SIAM J. OPTIM.
Vol. 27, No. 4, pp. 2563–2596

A FASTER ALGORITHM SOLVING A GENERALIZATION OF ISOTONIC MEDIAN REGRESSION AND A CLASS OF FUSED LASSO PROBLEMS*

DORIT S. HOCHBAUM† AND CHENG LU†

Abstract. Many applications in the areas of production, signal processing, economics, bioinformatics, and statistical learning involve a given partial order on certain parameters and a set of noisy observations of the parameters. The goal is to derive estimated values of the parameters that satisfy the partial order that minimize the *loss* of *deviations* from the given observed values. A prominent application of this setup is the well-known isotonic regression problem where the partial order is a total order. A commonly studied isotonic regression problem is the isotonic median regression (IMR) where the loss function is a sum of absolute value functions on the deviations of the estimated values from the observation values. We present here the most efficient algorithm to date for a generalization of IMR with any convex piecewise linear loss function as well as for variant models that include ℓ_1 norm penalty *separation/regularization* functions in the objective on the violation of the total order constraints. Such penalty minimization problems that feature convex piecewise linear deviation functions and ℓ_1 norm separation/regularization functions include classes of *nearly isotonic regression* and *fused lasso* problems as special cases. The algorithm devised here is therefore a unified approach for solving the generalized problem and all known special cases with a complexity that is the most efficient known to date. We present an empirical study showing that our algorithm outperforms the run times of a linear programming software, for simulated data sets.

Key words. isotonic median regression, quantile regression, fused lasso, partial order estimation, Markov random fields, minimum cut

AMS subject classifications. 90B10, 90C27, 62G08

DOI. 10.1137/15M1024081

1. Introduction.

A common problem in many applications is that noisy observations of parameters do not satisfy preset (partial) rank order requirements. The problem is to adjust the observations in order to identify estimated values, that satisfy the (partial) rank order constraints, while minimizing the total *loss* of *deviations* of the estimated values from the observation values. This problem, known as the *partial order estimation* problem, has applications in numerous fields including production [47, 28, 40, 25], signal processing [35], economics [42], bioinformatics [8, 9, 30], and statistical learning [5, 7, 6, 39, 24, 34].

The partial order estimation problem is often generalized to relax the requirement of satisfying the rank order constraints, and replacing the order constraints by penalty *separation* functions that impose costs for each violated constraint. These separation penalties are then traded off against the deviation penalties.

The problem studied here is a generalization of several well-known problems including isotonic median regression (IMR); we call this problem *generalized IMR* (GIMR). The GIMR problem is formulated as

$$\min_{x_1,\ldots,x_n} \sum_{i=1}^{n} f_i^{pl}(x_i;\{a_{i,j}\}_{j=1}^{q_i}) + \sum_{i=1}^{n-1} d_{i,i+1}(x_i - x_{i+1})_+ + \sum_{i=1}^{n-1} d_{i+1,i}(x_{i+1} - x_i)_+ \tag{1.1}$$

$$\text{(GIMR)} \quad \text{s.t. } \ell_i \le x_i \le u_i, \quad i = 1,\ldots,n,$$

*Received by the editors June 2, 2015; accepted for publication (in revised form) September 22, 2017; published electronically December 19, 2017.
http://www.siam.org/journals/siopt/27-4/M102408.html
Funding: The first author's research was supported in part by NSF award CMMI-1200592.
†Department of Industrial Engineering and Operations Research, University of California, Berkeley (hochbaum@ieor.berkeley.edu, chenglu@berkeley.edu).

where each deviation function $f_i^{pl}(x_i; \{a_{i,j}\}_{j=1}^{q_i})$ is an arbitrary convex piecewise linear function with q_i breakpoints $a_{i,1} < a_{i,2} < \cdots < a_{i,q_i}$ (the superscript "pl" stands for "piecewise linear"). The separation terms involve the nonnegative coefficients $d_{i,i+1}$ and $d_{i+1,i}$ for positive and negative separation penalties, and the notation $(x)_+$ is the positive part of x, $\max\{x, 0\}$. For values of $d_{i,i+1}$ that are sufficiently large, an optimal solution to GIMR satisfies the total rank order $x_1 \le x_2 \le \cdots \le x_n$. The set of feasible values for each x_i is contained in the interval $[\ell_i, u_i]$.

1.1. Special cases of GIMR and applications.

1.1.1. Models with deviation terms only. A commonly used special case of convex piecewise linear deviation functions, in the context of isotonic regression, is the ℓ_1 norm. An ℓ_1 deviation function is a sum of (weighted) absolute values of the differences between the estimated values and the respective observation values. There are a number of advantages for the use of the ℓ_1 deviation function: This function gives the exact maximum likelihood estimate if the noises in the observation values follow the Laplacian distribution [29, 10]; it is in general robust to heavy-tailed noises and to the presence of outliers [31, 48, 44]; it provides better preservation of the contrast and the invariance to global contrast changes [11, 44]. The ℓ_1 deviation function, in weighted or unweighted form, was used in a number of models and applications. IMR was studied in [37, 29, 10, 33], and its applications in statistics can be found in [36, 38, 39]. Additional areas in which IMR on partial order has been applied include chromosomal microarray analysis (CMA) [4] in bioinformatics and ordinal classification with monotonicity constraints [13]. We note that the algorithm of [4] is incorrect, and the complexity of the corrected version is considerably worse than the complexity of the algorithm of [22] for IMR.

The formulation of IMR with weights w_{ij} is

$$\begin{aligned} &\min_{x_1,\ldots,x_n} \sum_{i=1}^{n}\sum_{j=1}^{q_i} w_{ij}|x_i - a_{ij}| \\ &\text{(IMR)} \quad \text{s.t. } x_i \le x_{i+1}, \quad i = 1, \ldots, n-1. \end{aligned} \tag{1.2}$$

When each variable x_i is associated with only one observation a_i, i.e., $q_i = 1$, IMR is then referred to as the simple IMR (SIMR):

$$\begin{aligned} &\min_{x_1,\ldots,x_n} \sum_{i=1}^{n} w_i|x_i - a_i| \\ &\text{(SIMR)} \quad \text{s.t. } x_i \le x_{i+1}, \quad i = 1, \ldots, n-1. \end{aligned} \tag{1.3}$$

SIMR (1.3) was studied extensively, e.g., in [22, 3].

Some applications use deviation functions that are not ℓ_1 functions, but use only two pieces in the piecewise linear deviation function. One example is the *quantile deviation* function which preserves the robustness property of the ℓ_1 deviation function. The quantile deviation function of estimated variable x_i from observation a_i for parameter $\tau \in [0, 1]$ is

$$\rho_\tau(x_i; a_i) = \begin{cases} \tau(x_i - a_i) & \text{if } x_i - a_i \ge 0, \\ -(1-\tau)(x_i - a_i) & \text{if } x_i - a_i < 0. \end{cases} \tag{1.4}$$

Here the parameter τ represents the quantile of the observations of interest. For $\tau = \frac{1}{2}$ (half-quantile), the quantile deviation function is identical to the absolute value

function. Quantile deviations have been used in array-based comparative genomic hybridization (array-CGH) analysis in bioinformatics [15, 27].

Another special case of the convex piecewise linear function is the ϵ-*insensitive deviation* function, defined as follows [32]:

$$d_\epsilon(x_i; a_i) = \begin{cases} x_i - a_i - \epsilon & \text{if } x_i - a_i > \epsilon, \\ 0 & \text{if } |x_i - a_i| \le \epsilon, \\ -x_i + a_i - \epsilon & \text{if } x_i - a_i < -\epsilon. \end{cases} \tag{1.5}$$

The absolute value function is a special case of the 0-insensitive deviation function.

The use of isotonic regression for medical prognosis was discussed by Ryu, Chandrasekaran, and Jacob [41]. They considered convex piecewise linear deviation functions (each with 3 pieces) and a partial order that is derived from medical knowledge on the relative prognosis prospects for pairs of feature vectors.

1.1.2. Models that include separation/regularization terms. There are variant models of total order estimation that include penalty terms in the objective on the violation of total order constraints, instead of imposing those constraints. These penalty functions are often referred to as separation or regularization functions. We present here four such models that were presented for specific contexts.

Tibshirani, Hoefling, and Tibshirani [45], studied a "nearly isotonic" model on total order for the purpose of fitting global warming data on annual temperature anomalies. Here a_i is the observation of the temperature anomaly value at the ith year of the dataset:

$$\text{(nearly isotonic)} \quad \min_{x_1,\ldots,x_n} \ \frac{1}{2}\sum_{i=1}^{n}(x_i - a_i)^2 + \lambda \sum_{i=1}^{n-1}(x_i - x_{i+1})_+. \tag{1.6}$$

In this model (1.6), the tuning parameter $\lambda \ge 0$ measures the relative importance between the deviation terms and the separation terms. As $\lambda \to \infty$ the problem is equivalent to imposing the total order constraints of IMR (1.2) and SIMR (1.3). In model (1.6) the quadratic deviation terms are based on the Gaussian noise assumption on the observations. We note that the GIMR model differs from (1.6) in that instead of convex quadratic deviation functions it has convex piecewise linear deviation functions. This convex piecewise linear class of functions includes ℓ_1 deviation functions that are considered to be more appropriate as a model for Laplacian noises or heavy-tailed noises.

The separation term in the nearly isotonic model (1.6), $\lambda(x_i - x_{i+1})_+$, is "one-sided" in that it only penalizes the surplus of x_i over x_{i+1}. A more general separation term also penalizes the surplus of x_{i+1} over x_i, in the form of $\lambda'(x_{i+1} - x_i)_+$, leading to a "two-sided" separation penalty. If the two-sided penalty is symmetric ($\lambda = \lambda'$), it can be presented as the absolute value (ℓ_1) separation term $\lambda|x_i - x_{i+1}|$. This ℓ_1 separation penalty is known as *fused lasso* [46]. An example of the use of the fused lasso model is in array-CGH analysis [15, 27]. It is to estimate the ratio of gene copying numbers at each position in DNA sequences between tumor and normal cell samples, based on the biological knowledge that the ratios between adjacent positions in the DNA sequences are similar. Eilers and de Menezes, [15] proposed the following quantile fused lasso (Q-FL) model to identify the estimated log-ratio x_i, based on the

observed log-ratio a_i at the ith position:

$$\text{(Q-FL)} \quad \min_{x_1,\ldots,x_n} \sum_{i=1}^{n} \rho_\tau(x_i; a_i) + \lambda \sum_{i=1}^{n-1} |x_i - x_{i+1}|. \tag{1.7}$$

The deviation terms are the quantile deviation functions (1.4) and the ℓ_1 separation functions $|x_i - x_{i+1}|$ drive the log-ratios of adjacent positions to be similar. Later, Li and Zhu in [27] extended the above model to the following quantile weighted fused lasso (Q-wFL) model by considering the distances between adjacent positions, which is claimed to help improve the estimation [27]:

$$\text{(Q-wFL)} \quad \min_{x_1,\ldots,x_n} \sum_{i=1}^{n} \rho_\tau(x_i; a_i) + \lambda \sum_{i=1}^{n-1} \frac{1}{d_{i,i+1}} |x_i - x_{i+1}|, \tag{1.8}$$

where $d_{i,i+1} \in \mathbb{R}$ is the distance between the ith and the $(i+1)$th positions. Thus the closer the adjacent positions, the larger penalty on the log-ratio difference. The quantile deviation functions are also claimed in [15, 27] to be advantageous over the standard quadratic deviation functions (least squares mean regression).

In signal processing, Storath, Weinmann, and Unser [44], considered a fused lasso model with ℓ_1 deviation functions:

$$(\ell_1\text{-FL}) \quad \min_{x_1,\ldots,x_n} \sum_{i=1}^{n} w_i |x_i - a_i| + \lambda \sum_{i=1}^{n-1} |x_i - x_{i+1}|, \tag{1.9}$$

where the w_i's are nonnegative weights and $\lambda > 0$ is the model tuning parameter.

Recently, Kolmogorov, Pock, and Rolinek [26] studied a weighted fused lasso problem, which generalizes Q-FL (1.7), Q-wFL (1.8), and ℓ_1-FL (1.9), by allowing deviation functions to be general convex piecewise linear functions with $O(1)$ breakpoints each, and different weights on the ℓ_1 separation terms $|x_i - x_{i+1}|$ in place of the uniform coefficient λ. This problem is denoted by PL-wFL-$O(1)$, where PL stands for piecewise linear, and $O(1)$ indicates that each convex piecewise linear deviation function has $O(1)$ breakpoints.

1.2. Best algorithms for partial order estimation. Even though the GIMR problem is defined for total order, our algorithm uses key ideas from the best algorithms for partial order.

Partial order estimation with deviation terms only. The most general setup of the partial order estimation problem has general convex deviation functions, as well as partial order rather than total order. The fastest algorithm for the general partial order estimation problem was by Hochbaum and Queyranne [22] with complexity that is the sum of a minimum s,t-cut complexity, on a respective graph with a node for each variable and an arc for each rank order constraint, and the complexity of finding the minimum of the n convex deviation functions in the given intervals. This total complexity is provably best possible since, as shown in [22], the problem is a generalization of both minimum s,t-cut and of the problem of finding the minimum of each of the convex deviation functions.

Partial order estimation with separation terms. The Markov random fields model is the most general deviation-separation model. Given a partial order repre-

sented by a graph $G = (V, A)$, the problem is

$$\min_{x_i : i \in V} \sum_{i \in V} f_i(x_i) + \sum_{(i,j) \in A} g_{i,j}(x_i - x_j) \quad \text{s.t. } x_i \in X_i \quad \forall i \in V, \tag{1.10}$$

where each $f_i(x_i)$ is a deviation function, and each $g_{i,j}(x_i - x_j)$ is a separation function. X_i indicates the feasible set for each variable, typically integer or continuous values within an interval $[\ell_i, u_i]$. Best known algorithms for this problem under convexity assumptions of the separation functions were given in [18, 1, 2]. We are interested here in the Markov random fields where the separation terms are convex two-piecewise-linear functions each with a breakpoint at 0, which we refer to here as MRF. This MRF problem is a generalization of GIMR (1.1) and is formulated as

$$\min_{x_i : i \in V} \sum_{i \in V} f_i(x_i) + \sum_{(i,j) \in A} d_{i,j} \cdot (x_i - x_j)_+ \quad \text{(MRF)} \quad \text{s.t. } x_i \in X_i \quad \forall i \in V. \tag{1.11}$$

Here $f_i(x_i)$ is a general (not necessarily differentiable) convex deviation function, $d_{i,j}$ is a given nonnegative coefficient, and the associated graph is an arbitrary directed graph $G = (V, A)$.

Hochbaum in [18] gave an efficient algorithm to solve the MRF (1.11) in time $O(T(n, m) + n \log \frac{U}{\epsilon})$ (where $n = |V|, m = |A|$, $U = \max_i |X_i|$, and ϵ a parameter to be explained next). The first term $T(n, m)$ is the complexity of solving maximum flow (or minimum cut) on an associated graph of $G = (V, A)$. The second term $O(n \log \frac{U}{\epsilon})$ is the complexity of finding the minima of n convex functions with ϵ-accuracy. The value of $\log \frac{1}{\epsilon}$ indicates the number of significant digits in the solution. The ϵ-accuracy complexity model is the only way to solve problems that involve nonlinear and nonquadratic functions on digital computers. This issue is discussed in detail in [23, 22, 18] and in a review in [19]. If each X_i is an integer set, then $\epsilon = 1$ and the complexity becomes $O(T(n, m) + n \log U)$. It was shown in [18] that this algorithm is the fastest possible since the MRF (1.11) generalizes both the graph minimum cut problem and the problem of finding the minima of n convex functions over each feasible set X_i.

Note that for the convex piecewise linear functions discussed here, the continuous MRF algorithm is irrelevant since it is known a priori that the optimal solution can only take values that are the given breakpoints of the convex piecewise linear functions; see Lemma 4.1 in section 4. Thus the accuracy is determined by the input data.

1.3. Best algorithms for total order estimation.

Total order estimation with deviation terms only. For the special case that the partial order is a total order, the respective graph is a directed *path*. For the directed path Hochbaum and Queyranne's algorithm was shown to have complexity $O(n(\log n + \log(U/\epsilon)))$ [22], where U is the largest width of the interval in which any variable can take values, and ϵ is the solution accuracy, which is 1 for integer solutions. (It is important to note that the complexity term of $\log(U/\epsilon)$ is provably unavoidable when minimizing a nonlinear and nonquadratic convex function, as discussed in [22] and is based on the impossibility of strongly polynomial algorithms for nonlinear nonquadratic optimization proved in [17]). For quadratic deviation functions, the run time of Hochbaum and Queyranne's algorithm is $O(n \log n)$. Ahuja and Orlin in [3] derived a specialized algorithm for the total order case with complexity $O(n \log(U/\epsilon))$.

