved example with a detailed analysis, see Sect. 3.1.

Remark: On $k/(k-h+1)$-Competitiveness The classic competitive ratio of $k/(k-h+1)$ (versus opt with cache size $h \le k$) can be reproduced in the above settings as follows. For any set Q as described above, opt must meet the stronger constraint $\sum_{r_s \in Q-\{r_t\}} \lfloor x_s \rfloor \ge k-h+1$. In this scenario, the proof of Lemma 1 extends to show a ratio of $k/(k-h+1)$ (use that the variables are in $[0, 1]$, so there are at least $k-h+1$ variables x_j such that $x_j < y_j$, so $\tilde{c}_x(y)$ decreases at rate at least $k-h+1$).

3.1 Covering Constraint Generality; Upgradable Online Problems

Recall that the covering constraints in Submodular-Cost Covering need not be convex, only closed upwards. This makes them relatively powerful. The main purpose of this section is to illustrate this power, first by describing a simple example modeling file-segment requests in the `http:` protocol, then by using it to model *upgradable* online caching problems.

Http File-Segment Requests The `http:` protocol allows retrieval of segments of files. To model this, consider each file f as a group of arbitrary segments (e.g. bytes or pages). Let x_t be the *number of segments* of file r_t evicted before its next request. Let $c(r_t)$ be the cost to retrieve a single segment of file r_t, so the total cost is $\sum_t x_t c(r_t)$. Then (for example), if the cache can hold at most k segments total, model this with

[11] We assume the last request must stay cached. If not, don't subtract r_t from Q in each constraint. The competitive ratio is $k+1$.

constraints of the form (for a given subset Q) $\sum_{s \in Q} \max\{0, \mathsf{size}(r_s) - \lfloor x_s \rfloor\} \leq k$ (where $\mathsf{size}(r_s)$ is the total number of segments in r_s).

Running Algorithm 1 on an online request sequence gives the following online algorithm. At time t, respond to the file request r_t as follows. Bring all segments of r_t into the cache. Until the current set of segments in cache becomes cacheable, increase x_s for each file with a segment in cache (other than r_t) at rate $1/c(r_s)$. Meanwhile, whenever $\lfloor \min(x_s, \mathsf{size}(r_s)) \rfloor$ increases for some x_s, evict segment $\lfloor x_s \rfloor$ of r_s. Continue until the segments remaining in cache are cacheable.

The competitive ratio will be the maximum number of files in the cache. (In contrast, the obvious approach of modeling each segment as a separate cacheable item will give competitive ratio equal to the maximum number of individual segments ever in cache.)

Upgradable Caching The main point of this section is to illustrate the wide variety of online caching problems that can be reduced to online covering, and then solved via algorithms such as Algorithm 1.

An Upgradable Caching instance is specified by a maximum cache size k, a number d of hardware components, the eviction-cost function $\mathsf{cost}(\cdots)$, and, for each time step t (revealed in an online fashion) a request r_t and a cacheability predicate, $\mathsf{cacheable}_t(\cdots)$. As the online algorithm proceeds, it chooses not only how to evict items, but also how to upgrade the hardware configuration. The hardware configuration is modeled abstractly by a vector $\gamma \in \mathbb{R}^d_{\geq 0}$, where γ_i is the cost spent so far on upgrading the ith hardware component. Upgrading the hardware configuration is modeled by increasing the γ_i's, which (via $\mathsf{cost}(\cdots)$ and $\mathsf{cacheable}(\cdots)$), can decrease item eviction costs and increase the power of the cache.

In response to each request, if the requested item r_t is not in cache, it is brought in. The algorithm can then increase any of the γ_i's arbitrarily (increasing a given γ_i models spending to upgrade the ith hardware component). The algorithm must then evict items (other than r_t) from cache until the set Q of items remaining in cache is *cacheable*, that is, it satisfies the given predicate $\mathsf{cacheable}_t(Q, \gamma)$. The cost to evict any given item r_s is $\mathsf{cost}(r_s, \gamma)$ for γ at the time of eviction.

The eviction-cost function $\mathsf{cost}(\cdots)$ and each predicate $\mathsf{cacheable}_t(\cdots)$ must meet the following monotonicity restrictions. The eviction-cost function $\mathsf{cost}(r_s, \lambda)$ must be monotone non-increasing with respect to each λ_i. (Intuitively, upgrading the hardware can only decrease eviction costs.) The predicate $\mathsf{cacheable}_t(Q, \lambda)$ must be monotone with respect to Q and each λ_i. That is, increasing any single λ_i cannot cause the value of the predicate to switch from true to false. (Intuitively, upgrading the hardware can only increase the power of the cache.) Also, if a set Q is cacheable (for a given t and γ) then so is every subset $Q' \subseteq Q$. Finally, for simplicity of presentation, we assume that every cacheable set has cardinality k or less.

The cost of a solution is the total paid to evict items, plus the final hardware configuration cost, $\sum_{i=1}^{d} \gamma_i$. The competitive ratio is defined with respect to the minimum cost of any sequence of evictions that meets all the specified cacheability constraints.[12] Note that the offline solution may as well fix the optimal hardware config-

[12]This definition assumes that the request sequence and cacheability requirements are independent of the responses of the algorithm. In practice, even for standard paging, this assumption might not hold. For

uration at the start, before the first request, as this maximizes subsequent cacheability and minimizes subsequent eviction costs.

Standard File Caching is the special case when $\mathsf{cacheable}_t(Q, \gamma)$ is the predicate "$\sum_{r_s \in Q} \mathsf{size}(r_s) \le k$" and $\mathsf{cost}(r_s, \gamma)$ depends only on r_s; that is, $d = 0$. Using Upgradable caching with $d = 0$, one could model independent use of the cache by some interfering process: the cacheability predicate could be changed to $\mathsf{cacheable}_t(Q) \equiv$ "$\sum_{r_s \in Q} \mathsf{size}(r_s) \le k_t$", where each k_t is at most k but otherwise depends arbitrarily on t. Or, using Upgradable caching with $d = 1$, one could also model a cache that starts with size 1, with upgrades to larger sizes (up to a maximum of k) available for purchase at any time. Or, also with $d = 1$, one could model that upgrades of the network (decreasing the eviction costs of arbitrary items arbitrarily) are available for purchase at any time. One can also model fairly arbitrary restrictions on cacheability: for example (for illustration), one could require that, at odd times t, two specified files cannot both be in cache together, etc.

Next we describe how to reduce Upgradable Caching to online Submodular-Cost Covering with $\Delta = d + k$, giving (via Algorithm 1) a $(d + k)$-competitive online algorithm for Upgradable Caching. The resulting algorithm is a natural generalization of existing algorithms.

Theorem 2 (Upgradable caching) Upgradable Caching *has a* $(d + k)$*-competitive online algorithm, where* d *is the number of upgradable components and* k *is the maximum number of files ever held in cache.*

Proof Given an arbitrary Upgradable Caching instance with T requests, define a Submodular-Cost Covering instance $(c, \mathcal{C})$ over $\mathbb{R}^{d+T}_{\ge 0}$ as follows.

The variables are as follows. For $i = 1, \ldots, d$, variable γ_i is the amount invested in component i. For $t = 1, \ldots, T$, variable x_t is the cost (if any) incurred for evicting the tth requested item r_t at any time before its next request. Thus, a solution is a pair $(\gamma, x) \in \mathbb{R}^d_{\ge 0} \times \mathbb{R}^T_{\ge 0}$. The cost function is $c(\gamma, x) = \sum_{i=1}^{d} \gamma_i + \sum_{t=1}^{T} x_t$.

At any time t, let $A(t)$ denote the set of times of *active* requests, the times of the most recent requests to each item:

$$A(t) = \left\{ s \mid s \le t,\ \left(\forall s' \le t\right) r_{s'} = r_s \to s' \le s \right\}.$$

In what follows, in the context of the current request r_t at time t, we abuse notation by identifying each time $s \in A(t)$ with its requested item r_s. (This gives a bijection between $A(t)$ and the requested items.)

For any given subset $Q \subseteq A(t)$ of the currently active items, and any hardware configuration γ, either the set Q is cacheable or at least one item $s \in Q - \{t\}$ must be evicted by time t. In short, any feasible solution (γ, x) must satisfy the predicate

$$\mathsf{cacheable}_t(Q, \gamma) \quad \textbf{or} \quad \exists s \in Q - \{t\} \text{ such that } x_s \ge \mathsf{cost}(r_s, \gamma).$$

example, a fault incurred by one process may cause another process's requests to come earlier. In this case, the optimal offline strategy would choose responses that take into account the effects on inputs at subsequent times (possibly leading to a lower cost). Modeling this accurately seems difficult.

For a given t and Q, let $S_t(Q)$ denote the set of solutions (γ, x) satisfying the above predicate. The set $S_t(Q)$ is closed upwards (by the restrictions on cacheable and cost) and so is a valid covering constraint.

The online algorithm adapts Algorithm 1, as follows. It initializes $\gamma = x = \mathbf{0}$. After request r_t, the algorithm keeps in cache the set of active items whose eviction costs have not been paid, which we denote C:

$$C = C_t(\gamma, x) = \{t\} \cup \left\{s \in A(t) \mid x_s < \mathsf{cost}(r_s, \gamma)\right\}.$$

To respond to request r_t, as long as the cached set C is not legally cacheable (i.e., $\mathsf{cacheable}_t(C, \gamma)$ is false), the corresponding constraint, $S_t(C)$ is violated, and the algorithm performs an iteration of Algorithm 1 for that constraint. By inspection, this constraint depends on the following variables: every λ_i, and each x_s where r_s is cached and $s \neq t$ (that is, $s \in C - \{t\}$). Thus, the algorithm increases these variables at unit rate, until either *(a)* $x_s \geq \mathsf{cost}(r_s, \gamma)$ for some cached r_s and/or *(b)* $\mathsf{cacheable}_t(C, \gamma)$ becomes true (due to items leaving C and/or increases in γ). When case *(a)* happens, the algorithm evicts that r_s to maintain the invariant that the cached set is C, then continues with the new constraint for the new C. When case *(b)* happens, the currently cached set is legally cacheable, and the algorithms is done responding to request t,

This completes the description of the algorithm. For the analysis, we define the constraint collection $\mathcal{C}$ in the underlying Submodular Covering instance $(c, \mathcal{C})$ to contain *just those constraints $S_t(C)$ for which the algorithm, given the request sequence, does steps*. When the algorithm does a step at time t, the cached set C contains only t and items that stayed in cache (and were collectively cacheable) after the previous request. Since at most k items stayed in cache, by inspection, the underlying constraint $S_t(C)$ depends on at most $d + k$ variables in (γ, x). Thus, the degree Δ of $(c, \mathcal{C})$ is at most $d + k$.

For the Submodular-Cost Covering instance $(c, \mathcal{C})$, let (γ^*, x^*) and (γ', x'), respectively, be the solutions corresponding to opt and generated by the algorithm, respectively. For the original upgradable caching instance (distinct from $(c, \mathcal{C})$), let opt and $\mathcal{A}$ denote the costs of, respectively, the optimal solution and the algorithm's solution.

Then $\mathcal{A} \leq c(\gamma', x')$ because the algorithm paid at most x'_s to evict each evicted item r_s. (We use here that $x_s \geq \mathsf{cost}(r_s, \gamma)$ at the time of eviction, and x_s does not decrease after that; note that x_s may exceed $\mathsf{cost}(r_s, \gamma)$ because some items with positive x'_s might not be evicted.) The approximation guarantee for Algorithm 1 (Lemma 1) ensures $c(\gamma', x') \leq \Delta c(\gamma^*, x^*)$.

By transitivity $\mathcal{A} \leq c(\gamma', x') \leq \Delta\, c(\gamma^*, x^*) = \Delta\, \mathsf{opt}$. □

Flexibility in Tuning the Algorithm In practice, it is well known that a competitive ratio much lower than k is desirable and usually achievable for paging. Also, for file caching (where items have sizes), carefully tuned variants of Landlord (a.k.a. Greedy-Dual Size) outperform the original algorithms [23]. In this context, it is worth noting that the above algorithm can be adjusted, or tuned, in various ways while keeping its competitive ratio.

First, there is flexibility in how the algorithm handles "free" requests—requests to items that are already in the cache. When the algorithm is responding to request r_t, let t' be the most recent time that item was requested but was not in the cache at the time of the request. Let $F(t) = \{s \mid t' < s < t, r_s = r_t\}$ denote the times of these recent free requests to the item. Worst-case sequences have no free requests, and, although each free request r_s costs nothing, the analysis in the proof above charges x_s for it anyway.

