Operations Research Letters 43 (2015) 396–400

Contents lists available at ScienceDirect

Operations Research Letters

journal homepage: www.elsevier.com/locate/orl

Balanced splitting on weighted intervals

Wei Cao [a], Jian Li [a], Shimin Li [b,*], Haitao Wang [b]

[a] Institute for Interdisciplinary Information Sciences (IIIS), Tsinghua University, Beijing 100084, China

[b] Department of Computer Science, Utah State University, Logan, UT 84322, USA

ARTICLE INFO

Article history:
Received 19 December 2014
Received in revised form
8 May 2015
Accepted 8 May 2015
Available online 19 May 2015

Keywords:
Interval splitting
Combinatorial optimizations
Algorithms
Temporal databases
Multi-version databases

ABSTRACT

We study a problem on splitting intervals. Let $\mathcal{I}$ be a set of n intervals on a line L, with non-negative weights. Given any integer $k \geq 1$, we want to find k points on L to partition L into $k+1$ segments, such that the maximum cost of these segments is minimized, where the cost of each segment s is the sum of the weights of the intervals in $\mathcal{I}$ intersecting s. We present an $O(n \log n)$ time algorithm for this problem.

1. Introduction

We consider the problem of splitting weighted intervals in a balanced way. Let $\mathcal{I}$ be a set of n intervals on a line L, where each interval has a non-negative weight. Given an integer $k \geq 1$, we want to find k points on L to partition L into $k+1$ segments, such that the maximum cost of these segments is minimized, where the *cost* of each segment s is the sum of the weights of the intervals in $\mathcal{I}$ that "properly" intersect s (i.e., the intersection contains more than one point). The formal definition is given below.

Let $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ be a set of n intervals on a line L, and each interval I_i has a weight $w_i \geq 0$. For simplicity, we assume L is the x-axis, and depending on the context, any real value $x \in \mathbb{R}$ is also considered as the point on L with coordinate x, and vice versa. Each interval I_i is represented as $[l_i, r_i]$ with $l_i < r_i$, where l_i is its *left endpoint* and r_i is its *right endpoint*. Note that we consider each I_i as a closed interval including both endpoints.

For an integer $k \geq 1$, consider any k points $x_1, x_2, \ldots, x_k$ on L with $x_1 < x_2 < \cdots < x_k$, and we refer to these k points as *splitters*. For simplicity of discussion, let $x_0 = -\infty$ and $x_{k+1} = +\infty$. The above k splitters partition the line L into $k+1$ *open segments*: $s_i = (x_{i-1}, x_i)$ for $i = 1, 2, \ldots, k+1$. For each segment s_i, we define its *cost* $C(s_i)$ as the sum of the weights of the intervals of $\mathcal{I}$ that intersect s_i (e.g., see Fig. 1). Note that since each s_i is an open segment (i.e., it does not contain its two endpoints) and each interval of $\mathcal{I}$ is closed, if an interval I_i intersects s_i, then their intersection contains more than one point.

The *interval splitting* problem is to find k points/splitters $x_1, x_2, \ldots, x_k$ to partition L into $k+1$ open segments (as defined above) such that the maximum cost of all segments (i.e., $\max_{i=1}^{k+1} C(s_i)$) is minimized (e.g., see Fig. 1).

Previously, Le et al. [9] gave an $O(n \log n)$ time algorithm for a *special case* of this problem where $w_i = 1$ for each $1 \leq i \leq n$. Their algorithm, which is based on the observation that the maximum cost in any optimal solution must be an integer in $[1, n]$, does not work for our more general problem (see Section 4 for more discussions). In this paper, by developing new algorithmic techniques, we solve the general case in $O(n \log n)$ time.