Since U is generally assumed to be $\Omega(n)$, this complexity matches that of Hochbaum and Queyranne's algorithm, but Hochbaum and Queyranne's algorithm is applicable to the more general problem of partial order estimation.

Let $q = \sum_{i=1}^{n} q_i$ be the total number of breakpoints of the n convex piecewise linear deviation functions, $\{f_i^{pl}(x_i)\}_{i=1,\ldots,n}$, in GIMR. For the GIMR problem with deviation terms only (or very large d parameters), if the q breakpoints of the convex piecewise linear deviation functions are sorted, the complexity of Hochbaum and Queyranne's algorithm is $O(n(\log n + \log q)) = O(n \log q)$ $(q = \Omega(n))$, and otherwise the sorting complexity of $O(q \log n)$ is added.

Other known algorithms for IMR (1.2) include an algorithm in [10] of complexity $O(qn)$ and an algorithm of complexity $O(q \log^2 q)$ by [33]. GIMR-Algorithm shown here solves IMR (1.2) in time $O(q \log n)$, which improves the complexity of [10] and [33], and matches the complexity of Hochbaum and Queyranne's algorithm in [22]. For SIMR (1.3), the fastest algorithms to date, by [22, 3], have complexity $O(n \log n)$. GIMR-Algorithm solves SIMR (1.3) in the same complexity.

Total order estimation with separation terms. GIMR is a special case of MRF (1.11) where the graph G is a bidirectional path with node set $V = \{1, \ldots, n\}$, arc set $A = \{(i, i+1), (i+1, i)\}_{i=1,\ldots,n-1}$, and the deviation functions are convex piecewise linear. Therefore, any algorithm that solves MRF can solve GIMR. As discussed in section 1.2, Hochbaum's algorithm for MRF uses a parametric minimum cut procedure that runs in the complexity of solving a single minimum s,t-cut. Procedures known to date that can be used as parametric minimum cut and have this property (that they solve the parametric problem in the complexity of a single cut) are Hochbaum's pseudoflow (HPF) algorithm [20] and the push-relabel algorithm [16]. Both HPF algorithm and the push-relabel algorithm solve the parametric minimum cut in complexity $T(n, m) = O(nm \log \frac{n^2}{m})$ [16, 21]. A direct application of this MRF algorithm to GIMR (1.1) (total order graph) is of complexity $O(n^2 \log n + n \log q)$ since $m = O(n)$ and the deviation functions are convex piecewise linear, so finding n times the minima of such functions requires at most a binary search on the set of breakpoints. Our main contribution here can be viewed as an algorithm that speeds up Hochbaum's algorithm for GIMR by efficiently generating the respective minimum s,t-cuts for a path.

Other algorithms were devised for various special cases of total order estimation with separation terms: Eilers and de Menezes in [15], and Li and Zhu in [27], derived algorithms to solve Q-FL (1.7) and Q-wFL (1.8), respectively, both based on linear programming. They did not state concrete complexity results. For the ℓ_1-FL problem (1.9), Dümbgen and Kovac in [14] gave the best algorithm to-date with $O(n \log n)$ complexity. Recently Storath, Weinmann, and Unser in [44] proposed an algorithm for the ℓ_1-FL problem of complexity $O(n^2)$. GIMR-Algorithm solves all these problems in $O(n \log n)$ complexity.

For problem PL-wFL-$O(1)$, Kolmogorov, Pock, and Rolinek [26] derived an algorithm of complexity $O(n \log n)$, which, like our algorithm, uses a method for efficiently generating the respective minimum s,t-cuts for Hochbaum's algorithm and achieves the same complexity as ours (using a different methodology). Since PL-wFL-$O(1)$ generalizes Q-FL (1.7), Q-wFL (1.8), and ℓ_1-FL (1.9), this is also an alternative fastest algorithm for these problems. Note that Kolmogorov, Pock, and Rolinek in [26] also claimed an $O(n \log \log n)$ algorithm for the PL-wFL-$O(1)$ problem. However, the divide-and-conquer technique used in the algorithm requires the sorting of the breakpoints, which adds to the complexity $O(n \log n)$.

TABLE 1
Summary of comparison of complexities. Here LP stands for the complexity of solving a linear programming problem of size $O(n)$.

Problem	Deviation	Separation	Algorithm here	Algorithms to-date
GIMR	Convex piecewise linear	$d_{i,i+1}(x_i - x_{i+1})_+ + d_{i+1,i}(x_{i+1} - x_i)_+$	$O(q \log n)$	$O(n^2 \log n + n \log q)$ [18]
IMR	$\sum_{j=1}^{q_i} \lvert x_i - a_{ij} \rvert$		$O(q \log n)$	$O(n \log q + q \log n)$ [22]
SIMR	$\lvert x_i - a_i \rvert$		$O(n \log n)$	$O(n \log n)$ [22, 3]
Nearly isotonic	Convex piecewise linear	$\lambda(x_i - x_{i+1})_+$	$O(q \log n)$	$O(n^2 \log n + n \log q)$ [18]
Q-FL	$\rho_\tau(x_i; a_i)$	$\lambda \lvert x_i - x_{i+1} \rvert$	$O(n \log n)$	LP [15]
Q-wFL	$\rho_\tau(x_i; a_i)$	$\lambda \frac{1}{d_{i,i+1}} \lvert x_i - x_{i+1} \rvert$	$O(n \log n)$	LP [27]
ℓ_1-FL	$w_i \lvert x_i - a_i \rvert$	$\lambda \lvert x_i - x_{i+1} \rvert$	$O(n \log n)$	$O(n \log n)$ [14] $O(n^2)$ [44]
PL-wFL-$O(1)$	Convex $O(1)$-piecewise linear	$d_{i,i+1} \lvert x_i - x_{i+1} \rvert$	$O(n \log n)$	$O(n \log n)$ [26]

1.3.1. Summary of results. In Table 1 we provide a comparison of the complexity of our algorithm for GIMR and its special cases as compared to the best and recent algorithms' complexities known to date. As can be seen, our GIMR-Algorithm's complexity either matches the complexity of the best algorithms to date, or improves on them. And furthermore, GIMR-Algorithm is one unified algorithm for all these special cases, whereas, up till now specialized algorithms were devised for each category of the special cases.

We assess the empirical performance of GIMR-Algorithm by comparing our software implementation with Gurobi, a commercial linear programming solver, on simulated data sets of various sizes. The experimental results demonstrate that GIMR-Algorithm runs faster than Gurobi on the collection of simulated data sets, by approximately a factor of 10.

1.4. Overview. The paper is organized as follows: Notation and preliminaries are introduced in section 2. Then we provide a brief review of Hochbaum's algorithm [18] for the MRF problem (1.11) in section 3. Then we give an overview of GIMR-Algorithm in section 4, including additional notation that will help to present the algorithm and its analysis. Then we present GIMR-Algorithm in details in two steps: First, in section 5 we present GIMR-Algorithm for GIMR with ℓ_1 deviation functions, namely, ℓ_1-GIMR-Algorithm; then, in section 6, ℓ_1-GIMR-Algorithm is generalized to solving GIMR with arbitrary convex piecewise linear deviation functions, namely, GIMR-Algorithm. Experimental study that assess the performance of GIMR-algorithm is discussed in section 7. Concluding remarks are provided in section 8. The pseudocodes for the various subroutines used in ℓ_1-GIMR-Algorithm and GIMR-Algorithm are given in Appendices A and B.

2. Notation and preliminaries. The partial order estimation problem is represented on a directed graph $G = (V, A)$ with $n = |V|$ and $m = |A|$. Each decision variable x_i $(i = 1, \ldots, n)$ corresponds to node $i \in V$ and the (partial) ranking order constraints correspond to the arc set A where each (partial) ranking order constraint of the form $x_i \leq x_j$ is represented by an arc $(i, j) \in A$.

For the special case that the partial order is a total order, the respective graph is a directed path. A directed path of length n is an ordered list of nodes $(v_1, \ldots, v_n)$ so that $(v_i, v_{i+1}) \in A$ for all $i = 1, \ldots, n-1$. GIMR (1.1) is represented as a *bi-directional path* (bi-path) which is a union of two directed paths: $(1, 2, \ldots, n)$ and $(n, n-1, \ldots, 1)$. We call a graph $G = (V, A)$ a *bi-path graph* if for $V = \{1, \ldots, n\}$ the arc set is $A = \{(i, i+1), (i+1, i)\}_{i=1,\ldots,n-1}$.

Let the directed s,t*-graph* $G^{st} = (V_{st}, A_{st})$ be associated with graph $G = (V, A)$ such that $V_{st} = V \cup \{s, t\}$ and $A_{st} = A \cup A_s \cup A_t$. The appended node s is called the *source node* and t is called the *sink node*. The respective sets of *source adjacent arcs* and *sink adjacent arcs* are denoted as $A_s = \{(s, i) : i \in V\}$ and $A_t = \{(i, t) : i \in V\}$. Each arc $(i, j) \in A_{st}$ has an associated nonnegative capacity $c_{i,j}$.

For two subsets of nodes $V_1, V_2 \subseteq V_{st}$, we let the set of arcs of A_{st} directed from nodes of V_1 to nodes of V_2 be denoted by (V_1, V_2). The sum of the capacities of the arcs in (V_1, V_2) is denoted by $C(V_1, V_2) = \sum_{(i,j)\in(V_1,V_2)} c_{i,j}$.

An s,t*-cut* is a partition of V_{st}, $(\{s\} \cup S, T \cup \{t\})$, where $T = V \setminus S$. For simplicity, we refer to an s,t-cut partition as (S, T). We refer to S as the *source set* of the cut, excluding s. For each node $i \in V$, we define its *status* in graph G^{st} as $status(i) = s$ if $i \in S$ (referred as an s*-node*), otherwise $status(i) = t$ $(i \in T)$ (referred as a t*-node*).

The *capacity* of a cut (S, T) is defined as $C(\{s\} \cup S, T \cup \{t\})$. A *minimum cut* in s,t-graph G^{st} is an s,t-cut (S, T) that minimizes $C(\{s\} \cup S, T \cup \{t\})$. Hereafter, any reference to a minimum cut is to the unique minimum s,t-cut with the *maximal source set*. That means if there are multiple minimum cuts, then the one selected has a source set that is not contained in any other source set of a minimum cut.

A convex piecewise linear function $f_i^{pl}(x_i)$ is specified by its ascending list of q_i breakpoints, $a_{i,1} < a_{i,2} < \cdots < a_{i,q_i}$, and the slopes of the $q_i + 1$ linear pieces between every two consecutive breakpoints, denoted by $w_{i,0} < w_{i,1} < \cdots < w_{i,q_i}$. We assume that the n sets of breakpoints are disjoint and that the total number of breakpoints in the union is $q = \sum_{i=1}^{n} q_i$. Note that we make the disjoint breakpoint assumption only for convenience in presenting the algorithm. Our algorithm works in the same way even when a breakpoint is shared by multiple functions; for details see Remark 6.2 in section 6.

Let the sorted list of the union of q breakpoints of all the n convex piecewise linear functions be $a_{i_1,j_1} < a_{i_2,j_2} < \cdots < a_{i_q,j_q}$, where a_{i_k,j_k}, the kth breakpoint in the sorted list, is the breakpoint between the $(j_k - 1)$th and the j_kth linear pieces of function $f_{i_k}^{pl}(x_{i_k})$.

For $f_i^{pl}(x_i) = w_i|x_i - a_i|$, each function has one breakpoint a_i and two pieces of slopes $-w_i$ and w_i. Thus for this case the sorted list of all the n breakpoints is $a_{i_1} < a_{i_2} < \cdots < a_{i_n}$, where a_{i_k} is the single breakpoint of function $f_{i_k}^{pl}(x_{i_k})$.

3. Review of Hochbaum's algorithm for MRF. Recall that for MRF (1.11), the partial order is represented by a directed graph $G = (V, A)$. Hochbaum's algorithm constructs a *parametric graph* $G^{st}(\alpha) = (V_{st}, A_{st})$ associated with $G = (V, A)$, for any scalar value α in the union of the ranges of the decision variables, $\bigcup_i X_i$. The capacity of arc $(i, j) \in A$ is $c_{i,j} = d_{i,j}$. Each arc in $A_s = \{(s, i)\}_{i\in V}$ has capacity $c_{s,i} = \max\{0, -f_i'(\alpha)\}$ and each arc in $A_t = \{(i, t)\}_{i\in V}$ has capacity $c_{i,t} = \max\{0, f_i'(\alpha)\}$, where $f_i'(\alpha)$ is the right subgradient of function $f_i(\cdot)$ at argument α. (One can select instead the left subgradient.) Note that for any given value of α, either $c_{s,i} = 0$ or $c_{i,t} = 0$. Hochbaum's algorithm finds the minimum cuts in the parametric graph $G^{st}(\alpha)$, for all values of α, in the complexity of a single minimum s,t-cut. The key idea

of Hochbaum's algorithm is the *threshold theorem* which links the optimal solution of MRF (1.11) with the minimum cut partitions in $G^{st}(\alpha)$.

THEOREM 3.1 (threshold theorem, Hochbaum [18]). *For any given α, let S^* be the maximal source set of the minimum cut in graph $G^{st}(\alpha)$. Then there is an optimal solution $\boldsymbol{x}^*$ to MRF (1.11) satisfying $x_i^* \geq \alpha$ if $i \in S^*$ and $x_i^* < \alpha$ if $i \in T^*$.*

An important property of $G^{st}(\alpha)$ is that the capacities of source adjacent arcs are nonincreasing functions of α, the capacities of sink adjacent arcs are nondecreasing functions of α, and the capacities of all the other arcs are constants. This implies the following *nested cut property*.

LEMMA 3.2 (nested cut property [16, 18, 20]). *For any two parameter values $\alpha_1 \leq \alpha_2$, let $S^*_{\alpha_1}$ and $S^*_{\alpha_2}$ be the respective maximal source set of the minimum cuts of $G^{st}(\alpha_1)$ and $G^{st}(\alpha_2)$, then $S^*_{\alpha_1} \supseteq S^*_{\alpha_2}$.*

The threshold theorem is used to find an optimal solution to MRF (1.11): For each variable x_i, the largest value of α in X_i for which the corresponding node is still in the maximal source set is the optimal value of x_i. This can either be done with binary search, or as in Hochbaum's algorithm, all cut partitions for all values of α are identified with the use of a parametric cut procedure.

4. Overview of GIMR-Algorithm. The key idea used in our GIMR-Algorithm is to adapt the cut-derived threshold theorem of Hochbaum's MRF algorithm. But, instead of using a parametric cut procedure as in the MRF algorithm, our algorithm uses certain properties of the cut function, for the special case of a bi-path graph and convex piecewise linear functions, which lead to a more efficient procedure for computing all relevant cuts. This adaptation runs an order of magnitude faster than the direct application of Hochbaum's algorithm to GIMR, as explained in section 1.3.

The parametric graph $G^{st}(\alpha)$ associated with a bi-path graph G has the capacities of arcs $(i, i+1), (i+1, i) \in A$ as $c_{i,i+1} = d_{i,i+1}$ and $c_{i+1,i} = d_{i+1,i}$, respectively, and the capacities of the source and sink adjacent arcs as $c_{s,i} = \max\{0, -(f_i^{pl})'(\alpha)\}$ and $c_{i,t} = \max\{0, (f_i^{pl})'(\alpha)\}$, respectively. Based on the threshold theorem, Theorem 3.1, it is sufficient to solve the minimum cuts in the parametric graph $G^{st}(\alpha)$ for all values of α, in order to solve the GIMR (1.1). We next show that the values of α to be considered can be restricted to the set of breakpoints of the n convex piecewise linear functions, $\{f_i^{pl}(x_i)\}_{i=1,\ldots,n}$. This is proved in the following lemma.