The algorithm in the proof stops the step for the current constraint $S_t(C)$ and removes an item s from the cache C when some x_s reaches $\mathsf{cost}(r_s, \gamma)$. Modify the algorithm to stop the step (and remove s from C) sooner, specifically, when x_s reaches $\mathsf{cost}(r_s, \gamma) - \sum_{s' \in F(s)} x_{s'}$ for some $s \in C$ (effectively reducing the eviction cost of r_s by $\sum_{s' \in F(s)} x_{s'}$. The modified algorithm is still a specialization of Algorithm 1. Although the resulting solution (x, γ) may be infeasible, the approximation guarantee still applies, in that (x, γ) has cost at most Δ opt. The online solution $\mathcal{A}$ *is* feasible though, and has cost equal to the cost of (x, γ), and is thus Δ-competitive.

In the above description, each free request is used to reduce the effective cost of a later request to the same item. Whereas the unmodified algorithm generalizes LRU, the modified algorithm generalizes FIFO.

Even more generally, the sum over the free requests r_s of x_s can be *arbitrarily* distributed over the non-free requests to reduce their effective costs (leading to earlier eviction). Essentially the same analysis still shows k-competitiveness.

There is a second, independent source of flexibility—the rates at which the variables are increased in each step. As it specializes for File Caching, the algorithm in the proof raises each x_s at unit rate until x_s reaches $\mathsf{cost}(r_s)$. This raises the total cost $c(x, \gamma)$ at rate $\sum_{s \in C - \{t\}} 1 \leq k$, while (in the analysis of Algorithm 1) the residual cost of opt decreases at rate at least 1, implying a competitive ratio of k. In contrast, Landlord (effectively) raises each x_s at rate $\mathsf{size}(r_s)$ until x_s reaches $\mathsf{cost}(r_s)$. This raises $c(x, \gamma)$ more rapidly, at rate $\sum_{s \in C - \{t\}} \mathsf{size}(r_s)$, but this sum is also at most k (since all summed items fitted in the cache before r_t was brought in). This implies the (known) competitive ratio of k for Landlord. Generally, for items of size larger than 1, the algorithm could raise x_s at any rate in $[1, \mathsf{size}(r_s)]$. The more general algorithm still has competitive ratio at most k.

Analogous adjustments can be made in other applications of Algorithm 1. For some applications, adjusting the variables' relative rates of increase can lead to stronger theoretical bounds.

4 Stateless Online Algorithm and Randomized Generalization of Algorithm 2

This section describes two randomized algorithms for Submodular-Cost Covering: Algorithm 3—a *stateless* Δ-competitive online algorithm, and an algorithm that generalizes both that and Algorithm 2. For simplicity, in this section we assume each U_j has finite cardinality. (The algorithms can be generalized in various ways to arbitrary closed U_j, but the presentation becomes more technical.[13])

Algorithm 3 generalizes the Harmonic k-server algorithm as it specializes for Paging and Caching [56], and Pitt's weighted vertex cover algorithm [4].

[13] Here is one of many ways to modify Algorithm 3 to handle arbitrary closed U_j's. In each step, take β small enough so that for each $j \in \mathsf{vars}(S)$, either U_j contains the entire interval $[x_j, x_j + \beta]$, or U_j

Definition 5 (*Stateless* online algorithm) An online algorithm for a (non-canonical) SUBMODULAR-COST COVERING instance $(c, U, \mathcal{C})$ is *stateless* provided the only state it maintains is the current solution x, in which each x_j is assigned only values in U_j.

Although Algorithm 1 and Algorithm 2 maintain only the current partial solution $x \in \mathbb{R}^n_{\geq 0}$, for problems with variable-domain restrictions x_j may take values outside U_j. So these algorithms are not stateless.[14]

The stateless algorithm initializes each x_j to $\min U_j$. Given any constraint S, it repeats the following until S is satisfied: it chooses a random subset $J \subseteq \mathsf{vars}(S)$, then increases each x_j for $j \in J$ to its next allowed value, $\min\{\alpha \in U_j \mid \alpha > x_j\}$. The subset J can be any random subset such that, for some $\beta \geq 0$, for each $j \in \mathsf{vars}(S)$, $\Pr[j \in J]$ equals β/β_j, where β_j is the increase in $c(x)$ that would result from increasing x_j.

For example, one could take $J = \{r\}$ where r is chosen so that $\Pr[r = j] \propto 1/\beta_j$. Or take any $\beta \leq \min_j \beta_j$, then, independently for each $j \in \mathsf{vars}(S)$, take j in J with probability β/β_j. Or, choose $\tau \in [0, 1]$ uniformly, then take $J = \{j \mid \beta/\beta_j \geq \tau\}$.

Stateless algorithm for Submodular-cost Covering ALG. 3

Input: cost c, finite domains U, constraints $\mathcal{C}$

1. Initialize $x_j \leftarrow \min U_j$ for each j.
2. In response to each given constraint S, repeat the following until $x \in S$:
3. For each $j \in \mathsf{vars}(S)$:
4. If $x_j < \max U_j$:
5. Let $\hat{x}_j = \min\{\alpha \in U_j \mid \alpha > x_j\}$ be the next largest value in U_j.
6. Let β_j be the increase in $c(x)$ that would result from raising x_j to $\hat{x}_j$.
7. Else:
8. Let $\hat{x}_j = x_j$ and $\beta_j = \infty$.
9. If $(\forall j \in \mathsf{vars}(S))\ \beta_j = \infty$: Return "infeasible".
10. Increase x_j to $\hat{x}_j$ for all $j \in J$, where J is any random subset of $\mathsf{vars}(S)$ such that, for some $\beta \geq 0$, for each $j \in \mathsf{vars}(S)$, $\Pr[j \in J] = \beta/\beta_j$. Above interpret $0/0$ as 1. (Note that there are many ways to choose J with the necessary property.)

contains just x_j from that interval. For the latter type of j, take β_j and $\hat{x}_j$ as described in Algorithm 3. For the former type of j, take $\beta_j = \beta$ and take $\hat{x}_j$ to be the smallest value such that increasing x_j to $\hat{x}_j$ would increase $c(x)$ by β. Then proceed as above. (Taking β infinitesimally small gives the following process. For each $j \in \mathsf{vars}(S)$ simultaneously, x_j increases continuously at rate inversely proportional to its contribution to the cost, if it is possible to do so while maintaining $x_j \in U_j$, and otherwise x_j increases to its next allowed value randomly according to a Poisson process whose intensity is inversely proportional to the resulting expected increase in the cost.)

[14] The online solution is not x, but rather $x' \leq x$ defined from x by $x'_j = \max\{\alpha \in U_j \mid \alpha \leq x_j\}$ or something similar, so the algorithms maintain state other than the current online solution x'. For example, for paging problems, the algorithms maintain $x_t \in [0, 1]$ as they proceed, where a requested item r_s is currently evicted only once $x_s = 1$. To be stateless, they should maintain each $x_t \in \{0, 1\}$, where $x_s = 0$ iff page r_s is still in the cache.

In the case that each $U_j = \{0, 1\}$ and c is linear, one natural special case of the algorithm is to repeat the following as long as there is some unsatisfied constraint S:

Choose a single $k \in \{j \mid j \in \mathsf{vars}(S), x_j = 0\}$ *at random, so that* $\Pr[k = j] \propto 1/c_j$. *Set* $x_k = 1$.

Theorem 3 (Correctness of stateless Algorithm 3) *For online* Submodular-Cost Covering *with finite variable domains, Algorithm 3 is stateless. If the step sizes are chosen so the number of iterations has finite expectation* (*e.g. taking* $\beta = \Omega(\min_j \beta_j)$), *then it is* Δ*-competitive* (*in expectation*).

Proof By inspection the algorithm maintains each $x_j \in U_j$. It remains to prove Δ-competitiveness.

Consider any iteration of the repeat loop. Let x and x', respectively, denote x before and after the iteration. Let β and β_j be as in the algorithm.

First we observe that iteration increases the cost of algorithm's solution x by at most $\beta\Delta$ in expectation:

Claim 1 *Cost* $c(x)$ *increases by at most* $\sum_{j \in \mathsf{vars}(S)} (\beta/\beta_j)\beta_j = \beta|\mathsf{vars}(S)| \le \beta\Delta$ *in expectation*.

The claim follows easily by direct calculation and the submodularity of c.

Inequality (1) from the proof of Theorem 1 still holds: $\tilde{c}_x(y) - \tilde{c}_{x'}(y) \ge c(x' \wedge y) - c(x)$, so the next claim implies that the residual cost of any feasible $y \ge x$ decreases by at least β in expectation:

Claim 2 *For any feasible* $y \ge x$, $E_J[c(x' \wedge y) - c(x) \mid x] \ge \beta$.

Proof of Claim By Observation 2, there is a $k \in \mathsf{vars}(S)$ with $y_k > x_k$. Since $y_k \in U_k$, the algorithm's choice of $\hat{x}_k$ ensures $y_k \ge \hat{x}_k$. Let z be obtained from x by raising just x_k to $\hat{x}_k$. With probability β/β_k, the subroutine raises x_k to $\hat{x}_k \le y_k$, in which case $c(x' \wedge y) - c(x) \ge c(z) - c(x) = \beta_k$. This implies $E_J[c(x' \wedge y) - c(x) \mid x] \ge (\beta/\beta_k)\beta_k = \beta$, proving Claim 2.

Thus, for $y \ge x$, in each iteration, the residual cost of y decreases by at least β in expectation: $E_J[\tilde{c}_x(y) - \tilde{c}_{x'}(y) \mid x] \ge \beta$. By the argument at the end of the proof of Theorem 1, this implies the same for *all* feasible y (even if $y \not\ge x$).

In sum, the iteration increases the cost of x by at most $\Delta\beta$ in expectation, while decreasing the residual cost of any feasible y by at least β in expectation. By standard probabilistic arguments, this implies that the expected final cost of x is at most Δ times the initial residual cost of y (which equals the cost of y).

Formally, $c(x^t) + \Delta\tilde{c}_{x^t}(y)$ is a super-martingale, where random variable x^t denotes x after t iterations.

Let random variable T be the number of iterations. Using, respectively, $\tilde{c}_{x^T}(y) \ge 0$, a standard optional stopping theorem, and $\tilde{c}_{x^0}(y) = c(y) - c(x^0)$ (because $x^0 \le y$), the expected final cost $E[c(x^T)]$ is at most

$$E\left[c\left(x^T\right) + \Delta\tilde{c}_{x^T}(y)\right] \le E\left[c\left(x^0\right) + \Delta\tilde{c}_{x^0}(y)\right] = c\left(x^0\right) + \Delta\left(c(y) - c\left(x^0\right)\right) \le \Delta c(y).$$

□

Most General Randomized Algorithm Algorithm 2 raises the variables continuously, whereas Algorithm 3 steps each variable x_j through the successive values in U_j. For some instances, both of these choices can lead to slow running times. Next is an algorithm that generalizes both of these algorithms. The basic algorithm is simple, but the condition on β is more subtle. The analysis is a straightforward technical generalization of the previous analyses.

The algorithm has more flexibility in increasing variables. This may be important in distributed or parallel applications, where the flexibility allows implementing the algorithm so that it is guaranteed to make rapid (probabilistic) progress. (The flexibility may also be useful for dealing with limited-precision arithmetic.)

The algorithm is Algorithm 2, modified to call subroutine $\mathsf{random\_step}_c(x, S)$ (Algorithm 4, below) instead of $\mathsf{step}_c(x, S)$ to augment x in each iteration.

Subroutine $\mathsf{random\_step}_c$ ALG. 4

Input: current solution $x \in \bar{\mathbb{R}}^n_{\geq 0}$, unsatisfied constraint $S \in \mathcal{C}$

1. Fix an arbitrary probability $p_j \in [0, 1]$ for each $j \in \mathsf{vars}(S)$.
   ... above, taking each $p_j = 1$ gives Algorithm 2
2. Choose a step size $\beta \geq 0$ where β is at most expression (2) in Theorem 4.
3. For j with $p_j > 0$, let $\hat{x}_j$ be maximum such that raising x_j to $\hat{x}_j$ would raise $c(x)$ by at most β / p_j.
4. Choose a random subset[a] $J \subseteq \mathsf{vars}(S)$ s.t. $\Pr[j \in J] = p_j$ for $j \in \mathsf{vars}(S)$.
5. For $j \in J$, let $x_j \leftarrow \hat{x}_j$.