1.1. Applications and related work

As discussed in [9], the interval splitting problem has applications in load balancing for storing and processing data in temporal and multi-version databases. If we consider the x-axis L as the time, each interval represents a time period during which an object in databases is associated with the same value. Since an object may be associated with different values during different time periods, the task is to store and process a large number of intervals in a distributed store. To this end, one can split the intervals into "buckets" (corresponding to the segments of L in our problem) such that intervals from the same buckets can be stored in one node and processed by one core from a cluster of machines. One challenging

* Corresponding author.
E-mail addresses: cao-w13@mails.tsinghua.edu.cn (W. Cao), lijian83@mail.tsinghua.edu.cn (J. Li), shiminli@aggiemail.usu.edu (S. Li), haitao.wang@usu.edu (H. Wang).

http://dx.doi.org/10.1016/j.orl.2015.05.005
0167-6377/

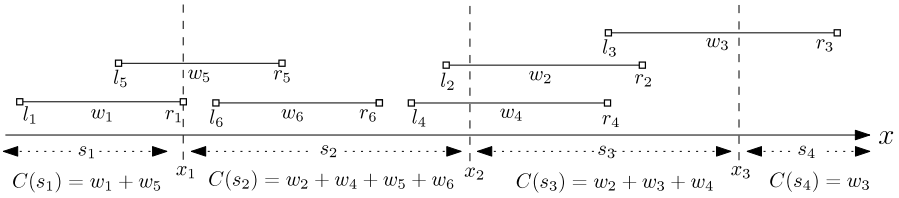

Fig. 1. Illustrating an example of the interval splitting problem for $k = 3$: Finding three points $x_1 < x_2 < x_3$ such that the maximum value of $\{C(s_1), C(s_2), C(s_3), C(s_4)\}$ is minimized.

problem is to achieve load-balancing in this process, i.e., no single node and core should store and process too many intervals. This is exactly (the special case of) our interval splitting problem. If each object has a weight, which may represent the difficulty or importance for storing and processing the object (and its corresponding intervals), then the problem becomes the general case of the interval splitting problem. Refer to [9] and the references therein for more discussions on temporal and multi-version databases.

The interval splitting problem is related to the classical interval scheduling problems. In the interval scheduling, each interval represents the time period during which a task needs to be executed. A subset of intervals is *compatible* if no two intervals overlap. One basic problem is to find a largest compatible set, and the problem can be solved by a simple greedy algorithm as shown in [8]. There are many other variations of problem; e.g., see [2,6,8,11,13].

Since problems related to intervals are normally very fundamental, there are many powerful tools dealing with these problems, such as interval graphs [5], interval trees [3], segment trees [4], etc. Unfortunately, none of these techniques seems useful for solving our interval splitting problem.

As discussed in [9], the interval splitting problem is also related to many other problems, e.g., finding optimal splitters for a set of one dimensional points [12], the array partitioning problems [7,10], etc.

1.2. Our approaches

We observe that there must exist an optimal solution in which every splitter is at the endpoint of an interval in $\mathcal{I}$. This observation implies that the objective value (i.e., the maximum cost of all segments s_i) of the optimal solution must be determined by two interval endpoints along with $-\infty$ and $+\infty$. This immediately gives $\Theta(n^2)$ *candidate* values for the optimal objective value since there are $2n$ interval endpoints. We can easily find the optimal objective value from these candidate values if we can solve the decision version of the problem: Given any value c, determine whether we can find k splitters such that the maximum cost of all segments s_i is no more than c.

Assume the $2n$ interval endpoints have already been sorted. We first present a greedy algorithm that can solve the decision version in $O(n)$ time. Then we use this algorithm to find the optimal objective value from the above candidate values. One difficulty is that since there are $\Theta(n^2)$ candidate values, computing them needs $\Omega(n^2)$ time. To reduce the running time, we manage to *implicitly* organize all the candidate values in $O(n)$ arrays and each array contains $O(n)$ elements in sorted order, and further, we give a data structure that can compute any candidate value in $O(1)$ time after $O(n)$ time preprocessing. Using this data structure and our decision algorithm, we apply a technique, called *binary search on sorted arrays* [1], to compute the optimal objective value in the above $O(n)$ sorted arrays. These efforts together lead to an $O(n \log n)$ time algorithm for solving the interval splitting problem.