LEMMA 4.1. *The minimum cuts in $G^{st}(\alpha)$ remain unchanged for α assuming any value between any two consecutive breakpoints in the sorted list of breakpoints of all the n convex piecewise linear functions, $\{f_i^{pl}(x_i)\}_{i=1,\ldots,n}$.*

Proof. Recall that only the capacities of the source and sink adjacent arcs depend on the values of α. Since each $f_i^{pl}(x_i)$ is convex piecewise linear in GIMR, the source and sink adjacent arc capacities remain constant for α between any two consecutive breakpoint values in the sorted list of breakpoints over all the n convex piecewise linear functions. Therefore the minimum cuts in $G^{st}(\alpha)$ remain unchanged as capacities of all the arcs in the parametric graph are unchanged. ∎

GIMR-Algorithm efficiently computes the minimum cuts of $G^{st}(\alpha)$ for subsequent values of α in the ascending list of breakpoints of all the n convex piecewise linear functions, $\{f_i^{pl}(x_i)\}_{i=1,\ldots,n}$.

Remark 4.2. For graphs such as bi-path graphs, a simple, linear complexity, dynamic programming algorithm solves the minimum cut in $G^{st}(\alpha)$ for a fixed value

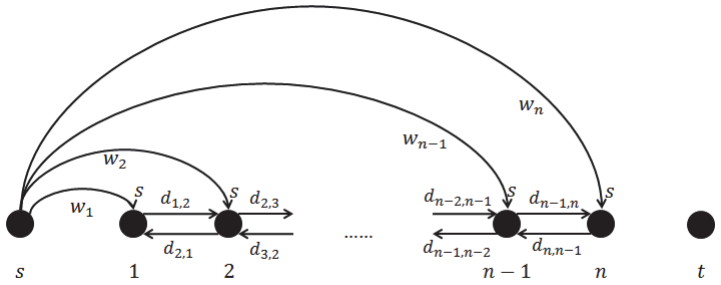

FIG. 1. *The structure of graph G_0. Arcs of capacity 0 are not displayed. Nodes 1 to n are labeled s on top as they are all in the maximal source set S_0 of the minimum cut in G_0.*

of α. A naive application of this dynamic programming algorithm setting α equal to each value of the q breakpoints, would render the running time $O(qn)$ for solving GIMR. Our GIMR-Algorithm solves GIMR in $O(q \log n)$ complexity which is significantly faster.

4.1. Additional notation. We introduce additional notation to facilitate the presentation of GIMR-Algorithm.

Let *interval* $[i, j]$ in G for $i \leq j$ be the subset of V, $\{i, i+1, \ldots, j-1, j\}$. If $i = j$, the interval $[i, i]$ is the singleton i. The notation $[i, j)$ and $(i, j]$ indicate the intervals $[i, j-1]$ and $[i+1, j]$, respectively. Let $[i, j] = \emptyset$ if $i > j$.

For an (S, T) cut, an *s-interval* is the maximal interval containing only s-nodes.

DEFINITION 4.3. *An interval $[i_\ell, i_r]$ of s-nodes in $G^{st}(\alpha)$ is said to be an* s-interval *if it is not strictly contained in another interval of only s-nodes.*

Node $i_\ell(i_r)$ is said to be the left (right) endpoint of the s-interval $[i_\ell, i_r]$. The definition of s-interval implies that for an s-interval $[i_\ell, i_r]$, if $i_\ell > 1$, then $i_\ell - 1$ is a t-node; if $i_r < n$, then $i_r + 1$ is a t-node.

Let G_k for $k \geq 1$ be the parametric graph $G^{st}(\alpha)$ for α equal to a_{i_k,j_k}, i.e., $G_k = G^{st}(a_{i_k,j_k})$ ($G_k = G^{st}(a_{i_k})$ for the ℓ_1 special case). For ease of presentation, we introduce $G_0 = G^{st}(a_{i_1,j_1} - \epsilon)$ (or $G_0 = G^{st}(a_{i_1} - \epsilon)$ in the ℓ_1 special case) for a small value of $\epsilon > 0$ (all values of $\epsilon > 0$ generate the same graph G_0). Let (S_k, T_k) be the minimum cut in G_k for $k \geq 0$. Recall that S_k is the maximal source set.

5. ℓ_1-GIMR-Algorithm.

This section describes an $O(n \log n)$ algorithm for the GIMR problem with ℓ_1 deviation functions, $f_i^{pl}(x_i) = w_i|x_i - a_i|$ with nonnegative coefficients w_i. This problem is referred to as ℓ_1-GIMR:

(5.1)

$$(\ell_1\text{-GIMR}) \quad \min_{x_1,\ldots,x_n} \sum_{i=1}^{n} w_i|x_i - a_i| + \sum_{i=1}^{n-1} d_{i,i+1}(x_i - x_{i+1})_+ + \sum_{i=1}^{n-1} d_{i+1,i}(x_{i+1} - x_i)_+.$$

The algorithm generates the respective minimum cuts of graphs G_k in increasing order of k. Based on the threshold theorem, Theorem 3.1, and the nested cut property, Lemma 3.2, we know that for each node $j = 1, \ldots, n$, $x_j^* = a_{i_k}$ for the index k such that $j \in S_{k-1}$ and $j \in T_k$.

In G_0, illustrated in Figure 1, $c_{s,i} = w_i$ and $c_{i,t} = 0$ for all $i = 1, \ldots, n$. The minimum cut in G_0 is $(\{s\} \cup V, \{t\})$.

For $k \geq 1$ graph G_k is obtained from graph G_{k-1} as follows: Capacity c_{s,i_k} is modified from w_{i_k} to 0 and capacity $c_{i_k,t}$ is modified from 0 to w_{i_k}. Other arcs' capacities remain unchanged. An illustration of G_k for $k \geq 1$ is provided in Figure 2.

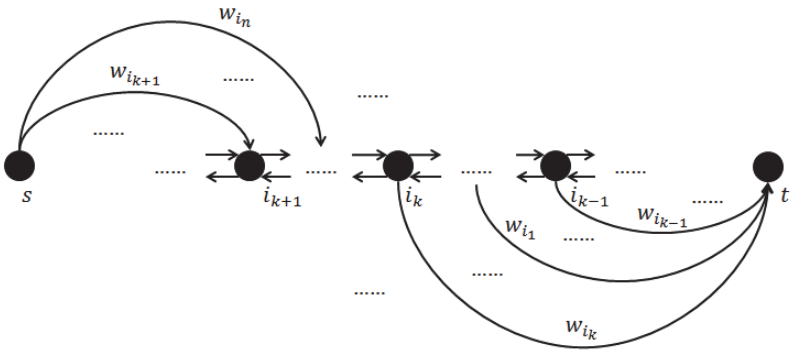

FIG. 2. *The structure of graph G_k. Arcs of capacity 0 are not displayed. Here node $i_k - 1$ appears to the right of i_k and $i_k + 1$ appears to the left, to illustrate that the order of the nodes in the graph, $(1, 2, \ldots, n)$, does not necessarily correspond to the order of the nodes according to the subscripts of the sorted breakpoints, $(i_1, i_2, \ldots, i_n)$.*

The key idea of the algorithm is to use a property, proved later, that the minimum cut in G_k is derived by updating the minimum cut in G_{k-1} on an interval of nodes that change their status from s to t. It is then shown that this interval of s-nodes can be found in amortized time $O(\log n)$. With this result, the total running time to solve ℓ_1-GIMR (5.1) is $O(n \log n)$. The remainder of this section is a proof of this main result, stated as Theorem 5.1.

THEOREM 5.1. *Given the minimum cut (S_{k-1}, T_{k-1}) in G_{k-1}, there is an amortized $O(\log n)$ algorithm for computing the minimum cut (S_k, T_k) in G_k.*

Note that the update of the graph from G_{k-1} to G_k involves only a change in the capacities of the source and sink adjacent arcs of i_k. The algorithm proceeds from G_0 to G_n by inspecting in order the nodes $i_1, i_2, \ldots, i_n$, the order of which is determined by the sorted list of breakpoints $a_{i_1} < a_{i_2} < \cdots < a_{i_n}$. Next we evaluate certain properties of node i_k.

For any node i, if $i \in T_{k-1}$, the nested cut property, Lemma 3.2, implies that i remains in the sink set for all subsequent cuts (i.e., $status(i) = t$ remains unchanged), and in particular $i \in T_k$. Hence an update of the minimum cut in G_k from the minimum cut in G_{k-1} can only involve shifting some nodes from source set S_{k-1} to sink set T_k (i.e., changing some nodes from s-nodes in G_{k-1} to t-nodes in G_k).

We first demonstrate that if node $i_k \in T_{k-1}$, then (S_{k-1}, T_{k-1}), the minimum cut in G_{k-1}, is also the minimum cut in G_k. This is proved in Lemma 5.2.

LEMMA 5.2. *If $i_k \in T_{k-1}$ then $(S_k, T_k) = (S_{k-1}, T_{k-1})$.*

Proof. Since $i_k \in T_{k-1}$, by the nested cut property, Lemma 3.2, $i_k \in T_k$.

The minimum cut in G_{k-1} satisfies

$$\begin{aligned} & C(\{s\} \cup S_{k-1}, T_{k-1} \cup \{t\}) \\ & \quad = \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\ & \quad = \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\}, T \cap ([1, i_k) \cup (i_k, n])) + C(S \cap ([1, i_k) \cup (i_k, n]), \{t\}) \\ & \qquad\qquad + C(S \cap [1, n], T \cap [1, n]) \Big\} + w_{i_k}, \end{aligned}$$

and the minimum cut in G_k satisfies

$$\begin{aligned}
&C(\{s\} \cup S_k, T_k \cup \{t\}) \\
&\quad = \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\
&\quad = \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\}, T \cap ([1, i_k) \cup (i_k, n])) + C(S \cap ([1, i_k) \cup (i_k, n]), \{t\}) \\
&\qquad\qquad + C(S \cap [1, n], T \cap [1, n]) \Big\} + 0.
\end{aligned}$$

Since the expressions inside the curly brackets are the same for both graphs, it follows that $C(\{s\} \cup S_{k-1}, T_{k-1} \cup \{t\}) - C(\{s\} \cup S_k, T_k \cup \{t\}) = w_{i_k}$, a constant. Therefore the total cut capacities in the two graphs differ by a constant. Recall that the minimum cut is unique as it is the maximal source set minimum cut. Hence the minimizer set S is the same for both G_{k-1} and G_k. ∎

We conclude that there is no update to the minimum cut in G_k from the minimum cut in G_{k-1} when $i_k \notin S_{k-1}$ (i.e., $i_k \in T_{k-1}$). As proved next, there is still no change to the minimum cut in G_k from the minimum cut in G_{k-1} when i_k is an s-node that does not change its status from G_{k-1} to G_k.

LEMMA 5.3. *If $i_k \in S_{k-1}$ and $i_k \in S_k$, then $(S_k, T_k) = (S_{k-1}, T_{k-1})$.*

Proof. The minimum cut in G_{k-1} satisfies

$$\begin{aligned}
&C(\{s\} \cup S_{k-1}, T_{k-1} \cup \{t\}) \\
&\quad = \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\
&\quad = \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\}, T \cap ([1, i_k) \cup (i_k, n])) + C(S \cap ([1, i_k) \cup (i_k, n]), \{t\}) \\
&\qquad\qquad + C(S \cap [1, n], T \cap [1, n]) \Big\} + 0.
\end{aligned}$$

And the minimum cut in G_k satisfies

$$\begin{aligned}
&C(\{s\} \cup S_k, T_k \cup \{t\}) \\
&\quad = \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\
&\quad = \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\}, T \cap ([1, i_k) \cup (i_k, n])) + C(S \cap ([1, i_k) \cup (i_k, n]), \{t\}) \\
&\qquad\qquad + C(S \cap [1, n], T \cap [1, n]) \Big\} + w_{i_k}.
\end{aligned}$$

Therefore, $C(\{s\} \cup S_{k-1}, T_{k-1} \cup \{t\}) - C(\{s\} \cup S_k, T_k \cup \{t\}) = -w_{i_k}$, and hence the minimizer set S is the same for both G_{k-1} and G_k. ∎

These two lemmas imply that the only case that will involve an update to the minimum cut in G_k is when $i_k \in S_{k-1}$ yet $i_k \in T_k$, i.e., when node i_k changes its status from an s-node in G_{k-1} to a t-node in G_k. It is shown next that in this case, if there is any node $j < i_k$ (on the left of i_k) that does not change its status from G_{k-1} to G_k (i.e., either j is an s-node in both G_{k-1} and G_k or j is a t-node in both G_{k-1} and G_k), then all nodes in the interval $[1, j]$ do not change their status from G_{k-1} to G_k; similarly, if there is any node $j' > i_k$ (on the right of i_k) that does not change its status from G_{k-1} to G_k, then all nodes in the interval $[j', n]$ do not change their status from G_{k-1} to G_k. This is proved formally in Lemma 5.4.

LEMMA 5.4. *Suppose that $i_k \in S_{k-1}$ and $i_k \in T_k$.*

(a) *If there is a node $j < i_k$ that does not change its status from G_{k-1} to G_k (i.e., either j is an s-node in both G_{k-1} and G_k or j is a t-node in both G_{k-1} and G_k), then all nodes in $[1, j]$ do not change their status from G_{k-1} to G_k.*

(b) *If there is a node $j' > i_k$ that does not change its status from G_{k-1} to G_k, then all nodes in $[j', n]$ do not change their status from G_{k-1} to G_k.*

Proof.

(a) The minimum cut in G_{k-1} satisfies

$$\begin{aligned}
&C(\{s\} \cup S_{k-1}, T_{k-1} \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\} \cup (S \cap [1,j]), (T \cap [1,j]) \cup \{t\}) \\
&\qquad + C(S \cap [j,n], T \cap [j,n]) + C(\{s\}, T \cap (j,n]) + C(S \cap (j,n], \{t\}) \Big\} \\
&= \min_{S \cap [1,j]} C(\{s\} \cup (S \cap [1,j]), (T \cap [1,j]) \cup \{t\}) \\
&\quad + \min_{S \cap [j,n]} \Big\{ C(S \cap [j,n], T \cap [j,n]) + C(\{s\}, T \cap (j,n]) + C(S \cap (j,n], \{t\}) \Big\}.
\end{aligned}$$

And the minimum cut in G_k satisfies

$$\begin{aligned}
&C(\{s\} \cup S_k, T_k \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\} \cup (S \cap [1,j]), (T \cap [1,j]) \cup \{t\}) \\
&\qquad + C(S \cap [j,n], T \cap [j,n]) + C(\{s\}, T \cap (j,n]) + C(S \cap (j,n], \{t\}) \Big\} \\
&= \min_{S \cap [1,j]} C(\{s\} \cup (S \cap [1,j]), (T \cap [1,j]) \cup \{t\}) \\
&\quad + \min_{S \cap [j,n]} \Big\{ C(S \cap [j,n], T \cap [j,n]) + C(\{s\}, T \cap (j,n]) + C(S \cap (j,n], \{t\}) \Big\}.
\end{aligned}$$

Since the arc capacities other than (s, i_k) and (i_k, t) in these two cut capacities are, respectively, the same, and the status of j does not change from G_{k-1} to G_k, the expressions $C(\{s\} \cup (S \cap [1,j]), (T \cap [1,j]) \cup \{t\})$ are of the same value for both graphs. As a result, the minimizer set $S \cap [1, j]$ is the same for both G_{k-1} and G_k.