[a] As in Algorithm 3, the events "$j \in J$" for $j \in \mathsf{vars}(S)$ can be dependent. See the last line of Algorithm 3.

The step-size requirement is a bit more complicated.

Theorem 4 (Correctness of randomized algorithm) *For* SUBMODULAR-COST COVERING *suppose, in each iteration of the randomized algorithm for a constraint $S \in \mathcal{C}$ and $x \notin S$, the step size $\beta \geq 0$ is at most*

$$\min\left\{E_J\left[c\left(x \uparrow_J^y\right) - c(x)\right] : y \geq x;\ y \in S\right\}, \tag{2}$$

where $x \uparrow_J^y$ is a random vector obtained by choosing a random subset J from the same distribution used in Line 4 of random_step *and then raising x_j to y_j for $j \in J$. Suppose also that the expected number of iterations is finite. Then the algorithm returns a Δ-approximate solution in expectation.*

Note that if $p = \mathbf{1}$, then (2) simplifies to $\tilde{c}_x(S)$. If c is linear, (2) simplifies to $\tilde{c}'_x(S)$ where $c'_j = p_j c_j$.

Proof The proof mirrors the proof of Theorem 3.

Fix any iteration. Let x and x', respectively, denote x before and after the iteration. Let p, β, $\hat{x}$, and J be as in random_step.

Claim 1 The expected increase in $c(x)$ is

$$E_J\big[c\big(x'\big) - c(x)|x\big] \leq \sum_{j \in \mathsf{vars}(S)} p_j \beta / p_j = \beta |\,\mathsf{vars}(S)| \leq \beta \Delta.$$

The claim follows easily by calculation and the submodularity of c.

Inequality (1) from the proof of Theorem 1 still holds: $\tilde{c}_x(y) - \tilde{c}_{x'}(y) \geq c(x' \wedge y) - c(x)$, so the next claim implies that the residual cost of any feasible $y \geq x$ decreases by at least β in expectation:

Claim 2 *For any feasible* $y \geq x$, $E_J[c(x' \wedge y) - c(x) \mid x] \geq \beta$.

Proof of Claim The structure of the proof is similar to the corresponding part of the proof of Theorem 1.

Recall that if y is feasible, then there must be at least one x_k with $k \in \mathsf{vars}(S)$ and $x_k < y_k$.

Subcase 1 When also there is an $\hat{x}_k < y_k$ for $k \in \mathsf{vars}(S)$ with $p_k > 0$.
In case of the event $k \in J$, raising x to $x' \wedge y$ raises x_k to $\hat{x}_k$, which alone (by Algorithm 4) costs β / p_k.
Thus, the expected cost to raise x to $x' \wedge y$ is at least $\Pr[k \in J]\, \beta / p_k = \beta$.
Subcase 2 Otherwise, $\hat{x}_j \geq y_j$ for all $j \in J$ (for all possible J).
In this case, $x' \wedge y \geq x \uparrow_J^y$ in all outcomes.
Thus, the expected cost to increase x to $x' \wedge y$ is at least the expected cost to increase x to $x \uparrow_J^y$.
By the assumption in the theorem, this is at least β. This proves Claim 2.

Claims 1 and 2 imply Δ-approximation via the argument in the final paragraphs of the proof of Theorem 3. □

5 Relation to Local-Ratio Method

The local-ratio method has most commonly been applied to problems with variables taking values in $\{0, 1\}$ and with linear objective function $c \cdot x$ (see [4, 6, 8, 9]; for one exception, see [7]). For example, [8] shows a form of equivalence between the primal-dual method and the local-ratio method, but that result only considers problems with solution space $\{0, 1\}^n$ (i.e., 0/1-variables). Also, the standard intuitive interpretation of local-ratio—that the algorithm reduces the coefficients in the cost vector c—works only for 0/1-variables.

Here we need to generalize to more general solution spaces. To begin, we first describe a typical local-ratio algorithm for a problem with variables over $\{0, 1\}$ (we use CIP-01). After that, we describe one way to extend the approach to more general variable domains. With that extension in place, we then recast Theorem 1 (the approximation ratio for Algorithm 2) as a local-ratio analysis.

Local-Ratio for $\{0,1\}$ *Variable Domains* Given a (non-canonical) LINEAR-COST COVERING instance $(c, U, \mathcal{C})$ where each $U_j = \{0,1\}$, the standard local-ratio approach gives the following Δ-approximation algorithm:

> *Initialize vector* $\ell = c$. *Let "the cost of* x *under* ℓ*" be* $\sum_j \ell_j x_j$. *Let* $\hat{x}(\ell)$ *be the maximal* $x \in \{0,1\}^n$ *that has zero cost under* ℓ *(i.e.,* $\hat{x}_j(\ell) = 1$ *if* $\ell_j = 0$*). As long as* $\hat{x}(\ell)$ *fails to meet some constraint* $S \in \mathcal{C}$*, repeat the following: Until* $\hat{x}(\ell) \in S$*, simultaneously for all* $j \in \mathsf{vars}(S)$ *with* $\ell_j > 0$*, decrease* ℓ_j *at unit rate. Finally, return* $\hat{x}(\ell)$.

The algorithm has approximation ratio $\Delta = \max_S |\mathsf{vars}(S)|$ by the following argument. Fix the solution x^a returned by the algorithm. An iteration for a constraint S decreases $\ell_j x_j^a$ for each $j \in \mathsf{vars}(S)$ at rate $x_j^a \le 1$, so it decreases $\ell \cdot x^a$ at rate at most Δ. On the other hand, in any feasible solution x^*, as long as the variables x_j for $j \in S$ are being decreased, at least one $j \in \mathsf{vars}(S)$ with $\ell_j > 0$ has $x_j^* = 1$ (otherwise $\hat{x}(\ell)$ would be in S). Thus the iteration decreases $\ell \cdot x^*$ at rate at least 1. From this it follows that $c \cdot x^a \le \Delta c \cdot x^*$ (details are left as an exercise).

This local-ratio algorithm is the same as Algorithm 1 for the case $U = \{0,1\}^n$ (and linear cost). To see why, observe that the modified cost vector ℓ in the local-ratio algorithm is implicitly keeping track of the residual problem for x in Algorithm 1. When the local-ratio algorithm reduces a cost ℓ_j at unit rate, for the same j, Algorithm 1 increases x_j at rate $1/c_j$. This maintains the mutual invariant $(\forall j)\ \ell_j = c_j(1 - x_j)$—that is, ℓ_j is the cost to raise x_j the rest of the way to 1. Thus, as they proceed together, *the* CIP-01 *instance* $(\ell, \mathcal{C})$ *defined by the current (lowered) costs* ℓ *is exactly the residual problem* $(\tilde{c}_x, \mathcal{C})$ *for the current* x *in Algorithm 1*. To confirm this, note that the cost of any y in the residual problem for x is $\tilde{c}_x(y) = \sum_j c_j \max(y_j - x_j, 0) = \sum_{j:y_j=1} c_j(1 - x_j)$, whereas in the local-ratio algorithm the cost for y under ℓ is $\sum_{j:y_j=1} \ell_j$, and by the mutual invariant above these are equal.

So, at least for linear-cost covering problems with $\{0,1\}$-variable domains, we can interpret local-ratio via residual costs, and vice versa. On the other hand, residual costs extend naturally to more general domains. Is it possible to likewise extend the local-ratio cost-reduction approach? Simply reducing some costs ℓ_j until some $\ell_j = 0$ does not work—$\ell_j x_j^a$ may decrease at rate faster than 1, and when ℓ_j reaches 0, it is not clear which value x_j should take in U_j.

Local Ratio for More General Domains One way to extend local-ratio to more general variable domains is as follows. Consider any (non-canonical) instance $(c, U, \mathcal{C})$ where c is linear. Assume for simplicity that each variable domain U_j is the same: $U_j = \{0, 1, \ldots, u\}$ for some u independent of j, and that all costs c_j are non-zero. For each variable x_j, instead of maintaining a single reduced cost ℓ_j, the algorithm will maintain a *vector* $\ell_j \in \mathbb{R}^u_{\ge 0}$ of reduced costs. Intuitively, ℓ_{jk} represents the cost to increase x_j from $k-1$ to k. (We are *almost* just reducing the general case to the 0/1 case by replacing each variable x_j by multiple copies, but that alone doesn't quite work, as it increases Δ by a factor of u.) Define the cost of any $x \in \{0, 1, \ldots, u\}^n$ under the current ℓ to be $\sum_j \sum_{k=1}^{x_j} \ell_{jk}$. As a function of the reduced costs ℓ, define $\hat{x}(\ell)$ to be the maximal zero-cost solution, i.e. $\hat{x}_j(\ell) = \max\{k \mid \sum_{i=1}^{k} \ell_{ji} = 0\}$.

The local-ratio algorithm initializes each $\ell_{jk} = c_j$, so that the cost of any x under ℓ equals the original cost of x (under c). The algorithm then repeats the following until $\hat{x}(\ell)$ satisfies all constraints.

1. Choose any constraint S that $\hat{x}(\ell)$ does not meet. Until $\hat{x}(\ell) \in S$, do:
2. Just until an ℓ_{jk} reaches zero, for all $j \in \mathsf{vars}(S)$ with $\hat{x}_j(\ell) < u$
3. simultaneously, lower ℓ_{jk_j} at unit rate, where $k_j = \hat{x}_j(\ell) + 1$.

Finally the algorithm returns $\hat{x}(\ell)$ (the maximal x with zero cost under the final ℓ).

One can show that this algorithm is a Δ-approximation algorithm (for Δ w.r.t. the original CIP-UB instance) by the following argument. Fix x^a and x^* to be, respectively, the algorithm's final solution and an optimal solution. In an iteration for a constraint S, as ℓ changes, the cost of x^a under ℓ decreases at rate at most Δ, while the cost of x^* under ℓ decreases at rate at least 1. We leave the details as an exercise.

In fact, the above algorithm is equivalent to Algorithm 1 for CIP-UB. If the two algorithms are run in sync, at any given time, the CIP-01 instance with modified cost ℓ exactly captures the residual problem for Algorithm 1.

Local-ratio for Submodular-Cost Covering The previous example illustrates the basic ideas underlying one approach for extending local-ratio to problems with general variable domains: decompose the cost into parts, one for each possible increment of each variable, then, to satisfy a constraint S, for each variable x_j with $j \in \mathsf{vars}(S)$, lower just the cost for that variable's next increment. This idea extends somewhat naturally even to infinite variable domains, and is equivalent to the residual-cost interpretation.

Next we tackle SUBMODULAR-COST COVERING in full generality. We recast the proof of Theorem 1 (the correctness of Algorithm 2) as a local-ratio proof. Formally, the minimum requirement for the local-ratio method is that the objective function can be decomposed into "locally approximable" objectives. The common cost-reduction presentation of local ratio described above gives one such decomposition, but others have been used (e.g. [7]). In our setting, the following local-ratio decomposition works. (We discuss the intuition after the lemma and proof.)

Lemma 2 (Local-ratio lemma) *Any algorithm returns a Δ-approximate solution x provided there exist $T \in \mathbb{Z}_{\geq 0}$ and $c^t : \mathbb{R}^n_{\geq 0} \to \mathbb{R}_{\geq 0}$ (for $t = 1, 2, \ldots, T$) and $r : \mathbb{R}^n_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that*

(a) *for any y, $c(y) = \sum_{t=1}^{T} c^t(y) + r(y)$,*
(b) *for all t, and any y and feasible x^*, $c^t(y) \leq c^t(x^*)\Delta$,*
(c) *the algorithm returns x such that $r(x) = 0$.*

Proof Properties (a)–(c) state that the cost function can be decomposed into parts, where, for each part $c^t()$, *any* solution y is Δ-approximate, and, for the remaining part $r()$, the solution x returned by the algorithm has cost zero. Since x is Δ-approximate w.r.t. each $c^t()$, and x has cost zero for the remaining part, x is Δ-approximate overall. Formally, let x^* be an optimal solution. By properties (a) and (c), (b), then (a),

respectively,

$$c(x) = \sum_{t=1}^{T} c^t(x) \le \sum_{t=1}^{T} c^t\left(x^*\right)\Delta + r\left(x^*\right)\Delta = c\left(x^*\right)\Delta.$$

□

In local-ratio as usually presented, the local-ratio algorithm determines the cost decomposition as it proceeds. The only state maintained by the algorithm after iteration t is the "remaining cost" function ℓ^t, defined by $\ell^t(y) = c(y) - \sum_{s \le t} c^s(y)$. In iteration t, the algorithm determines some portion c^t of ℓ^{t-1} satisfying Property (b) in the lemma and removes it from the cost. (This is the key step in designing the algorithm.) The algorithm stops when it has removed enough of the cost so that there is a feasible solution x^a with zero remaining cost ($\ell^T(x^a) = 0$), then returns that x^a (taking $r = \ell^T$ for Property (c) in the lemma). By the lemma, this x^a is a Δ-approximate solution.