The rest of the paper is organized as follows. We introduce some notations, definitions, and observations in Section 2. The algorithm for the decision problem is given in Section 3. In Section 4, we solve the interval splitting problem, which is referred to as the *optimization problem*.

2. Preliminaries

For ease of discussion, we make a general position assumption that no two intervals of $\mathcal{I}$ share the same endpoint, and our techniques can be easily adapted to the degenerate case.

We use an *open segment* to refer to a segment on L that does not include its endpoints. For any open segment s, let $\mathcal{I}(s)$ denote the set of intervals of $\mathcal{I}$ intersecting s, and let $C(s)$ denote the sum of the weights of the intervals in $\mathcal{I}(s)$ and we also call $C(s)$ the *cost* of s. For any point x on L, we let $\mathcal{I}(x)$ denote the set of intervals of $\mathcal{I}$ each of which contains x in its interior, and let $C(x)$ denote the sum of the weights of the intervals in $\mathcal{I}(x)$.

Let $X = \{x_1, x_2, \ldots, x_t\}$ be a set of points/splitters on L with $x_1 < x_2 < \cdots < x_t$, where t may or may not be equal to k. These splitters partition L into $t + 1$ open segments, and we denote by $C(X)$ the maximum cost of these open segments and $C(X)$ is referred to as the *cost* of X. We use C_{opt} to denote the cost of the set of splitters in any optimal solution of the interval splitter problem (for k splitters), and C_{opt} is also referred to as the *optimal objective value*.

Let E denote the set of all $2n$ endpoints of the intervals of $\mathcal{I}$. Due to our general position assumption, no two points of E have the same position. Let $e_1, e_2, \ldots, e_{2n}$ be the list of the points of E sorted on L from left to right.

We first prove Lemma 1. A similar observation has been made by Le et al. [9] for the special case where the weights of all intervals of $\mathcal{I}$ are 1, and here we extend their result to the general case.

Lemma 1. *For the interval splitting problem, there must exist an optimal solution in which every splitter is at the endpoint of an interval in $\mathcal{I}$ (i.e., every splitter is in E).*

Proof. Consider any optimal solution and assume $X = \{x_1, x_2, \ldots, x_k\}$ are the set of splitters sorted on L from left to right. We assume no two splitters in X have the same position since otherwise we could consider splitters at the same position as a single splitter.

If $X \subseteq E$, then we are done with the proof. Otherwise, consider any splitter x in X but not in E (i.e., $x \in X \setminus E$). For ease of discussions, we assume $x \in (e_1, e_{2n})$. Hence, there is some i with $1 \leq i \leq k-1$ such that $x \in (e_i, e_{i+1})$. If the open interval (e_i, e_{i+1}) contains some other splitters in X, then among such splitters, we let x represent the one closest to e_i. Hence, there is no splitter in the interval (e_i, x) (e.g., see Fig. 2).

An easy observation is that if we move x to e_i, the value $C(X)$ does not increase. Since X is an optimal solution, we further conclude $C(X)$ does not change and we have obtained another optimal solution after x moves to e_i. Notice that in the new optimal solution, the size $|X \setminus E|$ become one less than before. If in the new optimal solution the size $|X \setminus E|$ is zero, then we are done with the proof (i.e., we have found an optimal solution in which all splitters are in E); otherwise, we repeatedly apply the above "moving technique" until $|X \setminus E|$ becomes zero. The lemma thus follows. □

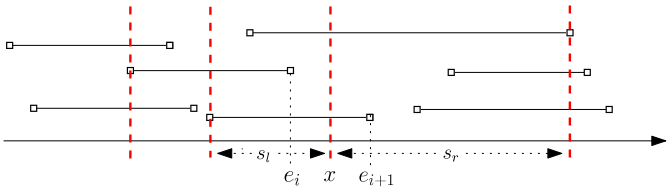

Fig. 2. Illustrating an example for the proof of Lemma 1: There are four splitters shown with the (red) dashed vertical segments, and the splitter x is in (e_i, e_{i+1}). s_l and s_r are the two open segments bounded by x. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

For any two points p and q on L, let $\overline{pq}$ be the *open* line segment whose endpoints are p and q (but $\overline{pq}$ does not include its endpoints). Recall that $\mathcal{I}(\overline{pq})$ is the set of intervals of $\mathcal{I}$ that intersect $\overline{pq}$, and $C(\overline{pq})$ is the sum of the weights of the intervals in $\mathcal{I}(\overline{pq})$.