(b) The minimum cut in G_{k-1} satisfies

$$\begin{aligned}
&C(\{s\} \cup S_{k-1}, T_{k-1} \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\}, T \cap [1,j')) + C(S \cap [1,j'), \{t\}) + C(S \cap [1,j'], T \cap [1,j']) \\
&\qquad + C(\{s\} \cup (S \cap [j',n]), (T \cap [j',n]) \cup \{t\}) \Big\} \\
&= \min_{S \cap [1,j']} \Big\{ C(\{s\}, T \cap [1,j')) + C(S \cap [1,j'), \{t\}) + C(S \cap [1,j'], T \cap [1,j']) \Big\} \\
&\quad + \min_{S \cap [j',n]} C(\{s\} \cup (S \cap [j',n]), (T \cap [j',n]) \cup \{t\}).
\end{aligned}$$

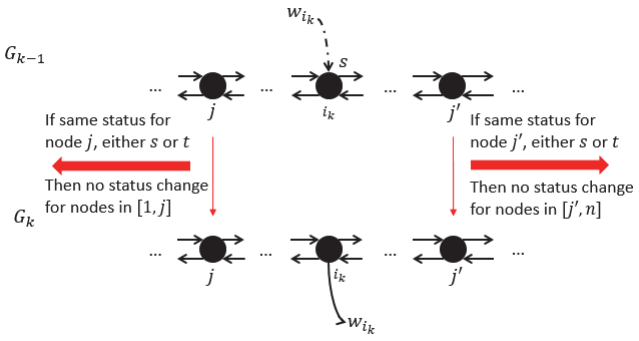

FIG. 3. *Illustration of Lemma* 5.4. $i_k \in S_{k-1}$ *(labeled* s *on top). If there is a node* $j < i_k$ *that does not change its status from* G_{k-1} *to* G_k *(i.e., either* j *is an* s*-node in both* G_{k-1} *and* G_k *or* j *is a* t*-node in both* G_{k-1} *and* G_k*), then all nodes in* $[1, j]$ *do not change their status from* G_{k-1} *to* G_k*; if there is a node* $j' > i_k$ *that does not change its status from* G_{k-1} *to* G_k*, then all nodes in* $[j', n]$ *do not change their status from* G_{k-1} *to* G_k*.*

And the minimum cut in G_k satisfies

$$
\begin{aligned}
&C(\{s\} \cup S_k, T_k \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} C(\{s\} \cup S, T \cup \{t\}) \\
&= \min_{\emptyset \subseteq S \subseteq V} \Big\{ C(\{s\}, T \cap [1, j')) + C(S \cap [1, j'), \{t\}) + C(S \cap [1, j'], T \cap [1, j']) \\
&\qquad + C(\{s\} \cup (S \cap [j', n]), (T \cap [j', n]) \cup \{t\}) \Big\} \\
&= \min_{S \cap [1, j']} \Big\{ C(\{s\}, T \cap [1, j')) + C(S \cap [1, j'), \{t\}) + C(S \cap [1, j'], T \cap [1, j']) \Big\} \\
&\quad + \min_{S \cap [j', n]} C(\{s\} \cup (S \cap [j', n]), (T \cap [j', n]) \cup \{t\}).
\end{aligned}
$$

Since the arc capacities other than (s, i_k) and (i_k, t) in these two cut capacities are, respectively, the same, and the status of j' does not change from G_{k-1} to G_k, the expressions $C(\{s\} \cup (S \cap [j', n]), (T \cap [j', n]) \cup \{t\})$ are of the same value for both graphs. As a result, the minimizer set $S \cap [j', n]$ is the same for both G_{k-1} and G_k. ∎

Lemma 5.4 is illustrated in Figure 3.

If there is a nonempty set of nodes that change their status from s in G_{k-1} to t in G_k, it must include i_k (otherwise by Lemmas 5.2 and 5.3, none of the nodes changes its status, which is a contradiction). Among all the nodes in $V = [1, n]$ that change from s in G_{k-1} to t in G_k, we denote the smallest node index as i^*_{k1} and the largest node index as i^*_{k2}, thus $i^*_{k1} \le i_k \le i^*_{k2}$. All nodes in the interval $[i^*_{k1}, i^*_{k2}]$ must change their status from s in G_{k-1} to t in G_k, because if there is a node $j \in [i^*_{k1}, i^*_{k2}] \setminus \{i_k\}$ whose status does not change, then Lemma 5.4 implies that either the status of i^*_{k1} does not change (when $j < i_k$) or the status of i^*_{k2} does not change (when $j > i_k$), which contradicts the choice of these nodes as nodes that do change their status. We conclude that if i_k changes its status, then all nodes that change their status form an interval of s-nodes containing i_k. This is stated in the following corollary.

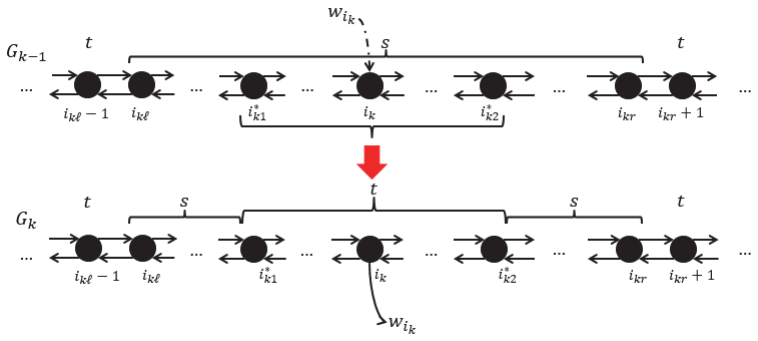

FIG. 4. *Illustration of Corollary* 5.5. $i_k \in S_{k-1}$. *Nodes are labeled on top* s *if they are* s*-nodes in* G_{k-1} *or* G_k. *Nodes are labeled on top* t *if they are* t*-nodes in* G_{k-1} *or* G_k. *All* s*-nodes in* $[i^*_{k1}, i^*_{k2}]$ *(possibly empty) in* G_{k-1}, *containing* i_k, *change to* t*-nodes in* G_k. *Note that* $[i^*_{k1}, i^*_{k2}]$ *is a subinterval of the* s*-interval w.r.t. node* i_k *in* G_{k-1}, $[i_{k\ell}, i_{kr}]$.

COROLLARY 5.5. *If* $i_k \in S_{k-1}$, *then all the nodes that change their status from* s *in* G_{k-1} *to* t *in* G_k *must form a (possibly empty) interval of* s*-nodes containing* i_k *in* G_{k-1}.

Corollary 5.5 is illustrated in Figure 4. Note that $[i^*_{k1}, i^*_{k2}]$ is a subinterval of the s-interval w.r.t. node i_k in G_{k-1}, $[i_{k\ell}, i_{kr}]$. Using Corollary 5.5 the problem of computing the minimum cut in G_k from the minimum cut in G_{k-1} is reduced to the problem of identifying the interval $[i^*_{k1}, i^*_{k2}]$.

5.1. Finding node status change interval. To identify the node status change interval $[i^*_{k1}, i^*_{k2}]$, we have the following lemma.

LEMMA 5.6. *The node status change interval* $[i^*_{k1}, i^*_{k2}]$ *is the optimal solution to the following optimization problem for* G_k*:*

$$
\begin{aligned}
\min_{[i_{k1}, i_{k2}]} \; & C(\{s\}, [i_{k1}, i_{k2}]) + C([i_{k\ell}, i_{kr}] \setminus [i_{k1}, i_{k2}], \{t\}) \\
& + C([i_{k\ell}, i_{kr}] \setminus [i_{k1}, i_{k2}], [i_{k1}, i_{k2}] \cup \{i_{k\ell} - 1, i_{kr} + 1\}) \\
\text{s.t. } & [i_{k1}, i_{k2}] = \emptyset \text{ or } i_k \in [i_{k1}, i_{k2}] \subseteq [i_{k\ell}, i_{kr}].
\end{aligned} \tag{5.2}
$$

Proof. The minimum cut (S_k, T_k) in G_k satisfies

$$
\begin{aligned}
& C(\{s\} \cup S_k, T_k \cup \{t\}) \\
& \quad = \min_{(S,T)} C(\{s\} \cup S, T \cup \{t\}) \\
& \quad = \min_{(S,T)} \Big\{ C(\{s\} \cup (S \cap [1, i_{k\ell} - 1]), (T \cap [1, i_{k\ell} - 1]) \cup \{t\}) \\
& \qquad\qquad + C(\{s\}, T \cap [i_{k\ell}, i_{kr}]) + C(S \cap [i_{k\ell}, i_{kr}], \{t\}) \\
& \qquad\qquad + C(S \cap [i_{k\ell} - 1, i_{kr} + 1], T \cap [i_{k\ell} - 1, i_{kr} + 1]) \\
& \qquad\qquad + C(\{s\} \cup (S \cap [i_{kr} + 1, n]), (T \cap [i_{kr} + 1, n]) \cup \{t\}) \Big\}.
\end{aligned} \tag{5.3}
$$

This minimization problem can be written as the sum of three minimization problems:

$$= \min_{S\cap[1,i_{k\ell}-1]} C(\{s\}\cup(S\cap[1,i_{k\ell}-1]),(T\cap[1,i_{k\ell}-1])\cup\{t\}) \tag{5.4}$$

$$+ \min_{T\cap[i_{k\ell},i_{kr}]}\Big\{C(\{s\},T\cap[i_{k\ell},i_{kr}])+C(S\cap[i_{k\ell},i_{kr}],\{t\}) + C(S\cap[i_{k\ell}-1,i_{kr}+1],T\cap[i_{k\ell}-1,i_{kr}+1])\Big\} \tag{5.5}$$

$$+ \min_{S\cap[i_{kr}+1,n]} C(\{s\}\cup(S\cap[i_{kr}+1,n]),(T\cap[i_{kr}+1,n])\cup\{t\}). \tag{5.6}$$

By Corollary 5.5, the minimizer set $S\cap[1,i_{k\ell}-1]$ in subproblem (5.4) is $S_{k-1}\cap[1,i_{k\ell}-1]$ and the minimizer set $S\cap[i_{kr}+1,n]$ in subproblem (5.6) is $S_{k-1}\cap[i_{kr}+1,n]$.

It remains to solve subproblem (5.5). Recall that since $i_{k\ell}-1$ and $i_{kr}+1$ are outside an s-interval, they are both t-nodes in G_k, therefore, $S\cap[i_{k\ell}-1,i_{kr}+1]=S\cap[i_{k\ell},i_{kr}]$. For $[i_{k1},i_{k2}]$ the interval of nodes that change their status from s to t, either $[i_{k1},i_{k2}]$ is empty, or else it must contain i_k (Corollary 5.5, $S\cap[i_{k\ell},i_{kr}]=[i_{k\ell},i_{kr}]\setminus[i_{k1},i_{k2}]$, and $T\cap[i_{k\ell},i_{kr}]=[i_{k1},i_{k2}]$). An interval $[i_{k1},i_{k2}]\subseteq[i_{k\ell},i_{kr}]$ is said to be *feasible* if it is either empty, $[i_{k1},i_{k2}]=\emptyset$, or else it contains i_k, $i_k\in[i_{k1},i_{k2}]$. The interval $[i^*_{k1},i^*_{k2}]$ is optimal if it is the feasible interval that minimizes the objective value of subproblem (5.5).

For any feasible interval $[i_{k1},i_{k2}]\subseteq[i_{k\ell},i_{kr}]$, we can rewrite the terms in subproblem (5.5) as

$$S\cap[i_{k\ell}-1,i_{kr}+1]=S\cap[i_{k\ell},i_{kr}]=[i_{k\ell},i_{kr}]\setminus[i_{k1},i_{k2}],\quad\text{and}$$
$$T\cap[i_{k\ell}-1,i_{kr}+1]=(T\cap[i_{k\ell},i_{kr}])\cup\{i_{k\ell}-1,i_{kr}+1\}=[i_{k1},i_{k2}]\cup\{i_{k\ell}-1,i_{kr}+1\}.$$

Substituting for these expressions in subproblem (5.5), it is rewritten as

$$\begin{aligned}\min_{[i_{k1},i_{k2}]}\ & C(\{s\},[i_{k1},i_{k2}])+C([i_{k\ell},i_{kr}]\setminus[i_{k1},i_{k2}],\{t\})\\ &+C([i_{k\ell},i_{kr}]\setminus[i_{k1},i_{k2}],[i_{k1},i_{k2}]\cup\{i_{k\ell}-1,i_{kr}+1\})\\ &\text{s.t. } [i_{k1},i_{k2}]=\emptyset \text{ or } i_k\in[i_{k1},i_{k2}]\subseteq[i_{k\ell},i_{kr}].\end{aligned}$$

This completes the proof. ∎

Next, we discuss how to solve the optimization problem (5.2). In the following equations, let $d_{0,1}=d_{1,0}=d_{n,n+1}=d_{n+1,n}=0$.

We evaluate the objective value of problem (5.2) for an empty interval solution, and compare it to the objective value of the optimal nonempty interval solution. The one that gives smaller objective value is the optimal solution to problem (5.2). For $[i_{k1},i_{k2}]=\emptyset$, the objective value of problem (5.2) is

$$Z(\emptyset)\triangleq C([i_{k\ell},i_{kr}],\{i_{k\ell}-1,i_{kr}+1,t\})=\sum_{i=i_{k\ell}}^{i_{kr}} c_{i,t}+d_{i_{k\ell},i_{k\ell}-1}+d_{i_{kr},i_{kr}+1}. \tag{5.7}$$

Let $[\hat{i}_{k1},\hat{i}_{k2}]$ be the optimal solution to the problem (5.2) restricted to nonempty intervals. Let the value of the objective function of problem (5.2) for $[\hat{i}_{k1},\hat{i}_{k2}]$ be $Z([\hat{i}_{k1},\hat{i}_{k2}])$. If $Z([\hat{i}_{k1},\hat{i}_{k2}])<Z(\emptyset)$ then the optimal solution is $[i^*_{k1},i^*_{k2}]=[\hat{i}_{k1},\hat{i}_{k2}]$, otherwise $[i^*_{k1},i^*_{k2}]=\emptyset$.

We next demonstrate that $\hat{i}_{k1}$ and $\hat{i}_{k2}$ can be found by solving two independent optimization problems.

LEMMA 5.7. *The optimal nonempty interval solution, $[\hat{i}_{k1}, \hat{i}_{k2}]$, can be found by solving two independent minimization problems for i_{k1} and i_{k2}, respectively.*

Proof. Problem (5.2) that restricted to nonempty intervals is equivalent to

$$\begin{aligned}
&\min_{\substack{i_{k1}:i_{k\ell}\le i_{k1}\le i_k\\ i_{k2}:i_k\le i_{k2}\le i_{kr}}} \Big\{ C(\{s\},[i_{k1},i_k]) + C([i_{k\ell},i_{k1}),\{t\}) + C([i_{k\ell},i_{k1}),[i_{k1},i_k]\cup\{i_{k\ell}-1\}) \\
&\qquad + C(\{s\},(i_k,i_{k2}]) + C((i_{k2},i_{kr}],\{t\}) + C((i_{k2},i_{kr}],[i_k,i_{k2}]\cup\{i_{kr}+1\}) \Big\} \\
&= \min_{i_{k1}:i_{k\ell}\le i_{k1}\le i_k} \Big\{ C(\{s\},[i_{k1},i_k]) + C([i_{k\ell},i_{k1}),\{t\}) + C([i_{k\ell},i_{k1}),[i_{k1},i_k]\cup\{i_{k\ell}-1\}) \Big\} \\
&+ \min_{i_{k2}:i_k\le i_{k2}\le i_{kr}} \Big\{ C(\{s\},(i_k,i_{k2}]) + C((i_{k2},i_{kr}],\{t\}) + C((i_{k2},i_{kr}],[i_k,i_{k2}]\cup\{i_{kr}+1\}) \Big\} \\
&\triangleq \min_{i_{k1}:i_{k\ell}\le i_{k1}\le i_k} \big\{ f_1(i_{k1}) \big\} + \min_{i_{k2}:i_k\le i_{k2}\le i_{kr}} \big\{ f_2(i_{k2}) \big\}.
\end{aligned}$$

Hence $\hat{i}_{k1}$ is found by solving the optimization $\min_{i_{k1}:i_{k\ell}\le i_{k1}\le i_k}\{f_1(i_{k1})\}$ and $\hat{i}_{k2}$ is found by solving the optimization problem $\min_{i_{k2}:i_k\le i_{k2}\le i_{kr}}\{f_2(i_{k2})\}$. ∎

We first show how to solve problem $\min_{i_{k1}:i_{k\ell}\le i_{k1}\le i_k}\{f_1(i_{k1})\}$. We evaluate the objective value $f_1(i_{k1})$ for $i_{k1} = i_{k\ell}$, and compare it to the optimal objective value for $i_{k1} \in [i_{k\ell}+1, i_k]$. The one that gives smaller objective value is the optimal solution $\hat{i}_{k1}$. For $i_{k1} = i_{k\ell}$, the objective value $f_1(i_{k\ell})$ is

$$f_1(i_{k\ell}) = \sum_{i=i_{k\ell}}^{i_k} c_{s,i}. \tag{5.8}$$

For $i_{k1} \in [i_{k\ell}+1, i_k]$, we have the following equation to express the objective value $f_1(i_{k1})$:

$$f_1(i_{k1}) = \sum_{i=i_{k\ell}}^{i_{k1}-1} c_{i,t} + \sum_{i=i_{k1}}^{i_k} c_{s,i} + d_{i_{k\ell},i_{k\ell}-1} + d_{i_{k1}-1,i_{k1}}. \tag{5.9}$$

Let $\tilde{i}_{k1}$ be the minimizer of $f_1(i_{k1})$ for $i_{k1} \in [i_{k\ell}+1, i_k]$. If there are multiple minima, $\tilde{i}_{k1}$ takes the largest index, due to the maximal source set requirement. To summarize,

$$\hat{i}_{k1} = \begin{cases} \tilde{i}_{k1} & \text{if } i_{k\ell}+1 \le i_k \text{ and } f_1(\tilde{i}_{k1}) \le f_1(i_{k\ell}), \\ i_{k\ell} & \text{otherwise.} \end{cases}$$