For a concrete example, consider the local-ratio algorithm for the linear-cost, 0/1-variable case described at the start of this section. Let T be the number of iterations. For $t = 0, 1, \ldots, T$, let ℓ^t be the modified cost vector at the end of iteration t (so ℓ^0 is the original cost vector). Define $c^t(y) = (\ell^t - \ell^{t-1}) \cdot y$ to be the decrease in the cost of y due to the change in ℓ in iteration t. Define $r(y) = \ell^T \cdot y$ to be the modified cost vector at termination (so the returned solution $x = \hat{x}(\ell^T)$ has $r(x) = 0$). It is easy to see that property (a) and (c) hold. To see that property (b) holds, recall that in iteration t the algorithm reduces all ℓ_j for $j \in \mathsf{vars}(S)$ with $\ell_j > 0$, simultaneously and continuously at unit rate. It raises each x_j to 1 when ℓ_j reaches 0. It stops once $x \in S$. At most Δ of the ℓ_j's are being lowered at any time, so the rate of decrease in $\ell \cdot y$ for any $y \in \{0, 1\}^n$ is at most Δ. But for any $x^* \in S$, the rate of decrease in $\ell \cdot x^*$ is at least 1, because at least one $j \in \mathsf{vars}(S)$ has $x^*_j = 1$ and $\ell_j > 0$ (otherwise x would be in S).

Next we describe how to generate such a decomposition of the cost c corresponding to a run of Algorithm 2 on an arbitrary Submodular-Cost Covering instance $(c, \mathcal{C})$. This gives an alternate proof of Theorem 1. The proof uses the previously described idea for extending local ratio to more general domains. Beyond that, it is slightly more complicated than the argument in the previous paragraph for two reasons: it handles submodular costs, and, more subtly, in an iteration for a constraint S, Algorithm 2 can increase variables more than enough to satisfy S (of course this is handled already in the previous analysis of Algorithm 2, which we leverage below).

Lemma 3 (Correctness of Algorithm 2 via local-ratio) *Algorithm 2, run on any instance $(c, \mathcal{C})$ of* Submodular-Cost Covering, *implicitly generates a cost decomposition $\{c^t\}$ and r as described in Lemma 2. Thus, Algorithm 2 gives a Δ-approximation.*

Proof Assume without loss of generality that $c(\mathbf{0}) = 0$. (Otherwise use cost function $c'(x) = c(x) - c(\mathbf{0})$. Then $c'(x)$ is still non-negative and non-decreasing, and, since $\Delta \ge 1$, the approximation ratio for c' implies it for c.)

Let x^t denote Algorithm 2's vector x after t iterations. Let T be the number of iterations.

Recall that $\tilde{c}_{x^t}$ is the cost in the residual problem $(\tilde{c}_{x^t}, \mathcal{C})$ for x after iteration t: $\tilde{c}_{x^t}(y) = c(x^t \vee y) - c(x^t)$.

Define c^t so that the "remaining cost" function ℓ^t (as discussed before the lemma) equals the objective $\tilde{c}_{x^t}$ in the residual problem for x^t. Specifically, take $c^t(y) = \tilde{c}_{x^{t-1}}(y) - \tilde{c}_{x^t}(y)$. Also define $r(y) = \tilde{c}_{x^T}(y)$.

These c^t and r have properties (a)–(c) from Lemma 2.

Properties (a) and (c) follow by direct calculation. To show (b), fix any y. Then $c^t(y) = c(x^t) - c(x^{t-1}) + c(x^{t-1} \vee y) - c(x^t \vee y) \leq c(x^t) - c(x^{t-1})$. So $c^t(y)$ is at most the increase in the cost $c(x)$ of x during iteration t. In the proof of Theorem 1, this increase in $c(x)$ in iteration t is shown to be at most $\Delta\beta$. Also, for any feasible x^*, the cost $\tilde{c}_x(x^*)$ for x^* in the residual problem for x is shown to reduce by at least β. But the reduction in $\tilde{c}_x(x^*)$ is exactly $c^t(x^*)$. Thus, $c^t(y) \leq \Delta\beta \leq \Delta c^t(x^*)$, proving Property (b). □

Each c^t in the proof captures the part of the cost c lying "between" x^{t-1} and x^t. For example, if c is linear, then $c^t(y) = \sum_j c_j |[0, y_j] \cap [x_j^{t-1}, x_j^t]|$. The choice of x^t in the algorithm guarantees property (b) in the lemma.

6 Relation to Primal-Dual Method; Local Valid Inequalities

Next we discuss how Algorithm 1 can be reinterpreted as a primal-dual algorithm.

It is folklore that local-ratio and primal-dual algorithms are "equivalent"; for example [8] shows a formal equivalence between the primal-dual method and the local-ratio method. But that result only applies to problems with solution space $\{0, 1\}^n$ (i.e., 0/1-variables), and the underlying arguments do not seem to extend directly to this more general setting.

Next we present two linear-program relaxations for Linear-Cost Covering, then use the second one to reprove Lemma 1 (that Algorithm 1 is a Δ-approximation algorithm for Linear-Cost Covering) using the primal-dual method.

Fix any Linear-Cost Covering instance $(c, \mathcal{C})$ in canonical form.

To simplify the presentation, assume at least one optimal solution to $(c, \mathcal{C})$ is finite (i.e., in $\mathbb{R}^n_{\geq 0}$).

For any $S \in \mathcal{C}$, let $\overline{S}$ denote the complement of S in $\bar{\mathbb{R}}^n_{\geq 0}$. Let $\overline{S}^*$ denote the closure of $\overline{S}$ under limit.

By Observation 2, if x is feasible, then, for any $S \in \mathcal{C}$ and $y \in \overline{S}$, x meets the *non-domination* constraint $x \not\leq_S y$ (that is, $x_j \geq y_j$ for some $j \in \mathsf{vars}(S)$). By a limit argument,[15] the same is true if $y \in \overline{S}^*$. In sum, if x is feasible, then x meets the non-domination constraint for every (S, y) where $S \in \mathcal{C}$ and $y \in \overline{S}^*$. For finite x, the converse is also true:

[15] If $x \in S$ and $y \in \overline{S}^*$, then y is the limit of some sequence $\{y^t\}$ of points in $\overline{S}$. Each y^t has $x^t_{j(t)} \geq y^t_{j(t)}$ for some $j(t) \in \mathsf{vars}(S)$. Since $|\mathsf{vars}(S)|$ is finite, for some $j \in \mathsf{vars}(S)$, the infinite subsequence $\{y^t \mid j(t) = j\}$ also has y as a limit point. Then y_j is the limit of the y^t_j's in this subsequence, each of which is at most x_j, so y_j is at most x_j.

Observation 4 *If $x \in \mathbb{R}^n_{\geq 0}$ meets the non-domination constraint for every $S \in \mathcal{C}$ and $y \in \overline{S}^*$, then x is feasible for $(c, \mathcal{C})$.*

Proof Assume x is not feasible. Fix an $S \in \mathcal{C}$ with $x \notin S$. Define $y(\varepsilon)$ by $y_j(\varepsilon) = x_j + \varepsilon$ so $\lim_{\varepsilon \to 0} y = x \notin S$. Since S is closed under limit, $y(\varepsilon') \notin S$ for some $\varepsilon' > 0$. Since x is finite, $x_j < y_j(\varepsilon')$ for each $j \in \mathsf{vars}(S)$. Thus, $x <_S y(\varepsilon')$ (i.e., x fails to meet the non-domination constraint for $(S, y(\varepsilon'))$). □

First Relaxation The non-domination constraints suggest this relaxation of $(c, \mathcal{C})$:

$$\mathsf{minimize} \quad c \cdot x \quad \text{subject to} \quad \left(\forall S \in \mathcal{C}, y \in \overline{S}^*\right) \sum_{j \in \mathsf{vars}(S)} x_j / y_j \geq 1.$$

Let $(c, \mathcal{R}^1)$ denote this Linear-Cost Covering instance. Call it *Relaxation 1*.

Observation 5 *Fix any $x \in \mathbb{R}^n_{\geq 0}$ that is feasible for $(c, \mathcal{R}^1)$.*
Then Δx is feasible for $(c, \mathcal{C})$.

Proof Fix any $S \in \mathcal{C}$ and $y \in \overline{S}^*$.
Then $\sum_{j \in \mathsf{vars}(S)} x_j / y_j \geq 1$. Thus, $\max_{j \in \mathsf{vars}(S)} x_j / y_j \geq 1/|\mathsf{vars}(S)|$.
Thus, $\max_{j \in \mathsf{vars}(S)} \Delta x_j / y_j \geq 1$.
That is, Δx meets the non-domination constraint for (any) (S, y).
By Observation 4, Δx is feasible for $(c, \mathcal{C})$. □

Corollary 2 (Relaxation gap for first relaxation) *The relaxation gap*[16] *for $(c, \mathcal{R}^1)$ is at most Δ.*

Proof Let x be a finite optimal solution for $(c, \mathcal{R}^1)$. By Obs. 5, Δx is feasible for $(c, \mathcal{C})$, and has cost $c \cdot (\Delta x) = \Delta(c \cdot x)$. Thus, the optimal cost for $(c, \mathcal{C})$ is at most Δ times the optimal cost for $(c, \mathcal{R}^1)$. □

Incidentally, $(c, \mathcal{R}^1)$ gives an ellipsoid-based Linear-Cost Covering Δ-approximation algorithm.[17]

Linear-Cost Covering Reduces to Set Cover From the Linear-Cost Covering instance $(c, \mathcal{C})$, construct an equivalent (infinite) Set Cover instance $(c', (E, \mathcal{F}))$ as follows. Recall the non-domination constraints: $x \not<_S y$ for each $S \in \mathcal{C}$ and $y \in \overline{S}^*$.

[16] The relaxation gap is the maximum, over all instances $(c, \mathcal{C})$ of Linear-Cost Covering, of the ratio [*optimal cost for* $(c, \mathcal{C})$]/[*optimal cost for its relaxation* $(c, \mathcal{R}^1)$].

[17] Briefly, run the ellipsoid method to solve $(c, \mathcal{R}^1)$ using a separation oracle that, given x, checks whether $\Delta x \in S$ for all $S \in \mathcal{C}$, and, if not, returns an inequality that x violates for $\mathcal{R}^1$ (from the proof of Observation 5). Either the oracle finds, for some x, that $\Delta x \in S$ for all S, in which case $x' = \Delta x$ is a Δ-approximate solution for $(c, \mathcal{C})$, or the oracle returns to the ellipsoid method a sequence of violated inequalities that, collectively, prove that $(c, \mathcal{R}^1)$ (and thus $(c, \mathcal{C})$) is infeasible.

Such a constraint is met if, for some $j \in \mathsf{vars}(S)$, x_j is assigned a value $r \geq y_j$. Introduce an element $e = (S, y)$ into the element set E for each pair (S, y) associated with such a constraint. For each $j \in [n]$ and $r \in \mathbb{R}_{\geq 0}$, introduce a set $s(j, r)$ into the set family $\mathcal{F}$, such that set $s(j, r)$ contains element (S, y) if assigning $x_j = r$ would ensure $x \not\leq_S y$ (i.e., would satisfy the non-domination constraint for (S, y)). That is, $s(j, r) = \{(S, y) \mid j \in \mathsf{vars}(S),\ r \geq y_j\}$. Take the cost of set $s(j, r)$ to be $c'_{jr} = rc_j$ (equal to the cost of assigning $x_j = r$).