From now on, we let E also include the two infinite points $-\infty$ and $+\infty$ on L. Let S_E consists of all values $C(\overline{pq})$ for any two points p and q in E. Lemma 1 implies the following corollary.

Corollary 1. $C_{opt} \in S_E$.

Proof. By Lemma 1, there is an optimal solution in which the set X of splitters is a subset of E. Hence, $C_{opt} = C(X)$. The splitters of X partition L into open segments and there must be a segment s_i such that $C(X) = C(s_i)$. Clearly, both endpoints of s_i are in E. By the definition of S_E, $C(s_i) = C_{opt}$ must be in S_E. □

For any value c, if there exists a set X of at most k splitters such that $C(X) \leq c$, then we call c a *feasible value* and call X a *feasible splitter set* with respect to c. For any given value c, the *decision version* of our interval splitting problem is to determine whether c is a feasible value, and if yes, find a feasible splitter set. For differentiation, we refer to our original interval splitting problem as the *optimization version*.

In the sequel, we will first present our algorithm for the decision problem in Section 3 and then solve the optimization problem in Section 4.

3. The decision problem

In this section, we solve the decision version of the problem. Our algorithm runs in $O(n)$ time after the points in E are sorted. Note that Le et al. [9] also gave a linear time algorithm (after the points in E are sorted), but their algorithm only works for the special case. Our algorithm solves the general case. In the following, we assume the points of E have been sorted.

Our algorithm uses the greedy approach. Let c be any given value. If c is a feasible value, the algorithm will find from left to right at most k splitters $x_1, x_2, \ldots$, that are feasible for c; otherwise it will report that c is not feasible.

Recall that for any point $x \in L$, $\mathcal{I}(x)$ is the set of intervals of $\mathcal{I}$ each of which contains x in its interior, and $C(x)$ is the sum of the weights of the intervals in $\mathcal{I}(x)$. We first give the following lemma, which will be useful later for proving the correctness of our algorithm.

Lemma 2. *If there is a point q on L with $C(q) > c$, then c is not a feasible value.*

Proof. Assume to the contrary that c is a feasible value. Let $X = \{x_1, x_2, \ldots, x_k\}$ be a feasible splitter set. Thus we have $C(X) \leq c$. Let $x_0 = -\infty$ and $x_{k+1} = +\infty$. Assume q is in $[x_{i-1}, x_i)$ for some index i. Let s_i be the open interval (x_{i-1}, x_i). Depending on whether $q = x_{i-1}$, there are two cases.

1. If $q \neq x_{i-1}$, then $q \in s_i$. Based on their definitions, we have $\mathcal{I}(q) \subseteq \mathcal{I}(s_i)$, and thus $C(q) \leq C(s_i)$. Note that $C(X) \geq C(s_i)$. Since $C(q) > c$, we obtain $C(X) \geq C(s_i) \geq C(q) > c$, which contradicts with that $C(X) \leq c$.
2. If $q = x_{i-1}$, then q is not in s_i. Let q' be a point to the right of q and infinitesimally close to q. Clearly, $q' \in s_i$. Further, it always holds that $C(q') \geq C(q)$. Consequently, we obtain $C(X) \geq C(s_i) \geq C(q') \geq C(q) > c$, which again contradicts with $C(X) \leq c$.