We next show how to solve the problem $\min_{i_{k2}:i_k\le i_{k2}\le i_{kr}}\{f_2(i_{k2})\}$. We evaluate the objective value $f_2(i_{k2})$ for $i_{k2} = i_{kr}$, and compare it to the optimal objective value for $i_{k2} \in [i_k, i_{kr}-1]$. The one that gives a smaller objective value is the optimal solution $\hat{i}_{k2}$. For $i_{k2} = i_{kr}$, the objective value $f_2(i_{kr})$ is

$$f_2(i_{kr}) = \sum_{i=i_k+1}^{i_{kr}} c_{s,i}. \tag{5.10}$$

For $i_{k2} \in [i_k, i_{kr}-1]$, we have the following equation to express the objective value $f_2(i_{k2})$:

$$f_2(i_{k2}) = \sum_{i=i_k+1}^{i_{k2}} c_{s,i} + \sum_{i=i_{k2}+1}^{i_{kr}} c_{i,t} + d_{i_{kr},i_{kr}+1} + d_{i_{k2}+1,i_{k2}}. \tag{5.11}$$

Let $\tilde{i}_{k2}$ be the minimizer of $f_2(i_{k2})$ for $i_{k2} \in [i_k, i_{kr} - 1]$. If there are multiple minima, $\tilde{i}_{k2}$ takes the smallest index, due to the maximal source set requirement. To summarize,

$$\hat{i}_{k2} = \begin{cases} \tilde{i}_{k2} & \text{if } i_k \le i_{kr} - 1 \text{ and } f_2(\tilde{i}_{k2}) \le f_2(i_{kr}), \\ i_{kr} & \text{otherwise.} \end{cases}$$

Finally the value of the objective function of problem (5.2) for $[\hat{i}_{k1}, \hat{i}_{k2}]$ is

$$Z([\hat{i}_{k1}, \hat{i}_{k2}]) = f_1(\hat{i}_{k1}) + f_2(\hat{i}_{k2}).$$

5.2. Data structure to find node status change interval efficiently. We observe that (5.7) to (5.11) share two operations, one is the sum of capacities of source adjacent arcs of nodes in an interval $[i, j]$, $\sum_{i'=i}^{j} c_{s,i'}$, and the other is the sum of capacities of sink adjacent arcs of nodes in an interval $[i, j]$, $\sum_{i'=i}^{j} c_{i',t}$. It will be convenient to rewrite these two sums of capacities as

$$\sum_{i'=i}^{j} c_{s,i'} = \sum_{i'=1}^{j} c_{s,i'} - \sum_{i'=1}^{i-1} c_{s,i'},$$
$$\sum_{i'=i}^{j} c_{i',t} = \sum_{i'=1}^{j} c_{i',t} - \sum_{i'=1}^{i-1} c_{i',t}.$$

To derive these sums easily, we maintain two arrays, $(sa(i))_{i=0,1,\ldots,n}$ and $(ta(i))_{i=0,1,\ldots,n}$. $sa(i)$ is the sum of capacities of source adjacent arcs of nodes in $[1, i]$ and $ta(i)$ is the sum of capacities of sink adjacent arcs of nodes in $[1, i]$. Formally,

$$sa(0) = 0; sa(i) = C(\{s\}, [1, i]) = \sum_{j=1}^{i} c_{s,j} \ (i = 1, \ldots, n); \tag{5.12}$$

$$ta(0) = 0; ta(i) = C([1, i], \{t\}) = \sum_{j=1}^{i} c_{j,t} \ (i = 1, \ldots, n). \tag{5.13}$$

Note that both arrays can also be defined recursively as

$$sa(0) = 0; sa(i) = sa(i-1) + c_{s,i} \ (i = 1, \ldots, n);$$
$$ta(0) = 0; ta(i) = ta(i-1) + c_{i,t} \ (i = 1, \ldots, n).$$

The two arrays, along with two others to be introduced, will be used throughout the algorithm.

Equations (5.7), (5.8), and (5.10) in terms of the two arrays result in

$$\begin{aligned} Z(\emptyset) &= \sum_{i=i_{k\ell}}^{i_{kr}} c_{i,t} + d_{i_{k\ell}, i_{k\ell}-1} + d_{i_{kr}, i_{kr}+1} \\ &= ta(i_{kr}) - ta(i_{k\ell} - 1) + d_{i_{k\ell}, i_{k\ell}-1} + d_{i_{kr}, i_{kr}+1}, \\ f_1(i_{k\ell}) &= \sum_{i=i_{k\ell}}^{i_k} c_{s,i} = sa(i_k) - sa(i_{k\ell} - 1), \\ f_2(i_{kr}) &= \sum_{i=i_k+1}^{i_{kr}} c_{s,i} = sa(i_{kr}) - sa(i_k). \end{aligned}$$

Equation (5.9) can be rewritten as

$$\begin{aligned} f_1(i_{k1}) &= \sum_{i=i_{k\ell}}^{i_{k1}-1} c_{i,t} + \sum_{i=i_{k1}}^{i_k} c_{s,i} + d_{i_{k\ell},i_{k\ell}-1} + d_{i_{k1}-1,i_{k1}} \\ &= \big(ta(i_{k1}-1) - ta(i_{k\ell}-1)\big) + \big(sa(i_k) - sa(i_{k1}-1)\big) + d_{i_{k\ell},i_{k\ell}-1} + d_{i_{k1}-1,i_{k1}}. \end{aligned}$$

Next we introduce a third array $(tms(i))_{i=0,1,\ldots,n}$ defined as

$$tms(i) = ta(i) - sa(i) + d_{i,i+1} \ (i = 0, 1, \ldots, n). \tag{5.14}$$

With these arrays, (5.9) can be simplified to

$$f_1(i_{k1}) = tms(i_{k1}-1) - ta(i_{k\ell}-1) + sa(i_k) + d_{i_{k\ell},i_{k\ell}-1}.$$

Recall that the above equation is the expression of $f_1(i_{k1})$ for $i_{k1} \in [i_{k\ell}+1, i_k]$, and we want to find the minimizer of $f_1(i_{k1})$ for $i_{k1} \in [i_{k\ell}+1, i_k]$. The only term in $f_1(i_{k1})$ that depends on i_{k1} is $tms(i_{k1}-1)$, thus the minimizer $\tilde{i}_{k1}$ of $f_1(i_{k1})$, is also the minimizer of $tms(i_{k1}-1)$, for $i_{k1} \in [i_{k\ell}+1, i_k]$:

$$\tilde{i}_{k1} = \operatorname{argmin}_{i_{k1}:i_{k\ell}+1 \le i_{k1} \le i_k} \big\{f_1(i_{k1})\big\} = \operatorname{argmin}_{i_{k1}:i_{k\ell}+1 \le i_{k1} \le i_k} \big\{tms(i_{k1}-1)\big\}. \tag{5.15}$$

Similarly, for $f_2(i_{k2})$ $(i_{k2} \in [i_k, i_{kr}-1])$, (5.11) can be rewritten as

$$\begin{aligned} f_2(i_{k2}) &= \sum_{i=i_k+1}^{i_{k2}} c_{s,i} + \sum_{i=i_{k2}+1}^{i_{kr}} c_{i,t} + d_{i_{kr},i_{kr}+1} + d_{i_{k2}+1,i_{k2}} \\ &= \big(sa(i_{k2}) - sa(i_k)\big) + \big(ta(i_{kr}) - ta(i_{k2})\big) + d_{i_{kr},i_{kr}+1} + d_{i_{k2}+1,i_{k2}}. \end{aligned}$$

A final, fourth, array $(smt(i))_{i=0,1,\ldots,n}$ is

$$smt(i) = sa(i) - ta(i) + d_{i+1,i} \ (i = 0, 1, \ldots, n). \tag{5.16}$$

Then (5.11) can be further simplified to

$$f_2(i_{k2}) = smt(i_{k2}) - sa(i_k) + ta(i_{kr}) + d_{i_{kr},i_{kr}+1}.$$

Recall that the above equation is the expression of $f_2(i_{k2})$ for $i_{k2} \in [i_k, i_{kr}-1]$, and we want to find the minimizer of $f_2(i_{k2})$ for $i_{k2} \in [i_k, i_{kr}-1]$. The only term in $f_2(i_{k2})$ that depends on i_{k2} is $smt(i_{k2})$, thus the minimizer $\tilde{i}_{k2}$ of $f_2(i_{k2})$, is also the minimizer of $smt(i_{k2})$ for $i_{k2} \in [i_k, i_{kr}-1]$:

$$\tilde{i}_{k2} = \operatorname{argmin}_{i_{k2}:i_k \le i_{k2} \le i_{kr}-1} \big\{f_2(i_{k2})\big\} = \operatorname{argmin}_{i_{k2}:i_k \le i_{k2} \le i_{kr}-1} \big\{smt(i_{k2})\big\}. \tag{5.17}$$

To summarize, we introduced here four arrays: $(sa(i))_{i=0,1,\ldots,n}$ in (5.12), $(ta(i))_{i=0,1,\ldots,n}$ in (5.13), $(tms(i))_{i=0,1,\ldots,n}$ in (5.14), and $(smt(i))_{i=0,1,\ldots,n}$ in (5.16).

With the four arrays, $\tilde{i}_{k1}$ is identified by finding the minimum value of a subarray of array $(tms(i))_{i=0,1,\ldots,n}$ according to (5.15), $\tilde{i}_{k2}$ is identified by finding the minimum value of a subarray of array $(smt(i))_{i=0,1,\ldots,n}$ according to (5.17). After we identify $\tilde{i}_{k1}$ and $\tilde{i}_{k2}$, we evaluate and compare $f_1(i_{k\ell})$ to $f_1(\tilde{i}_{k1})$ to identify $\hat{i}_{k1}$, and evaluate and compare $f_2(i_{kr})$ to $f_2(\tilde{i}_{k2})$ to identify $\hat{i}_{k2}$. The process also gives us the objective value of $Z([\hat{i}_{k1}, \hat{i}_{k2}])$. Finally, we evaluate $Z(\emptyset)$ and compare it to $Z([\hat{i}_{k1}, \hat{i}_{k2}])$ to identify $[i^*_{k1}, i^*_{k2}]$. Evaluating all the above objective values involves querying different specific elements of the four arrays.

In our algorithm, we implement the four arrays using a data structure introduced in Appendix B. Using the data structure, the operations of identifying the minimum value of any subarray of an array; querying a specific element of an array; updating the arrays from graph G_{k-1} to G_k, can all be done efficiently, in complexity $O(\log n)$ per operation. Therefore the node status change interval $[i^*_{k1}, i^*_{k2}]$ and the array updates can be computed efficiently.

5.3. The complete ℓ_1-GIMR-Algorithm. To summarize, the algorithm proceeds from G_{k-1} to G_k by checking the status of node i_k. If this node is an s-node then it is possible that the minimum cut in G_{k-1} is changed when the graph is updated to G_k. In that case, the algorithm identifies the node status change interval with respect to i_k, $[i^*_{k1}, i^*_{k2}]$. If this interval is not empty then the nodes in the interval change their status from s to t. This triggers a change in the set of s-intervals of G_{k-1} by decomposing one s-interval to up to two new s-intervals in G_k. Once the s-intervals have been updated, the iteration for k terminates.

We next present the pseudocode for ℓ_1-GIMR-Algorithm, followed by an explanation of the subroutines used:

ℓ_1-GIMR-Algorithm
input: $\{w_i, a_i\}_{i=1,\ldots,n}$ and $\{d_{i,i+1}, d_{i+1,i}\}_{i=1,\ldots,n-1}$ in ℓ_1-GIMR (5.1).
output: An optimal solution $\{x^*_i\}_{i=1,\ldots,n}$.
begin
1 Sort the a_i's as $a_{i_1} < a_{i_2} < \cdots < a_{i_n}$;
2 initialization();
3 **for** $k := 1, \ldots, n$:
4 {Update graph}update_arrays$(i_k, -w_{i_k}, w_{i_k})$;
5 **if** $status(i_k) = s$ **then**
6 $[i_{k\ell}, i_{kr}] :=$ get_s_interval(i_k);
7 $[i^*_{k1}, i^*_{k2}] :=$ find_status_change_interval$(i_{k\ell}, i_k, i_{kr})$;
8 **if** $[i^*_{k1}, i^*_{k2}] \neq \emptyset$ **then**
9 **for** $i \in [i^*_{k1}, i^*_{k2}]$: $x^*_i := a_{i_k}$, $status(i) := t$;
10 update_s_interval$(i_{k\ell}, i^*_{k1}, i^*_{k2}, i_{kr})$;
11 **end if**
12 **end if**
13 **end for**
14 **return** $\{x^*_i\}_{i=1,\ldots,n}$;
end

At line 2, We use initialization() to initialize all the data structures for G_0 that are needed in the algorithm, including the set of s-intervals, the four arrays, and the status of all the nodes. The data structure for the set of s-intervals is introduced in Appendix A. G_0 contains a single s-interval $[1, n]$. Appendix A.1 shows that initializing the set of s-intervals containing a single s-interval $[1, n]$ is done in $O(1)$ time using the data structure. The data structure for the four arrays is introduced in Appendix B.

Appendix B.1 shows that initializing the four arrays for G_0 is done in time $O(n \log n)$ using the data structure. We implement the status of all the nodes as a simple boolean array such that $status(i)$ is the status of node i for $i = 1, \ldots, n$. As all nodes are in the maximal source set in the minimum cut in G_0, initially $status(i) = s$ for all $i = 1, \ldots, n$, which is initialized in $O(n)$ time. Hence the complexity of initialization() is $O(n \log n)$.

At line 3, the **for** loop computes, in the kth iteration, the minimum cut in G_k from the minimum cut in G_{k-1}. At line 4 we first call subroutine update_arrays$(i_k, -w_{i_k}, w_{i_k})$ to update the values of the four arrays from G_{k-1} to G_k. Recall that graph G_k is obtained from graph G_{k-1} by changing only the capacities c_{s,i_k} and $c_{i_k,t}$. Thus the values of the four arrays are updated as follows:

$$\begin{aligned}
&\forall i \in [i_k, n]: \\
&\quad sa(i) := sa(i) - w_{i_k}, \\
&\quad ta(i) := ta(i) + w_{i_k}, \\
&\quad tms(i) := tms(i) + 2w_{i_k}, \\
&\quad smt(i) := smt(i) - 2w_{i_k}.
\end{aligned}$$

Note that the above operations are all to add the same constant to a subarray. We show in Appendix B.2 that adding the same constant to a subarray of size $O(n)$ can be done in complexity $O(\log n)$ using the data structure for the array. Hence the complexity of update_arrays is $O(\log n)$ with a pseudocode provided in Appendix B.2.

At line 5 we check whether i_k is an s-node in G_{k-1}, that is, whether $i_k \in S_{k-1}$, and if so, there is a potential change of status of nodes. If i_k is an s-node, we proceed to the **if** statement to identify the node status change interval $[i^*_{k1}, i^*_{k2}]$. We first find the s-interval $[i_{k\ell}, i_{kr}]$ w.r.t. node i_k in G_{k-1}. This is implemented in subroutine $[i_{k\ell}, i_{kr}] :=$ get_s_interval(i_k) at line 6. The data structure for the (disjoint) s-intervals maintains them sorted in increasing order of their left endpoints. This allows us to identify $[i_{k\ell}, i_{kr}]$ with the binary search in $O(\log n)$ time. The pseudocode of get_s_interval is given in Appendix A.2. With the values of i_k and $[i_{k\ell}, i_{kr}]$, the algorithm proceeds to subroutine $[i^*_{k1}, i^*_{k2}] :=$ find_status_change_interval$(i_{k\ell}, i_k, i_{kr})$ at line 7 to identify $[i^*_{k1}, i^*_{k2}]$ by solving the optimization problem (5.2) according to the procedure shown in section 5.2. Appendix B shows that using the data structure, it takes $O(\log n)$ time to identify the minimum value of any subarray of an array and $O(\log n)$ time to query a specific element of an array. Hence the complexity of find_status_change_interval is $O(\log n)$. The pseudocode for find_status_change_interval is in Appendix B.3.