Observation 6 (Reduction to Set Cover) *The* Linear-Cost Covering *instance* $(c, \mathcal{C})$ *is equivalent to the above* Set Cover *instance* $(c', (E, \mathcal{F}))$. *By "equivalent" we mean that each feasible solution* x *to* $(c, \mathcal{C})$ *corresponds to a set cover* X *for* $(E, \mathcal{F})$ (*where* $s(j, r) \in X$ *iff* $x_j = r$) *and, conversely, each set cover* X *for* $(E, \mathcal{F})$ *corresponds to a feasible solution* x *to* $(c, \mathcal{C})$ (*where* $x_j = \sum_{r:s(j,r)\in X} r$). *Each correspondence preserves cost.*

The observation is a consequence of Observation 4.
Note that above reduction increases Δ.

Second Relaxation, via Set Cover Relaxation 2 is the standard LP relaxation of Set Cover, applied to the equivalent Set Cover instance $(c', (E, \mathcal{F}))$ above, with a variable X_{jr} for each set $s(j, r) \in \mathcal{F}$:

$$\text{minimize} \sum_{j,r} rc_j X_{jr} \quad \text{subject to} \quad \left(\forall S \in \mathcal{C}, y \in \overline{S}^*\right) \sum_{j \in \mathsf{vars}(S)} \sum_{r \geq y_j} X_{jr} \geq 1.$$

(There is a technicality in the definition above—the index r of the inner sum ranges over $[y_j, \infty)$. Should one sum, or integrate, over r? Either can be appropriate—the problem and its dual will be well-defined and weak duality will hold either way. Here we restrict attention to solutions X with finite support, so we sum. The same issue arises in the dual below.)

We denote the above relaxation $(c', \mathcal{R}^2)$. By Observation 6, any feasible solution x to $(c, \mathcal{C})$ gives a feasible solution to $(c, \mathcal{R}^2)$ of the same cost (via $X_{jr} = 1$ iff $r = x_j$ and $X_{jr} = 0$) otherwise. Incidentally, any feasible solution X to $(c', \mathcal{R}^2)$ also gives a solution x to $(c, \mathcal{R}^1)$ of the same cost, via $x_j = \sum_r rX_{jr}$. That is, Relaxation 1 is a relaxation of Relaxation 2. The converse is not generally true.[18]

Dual of Set-Cover Relaxation The linear-programming dual of Relaxation 2 is the standard Set Cover dual: fractional packing of elements under (capacitated) sets. We use a variable z_e for each element e:

$$\text{maximize} \sum_{e \in E} z_e \quad \text{subject to} \quad \left(\forall s(j, r) \in \mathcal{F}\right) \sum_{e \in s(j,r)} z_e \leq rc_j.$$

[18]The instance $(c, \mathcal{C})$ defined by $\min\{x_1 + x_2 \mid x \in \mathbb{R}^2_{\geq 0}; x_1 + x_2 \geq 1\}$ has optimum cost 1. In its first relaxation $(c, \mathcal{R}^1)$, $x_1 = x_2 = 1/4$ with cost 1/2 is feasible. But one can show (via duality) that $(c', \mathcal{R}^2)$ has optimal cost at least 1.

Recall $E = \{(S, y) \mid S \in \mathcal{C}, y \in \overline{S}^*\}$; $s(j, r) = \{(S, y) \in E \mid j \in \mathsf{vars}(S),\ r \geq y_j\}$.

We now describe the primal-dual interpretation of Algorithm 1.

Lemma 4 (Primal-dual analysis of Algorithm 1) *Algorithm 1 can be augmented to compute, along with the solution x to $(c, \mathcal{C})$, a solution z to the dual of Relaxation 2 such that $c \cdot x$ is at most Δ times the cost of z. Thus, Algorithm 1 is a Δ-approximation algorithm.*

Proof Initialize $z = \mathbf{0}$. Consider an iteration of Algorithm 1 for some constraint S'. Let x and x', respectively, be the solution x before and after the iteration. Fix element $e' = (S', x')$. Augment Algorithm 1 to raise[19] the dual variable $z_{e'}$ by β. This increases the dual cost by β. Since the iteration increases the cost of x by at most $\beta\Delta$, the iteration maintains the invariant that the cost of x is at most Δ times the dual cost.

To finish, we show the iteration maintains dual feasibility. For any element $e = (S, y) \in E$, let $S(e)$ denote S. Increasing the dual variable $z_{e'}$ by β maintains the following invariant:

$$\text{for all } j \in [n], \quad x_j c_j = \sum_{e : j \in \mathsf{vars}(S(e))} z_e.$$

The invariant is maintained because $z_{e'}$ occurs in the sum iff $j \in \mathsf{vars}(S(e')) = \mathsf{vars}(S')$, and each x_j is increased (by β/c_j) iff $j \in \mathsf{vars}(S')$, so the iteration increases both sides of the equation equally.

Now consider any dual constraint that contains the raised variable $z_{e'}$. Fix the pair (j, r) defining the dual constraint. That $e' \in s(j, r)$ implies $j \in \mathsf{vars}(S')$ and $x'_j \leq r$. Each dual variable z_e that occurs in this dual constraint has $j \in \mathsf{vars}(S(e))$. But, by the invariant, at the end of the iteration, the sum of *all* dual variables z_e with $j \in \mathsf{vars}(S(e))$ equals $x'_j c_j$. Since $x'_j \leq r$, this sum is at most $r c_j$. Thus, the dual constraint remains feasible at the end of the iteration. □

6.1 Valid Local Inequalities; the "Price of Locality"

Here is one general way of characterizing the analyses in this paper in terms of valid inequalities. Note that each of the valid inequalities that is used in Relaxation 1 from Sect. 6 can be obtained by considering some single constraint "$x \in S$" in isolation, and adding valid inequalities for just that constraint. Call such a valid inequality "local". This raises the following question: What if we were to add *all* local valid inequalities (ones that can be obtained by looking at each S in isolation)? What can we say about the relaxation gap of the resulting polytope?

Formally, fix any Submodular-Cost Covering instance $\min\{c(x) \mid x \in S \text{ for all } S \in \mathcal{C}\}$. Consider the "local" relaxation $(c, \mathcal{L})$ obtained as follows. For each constraint $S \in \mathcal{C}$, let $\mathsf{conv}(S)$ denote the convex closure of S. Then let $\mathcal{L} = \{\mathsf{conv}(S) \mid S \in \mathcal{C}\}$.

[19] In fact this dual variable must be 0 before this, because $x'_j > x_j$ for some j, so this dual variable has not been raised before.

Equivalently, for each $S \in \mathcal{C}$, let $\mathcal{L}_S$ contain all of the linear inequalities on variables in $\mathsf{vars}(S)$ that are valid for S, then let $\mathcal{L} = \bigcup_{S \in \mathcal{C}} \mathcal{L}_S$. For Linear-Cost Covering, Relaxation 1 above is a relaxation of $(c, \mathcal{L})$, so Corollary 2 implies that the gap is at most Δ. It is not hard to find examples[20] showing that the gap is at least Δ.

Of course, if we add *all* (not just local) valid inequalities for the feasible region $\bigcap_{S \in \mathcal{C}} S$, then every extreme point of the resulting feasible region is feasible for $(c, \mathcal{C})$, so the relaxation gap would be 1.

7 Fast Implementations for Special Cases of Submodular-Cost Covering

This section has a linear-time implementation of Algorithm 2 for Facility Location (and thus also for Set Cover and Vertex Cover), a nearly linear-time implementation for CMIP-UB, and an $O(N\widehat{\Delta}\log\Delta)$-time implementation for two-stage probabilistic CMIP-UB. (Here N is the number of non-zeroes in the constraint matrix and $\widehat{\Delta}$ is the maximum, over all variables x_j, of the number of constraints that constrain that variable.) The section also introduces a two-stage probabilistic version of Submodular Covering, and shows that it reduces to ordinary Submodular Covering.

For Facility Location, Δ is the maximum number of facilities that might serve any given customer. For Set Cover, Δ is the maximum set size. For Vertex Cover, $\Delta = 2$.

7.1 Linear-Time Implementations for Facility Location, Set Cover, and Vertex Cover

The standard integer linear program for Facility Location is not a covering linear program due to constraints of the form "$x_{ij} \le y_j$". Also, the standard reduction of Facility Location to Set Cover increases Δ exponentially. For these reasons, we formulate Facility Location directly as the following special case of Submodular-Cost Covering, taking advantage of submodular cost:

$$\text{minimize} \quad \sum_j f_j \max_i x_{ij} + \sum_{ij} d_{ij} x_{ij}$$

$$\text{subject to (for each customer } i) \quad \max_{j \in N(i)} x_{ij} \ge 1.$$

[20] Here is an example in $\mathbb{R}^2$. For $v \in \mathbb{R}^2$, let $|v|$ denote the 1-norm $\sum_i |v_i|$. For each $v \in \mathbb{R}^2_{\ge 0}$ such that $|v| = 1$, define constraint set $S_v = \{x \in \mathbb{R}^2_{\ge 0} : (\exists j) x_j \ge v_j\}$. Consider the covering problem $\min\{|x| : (\forall v) x \in S_v\}$.

Each constraint $x \in S_v$ excludes points dominated by v, so the intersection of all S_v's is $\{x \in \mathbb{R}^2_{\ge 0} : |x| \ge 1\}$. On the other hand, since S_v contains the points $(v_1, 0)$ and $(0, v_2)$, $\mathsf{conv}(S_v)$ must contain $x = v_2(v_1, 0) + v_1(0, v_1) = (v_1 v_2, v_1 v_2)$, where $v_1 v_2 \le (1/2)^2 = 1/4$. Thus, each $\mathsf{conv}(S_v)$ contains $x = (1/4, 1/4)$, with $|x| = 1/2$. Thus, the relaxation gap of $(c, \mathcal{L})$ for this instance is at least 2.

Another example with $\Delta = 2$, this time in $\mathbb{R}^n_{\ge 0}$. Consider the sets $S_{ij} = \{x \in \mathbb{R}^n_{\ge 0} : \max(x_i, x_j) \ge 1\}$. Consider the covering problem $\min\{|x| : (\forall i, j) x \in S_{ij}\}$. Each point $x \in \bigcap_{ij} S_{ij}$ has $|x^*| \ge (n-1)/n$, but $x = (1/2, 1/2, 1/2, \ldots, 1/2)$ is in each $\mathsf{conv}(S)$, and $|x| = n/2$, so the relaxation gap of $(c, \mathcal{L})$ is at least 2.

Above $j \in N(i)$ if customer i can use facility j. ($N(i) = \mathsf{vars}(S_i)$ where S_i is the constraint above for customer i.)

Theorem 5 (Linear-time implementations) *For* (*non-metric*) Facility Location, Set Cover, *and* Vertex Cover, *the greedy Δ-approximation algorithm* (*Algorithm 2*) *has a linear-time implementation.*

Proof The implementation is as follows.

1. Start with all $x_{ij} = 0$. Then, for each customer i, in any order, do the following:
2. Let $\beta = \min_{j \in N(i)}[d_{ij} + f_j(1 - \max_{i'} x_{i'j})]$ (the minimum cost to raise x_{ij} to 1 for any $j \in N(i)$).
3. For each $j \in N(i)$, raise x_{ij} by $\min[\beta/d_{ij}, (\beta + f_j \max_{i'} x_{i'j})/(d_{ij} + f_j)]$.
4. Assign each customer i to any facility $j(i)$ with $x_{ij(i)} = 1$.
5. Open the facilities that have customers.

Line 3 raises the x_{ij}'s just enough to increase the cost by β per raised x_{ij} and to increase $\max_{j \in N(i)} x_{ij}$ to 1.

By maintaining, for each facility j, $\max_i x_{ij}$, the implementation can be done in linear time, $O(\sum_i |N(i)|)$.