The lemma is thus proved. □

We first describe the main idea of our algorithm and then flesh out the details. The algorithm starts with setting x_0 to $-\infty$ (note that x_0 is not a splitter). Assume x_{i-1} has already been computed for any $1 \leq i \leq k$. Our algorithm sweeps a point x from x_{i-1} to the right as far as possible to find x_i. For any $x > x_{i-1}$, recall that $C(\overline{x_{i-1}x})$ is the sum of the weights of the intervals in $\mathcal{I}$ that intersect the open segment $\overline{x_{i-1}x} = (x_{i-1}, x)$. During the rightward sweeping of x, as long as $C(\overline{x_{i-1}x}) \leq c$, we continue to move x rightwards. But if moving x rightwards will make the value $C(\overline{x_{i-1}x})$ larger than c, then we stop and put the next splitter x_i at the current position of x; if the above situation happens when $x = x_{i-1}$, then we terminate the algorithm and conclude that c is not a feasible solution. If x has moved to the right of all intervals of $\mathcal{I}$, then we terminate the algorithm and conclude that c is feasible value. In addition, if the algorithm has already put k splitters (i.e., $k = i - 1$) but still need to put the next splitter x_{k+1}, then we conclude that c is not a feasible solution and terminate the algorithm. The details on how to implement the algorithm are given below.

Our algorithm will maintain an *invariant* that each splitter (i.e., x_i for $1 \leq i \leq k$) computed by the algorithm is at the left endpoint of an interval of $\mathcal{I}$. Assume x_{i-1} has just been computed and x is at x_{i-1}. In order to compute the value $C(\overline{x_{i-1}x})$ during the rightward sweeping of x, we need to know the value $C(x_{i-1})$. We assume $C(x_{i-1})$ is already known when x is at x_{i-1}. Initially when $i = 1$, we set $x_{i-1} = -\infty$ and $C(-\infty) = 0$. Further, during the sweeping of x, we will maintain the value $C(x)$, which will be used to compute $C(x_i)$ once the next splitter x_i is determined. We will show that after x_i is determined, x_i is at the left endpoint of an interval and $C(x_i)$ is computed correctly.

During the sweeping of x, an *event* happens when x encounters a point of E. Suppose we have just computed x_{i-1}. For the case where $i \geq 2$, before we sweep x rightwards, we first process this *beginning event* for $x = x_{i-1}$ as follows.

For any interval $I \in \mathcal{I}$, we use $w(I)$ to denote its weight.

Since $i \geq 2$, by our algorithm invariant, x_{i-1} is the left endpoint of an interval, denoted by I. Also recall that $C(x_{i-1})$ is known. If $C(x_{i-1}) + w(I) > c$, then we conclude that c is not a feasible solution and terminate the algorithm. The correctness is proved in the following lemma.

Lemma 3. *If $C(x_{i-1}) + w(I) > c$, then c is not a feasible value.*

Proof. Consider any point q to the right of x_{i-1} and infinitesimally close to x_{i-1}. Since x_{i-1} is the left endpoint of I, it holds that $\mathcal{I}(q) = \mathcal{I}(x_{i+1}) \cup \{I\}$, and thus, $C(q) = C(x_{i-1}) + w(I)$. It follows that $C(q) > c$. By Lemma 2, c is not a feasible value. □

If $C(x_{i-1}) + w(I) \leq c$, then we are "safe" to move x rightwards. We also set $C(\overline{x_{i-1}x}) = C(x) = C(x_{i-1}) + w(I)$. One can verify that the above values are correct when x is moving rightwards before the next event happens. This finishes our processing on the beginning event for $x = x_{i-1}$.

Below, we discuss the general events after the beginning event. Suppose the next event is at a point e in E (with $x_{i-1} < e < +\infty$) and assume that $C(x)$ and $C(\overline{x_{i-1}x})$ have been correctly maintained for x right before x arrives at e. We process the event e as follows. Let I be the interval for which e is its endpoint. Depending on whether e is the right or left endpoint of I, there are two cases.

1. If e is the right endpoint of I, then we update $C(x)$ by setting $C(x) = C(x) - w(I)$ and continue to move x rightwards and proceed on the next event after e. Note that we do not need to change the value $C(\overline{x_{i-1}x})$.