If $[i^*_{k1}, i^*_{k2}]$ is nonempty, checked at line 8, we proceed to line 9 to record the optimal values of all x_i for all node i in the node status change interval $[i^*_{k1}, i^*_{k2}]$, as $x^*_i = a_{i_k}$, and update the status of node i from s in G_{k-1} to t in G_k. The s-interval $[i_{k\ell}, i_{kr}]$ in G_{k-1} is then decomposed into at most two new nonempty s-intervals in G_k, $[i_{k\ell}, i^*_{k1} - 1]$ (if $i^*_{k1} > i_{k\ell}$) and $[i^*_{k2} + 1, i_{kr}]$ (if $i^*_{k2} < i_{kr}$), while all the other s-intervals in G_{k-1} remain unchanged in G_k. This is achieved by subroutine update_s_interval$(i_{k\ell}, i^*_{k1}, i^*_{k2}, i_{kr})$ at line 10 by removing the s-interval $[i_{k\ell}, i_{kr}]$ from the data structure and inserting the decomposed nonempty s-intervals $[i_{k\ell}, i^*_{k1} - 1]$ and $[i^*_{k2} + 1, i_{kr}]$ into the data structure. Appendix A.3 shows that the data structure can complete the above operations, while keeping the s-intervals sorted, in $O(\log n)$ time. Hence the complexity of update_s_interval is $O(\log n)$. The pseudocode

of update_s_interval is in Appendix A.3. The optimal solution is returned at line 14.

It takes $O(n \log n)$ time to sort the breakpoints a_i's. In each iteration of the **for** loop starting at line 3, each of the four subroutines called takes $O(\log n)$ time. For each node $i \in V = [1, n]$, its corresponding decision variable gets optimal value assigned exactly once, and it changes status from s to t exactly once over the n iterations. Thus the amortized complexity of line 9 is $O(1)$ in each iteration. Therefore each iteration takes amortized time $O(\log n)$. This completes the proof of Theorem 5.1.

In addition, reading the input data and outputting the optimal solution take $O(n)$ time in total. Thus the total complexity of ℓ_1-GIMR-Algorithm is $O(n \log n)$. We therefore conclude the following.

THEOREM 5.8. ℓ_1-GIMR-Algorithm *solves problem* ℓ_1*-GIMR* (5.1) *in* $O(n \log n)$ *time.*

6. Extending ℓ_1-GIMR-Algorithm to GIMR-Algorithm. The key ideas used in ℓ_1-GIMR-Algorithm are extended here for GIMR (1.1) of general convex piecewise linear deviation functions. The adjustments required are described below.

First we note that without loss of generality, any convex piecewise linear deviation function $f_i^{pl}(x_i)$ with box constraints for the variable $\ell_i \le x_i \le u_i$, is equivalent to a convex piecewise linear function without the box constraints:

$$\tilde{f}_i^{pl}(x_i) = \begin{cases} f_i^{pl}(\ell_i) - M(x_i - \ell_i) & \text{for } x_i < \ell_i, \\ f_i^{pl}(x_i) & \text{for } \ell_i \le x_i \le u_i, \\ f_i^{pl}(u_i) + M(x_i - u_i) & \text{for } x_i > u_i \end{cases}$$

for M sufficiently large. Therefore GIMR is unconstrained, without loss of generality, with the first piece of each convex piecewise linear function having negative (nonpositive) slope ($w_{i,0} = -M$) and the last piece of each convex piecewise linear function having positive (nonnegative) slope ($w_{i,q_i} = M$).

The running time of GIMR-Algorithm is proved in Theorem 6.1.

THEOREM 6.1. *GIMR* (1.1) *is solved in* $O(q \log n)$ *time, where* q *is the total number of breakpoints of the* n *arbitrary convex piecewise linear deviation functions.*

Proof. For GIMR (1.1), the structure of G_0 remains as in Figure 1 as for ℓ_1-GIMR (5.1) with $c_{s,i} = -w_{i,0} > 0$ and $c_{i,t} = 0$ for all $i = 1, \ldots, n$. Thus the minimum cut in G_0 is $(\{s\} \cup V, \{t\})$. Hence subroutine initialization() is still valid for GIMR (1.1) in the same complexity.

For GIMR (1.1), as for ℓ_1-GIMR (5.1), all arc capacities other than c_{s,i_k} and $c_{i_k,t}$ are the same for both G_{k-1} and G_k. But the construction of G_k from G_{k-1} is more complicated than that in ℓ_1-GIMR. Recall that from G_{k-1} to G_k, the right subgradient of $f_{i_k}^{pl}$ changes from w_{i_k,j_k-1} to w_{i_k,j_k}. Thus depending on the signs of w_{i_k,j_k-1} and w_{i_k,j_k}, we have the following three possible cases:

Case 1. $w_{i_k,j_k-1} \le 0$, $w_{i_k,j_k} \le 0$: c_{s,i_k} is changed from $-w_{i_k,j_k-1}$ to $-w_{i_k,j_k}$.

Case 2. $w_{i_k,j_k-1} \le 0$, $w_{i_k,j_k} \ge 0$: c_{s,i_k} is changed from $-w_{i_k,j_k-1}$ to 0 and $c_{i_k,t}$ is changed from 0 to w_{i_k,j_k}.

Case 3. $w_{i_k,j_k-1} \ge 0$, $w_{i_k,j_k} \ge 0$: $c_{i_k,t}$ is changed from w_{i_k,j_k-1} to w_{i_k,j_k}.

The capacities of other arcs do not change.

Accordingly, the four arrays are updated for each one of these three cases as follows:

Case 1. $w_{i_k,j_k-1} \le 0$, $w_{i_k,j_k} \le 0$:

$$\begin{aligned}&\forall i \in [i_k, n]:\\ &\quad sa(i) := sa(i) - (w_{i_k,j_k} - w_{i_k,j_k-1}),\\ &\quad tms(i) := tms(i) + (w_{i_k,j_k} - w_{i_k,j_k-1}),\\ &\quad smt(i) := smt(i) - (w_{i_k,j_k} - w_{i_k,j_k-1}).\end{aligned}$$

Case 2. $w_{i_k,j_k-1} \le 0$, $w_{i_k,j_k} \ge 0$:

$$\begin{aligned}&\forall i \in [i_k, n]:\\ &\quad sa(i) := sa(i) + w_{i_k,j_k-1},\\ &\quad ta(i) := ta(i) + w_{i_k,j_k},\\ &\quad tms(i) := tms(i) + (w_{i_k,j_k} - w_{i_k,j_k-1}),\\ &\quad smt(i) := smt(i) - (w_{i_k,j_k} - w_{i_k,j_k-1}).\end{aligned}$$

Case 3. $w_{i_k,j_k-1} \ge 0$, $w_{i_k,j_k} \ge 0$:

$$\begin{aligned}&\forall i \in [i_k, n]:\\ &\quad ta(i) := ta(i) + (w_{i_k,j_k} - w_{i_k,j_k-1}),\\ &\quad tms(i) := tms(i) + (w_{i_k,j_k} - w_{i_k,j_k-1}),\\ &\quad smt(i) := smt(i) - (w_{i_k,j_k} - w_{i_k,j_k-1}).\end{aligned}$$

Although seemingly more complicated, all the above operations amount to adding a constant to a subarray, which can be done efficiently using the data structure for the four arrays. The above update is done by calling the subroutine update_arrays$(i_k, w_{i_k,j_k-1}, w_{i_k,j_k})$ in complexity $O(\log n)$ (see Appendix B.2 for pseudocode). Note that the ℓ_1 deviation function in ℓ_1-GIMR (5.1) is a special case of Case 2 above where $w_{i_k,j_k-1} < 0$, $w_{i_k,j_k} > 0$, and $-w_{i_k,j_k-1} = w_{i_k,j_k}$.

On the other hand, since all arc capacities other than c_{s,i_k} and $c_{i_k,t}$ are the same for both G_{k-1} and G_k, Lemmas 5.2, 5.3, and 5.4 and Corollary 5.5 for ℓ_1-GIMR (5.1) hold true for GIMR (1.1). As a result, the procedure to identify the node status change interval $[i^*_{k1}, i^*_{k2}]$ in graph G_k for ℓ_1-GIMR, shown in sections 5.1 and 5.2, also applies to GIMR. This implies that subroutines get_s_interval, find_status_change_interval, and update_s_interval are still valid for GIMR in the same complexity, respectively, as for ℓ_1-GIMR. Thus Theorem 5.1 holds for GIMR.

The complete GIMR-Algorithm follows.

GIMR-Algorithm

input: $\{\{a_{i,1}, \ldots, a_{i,q_i}\}, \{w_{i,0}, \ldots, w_{i,q_i}\}\}_{i=1,\ldots,n}$ and $\{d_{i,i+1}, d_{i+1,i}\}_{i=1,\ldots,n-1}$ in GIMR (1.1).

output: An optimal solution $\{x^*_i\}_{i=1,\ldots,n}$.

begin

Sort the breakpoints of all the n convex piecewise linear functions as $a_{i_1,j_1} < a_{i_2,j_2} < \cdots < a_{i_q,j_q}$;

initialization();

for $k := 1, \ldots, q$:

{Update graph}update_arrays(i_k, w_{i_k,j_k-1}, w_{i_k,j_k});

if $status(i_k) = s$ **then**

$[i_{k\ell}, i_{kr}] :=$ get_s_interval(i_k);

```
                [i*_{k1}, i*_{k2}] := find_status_change_interval(i_{kℓ}, i_k, i_{kr});
                if [i*_{k1}, i*_{k2}] ≠ ∅ then
                    for i ∈ [i*_{k1}, i*_{k2}]: x*_i := a_{i_k,j_k}, status(i) := t;
                    update_s_interval(i_{kℓ}, i*_{k1}, i*_{k2}, i_{kr});
                end if
            end if
        end for
        return {x*_i}_{i=1,...,n};
end
```

Reading the input data takes $O(q)$ time. Sorting the q breakpoints from n ordered lists takes $O(q \log n)$ time [12]. The amortized complexity of each iteration in the **for** loop remains $O(\log n)$, thus for the q iterations the total complexity is $O(q \log n)$. Finally it takes $O(n)$ time to output the optimal solution. Therefore the total complexity of GIMR-Algorithm is $O(q \log n)$. ∎

Remark 6.2. The discussion above and algorithms' presentations assume that all breakpoints are distinct. However, when this is not the case and a breakpoint is shared by more than one function, the algorithms still work in the same way, by breaking ties arbitrarily: A breakpoint is associated with a function or a variable, or with multiple functions and variables. The ordering of the variables that correspond to the same breakpoint can be selected arbitrary. To see that, consider a "perturbed" problem, in which small perturbations are applied to the original shared breakpoints so as to break the ties. The values that the optimal variables would then assume are either values of the breakpoints or the perturbed breakpoints. Since the perturbations can be made arbitrarily small, it follows that the optimal variables values will be among the "unperturbed" breakpoints.

Remark 6.3. The GIMR-Algorithm is applicable for solving, not only the continuous, but also the integer GIMR (1.1) problem. In [18] it is proved in the threshold theorem (Theorem 3.1) that the integer optimal solution can only take values at arguments where at least one function's integer subgradient, $f'(x) = f(x+1) - f(x)$, changes. For piecewise linear functions that can have noninteger breakpoints, the subgradients can only change at integer arguments that are the breakpoints of the piecewise linear functions rounded up or down. Therefore instead of considering only the q breakpoints, as is the continuous case proved in Lemma 4.1, the integer case requires us to consider up to $2q$ breakpoints. Therefore the running time of GIMR-Algorithm for the integer version of GIMR is the same as for the continuous version, $O(q \log n)$.

7. Experimental study. We implement GIMR-Algorithm in C++ in Microsoft Visual Studio 2015. We use the "set" data structure object in C++ standard template library (STL) to maintain the set of s-intervals in the algorithm. The dynamic path data structure has been implemented according to [43]. In order to assess the performance of GIMR-Algorithm in practice, we compare our software implementation with Gurobi, a commercial linear programming solver, on 30 simulated data sets of various sizes. Both algorithms are run on the same laptop with an Intel(R) Core i7-6820HQ CPU at 2.70 GHz, 32 GB RAM, and 64-bit Windows 10 operating system.

The GIMR problem can be formulated as a linear programming problem: Let $b_{i,j} = f_i^{pl}(a_{i,j})$ for $i = 1, \ldots, n; j = 1, \ldots, q_i$. It is easy to see that the following linear programming problem has the same optimal solution as GIMR (1.1), where u_i is the upper envelope of the q_i lines, $\{w_{i,0}(x_i - a_{i,1}) + b_{i,1}, \{w_{i,j}(x_i - a_{i,j}) + b_{i,j}\}_{j=1,\ldots,q_i}\}$,

TABLE 2

Running time (in seconds) comparison between GIMR-Algorithm *and Gurobi for solving GIMR* (1.1). *The numbers reported are the average running times (standard deviations).*

	Time (in seconds)	
$(n, \bar{q})$	GIMR-Algorithm	Gurobi
(100, 100)	0.17(0.013)	1.64(0.034)
(100, 1000)	1.47(0.079)	15.59(0.29)
(1000, 100)	1.97(0.035)	16.37(0.50)
(1000, 1000)	16.33(0.12)	148.46(0.40)
(1000, 10000)	174.06(9.86)	1608.36(61.66)
(10000, 1000)	190.53(3.62)	1559.07(63.41)

which correspond to the q_i linear pieces of function $f_i^{pl}(x_i)$:

$$\begin{aligned}
\min_{\substack{\{u_i,x_i\}_{i=1,\dots,n}\\ \{z_{i,i+1},z_{i+1,i}\}_{i=1,\dots,n-1}}} \quad & \sum_{i=1}^{n} u_i + \sum_{i=1}^{n-1} d_{i,i+1} z_{i,i+1} + \sum_{i=1}^{n-1} d_{i+1,i} z_{i+1,i} \\
\text{s.t. } & u_i \geq w_{i,0}(x_i - a_{i,1}) + b_{i,1},\ i = 1,\dots,n, \\
& u_i \geq w_{i,j}(x_i - a_{i,j}) + b_{i,j},\ i = 1,\dots,n; j = 1,\dots,q_i, \\
& x_i - x_{i+1} \leq z_{i,i+1},\ i = 1,\dots,n-1, \\
& x_{i+1} - x_i \leq z_{i+1,i},\ i = 1,\dots,n-1, \\
& \ell_i \leq x_i \leq u_i,\ i = 1,\dots,n, \\
& z_{i,i+1}, z_{i+1,i} \geq 0,\ i = 1,\dots,n-1.
\end{aligned}$$

The simulated data sets. In the generated data sets there are no box constraints. For each convex piecewise linear deviation function, we let the slope of the first linear piece be negative and the slope of the last linear piece be positive. That guarantees that the problem has an optimal solution in a bounded interval. In the separation terms, we set $d_{i,i+1} = d_{i+1,i} = d_i$, thus $d_{i,i+1}(x_i - x_{i+1})_+ + d_{i+1,i}(x_{i+1} - x_i)_+ = d_i|x_i - x_{i+1}|$. We set the number of breakpoints of each piecewise linear function $f_i^{pl}(x_i)$, q_i, to all be equal to a common value $\bar{q}$. Thus the total number of breakpoints of the n convex piecewise linear functions is $q = n\bar{q}$. For each pair of $(n, \bar{q})$, we generate 5 random problem instances, by randomly generating 5 groups of $\bar{q} + 1$ slope values, $\bar{q}$ breakpoints (for each convex piecewise linear deviation function), and d_i coefficients for the separation terms. The slopes of each $f_i^{pl}(x_i)$ are randomly generated in increasing order as follows: We first sample the value of $w_{i,0}$ from a uniform distribution on $(-\bar{q}, 0)$. Each subsequent breakpoint $w_{i,j}$ $(j = 1,\dots,q_i)$ is generated by adding a uniformly sampled random real value from $(0, 100)$ to $w_{i,j-1}$. The breakpoints of $f_i^{pl}(x_i)$ are generated in increasing order as follows: A first value, denoted as $a_{i,0}$, is sampled with uniform distribution from $(-\bar{q}, 0)$. This value is not a breakpoint. Each subsequent breakpoint $a_{i,j}$ $(j = 1,\dots,q_i)$ is generated by adding a uniformly sampled real value from $(0, 100)$ to $a_{i,j-1}$. This guarantees that the generated slopes and breakpoints are strictly increasing in each convex piecewise linear function. Each d_i value is sampled uniformly from the interval $(0, \bar{q})$.

We compare the average running times of GIMR-Algorithm and Gurobi for the 5 random instances of GIMR for each pair of $(n, \bar{q})$. We report the average running times(standard deviations) for all six families of problem instances in Table 2.

From Table 2 one can see that GIMR-Algorithm is approximately 10 times faster than Gurobi for each problem size with a smaller standard deviation.