Set Cover is the special case when $d_{ij} = 0$; Vertex Cover is the further special case $\Delta = 2$. □

7.2 Nearly Linear-Time Implementation for CMIP-UB

This section describes a nearly linear-time implementation of Algorithm 2 for Covering Mixed Integer Linear Programs with upper bounds on the variables (CMIP-UB), that is, problems of the form

$$\min\left\{c \cdot x \mid x \in \mathbb{R}^n_{\geq 0};\ Ax \geq B;\ x \leq u;\ (\forall j \in I)\ x_j \in \mathbb{Z}\right\},$$

where $c \in \mathbb{R}^n_{\geq 0}$, $A \in \mathbb{R}^{m \times n}_{\geq 0}$ and $B \in \mathbb{R}^n_{\geq 0}$ have no negative entries. The set I contains the indices of the variables that are restricted to take integer values, while $u \in \bar{\mathbb{R}}^n_{\geq 0}$ gives the upper bounds on the variables. Δ is the maximum number of non-zeroes in any row of A. We prove the following theorem:

Theorem 6 (Implementation for CMIP-UB) *For* CMIP-UB, *Algorithm 2 can be implemented to return a Δ-approximation in $O(N \log \Delta)$ time, where N is the total number of non-zeroes in the constraint matrix.*

Proof Fix any CMIP-UB instance as described above. For each constraint $A_i x \geq B_i$ (each row of A), do the following. For presentation (to avoid writing the subscript i), rewrite the constraint as $a \cdot x \geq b$ (where $a = A_i$ and $b = B_i$). Then bring the constraint into canonical form, as follows. Assume for simplicity of presentation that integer-valued variables in S come before the other variables (that is, $I \cap \mathsf{vars}(S) = \{1, 2, \ldots, \ell\}$ for some ℓ). Assume for later in the proof that these

ℓ variables are ordered so that $a_1 \geq a_2 \geq \cdots \geq a_\ell$. (These assumptions are without loss of generality.) Now incorporate the variable-domain restrictions ($x \leq u$ and $(\forall j \in I)\ x_j \in \mathbb{Z}$) into the constraint by rewriting it as follows:

$$\sum_{j=1}^{\ell} a_j \lfloor \min(x_j, u_j) \rfloor + \sum_{j>\ell} a_j \min(x_j, u_j) \geq b.$$

(canonical constraint S for $A_i x \geq B_i$)

Let $\mathcal{C}$ be the collection of such canonical constraints, one for each original covering constraint $A_i x \geq B_i$.

Intuition The algorithm focuses on a single unsatisfied $S \in \mathcal{C}$, repeating an iteration of Algorithm 2 (raising the variables x_j for $j \in \mathsf{vars}(S)$) until S is satisfied. It then moves on to another unsatisfied S, and so on, until all constraints are satisfied. While working with a particular constraint S, it increases each x_j for $j \in \mathsf{vars}(S)$ by β/c_j for some β. We must choose $\beta \leq \tilde{c}_x(S)$ (the optimal cost to augment x to satisfy S), thus each step requires some lower bound on $\tilde{c}_x(S)$. But the steps must also be large enough to satisfy S quickly.

For intuition, consider first the case when S has no variable upper bounds (each $u_j = \infty$) and no floors. In this case, the optimal augmentation of x to satisfy S simply raises the single most cost-effective variable x_j (minimizing a_j/c_j) to satisfy S, so $\tilde{c}_x(S)$ is easy to calculate exactly and taking $\beta = \tilde{c}_x(S)$ satisfies S in one iteration.

Next consider the case when S has some variable upper bounds (finite u_j). In this case, we take β to be the minimum cost to *either* satisfy S *or* bring some variable to its upper bound (we call this *saturating* the variable). This β is easy to calculate, and will satisfy S after at most $\mathsf{vars}(S)$ iterations (as each variable can be saturated at most once).

Finally, consider the case when S also has floors. This complicates the picture considerably. The basic idea is to relax (remove) the floors, satisfy the relaxed constraint as described above, and then reintroduce the floors one by one. We reintroduce a floor only once the constraint *without* that floor is already satisfied. This ensures that *the constraint with the floor will be satisfied if the term with the floor increases even once*. (If the term for a floored variable x_j increases, we say x_j is *bumped*.) We also reintroduce the floors in a particular order—in order of decreasing a_j. This ensures that introducing one floor (which lowers the value of the left-hand side) does not break the property in italics above for previously reintroduced floors.

The above approach ensures that S will be satisfied in $O(\mathsf{vars}(S))$ iterations. A careful but straightforward use of heaps allows all the iterations for S to be done in $O(\mathsf{vars}(S) \log \Delta)$ time. This will imply the theorem.

Here are the details. To specify the implementation of Algorithm 2, we first specify how, in each iteration, for a given constraint $S \in \mathcal{C}$ and $x \notin S$, the implementation chooses the step size β. It starts by finding a relaxation S^h of S (that is, $S \subseteq S^h$, so $\tilde{c}_x(S^h) \leq \tilde{c}_x(S)$). Having chosen the relaxation, the algorithm then takes β to be the minimum cost needed to raise any *single* variable x_j (with $j \in \mathsf{vars}(S)$) just enough to either satisfy the relaxation S^h or to cause $x_j = u_j$.

The relaxation S^h is as follows. Remove all floors from S, then add in just enough floors (from left to right), so that the resulting constraint is unsatisfied. Let S^h be the resulting constraint, where h is the number of floors added in. Formally, for $h = 0, 1, \ldots, \ell$, define $f^h(x) = \sum_{j=1}^{h} a_j \lfloor \min(x_j, u_j) \rfloor + \sum_{j>h} a_j \min(x_j, u_j)$ to be the left-hand side of constraint S above, with only the first h floors retained. Then fix $h = \min\{h \geq 0 \mid f^h(x) < b\}$, and take $S^h = \{x \mid f^h(x) \geq b\}$.

Next we show that this β satisfies the constraint in Algorithm 2.

Lemma 5 (Validity of step size) *For S, $x \notin S$, and β as described above, $\beta \in [0, \tilde{c}_x(S)]$.*

Proof As $S \subseteq S^h$, it suffices to prove $\beta \leq \tilde{c}_x(S^h)$. Recall that a variable x_j is *saturated* if $x_j = u_j$. Focus on the unsaturated variables in $\mathsf{vars}(S)$. We must show that if we wish to augment (increase) some variables just enough to saturate a variable or bring x into S^h, then we can achieve this at minimum cost by increasing a single variable. This is certainly true if we saturate a variable: only that variable needs to be increased. A special case of this is when some c_i is 0—we can saturate x_i at zero cost, which is minimum. Therefore, consider the case where all c_i's are positive and the variable increases bring x into S^h.

Let P be the set of unsaturated variables in $\{x_1, \ldots, x_h\}$, and let Q be the set of unsaturated variables among $\{x_j \mid j > h\}$. Consider increasing a variable $x_j \in P$. Until x_j is bumped (i.e., the term $\lfloor x_j \rfloor + 1$ increases because x_j reaches its next higher integer), $f^h(x)$ remains unchanged, but the cost increases. Thus, if it is optimal to increase x_j at all, x_j must be bumped. When x_j is bumped, $f^h(x)$ jumps by a_j, which (by the ordering of coefficients) is at least a_h, which (by the choice of h) is sufficient to bring x into S^h. Thus, if the optimal augmentation increases a variable in P, then the only variable that it increases is that one variable, which is bumped once.

The only remaining case is when the optimal augmentation of x increases only variables from Q. Let $x_k = \arg\min\{c_j/a_j \mid x_j \in Q\}$. Clearly it is not advantageous to increase any variable in Q other than x_k. (Let $\delta_j \geq 0$ denote the amount by which we increase $x_j \in Q$. If $\delta_j > 0$ for some $j \neq k$, then we can set $\delta_j = 0$ and instead increase δ_k by $a_j \delta_j / a_k$. this will leave the increase in $f^h(x)$ intact, so x will still be brought into S^h, yet will not inflate the cost increase, because the cost will decrease by $c_j \delta_j$, but increase by $c_k a_j \delta / a_k \leq c_j \delta_j$, where the inequality holds by the definition of k.) □

By the lemma and Theorem 1, with this choice of β, the algorithm gives a Δ-approximation. It remains to bound the running time.

Lemma 6 (Iterations) *For each $S \in \mathcal{C}$, the algorithm does at most $2|\mathsf{vars}(S)|$ iterations for S.*

Proof Recall that, in a given iteration, β is the minimum such that raising some single x_k by β/c_k (with $k \in \mathsf{vars}(S)$ and $x_k < u_k$) is enough to saturate x_k or bring x into S^h. If the problem is feasible, $\beta < \infty$ so there is such an x_k. Each iteration increases x_j for *all* $j \in \mathsf{vars}(S)$ by β/c_j, so must increase this x_k by β/c_k. Thus, the iteration either saturates x_k or brings x into S^h.

The number of iterations for S that saturate variable is clearly at most $|\mathsf{vars}(S)|$. The number of iterations for S that satisfy that iteration's relaxation (bringing x into S^h) is also at most $|\mathsf{vars}(S)|$, because, by the choice of h, in the next iteration for S the relaxation index h will be at least 1 larger. Thus, there are at most $2|\mathsf{vars}(S)|$ iterations for S before $x \in S$. □

The obvious implementation of an iteration for a given constraint S runs in time $O(|\mathsf{vars}(S)|)$ (provided the constraint's a_j's are sorted in a preprocessing step). By the lemma, the obvious implementation thus yields total time $O(\sum_S |\mathsf{vars}(S)|^2) \leq O(\sum_S |\mathsf{vars}(S)|\Delta) = O(N\Delta)$.

To complete the proof of Theorem 6, we show how to use standard heap data structures to implement the above algorithm to run in $O(N \log \Delta)$ time. The implementation considers the constraints $S \in \mathcal{C}$ in any order. For a given S, it repeatedly does iterations for that S until $x \in S$. As the iterations for a given S proceed, the algorithm maintains the following quantities:

- A fixed vector x^b, which is x at the start of the first iteration for S, initialized in time $O(|\mathsf{vars}(S)|)$.
- A variable τ, tracking the sum of the β's for S so far (initially 0). Crucially, the current x then satisfies $x_j = x_j^b + \tau/c_j$ for $j \in \mathsf{vars}(S)$. While processing a given S, we use this to represent x implicitly.

  We then use the following heaps to find each *breakpoint* of τ—each value at which a variable becomes saturated, is bumped, or at which S^h is satisfied and the index h of the current relaxation S^h increases. We stop when S^ℓ (that is, S) is satisfied.
- A heap containing, for each unsaturated variable x_j in $\mathsf{vars}(S)$, the value $c_j(u_j - x_j^b)$ of τ at which x_j would saturate. This value does not change until x_j is saturated, at which point the value is removed from the heap.
- A heap containing, for each unsaturated integer variable x_j $(j \leq h)$ in S^h, the value of τ at which x^j would next be bumped. This value is initially $c_j(1 - (x_j^b - \lfloor x_j^b \rfloor))$. It changes only when x_j is bumped, at which point it increases by c_j.
- A heap containing, for each unsaturated non-integer variable x_j $(j > h)$ in S^j, the ratio c_j/a_j. This value does not change. It is removed from the heap when x_j is saturated.
- The current *derivative* d of $f^h(x)$ with respect to τ, which is $d = \sum_{j>h, x_j<u_j} a_j/c_j$. This value changes by a single term whenever a variable is saturated or h increases.
- The current slack $b^h = b - f^h(x)$ of S^h, updated at each breakpoint of τ.

In each iteration, the algorithm queries the min-values of each of the three heaps. It uses the three values to calculate the minimum value of τ at which, respectively, a variable would become saturated, a variable would be bumped, or a single (non-integer) variable's increase would increase $f^h(x)$ by the slack b^h. It then increases τ to the minimum of these three values. (This corresponds to doing a step of Algorithm 1 with β equal to the increase in τ.) With the change in τ, it detects each saturation, bump, and increment of h that occurs, uses the derivative to compute the increase in $f^h(x)$, then updates the data structures accordingly. (For example, it removes saturated variables' keys from all three heaps.)

After the algorithm has finished all iterations for a given constraint S, it explicitly sets $x_j \leftarrow x_j^b + \tau/c_j$ for $j \in \mathsf{vars}(S)$, discards the data structures for S, and moves on to the next constraint.

The heap keys for a variable x_j change (and are inserted or removed) only when that particularly variable is bumped, or saturated, or when h increases to j. Each variable is saturated at most once, and h increases at most $\ell \le \mathsf{vars}(S)$ times, and thus there are at most $\mathsf{vars}(S)$ bumps (as each bump increases h by at least 1). Thus, during all iterations for S, the total number of breakpoints and heap key changes is $O(\mathsf{vars}(S))$. Since each heap operation takes $O(\log \Delta)$ time, the overall time is then $O(\sum_{S\in\mathcal{C}} |\mathsf{vars}(S)| \log \Delta) = O(N \log \Delta)$, where N is the number of non-zeros in A.