   If e is the rightmost point of E, then we terminate the algorithm and conclude that c is a feasible value, and the set of $i-1$ splitters $x_1, x_2, \ldots, x_{i-1}$ that have been computed so far is a feasible splitter set.
2. If e is the left endpoint of I, then we first check whether $C(\overline{x_{i-1}x}) + w(I) \leq c$. If yes, we set $C(x) = C(x) + w(I)$ and $C(\overline{x_{i-1}x}) = C(\overline{x_{i-1}x}) + w(I)$, and continue to move x rightward and proceed on the next event after e.

   If $C(\overline{x_{i-1}x}) + w(I) > c$, we need to put the next splitter x_i at e. But if $i = k+1$, then we terminate the algorithm and conclude that c is not a feasible value because we are only allowed to have k splitters. If $i < k+1$, then we let $x_i = e$ and proceed on finding the next splitter x_{i+1}. Note that x_i is at the left endpoint of I, which maintains the algorithm invariant. Also, it is easy to see that $C(x_i) = C(x)$.

This finishes the description of our algorithm. For the running time, since the points of E have already been sorted, after processing each event, we can find the next event point in constant time. Also, processing each event takes only constant time. Hence, the total time of the algorithm is $O(n)$. The correctness of the algorithm can be seen from Lemma 3 as well as the fact that our algorithm always tries to push the splitters rightward on L as far as possible.

As a summary, we have the following result.

Theorem 1. *Suppose the endpoints of all intervals in $\mathcal{I}$ have been sorted. The decision version of the interval splitting problem can be solved in $O(n)$ time.*

4. The optimization problem

In this section, we solve the optimization version of the interval splitting problem, with the help of Corollary 1 and Theorem 1. In the following, we refer to our algorithm for the decision problem in Theorem 1 as the *decision algorithm*.

Recall that C_{opt} is the optimal objective value. If we know the value C_{opt}, then we can compute an optimal solution by using our decision algorithm. Specifically, we apply our decision algorithm on $c = C_{opt}$, and the algorithm will find a feasible splitter set, which is an optimal solution. Hence, to solve the optimization problem, the key is to compute C_{opt}, which is our focus below.

Note that in the special case where the weight of each interval of $\mathcal{I}$ is 1, an easy observation is that C_{opt} must be an integer in $[1, n]$. Thus, using the decision algorithm, we can easily compute C_{opt} in $O(n \log n)$ time by doing binary search on the integer sequence from 1 to n. This is exactly the approach used in [9] (by using their own decision algorithm, which works only for the special case). In our general problem, however, this approach does not work because C_{opt} may not be an integer. We propose a new approach, as follows.

4.1. Computing the optimal objective value C_{opt}

Recall that the set S_E consists of all values $C(\overline{pq})$ for any two points p and q in E. By Corollary 1, we have $C_{opt} \in S_E$. One straightforward way to compute C_{opt} is to first compute all values in the set S_E and sort them. Then, using our decision algorithm in Theorem 1, we can compute C_{opt} by doing binary search on the sorted list of the values in S_E. However, since $|S_E| = \Theta(n^2)$, this approach takes $\Omega(n^2)$ time. In the following, we give an $O(n \log n)$ time algorithm.

Recall that E also includes $-\infty$ and $+\infty$. We first organize the values in S_E into $O(n)$ sorted arrays and each array has $O(n)$ elements. Note that our algorithm does not do this organization explicitly.

Let $e_0, e_1, \ldots, e_{2n+1}$ be the list of the values of E sorted on L from left to right, with $e_0 = -\infty$ and $e_{2n+1} = +\infty$. For any i and j with $0 \leq i < j \leq 2n+1$, define $w(i, j) = C(\overline{e_i e_j})$. Clearly, $S_E = \{w(i, j) \mid 0 \leq i < j \leq 2n+1\}$. Below is a self-evident observation that shows a monotonicity property of $w(i, j)$.