8. Concluding remarks. We describe here an efficient algorithm that solves GIMR (1.1), generalizing isotonic median regression and a class of fused lasso problems with wide applications in signal processing, bioinformatics, and statistical learning. The algorithm proposed here is the first known unified, and most efficient in terms of complexity to date, for IMR, SIMR, and fused lasso problems with convex piecewise linear deviation functions. The latter includes the Q-FL and the Q-wFL problems. For all these problems our algorithm improves or matches on previous complexities of a collection of specialized algorithms and offers a unified framework for all these problems. The unified framework here is also amenable to extensions to other generalized versions of GIMR, that include generalized IMR on simple structure graphs, such as directed trees or cycles. The algorithm devised here is also shown to work well in practice, as demonstrated in an empirical study.

Appendix A. Red-black tree data structure to maintain *s*-intervals. A red-black tree is a binary search tree. Each node of the tree contains the following five fields [12]:
color: The "color" of a node. Its value is either RED or BLACK.
key: The "key" value of a node. It is a scalar.
left, right: The pointers to the left and the right child of a node. If the corresponding child does not exist, the corresponding pointer has value NIL.
p: The pointer to the parent of a node. If the node is the root node, the pointer value is NIL.

As it is a binary search tree, the keys of the nodes are comparable. Furthermore, it has the following two properties [12]:

1. Binary-search-tree property: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[y] \geq key[x]$.

2. Tree height property: A red-black tree with n nodes has height at most $2\log(n+1)$.

We use a red-black tree data structure T to represent the set of s-intervals. Each node of the tree represents one s-interval. Due to the disjointness property of s-intervals, every s-interval is uniquely identified by its two endpoints. Thus we extend the key field from a scalar to a *tuple*: For a node x representing an s-interval $[i_\ell, i_r]$, the key field of x, $key[x]$, contains two values, $key[x].first$ and $key[x].second$, such that $key[x].first = i_\ell$ and $key[x].second = i_r$. As a result, we also define a comparison between the *key tuples* of two nodes, which can also be viewed as a comparison between their corresponding s-intervals: For any two nodes x_1 (representing an s-interval $[i_{\ell 1}, i_{r1}]$) and x_2 (representing an s-interval $[i_{\ell 2}, i_{r2}]$), we define

1. $key[x_1] < key[x_2]$: if $key[x_1].second = i_{r1} < i_{\ell 2} = key[x_2].first$. It is the case where $[i_{\ell 1}, i_{r1}]$ is on the left of $[i_{\ell 2}, i_{r2}]$;

2. $key[x_1] = key[x_2]$: if $key[x_1].first = i_{\ell 1} = i_{\ell 2} = key[x_2].first$. It implies that $key[x_1].second = i_{r1} = i_{r2} = key[x_2].second$. It is the case where x_1 and x_2 refer to the same tree node and the same s-interval;

3. $key[x_1] > key[x_2]$: if $key[x_1].first = i_{\ell 1} > i_{r2} = key[x_2].second$. It is the case where $[i_{\ell 1}, i_{r1}]$ is on the right of $[i_{\ell 2}, i_{r2}]$.

As s-intervals do not overlap, the comparison of any two (possibly identical) nodes in the tree must fall into exactly one of the above three outcomes.

The above is our only extension of the red-black tree discussed in [12] in our algorithm. Since any two different nodes in the tree represent different s-intervals,

the binary-search-tree property still holds with the two inequalities being strict, i.e., " $\leq$ " $\rightarrow$ " $<$ ". The tree height property still satisfies as its proof in [12] does not involve the *key* fields.

A.1. Initializing the red-black tree with a single *s*-interval. In each of the two algorithms presented in the paper, the red-black tree is initialized with a single s-interval $[1, n]$. This is achieved by a subroutine $z :=$ new_node(i_ℓ, i_r) initializing a new node z for s-interval $[i_\ell, i_r]$, such that $color[z] = RED$, $key[z].first = i_\ell, key[z].second = i_r$, and $left[z] = right[z] = p[z] = NIL$ [12]. This is done in $O(1)$ time [12]. Thus the initialization of the tree is completed by calling $z :=$ new_node$(1, n)$. Node z is the root of the tree.

A.2. Pseudocode of subroutine $[i_{k\ell}, i_{kr}] :=$ get_s_interval(i_k). Let $root[T]$ represent the root node of the red-black tree T. The pseudocode of get_s_interval is

$[i_{k\ell}, i_{kr}] :=$ get_s_interval(i_k)
begin
  $z := root[T]$;
  while $z \neq NIL$:
    if $i_k \geq key[z].first$ and $i_k \leq key[z].second$ {node i_k is in the s-interval represented by node z}
    then $i_{k\ell} := key[z].first$, $i_{kr} := key[z].second$; **return** $[i_{k\ell}, i_{kr}]$;
    else if $i_k < key[z].first$
    then $z := left[z]$;
    else $z := right[z]$;
    end if
  end while
end

The correctness of the pseudocode is justified by the binary-search-tree property with the extended comparison for the key tuples. The complexity is determined by the height of the tree. Note that since all s-intervals are originally decomposed from the initial s-interval $[1, n]$, then the number of s-intervals generated throughout the algorithm is at most n. Thus the red-black tree has at most n nodes. Therefore the tree height is at most $2\log(n + 1)$. As a result, the complexity of get_s_interval is $O(\log n)$.

A.3. Pseudocode of subroutine update_s_interval$(i_{k\ell}, i^*_{k1}, i^*_{k2}, i_{kr})$. Cormen et al. in [12] define and analyze the following three operations on a red-black tree with scalar *key* values:

1. TREE-SEARCH(T, k): Searching for a node in red-black tree T with a given key value k. It returns a pointer to a node with key k if one exists; otherwise it returns NIL. In our case the key value is a tuple and the comparison of scalar key values is extended to key tuples.
2. RB-INSERT(T,z): Inserting a node z into red-black tree T. The pseudocode involves comparing the key values of two nodes, where we can simply apply our definition of a key tuple comparison. As a result, *literally* there is no change to the pseudocode of RB-INSERT(T, z) in our extension.
3. RB-DELETE(T, z): Deleting a node z from red-black tree T. The pseudocode does not involve the key field, hence it is directly applicable to our extension.

Cormen et al. in [12] prove that each of the above operations has complexity $O(\log n)$ for a tree of at most n nodes. The complexities are the same in our case with key tuples.

update_s_interval is implemented by calling the above three built-in operations, and the new_node subroutine. It changes the red-black tree T.

update_s_interval$(i_{k\ell}, i^*_{k1}, i^*_{k2}, i_{kr})$
begin
$z :=$ TREE-SEARCH$(T, (i_{k\ell}, i_{kr}))$;
RB-DELETE(T, z);
if $i_{k\ell} \le i^*_{k1} - 1$ **then** $z :=$ new_node$(i_{k\ell}, i^*_{k1} - 1)$; RB-INSERT$(T, z)$; **end if**
if $i^*_{k2} + 1 \le i_{kr}$ **then** $z :=$ new_node$(i^*_{k2} + 1, i_{kr})$; RB-INSERT(T, z); **end if**
end

As the tree has at most n nodes, each call to update_s_interval takes $O(\log n)$ time.

Appendix B. Dynamic path data structure to maintain the four arrays. Dynamic path [43] is a data structure for a collection of vertex-disjoint paths. Each path in the collection is an undirected symmetric path, where each edge has a real-valued cost.

Each internal vertex of a path has two edges adjacent to it. To distinguish the two edges, we define the following terminology. We designate one end of the path as *head* and the other end as *tail* [43]. For any internal vertex v, we define the vertex *before* vertex v as the index of the adjacent vertex of v that is closer to the head of the path. Similarly, we define the vertex *after* vertex v as the index of the adjacent vertex of v that is closer to the tail of the path [43]. For head vertex v, the vertex before vertex v does not exist. Likewise, for tail vertex v, the vertex after vertex v does not exist. The designation of the head and tail vertices is arbitrary. If the head and tail vertices are reversed, the references to the vertices before and after vertex v are also reversed accordingly.

The following 11 operations are supported in dynamic paths [43]:

$p :=$ *path*(v): Return the path p containing vertex v.

$v :=$ *head*(p): Return the head vertex v of path p.

$v :=$ *tail*(p): Return the tail vertex v of path p.

$u :=$ *before*(v): Return the vertex u before vertex v on *path*(v). If v is the head of the path, return NIL.

$u :=$ *after*(v): Return the vertex u after vertex v on *path*(v). If v is the tail of the path, return NIL.

$x :=$ *pcost*(v): Return the real-valued cost x of the edge $(v,$ *after*$(v))$. If vertex v is the tail of the path, return NIL.

$v :=$ *pmincost*(p): Return the vertex v closest to *tail*(p) such that $(v,$ *after*$(v))$ has minimum cost among edges on path p. If p contains only one vertex (degenerate case), return NIL.

pupdate(p, x): Add real value x to the cost of every edge on path p.

reverse(p): Reverse the direction of path p, making the head the tail and vice versa.

$p_3 :=$ *concatenate*(p_1, p_2, x): Merge paths p_1 and p_2 by adding the edge $($*tail*$(p_1),$ *head*$(p_2))$ of real-valued cost x. Return the merged path p_3.

$[p_1, p_2, x, y] :=$ *split*(v): Divide *path*(v) into (up to) three parts by deleting the edges incident to v. Return a list $[p_1, p_2, x, y]$, where p_1 is the subpath consisting of all vertices from *head*$($*path*$(v))$ to *before*(v), p_2 is the subpath consisting of all vertices from *after*(v) to *tail*$($*path*$(v))$, x is the cost of the deleted edge $($*before*$(v), v)$, and y is the cost of the deleted edge $(v,$ *after*$(v))$. If v is originally the head of *path*(v), p_1 is NIL and x is undefined; if v is originally the tail of *path*(v), p_2 is NIL and y is undefined.

A dynamic path is implemented as a full balanced binary tree [43]. Each vertex of the path is constructed as a leaf node of the tree and each edge of the path is constructed as a nonleaf node of the tree, which stores the edge cost as a node field. Besides, each node in the tree contains various other fields in support of efficient implementation of the above 11 operations on dynamic paths. The complete details of the binary tree implementation of a dynamic path was presented in [43, Chap. 4].

We highlight the complexity of each of the above operations: Sleator and Tarjan in [43] show that, for a collection of dynamic paths with a total of $O(n)$ vertices, $head(p)$, $tail(p)$, $pupdate(p, x)$, and $reverse(p)$ each takes $O(1)$ time and $path(v)$, $before(v)$, $after(v)$, $pcost(v)$, $pmincost(p)$, $concatenate(p_1, p_2, x)$, and $split(v)$ each takes $O(\log n)$ time.

We define the following two additional split operations that are more convenient to use for our purpose:

$[p_1, p_2, x] :=$ *split-before*(v): Divide $path(v)$ into (up to) two parts by deleting the edge $(before(v), v)$. Return a list $[p_1, p_2, x]$, where p_1 is the subpath consisting of all vertices from $head(path(v))$ to $before(v)$, p_2 is the subpath consisting of all vertices from v to $tail(path(v))$, x is the cost of the deleted edge $(before(v), v)$. If v is originally the head of $path(v)$, p_1 is NIL and x is undefined.

$[p_1, p_2, y] :=$ *split-after*(v): Divide $path(v)$ into (up to) two parts by deleting the edge $(v, after(v))$. Return a list $[p_1, p_2, y]$, where p_1 is the subpath consisting of all vertices from $head(path(v))$ to v, p_2 is the subpath consisting of all vertices from $after(v)$ to $tail(path(v))$, y is the cost of the deleted edge $(v, after(v))$. If v is originally the tail of $path(v)$, p_2 is NIL and y is undefined.

Both *split-before* and *split-after* can be implemented efficiently using *concatenate* and *split*:

$[p_1, p_2, x] :=$ *split-before*(v)
begin
  $[p_1, p_2, x, y] :=$ *split*(v);
  if $p_2 \neq NIL$ **then** $p_2 :=$ *concatenate*(v, p_2, y);
  else $p_2 := [v, v]$; {a path with single vertex v}
  end if
  return $[p_1, p_2, x]$;
end
$[p_1, p_2, y] :=$ *split-after*(v)
begin
  $[p_1, p_2, x, y] :=$ *split*(v);
  if $p_1 \neq NIL$ **then** $p_1 :=$ *concatenate*(p_1, v, x);
  else $p_1 := [v, v]$; {a path with single vertex v}
  end if
  return $[p_1, p_2, y]$;
end

Since each subroutine calls one split and one concatenate operation, the complexity of *split-before*(v) and *split-after*(v) is each $O(\log n)$. We include *split-before*(v) and *split-after*(v) into our pool of dynamic path operations.

B.1. Initializing the four arrays for G_0. For GIMR (1.1), the four arrays are initiated for G_0 as follows:

begin
  $sa(0) := 0$, $sa(i) := sa(i-1) - w_{i,0}$ for $i = 1, \ldots, n$;
  $p_{sa} :=$ init_dynamic_path$((sa(i))_{i=0,1,\ldots,n})$;

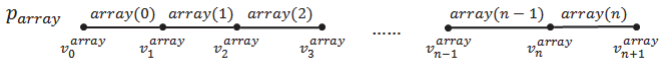

FIG. 5. *p_{array} is a dynamic path constructed from array $(array(i))_{i=0,1,\ldots,n}$. In p_{array}, we designate vertex v_0^{array} as head and vertex v_{n+1}^{array} as tail. p_{array} is implemented as a single full balanced binary tree of $n+1$ nonleaf nodes and $n+2$ leaf nodes.*

$ta(i) := 0$, for $i = 0, 1, \ldots, n$;
$p_{ta} :=$ init_dynamic_path$((ta(i))_{i=0,1,\ldots,n})$;
$tms(0) := 0$, $tms(i) := ta(i) - sa(i) + d_{i,i+1}$, for $i = 1, \ldots, n-1$, $tms(n) := ta(n) - sa(n)$;
$p_{tms} :=$ init_dynamic_path$((tms(i))_{i=0,1,\ldots,n})$;
$smt(0) := 0$, $smt(i) := sa(i) - ta(i) + d_{i+1,i}$, for $i = 1, \ldots, n-1$; $smt(n) := sa(n) - ta(n)$;
$p_{smt} :=$ init_dynamic_path$((smt(i))_{i=0,1,\ldots,n})$;

end

Note that for ℓ_1-GIMR (5.1), $w_{i,0} = -w_i$. An ordinary array is converted into a dynamic path by subroutine $p_{array} :=$ init_dynamic_path$((array(i))_{i=0,1,\ldots,n})$, which takes as argument an array $(array(i))_{i=0,1,\ldots,n}$ and returns a dynamic path p_{array} constructed from the array.

Let $(array(i))_{i=0,1,\ldots,n}$ be any of the four arrays of $n+1$ elements. We can construct a dynamic path p_{array} of $n+2$ vertices, from vertex v_0^{array} to v_{n+1}^{array}, such that the cost of edge $(v_i^{array}, v_{i+1}^{array})$ is $array(i)$ for $i = 0, 1, \ldots, n$ (see Figure 5).

We use the *concatenate* operation to construct p_{array} from array $(array(i))_{i=0,1,\ldots,n}$. The pseudocode is the following:

$p_{array} :=$ init_dynamic_path$((array(i))_{i=0,1,\ldots,n})$
begin
  Initialize dynamic path p_{array} of a single vertex v_0^{array}, i.e., p_{array} $:= [v_0^{array}, v_0^{array}]$;
  for $i := 0, \ldots, n$:
    Create dynamic path q of a single vertex v_{i+1}^{array}, i.e., $q := [v_{i+1}^{array}, v_{i+1}^{array}]$;
    $p_{array} := concatenate(p_{array}, q, array(i))$;
  end for
  return p_{array};
end

It takes $O(1)$ time to create a single vertex dynamic path [43], therefore, the complexity of the subroutine is $O(n \log n)$.

Note that by the definition of the concatenate operation, p_{array} has vertex v_0^{array} as head and vertex v_{n+1}^{array} as tail. Recall that $pcost(v)$ returns the cost of edge $(v, after(v))$, hence $array(i)$ is accessed by calling $pcost(v_i^{array})$.