This proves the theorem. □

7.3 Two-Stage (Probabilistic) Submodular-Cost Covering

An instance of *two-stage* Submodular-Cost Covering is a tuple $(W, p, (c, \mathcal{C}))$ where $(c, \mathcal{C})$ is an instance of Submodular-Cost Covering over n variables (so $S \subseteq \bar{\mathbb{R}}^n_{\ge 0}$ for each $S \in \mathcal{C}$), $W : \bar{\mathbb{R}}^{|\mathcal{C}|\times n}_{\ge 0} \to \bar{\mathbb{R}}_{\ge 0}$ is a non-decreasing, submodular, continuous *first-stage* objective function, and, for each $S \in \mathcal{C}$, the *activation probability* of S is p_S. A solution is a collection $X = [x^S]_{S\in\mathcal{C}}$ of vectors $x^S \in \bar{\mathbb{R}}^n_{\ge 0}$, one for each constraint $S \in \mathcal{C}$, such that $x^S \in S$. Intuitively, x^S specifies how S will be satisfied if S is activated, which happens with probability p_S. As usual $\Delta = \max_{S\in\mathcal{C}} |\mathsf{vars}(S)|$.

The solution should minimize the cost $w(X)$ of X, as defined by the following random experiment. Each constraint S is independently *activated* with probability p_S. This defines a Submodular-Cost Covering instance $(c, \mathcal{A})$ where $\mathcal{A} = \{S \in \mathcal{C} \mid S \text{ is activated}\} \subseteq \mathcal{C}$, and the solution $x^{\mathcal{A}}$ for that instance defined by $x_j^{\mathcal{A}} = \max\{x_j^S \mid S \in \mathcal{A}\}$. Intuitively, $x^{\mathcal{A}}$ is the minimal vector that meets the first-stage commitment to satisfy each activated constraint S with x^S. The cost $w(X)$ is then $W(X) + E_{\mathcal{A}}[c(x^{\mathcal{A}})]$, the first-stage cost $W(X)$ (modeling a "preparation" cost) plus the (expectation of the) second-stage cost $c(x^{\mathcal{A}})$ (modeling an additional cost for assembling the final solution to the second-stage Submodular-Cost Covering instance $(c, \mathcal{A})$).

Facility-Location Example For example, consider a Set Cover instance $(c, \mathcal{C})$ with elements $[m]$ and sets $s(j) \subseteq [m]$ for $j \in [n]$. That is, $\mathsf{minimize}\, c \cdot x \,\mathsf{subject\ to}\, x \in \mathbb{R}^n_{\ge 0}$, $(\forall i \in [m]) \max_{j:i\in s(j)} x_j \ge 1$.

Extend this to a two-stage Set Cover instance $(W, p, (c, \mathcal{C}))$ where $W_{ij} \ge 0$ and each $p_i = 1$. Let $X = [x^i]_i$ be any (minimal) feasible solution to this instance. That is, $x^i \in \{0, 1\}^n$ says that element i chooses the set $s(j)$ where $x_j^i = 1$. All constraints are activated in the second stage, so each $x_j^{\mathcal{A}} = \max\{x_j^i \mid i \in s(j)\}$. That is, $x_j^{\mathcal{A}} = 1$ iff any element i has chosen set $s(j)$. The cost $w(X)$ is $\sum_{ij} W_{ij} x_j^i + \sum_j c_j \max\{x_j^i \mid i \in s(j)\}$.

Note that this two-stage Set Cover problem exactly models Facility Location. The first-stage cost W captures the assignment cost; the second-stage cost c captures the opening cost.

Consider again general two-stage Submodular-Cost Covering. A Δ-approximation algorithm for it follows immediately from the following observation:

Observation 7 *Two-stage* Submodular-Cost Covering *reduces to* Submodular-Cost Covering (*preserving* Δ).

Proof Any two-stage instance $(W, p, (c, \mathcal{C}))$ over n variables is equivalent to a standard instance $(w, \mathcal{C}')$ over $n|\mathcal{C}|$ variables $(X = [x^S]_{S\in\mathcal{C}})$ where $w(X)$ is the cost of X for the two-stage instance as defined above, and, for each $S \in \mathcal{C}$, there is a corresponding constraint $x^S \in S$ on X in $\mathcal{C}'$. One can easily verify that the cost $w(X)$ is submodular, non-decreasing, and continuous because $W(X)$ and $c(x)$ are. □

Next we describe a fast implementation of Algorithm 2 for two-stage CMIP-UB—the special case of two-stage Submodular-Cost Covering where W is linear and the pair $(c, \mathcal{C})$ form a CMIP-UB instance.

Theorem 7 (Implementation for two-stage CMIP-UB) *For two-stage* CMIP-UB:

(a) *Algorithm 2 can be implemented to return a Δ-approximation in $O(N\widehat{\Delta}\log\Delta)$ time, where $\widehat{\Delta}$ is the maximum number of constraints per variable and N is the input size $\sum_{S\in\mathcal{C}}|\mathsf{vars}(S)|$.*

(b) *When $p = \mathbf{1}$, the algorithm can be implemented to run in time $O(N\log\Delta)$. (The case $p = \mathbf{1}$ of two-stage* CMIP-UB *generalizes* CMIP-UB *and* Facility Location.)

Proof Fix an instance $(W, p, (c, \mathcal{C}))$ of two-stage CMIP-UB. Let $(w, \mathcal{C}')$ be the equivalent instance of standard Submodular-Cost Covering from Observation 7 over variable vector $X = [x^S]_{S\in\mathcal{C}}$. Let random variable $x^{\mathcal{A}}$ be as described in the problem definition ($x_j^{\mathcal{A}} = \max\{x_j^S \mid S \text{ active}\}$), so that $w(X) = W \cdot X + E[c \cdot x^{\mathcal{A}}]$.

We implement Algorithm 2 for the Submodular-Cost Covering instance $(w, \mathcal{C}')$. In an iteration of the algorithm for a constraint S on x^S, the algorithm computes β as follows. Recall that the variables in X being increased (to satisfy $x^S \in S$) are x_j^S for $j \in \mathsf{vars}(S)$. The derivative of $w(X)$ with respect to x_j^S is

$$\begin{aligned} c'_j &= W_j^S + c_j \Pr\left[x_j^S \text{ determines } x_j^{\mathcal{A}}\right] \\ &= W_j^S + c_j p_S \prod\left\{1 - p_R \mid x_j^R > x_j^S,\ j \in \mathsf{vars}(R)\right\}. \end{aligned}$$

The derivative will be constant (that is, $w(X)$ will be linear in x^S) until x_j^S reaches its next breakpoint $t_j = \min\{x_j^R \mid x_j^R > x_j^S,\ j \in \mathsf{vars}(R)\}$. Define $\beta_t = \min\{(t_j - x_j^S)c'_j \mid j \in \mathsf{vars}(S)\}$ to be the minimum cost to bring any x_j^S to its next breakpoint.

Let w' be the vector defined above (the gradient of w with respect to x^S). Let β' be the step size that the algorithm in Theorem 6 would compute given the linear cost w'. That is, that it would compute in an iteration for constraint $x^S \in S$ given the CMIP-UB instance $(w', \{S\})$ and the current x^S.

The algorithm here computes β_t and β' as defined above, then takes the step size β to be $\beta = \min(\beta_t, \beta')$. This β is a valid lower bound on $\tilde{c}_X(S)$, because β_t is the minimum cost to bring any x_j^S to its next breakpoint, while $\beta' \le \tilde{c}'_{x^S}(S)$ is a lower bound on the cost to satisfy S without bringing any x_j^S to a breakpoint. Thus, by Theorem 1, this algorithm computes a Δ-approximation.

The algorithm is as follows. It considers the constraints in any order. For each constraint S, it does iterations for that S, with step size β defined above, until S is satisfied.

Lemma 7 (Iterations) *For each $S \in \mathcal{C}$, the algorithm does at most $|\mathsf{vars}(S)|(\widehat{\Delta}+2)$ iterations for S.*

Proof An iteration may cause some x_j^S to reach its next breakpoint t_j. By inspection of the breakpoints t_j, each x_j^S can cross at most $\widehat{\Delta}$ breakpoints (one for each constraint R on x_j in the original instance). Thus, there are at most $|\mathsf{vars}(S)|\widehat{\Delta}$ such iterations. In each remaining iteration the step size β equals the step size β' from the algorithm in Theorem 6. Following the proof of Lemma 6 in Theorem 6, there are at most $2|\mathsf{vars}(S)|$ such iterations. (In each such iteration, either some variable x_j^S reaches its upper bound u_j for the first time, or the constraint $x_j^S \in S^h$ is satisfied for the current relaxation S^h of S. By inspection, S^h depends only on the current x^S and the constraint S, and not on the cost function w'. Thus, as in the proof of Lemma 6, after an iteration for S where the current S^h is satisfied, in the next iteration, h will be at least one larger. That can happen at most $|\mathsf{vars}(S)|$ times.) □

To complete the proof of Theorem 7, we prove that algorithm can be implemented to take time $O(N\widehat{\Delta}\log\Delta)$, or, if $p = \mathbf{1}$, time $O(N\log\Delta)$.

As the algorithm does iterations for S, the algorithm maintains the data structures described at the end of the proof of Theorem 6, with the following adjustments. When some x_j^S reaches its next breakpoint and w'_j increases, the algorithm

- raises x_j^b to maintain the invariant $x_j = x_j^b + \tau/w'_j$;
- updates the derivative d to account for the change in the term a_j/c_j (if present in the derivative), and
- updates the values for key j in the three heaps (where present).

By inspection of the proof of Theorem 6, these adjustments are enough to maintain the data structures correctly throughout all iterations for S. The updates take $O(\log\Delta)$ time per breakpoint. Thus, the total time for the adjustments is $O(\sum_S |\mathsf{vars}(S)|\widehat{\Delta}\log\Delta)$, which is $O(N\widehat{\Delta}\log\Delta)$.

To compute β_t in each iteration, the algorithm does the following. As it is doing iterations for a particular constraint S, recall that τ is the sum of the β's for S so far (from the proof of Theorem 6). The algorithm maintains a fourth heap containing values $\{\tau + (t_j - x_j^S)w'_j \mid j \in \mathsf{vars}(S)\}$ (the values in the definition of β_t, plus τ). Then β_t is the minimum value in this heap, minus τ.

Then x_j^S reaches a breakpoint (and w'_j changes) if and only if $\beta = \beta_t$ and key j has minimum value in this heap. When that happens, the algorithm finds the next breakpoint t'_j for j (as described in the next paragraph) and updates j's value in the fourth heap. The total time spent maintaining the fourth heap is $O(\log\Delta)$ per breakpoint, $O(\sum_S \sum_{j\in\mathsf{vars}(S)} \widehat{\Delta}\log\Delta) = O(N\widehat{\Delta}\log\Delta)$.

The algorithm computes the breakpoints t_j efficiently as follows. Throughout the entire computation (not just the iterations for S), the algorithm maintains, for each j,

an array of j's variables in X, that is, $\{x_j^R \mid R \in \mathcal{C}, j \in \mathsf{vars}(R)\}$, sorted by the variables' current values (initially all 0). Then t_j is the value of the first x_j^R after x_j^S in j's list. When x_j^S reaches its breakpoint t_j (detected as described in the previous paragraph), the algorithm updates the list order by swapping x_j^S with the x_j^R following it in the list (the one with value t_j). The next breakpoint is then the value of the variable $x_j^{R'}$ that was after x_j^R and is now after x_j^S. The time spent computing breakpoints in this way is proportional to the total number of swaps, which is proportional to the total number of breakpoints, which is at most $\sum_S \sum_{j \in \mathsf{vars}(S)} \widehat{\Delta} = N\widehat{\Delta}$.

This concludes the proof for the general case.

When $p = \mathbf{1}$, note that in this case the product in the equation for c'_j is 1 if $x_j^S = \max_R x_j^R$ and 0 otherwise. So each constraint S has at most one breakpoint per variable, and the total time for the adjustments above reduces to $O(\sum_S |\mathsf{vars}(S)| \log \Delta) = O(N \log \Delta)$. As in the proof of Theorem 6, the remaining operations also take $O(N \log \Delta)$ time.

This concludes the proof of the theorem. □

Acknowledgements The authors gratefully acknowledge Marek Chrobak for useful discussions, and two anonymous reviewers for careful and constructive reviews that helped improve the presentation.

This work was partially supported by National Science Foundation (NSF) grants CNS-0626912 and CCF-0729071.

Appendix

Proof of Observation 1 (reduction to canonical form) Here is the reduction: Let $(c, U, \mathcal{C})$ be any instance of Submodular-Cost Covering. Construct its canonical form $(c, \mathcal{C}')$ as follows. First, assume without loss of generality that $\min U_j = 0$ for each j. (If not, let $\ell_j = \min U_j$, then apply the translation $x \leftrightarrow x' + \ell$ to the cost and feasible region: rewrite the cost $c(x)$ as $c'(x') = c(x' + \ell)$; rewrite each constraint "$x \in S$" as "$x' \in S - \ell$"; replace each domain U_j by $U'_j = U_j - \ell_j$.)

Next, define $\mu_j(x) = \max\{\alpha \in U_j \mid \alpha \le x_j\}$ (that is, $\mu(x)$ is x with each coordinate lowered into U_j). For each constraint S in $\mathcal{C}$, put a corresponding constraint "$\mu(x) \in S$" in $\mathcal{C}'$. The new constraint is closed upwards and closed under limit because S is and μ is non-decreasing. It is not hard to verify that any solution x to the canonical instance $(c, \mathcal{C}')$ gives a corresponding solution $\mu(x)$ to the original instance $(c, U, \mathcal{C})$, and that this reduction preserves Δ-approximation. □

References

1. Albers, S.: On generalized connection caching. Theory Comput. Syst. **35**(3), 251–267 (2002)
2. Bansal, N., Buchbinder, N., Naor, J.S.: A primal-dual randomized algorithm for weighted paging. In: The Forty-third IEEE Symposium on Foundations of Computer Science, pp. 507–517 (2007)
3. Bansal, N., Buchbinder, N., Naor, J.S.: Randomized competitive algorithms for generalized caching. In: The Fourtieth ACM Symposium on Theory of Computing, pp. 235–244 (2008)
4. Bar-Yehuda, R.: One for the price of two: A unified approach for approximating covering problems. Algorithmica **27**(2), 131–144 (2000)

5. Bar-Yehuda, R., Even, S.: A linear-time approximation algorithm for the Weighted Vertex Cover problem. J. Algorithms **2**(2), 198–203 (1981)
6. Bar-Yehuda, R., Even, S.: A local-ratio theorem for approximating the weighted vertex cover problem. Ann. Discrete Math. **25**(27–46), 50 (1985)
7. Bar-Yehuda, R., Rawitz, D.: Efficient algorithms for integer programs with two variables per constraint. Algorithmica **29**(4), 595–609 (2001)
8. Bar-Yehuda, R., Rawitz, D.: On the equivalence between the primal-dual schema and the local-ratio technique. SIAM J. Discrete Math. **19**(3), 762–797 (2005)
9. Bar-Yehuda, R., Bendel, K., Freund, A., Rawitz, D.: Local ratio: a unified framework for approximation algorithms. ACM Comput. Surv. **36**(4), 422–463 (2004)
10. Bertsimas, D., Vohra, R.: Rounding algorithms for covering problems. Math. Program. **80**(1), 63–89 (1998)
11. Borodin, A., Cashman, D., Magen, A.: How well can primal-dual and local-ratio algorithms perform? In: The Thirty-Second International Colloquium on Automata, Languages and Programming (2005)
12. Borodin, A., Cashman, D., Magen, A.: How well can primal-dual and local-ratio algorithms perform? ACM Trans. Algorithms **7**(3), 29 (2011).
13. Buchbinder, N., Naor, J.S.: Online primal-dual algorithms for covering and packing problems. In: The Thirteenth European Symposium on Algorithms. Lecture Notes in Computer Science, vol. 3669, pp. 689–701 (2005)
14. Buchbinder, N., Naor, J.S.: Online primal-dual algorithms for covering and packing problems. Math. Oper. Res. **34**(2), 270–286 (2009)
15. Cao, P., Irani, S.: Cost-aware www proxy caching algorithms. In: The 1997 USENIX Symposium on Internet Technology and Systems, pp. 193–206 (1997)
16. Carr, R.D., Fleischer, L.K., Leung, V.J., Phillips, C.A.: Strengthening integrality gaps for capacitated network design and covering problems. In: The Eleventh ACM-SIAM Symposium on Discrete Algorithms, pp. 106–115 (2000)
17. Chrobak, M., Karloff, H., Payne, T., Vishwanathan, S.: New results on server problems. SIAM J. Discrete Math. **4**(2), 172–181 (1991)
18. Chudak, F.A., Nagano, K.: Efficient solutions to relaxations of combinatorial problems with submodular penalties via the Lovász extension and non-smooth convex optimization. In: The Eighteenth ACM-SIAM Symposium on Discrete Algorithms, pp. 79–88 (2007)
19. Chvátal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**, 233–235 (1979)
20. Cohen, E., Kaplan, H., Zwick, U.: Connection caching. In: The Thirty-First ACM Symposium on Theory of Computing, pp. 612–621 (1999)
21. Cohen, E., Kaplan, H., Zwick, U.: Connection caching under various models of communication. In: The Twelfth ACM Symposium on Parallel Algorithms and Architectures, pp. 54–63 (2000)
22. Cohen, E., Kaplan, H., Zwick, U.: Connection caching: Model and algorithms. J. Comput. Syst. Sci. **67**(1), 92–126 (2003)
23. Dilley, J., Arlitt, M., Perret, S.: Enhancement and validation of Squid's cache replacement policy. In: Fourth International Web Caching Workshop (1999)
24. Dinur, I., Safra, S.: On the hardness of approximating minimum vertex cover. Ann. Math. **162**, 439–485 (2005)
25. Fiat, A., Karp, R.M., Luby, M., McGeoch, L.A., Sleator, D.D., Young, N.E.: Competitive paging algorithms. J. Algorithms **12**, 685–699 (1991)
26. Gonzales, T. (ed.): Approximation Algorithms and Metaheuristics (Greedy Methods). Taylor & Francis, London (2007)
27. Hall, N.G., Hochbaum, D.S.: A fast approximation algorithm for the multicovering problem. Discrete Appl. Math. **15**(1), 35–40 (1986)
28. Halldórsson, M.M., Radhakrishnan, J.: Greed is good: Approximating independent sets in sparse and bounded-degree graphs. In: The Twenty-Sixth ACM Symposium on Theory of Computing, pp. 439–448 (1994)
29. Halldórsson, M.M., Radhakrishnan, J.: Greed is good: Approximating independent sets in sparse and bounded-degree graphs. Algorithmica **18**(1), 145–163 (1997)
30. Halperin, E.: Improved approximation algorithm for the Vertex Cover problem in graphs and hypergraphs. SIAM J. Comput. **31**(5), 1608–1623 (2002)
31. Håstad, J.: Some optimal inapproximability results. J. ACM **48**(4), 798–859 (2001)
32. Hayrapetyan, A., Swamy, C., Tardos, E.: Network design for information networks. In: The Sixteenth ACM-SIAM Symposium on Discrete Algorithms, pp. 933–942 (2005)

33. Hochbaum, D.S.: Approximation algorithms for the Set Covering and Vertex Cover problems. SIAM J. Comput. **11**, 555–556 (1982)
34. Hochbaum, D.S.: Efficient bounds for the Stable Set, Vertex Cover, and Set Packing problems. Discrete Appl. Math. **6**, 243–254 (1983)
35. Hochbaum, D.S.: Approximation Algorithms for NP-hard Problems. PWS Publishing, Boston (1996)
36. Iwata, S., Nagano, K.: Submodular function minimization under covering constraints. In: The Fiftieth IEEE Symposium on Foundations of Computer Science, pp. 671–680 (2009)
37. Johnson, D.S.: Approximation algorithms for combinatorial problems. In: The Fifth ACM Symposium on Theory of Computing, pp. 38–49 (1973)
38. Johnson, D.S.: Approximation algorithms for combinatorial problems. J. Comput. Syst. Sci. **9**(3), 256–278 (1974)
39. Karlin, A.R., Manasse, M.S., Rudolph, L., Sleator, D.D.: Competitive snoopy caching. Algorithmica **3**, 77–119 (1988)
40. Khot, S., Regev, O.: Vertex Cover might be hard to approximate to within 2-ε. J. Comput. Syst. Sci. **74**, 335–349 (2008)
41. Kolliopoulos, S.G., Young, N.E.: Approximation algorithms for covering/packing integer programs. J. Comput. Syst. Sci. **71**(4), 495–505 (2005)
42. Koufogiannakis, C., Young, N.E.: Distributed and parallel algorithms for weighted vertex cover and other covering problems. In: The Twenty-Eighth ACM Symposium on Principles of Distributed Computing, pp. 171–179 (2009)
43. Koufogiannakis, C., Young, N.E.: Distributed fractional packing and maximum weighted b-matching via tail-recursive duality. In: The Twenty-Third International Symposium on Distributed Computing, pp. 221–238 (2009)
44. Koufogiannakis, C., Young, N.E.: Greedy δ-approximation algorithm for covering with arbitrary constraints and submodular cost. In: The Thirty-Sixth International Colloquium on Automata, Languages, and Programming. Lecture Notes in Computer Science, vol. 5555, pp. 634–652 (2009)
45. Koufogiannakis, C., Young, N.E.: Distributed algorithms for covering, packing and maximum weighted matching. Distrib. Comput. **24**, 45–63 (2011)
46. Kuhn, F., Moscibroda, T., Wattenhofer, R.: The price of being near-sighted. In: The Seventeenth ACM-SIAM Symposium on Discrete Algorithm, pp. 980–989 (2006)
47. Lotker, Z., Patt-Shamir, B., Rawitz, D.: Rent, lease or buy: Randomized algorithms for multislope ski rental. In: The Twenty-Fifth Symposium on Theoretical Aspects of Computer Science, pp. 503–514 (2008)
48. Lovász, L.: On the ratio of optimal integral and fractional covers. Discrete Math. **13**(4), 383–390 (1975)
49. McGeoch, L.A., Sleator, D.D.: A strongly competitive randomized paging algorithm. Algorithmica **6**(1), 816–825 (1991)
50. Monien, B., Speckenmeyer, E.: Ramsey numbers and an approximation algorithm for the Vertex Cover problem. Acta Inform. **22**, 115–123 (1985)
51. Orlin, J.B.: A faster strongly polynomial time algorithm for submodular function minimization. In: The Twelfth Conference on Integer Programming and Combinatorial Optimization, pp. 240–251 (2007)
52. Orlin, J.B.: A faster strongly polynomial time algorithm for submodular function minimization. Math. Program. **118**(2), 237–251 (2009)
53. Papadimitriou, C.H., Yannakakis, M.: Linear programming without the matrix. In: The Twenty-Fifth ACM Symposium on Theory of Computing, pp. 121–129 (1993)
54. Pritchard, D.: Approximability of sparse integer programs. In: The Seventeenth European Symposium on Algorithms. Lecture Notes in Computer Science, vol. 5757, pp. 83–94 (2009)
55. Pritchard, D., Chakrabarty, D.: Approximability of sparse integer programs. Algorithmica **61**(1), 75–93 (2011)
56. Raghavan, P., Snir, M.: Memory versus randomization in on-line algorithms. IBM J. Res. Dev. **38**(6), 683–707 (1994)
57. Ravi, R., Sinha, A.: Hedging uncertainty: Approximation algorithms for stochastic optimization problems. Math. Program. **108**(1), 97–114 (2006)
58. Shmoys, D., Swamy, C.: Stochastic optimization is (almost) as easy as deterministic optimization. In: The Forty-Fifth IEEE Symposium on Foundations of Computer Science, pp. 228–237 (2004)
59. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985)

60. Srinivasan, A.: Improved approximation guarantees for packing and covering integer programs. SIAM J. Comput. **29**, 648–670 (1999)
61. Srinivasan, A.: New approaches to covering and packing problems. In: The Twelfth ACM-SIAM Symposium on Discrete Algorithms, pp. 567–576 (2001)
62. Vazirani, V.V.: Approximation Algorithms. Springer, Berlin (2001)
63. Young, N.E.: On-line caching as cache size varies. In: The Second ACM-SIAM Symposium on Discrete Algorithms, pp. 241–250 (1991)
64. Young, N.E.: The k-server dual and loose competitiveness for paging. Algorithmica **11**, 525–541 (1994)
65. Young, N.E.: On-line file caching. In: The Twelfth ACM-SIAM Symposium on Discrete Algorithms, pp. 82–86 (1998)
66. Young, N.E.: On-line file caching. Algorithmica **33**(3), 371–383 (2002)