B.2. Pseudocode of subroutine update_arrays($i_k, w_{i_k,j_k-1}, w_{i_k,j_k}$). We define subroutine update_constant(p_{array}, i_k, w) that adds a constant value w to the subpath of dynamic path p_{array} that corresponds to the subarray from $array(i_k)$ to $array(n)$.

update_constant(p_{array}, i_k, w)
begin
  $[p, q, x] := split\text{-}before(v_{i_k}^{array})$; {$q$ is the subpath corresponding to the subarray $array(i_k)$ to $array(n)$}
  $pupdate(q, w)$; $\{\forall j \in [i_k, n] : array(j) := array(j) + w\}$

if $p \neq NIL$ **then** $p_{array} := concatenate(p, q, x)$; **else** $p_{array} := q$; **end if** {Merge the split vertex-disjoint paths p and q back to a single dynamic path p_{array} corresponding to the whole array $(array(i))_{i=0,1,\ldots,n}$}

end

The complexity of each call to update_constant is $O(\log n)$.

With update_constant, update_arrays is implemented as follows:

update_arrays$(i_k, w_{i_k,j_k-1}, w_{i_k,j_k})$

begin

if $w_{i_k,j_k-1} \leq 0$ and $w_{i_k,j_k} \leq 0$ **then**
update_constant$(p_{sa}, i_k, -(w_{i_k,j_k} - w_{i_k,j_k-1}))$;
update_constant$(p_{tms}, i_k, w_{i_k,j_k} - w_{i_k,j_k-1})$;
update_constant$(p_{smt}, i_k, -(w_{i_k,j_k} - w_{i_k,j_k-1}))$;
else if $w_{i_k,j_k-1} \leq 0$ and $w_{i_k,j_k} \geq 0$ **then**
update_constant$(p_{sa}, i_k, w_{i_k,j_k-1})$;
update_constant$(p_{ta}, i_k, w_{i_k,j_k})$;
update_constant$(p_{tms}, i_k, w_{i_k,j_k} - w_{i_k,j_k-1})$;
update_constant$(p_{smt}, i_k, -(w_{i_k,j_k} - w_{i_k,j_k-1}))$;
else if $w_{i_k,j_k-1} \geq 0$ and $w_{i_k,j_k} \geq 0$ **then**
update_constant$(p_{ta}, i_k, w_{i_k,j_k} - w_{i_k,j_k-1})$;
update_constant$(p_{tms}, i_k, w_{i_k,j_k} - w_{i_k,j_k-1})$;
update_constant$(p_{smt}, i_k, -(w_{i_k,j_k} - w_{i_k,j_k-1}))$;
end if

end

As update_arrays makes constant a number of calls to update_constant, then the complexity of update_arrays is $O(\log n)$.

For the special case of ℓ_1-GIMR (5.1), update_arrays is called with argument $w_{i_k,j_k-1} = -w_{i_k}$ and $w_{i_k,j_k} = w_{i_k}$.

B.3. A pseudocode of subroutine $[i^*_{k1}, i^*_{k2}]$. The following pseudocode operates on the dynamic paths of the four arrays:

$[i^*_{k1}, i^*_{k2}]$:= find_status_change_interval$(i_{k\ell}, i_k, i_{kr})$

begin

{Identify $\hat{i}_{k1}$}
if $i_{k\ell} = i_k$ **then** $\hat{i}_{k1} := i_{k\ell}$, $f_1(\hat{i}_{k1}) = pcost(v^{sa}_{i_k}) - pcost(v^{sa}_{i_{k\ell}-1})$;
else
{Identify $\tilde{i}_{k1} \in [i_{k\ell}+1, i_k]$}
$[p_1, p_2, x] := split\text{-}before(v^{tms}_{i_{k\ell}})$; {Path p_{tms} is split into path p_1 and path p_2}
$[q_1, q_2, y] := split\text{-}after(v^{tms}_{i_k})$; {Path p_2 is split into path q_1 and path q_2. Path q_1 corresponds to the subarray $tms(i_{k\ell})$ to $tms(i_k - 1)$}
$\tilde{i}_{k1} := pmincost(q_1) + 1$;
{Recover a single dynamic path p_{tms} for $(tms(i))_{i=0,1,\ldots,n}$}
if $q_2 \neq NIL$ **then** $p_{tms} := concatenate(q_1, q_2, y)$; **else** $p_{tms} := q_1$; **end if**
if $p_1 \neq NIL$ **then** $p_{tms} := concatenate(p_1, p_{tms}, x)$; **end if**
$f_1(\tilde{i}_{k1}) := pcost(v^{tms}_{\tilde{i}_{k1}-1}) - pcost(v^{ta}_{i_{k\ell}-1}) + pcost(v^{sa}_{i_k}) + d_{i_{k\ell},i_{k\ell}-1}$;
$f_1(i_{k\ell}) := pcost(v^{sa}_{i_k}) - pcost(v^{sa}_{i_{k\ell}-1})$;
if $f_1(\tilde{i}_{k1}) \leq f_1(i_{k\ell})$ **then** $\hat{i}_{k1} := \tilde{i}_{k1}$, $f_1(\hat{i}_{k1}) := f_1(\tilde{i}_{k1})$; **else** $\hat{i}_{k1} := i_{k\ell}$, $f_1(\hat{i}_{k1}) := f_1(i_{k\ell})$; **end if**
end if

{Identify $\hat{i}_{k2}$}
if $i_{kr} = i_k$ **then** $\hat{i}_{k2} := i_{kr}$; $f_2(\hat{i}_{k2}) := pcost(v^{sa}_{i_{kr}}) - pcost(v^{sa}_{i_k})$;
else
{Identify $\tilde{i}_{k2} \in [i_k, i_{kr} - 1]$}
$[p_1, p_2, x] := \textit{split-before}(v^{smt}_{i_k})$; {Path p_{smt} is split into path p_1 and path p_2}
$[q_1, q_2, y] := \textit{split-after}(v^{smt}_{i_{kr}})$; {Path p_2 is split into path q_1 and path q_2. Path q_1 corresponds to the subarray $smt(i_k)$ to $smt(i_{kr} - 1)$}
$\textit{reverse}(q_1)$; {Make $v^{smt}_{i_k}$ as the tail and $v^{smt}_{i_{kr}}$ as the head}
$\tilde{i}_{k2} := \textit{pmincost}(q_1)$;
$\textit{reverse}(q_1)$; {Resume $v^{smt}_{i_k}$ as the head and $v^{smt}_{i_{kr}}$ as the tail}
{Recover a single dynamic path p_{smt} for $(smt(i))_{i=0,1,\ldots,n}$}
if $q_2 \neq NIL$ **then** $p_{smt} := \textit{concatenate}(q_1, q_2, y)$; **else** $p_{smt} := q_1$; **end if**
if $p_1 \neq NIL$ **then** $p_{smt} := \textit{concatenate}(p_1, p_{smt}, x)$; **end if**
$f_2(\tilde{i}_{k2}) := pcost(v^{smt}_{\tilde{i}_{k2}}) - pcost(v^{sa}_{i_k}) + pcost(v^{ta}_{i_{kr}}) + d_{i_{kr}, i_{kr}+1}$;
$f_2(i_{kr}) := pcost(v^{sa}_{i_{kr}}) - pcost(v^{sa}_{i_k})$;
if $f_2(\tilde{i}_{k2}) \leq f_2(i_{kr})$ **then** $\hat{i}_{k2} := \tilde{i}_{k2}$, $f_2(\hat{i}_{k2}) := f_2(\tilde{i}_{k2})$; **else** $\hat{i}_{k2} := i_{kr}$, $f_2(\hat{i}_{k2}) := f_2(i_{kr})$; **end if**
end if
$Z([\hat{i}_{k1}, \hat{i}_{k2}]) := f_1(\hat{i}_{k1}) + f_2(\hat{i}_{k2})$;
$Z(\emptyset) := pcost(v^{ta}_{i_{kr}}) - pcost(v^{ta}_{i_{k\ell}-1}) + d_{i_{k\ell}, i_{k\ell}-1} + d_{i_{kr}, i_{kr}+1}$;
if $Z(\emptyset) \leq Z([\hat{i}_{k1}, \hat{i}_{k2}])$ **then** $[i^*_{k1}, i^*_{k2}] := \emptyset$; **else** $[i^*_{k1}, i^*_{k2}] := [\hat{i}_{k1}, \hat{i}_{k2}]$; **end if**
return $[i^*_{k1}, i^*_{k2}]$;
end

The number of calls to the dynamic path operations is constant. More precisely, there are 16 calls to pcost, 2 calls to split-before, 2 calls to split-after, 2 calls to pmincost, 4 calls to concatenate, and 2 calls to reverse. Therefore the complexity of find_status_change_interval is $O(\log n)$.

REFERENCES

[1] R. K. Ahuja, D. S. Hochbaum, and J. B. Orlin, *Solving the convex cost integer dual network flow problem*, Manag. Sci., 49 (2003), pp. 950–964.

[2] R. K. Ahuja, D. S. Hochbaum, and J. B. Orlin, *A cut-based algorithm for the convex dual of the minimum cost network flow problem*, Algorithmica, 39 (2004), pp. 189–208.

[3] R. K. Ahuja and J. B. Orlin, *A fast scaling algorithm for minimizing separable convex functions subject to chain constraints*, Oper. Res., 49 (2001), pp. 784–789.

[4] S. Angelov, B. Harb, S. Kannan, and L.-S. Wang, *Weighted isotonic regression under the L_1 norm*, in Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithm, SODA, SIAM, Philadelphia, 2006, pp. 783–791.

[5] M. Ayer, H. D. Brunk, G. M. Ewing, W. T. Reid, and E. Silverman, *An empirical distribution function for sampling with incomplete information*, Ann. Math. Stat., 26 (1955), pp. 641–647.

[6] R. E. Barlow, D. J. Bartholomew, J. M. Bremner, and H. D. Brunk, *Statistical Inference under Order Restrictions: The Theory and Application of Isotonic Regression*, Wiley, New York, 1972.

[7] R. E. Barlow and H. D. Brunk *The isotonic regression problem and its dual*, J. Amer. Statist. Assoc., 67 (1972), pp. 140–147.

[8] D. J. Bartholomew, *A test for homogeneity for ordered alternatives*, Biometrika, 46 (1959), pp. 36–48.

[9] D. J. Bartholomew, *A test for homogeneity for ordered alternatives* II, Biometrika, 46 (1959), pp. 328–335.

[10] N. Chakravarti, *Isotonic median regression: A linear programming approach*, Math. Oper. Res., 14 (1989), pp. 303–308.
[11] T. F. Chan and S. Esedoglu, *Aspects of total variation regularized L^1 function approximation*, SIAM J. Appl. Math., 65 (2005), pp. 1817–1837.
[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 2009.
[13] K. Dembczyński, W. Kotlowski, and R. Slowiński, *Learning rule ensembles for ordinal classification with monotonicity constraints*, Fund. Inform. 94 (2009), pp. 163–178.
[14] L. Dümbgen and A. Kovac, *Extensions of smoothing via taut strings*, Electron. J. Stat., 3 (2009), pp. 41–75.
[15] P. H. C. Eilers and R. X. de Menezes, *Quantile smoothing of array CGH data*, Bioinformatics, 21 (2005), pp. 1146–1153.
[16] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan, *A fast parametric maximum flow algorithm and applications*, SIAM J. Comput., 18 (1989), pp. 30–55.
[17] D. S. Hochbaum, *Lower and upper bounds for the allocation problem and other nonlinear optimization problems*, Math. Oper. Res., 19 (1994), pp. 390–409.
[18] D. S. Hochbaum, *An efficient algorithm for image segmentaion, Markov random fields, and related problems*, J. ACM, 48 (2001), pp. 686–701.
[19] D. S. Hochbaum, *Complexity and algorithms for nonlinear optimization problems*, Ann. Oper. Res., 153 (2007), pp. 257–296.
[20] D. S. Hochbaum, *The pseudoflow algorithm: A new algorithm for the maximum flow problem*, Oper. Res., 58 (2008), pp. 992–1009.
[21] D. S. Hochbaum and J. B. Orlin, *Simplifications and speedups of the pseudoflow algorithm*, Networks, 61 (2013), pp. 40–57.
[22] D. S. Hochbaum and M. Queyranne, *Minimizing a convex cost closure set*, SIAM J. Discrete Math., 16 (2003), pp. 192–207.
[23] D. S. Hochbaum and J. G. Shanthikumar, *Convex separable optimization is not much harder than linear optimization*, J. ACM, 37 (1990), pp. 843–862.
[24] A. T. Kalai and R. Sastry, *The isotron algorithm: High-dimensional isotonic regression*, in Proceedings of Computational Learning Theory, COLT, Montreal, Quebec, 2009.
[25] Y. Kaufman and A. Tamir, *Locating service centers with precedence constraints*, Discrete Appl. Math., 47 (1993), pp. 251–261.
[26] V. Kolmogorov, T. Pock, and M. Rolinek, *Total variation on a tree*, SIAM J. Imaging Sci., 9 (2016), pp. 605–636.
[27] Y. J. Li and J. Zhu, *Analysis of array CGH data for cancer studies using fused quantile regression*, Bioinformatics, 23 (2007), pp. 2470–2476.
[28] W. L. Maxwell and J. A. Muckstadt, *Establishing consistent and realistic reorder intervals in production/distribution systems*, Oper. Res., 33 (1985), pp. 1316–1341.
[29] J. A. Mendendez and B. Salvador, *An algorithm for isotonic median regression*, Comput. Statist. Data Anal., 5 (1987), pp. 399–406.
[30] R. E. Miles, *The complete amalgamation into blocks, by weighted means, of a finite set of real numbers*, Biometrika, 46 (1959), pp. 317–327.
[31] M. Nikolova, *Minimizers of cost-functions involving nonsmooth data-fidelity terms. Application to the processing of outliers*, SIAM J. Numer. Anal., 40 (2002), pp. 965–994.
[32] A. Painsky and S. Rosset, *Isotonic modeling with non-differentiable loss functions with application to lasso regularization*, IEEE Trans. Pattern Anal. Mach. Intell., 38 (2016), pp. 308–321.
[33] P. M. Pardalos, G. L. Xue, and L. Yong, *Efficient computation of an isotonic median regression*, Appl. Math. Lett., 8 (1995), pp. 67–70.
[34] K. Punera and J. Ghosh, *Enhanced hierarchical classification via isotonic smoothing*, in Proceedings of the 17th International Conference on World Wide Web, WWW, ACM, New York, 2008, pp. 151–160.
[35] A. Restrepo and A. C. Bovik, *Locally monotonic regression*, IEEE Trans. Signal Process., 41 (1993), pp. 2796–2810.
[36] T. Robertson and P. Waltman, *On estimating monotone parameters*, Ann. Math. Statist., 39 (1968), pp. 1030–1039.
[37] T. Robertson and F. T. Wright, *Multiple isotonic median regression*, Ann. Statist., 1 (1973), pp. 422–432.
[38] T. Robertson and F. T. Wright, *Algorithms in order restricted statistical inference and the Cauchy mean value property*, Ann. Statist., 8 (1980), pp. 645–651.
[39] T. Robertson, F. T. Wright, and R. L. Dykstra, *Order Restricted Statistical Inference*, Wiley, New York, 1988.

[40] R. Roundy, *A 98%-effective integer-ratio lot-sizing for one-warehouse multi-retailer systems*, Manag. Sci., 31 (1985), pp. 1416–1430.
[41] Y. U. Ryu, R. Chandrasekaran, and V. Jacob, *Prognosis using an isotonic prediction technique*, Manag. Sci., 50 (2004), pp. 777–785.
[42] T. S. Shively, S. G. Walker, and P. Damien, *Nonparametric function estimation subject to monotonicity, convexity and other shape constraints*, J. Econometrics, 161 (2011), pp. 166–181.
[43] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–391.
[44] M. Storath, A. Weinmann, and M. Unser, *Exact algorithms for L^1-TV regularization of real-valued or circle-valued signals*, SIAM J. Sci. Comput., 38 (2016), pp. A614–A630.
[45] R. J. Tibshirani, H. Hoefling, and R. Tibshirani, *Nearly-isotonic regression*, Technometrics, 53 (2011), pp. 54–61.
[46] R. J. Tibshirani, M. Saunders, S. Rosset, J. Zhu, and K. Knight, *Sparsity and smoothness via the fused lasso*, J. Roy. Statist. Soc., Ser. B, 67 (2005), pp. 91–108.
[47] A. F. Veinott, Jr., *Least d-majorized network flows with inventory and statistical applications*, Manag. Sci., 17 (1971), pp. 547–567.
[48] H. S. Wang, G. D. Li, and G. H. Jiang, *Robust regression shrinkage and consistent variable selection through the LAD-lasso*, J. Bus. Econom. Statist., 25 (2007), pp. 347–355.