Observation 1. *For any i, if $i < j_1 \leq j_2$, then $w(i, j_1) \leq w(i, j_2)$.*

For each $i = 0, 1, \ldots, 2n+1$, we define an array $A_i[0 \cdots 2n+1]$ of $2n+2$ elements as follows. For each j with $0 \leq j \leq 2n+1$, define $A_i[j]$ to be $w(i, j)$ if $i < j$ and 0 otherwise. By Observation 1, elements in each array A_i are sorted in ascending order. It is not difficult to see that S_E is the union of all elements in the arrays A_i, $0 \leq i \leq 2n+1$, i.e., $S_E = \bigcup_{i=0}^{2n+1} A_i$.

Since $C_{opt} \in S_E$, our goal is to find C_{opt} in $\bigcup_{i=0}^{2n+1} A_i$. To this end, although we cannot afford to explicitly compute all elements of these arrays, based on the following Lemma 4, with linear time preprocessing, we can obtain any element of these arrays in constant time whenever we need it.

Lemma 4. *With $O(n)$ time preprocessing, for any query (i, j) with $i < j$, we can compute the value $w(i, j) = A_i[j]$ in constant time.*

Before proving Lemma 4, we show how to compute C_{opt} with the help of Lemma 4. We use a technique, called *binary search on sorted arrays*, which was developed in [1]. We first briefly discuss this technique.

Assume there is a "black-box" decision procedure σ available such that given any value α, σ can report whether α is a feasible value in $O(T)$ time, and further, if α is a feasible value, then any value larger than α is also a feasible value. Given a set of M arrays B_i, $1 \leq i \leq M$, each containing N elements in sorted order, the goal is to find the smallest feasible value δ in $\bigcup_{i=1}^{M} B_i$. Suppose given its indices, any element of these arrays can be obtained in constant time. An algorithm is presented in [1] with the following result.

Lemma 5 ([1])**.** *The smallest feasible value δ in $\bigcup_{i=1}^{M} B_i$ can be found in $O((M+T)\log(MN))$ time.*

For solving our problem, we can use the above result to find C_{opt} in $\bigcup_{i=0}^{2n+1} A_i$ as follows. The following observation is self-evident.

Observation 2. *If a value c is a feasible value for the decision problem, then any value larger than c is also a feasible value.*

Hence, C_{opt} is the smallest feasible value in $\bigcup_{i=0}^{2n+1} A_i$. Our linear time decision algorithm in Theorem 1 can play the role of the black-box σ with $T = O(n)$. Further, we have already shown that given any i and j, we can compute the element $A_i[j]$ in constant time. Therefore, we can apply the technique in Lemma 5 (with $M = N = 2n+2$ and $T = O(n)$) to compute C_{opt} in $O(n \log n)$ time.

In summary, we have the following result.

Theorem 2. *The optimization version of the interval splitting problem can be solved in $O(n \log n)$ time.*

4.2. Proving Lemma 4

It remains to prove Lemma 4. Consider any query (i, j) with $i < j$. Our goal is to compute $w(i, j) = C(\overline{e_i e_j})$. We begin with some observations.

Recall that $e_0, e_1, \ldots, e_{2n+1}$ are the sorted list of points of E. For each t with $0 \le t \le 2n+1$, define $\mathcal{I}_t$ to be the set of intervals of $\mathcal{I}$ whose left endpoints are strictly to the left of e_t. Recall that for any point x on L, $\mathcal{I}(x)$ is the set of intervals of $\mathcal{I}$ each of which contains x in its interior. Also recall that $\mathcal{I}(\overline{e_i e_j})$ is the set of intervals of $\mathcal{I}$ that intersect the open segment $\overline{e_i e_j}$. We have the following lemma.

Lemma 6. *$\mathcal{I}(\overline{e_i e_j}) = \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$ and $\mathcal{I}(e_i) \cap (\mathcal{I}_j \setminus \mathcal{I}_i) = \emptyset$.*

Proof. We first prove $\mathcal{I}(\overline{e_i e_j}) = \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$. To this end, we show below that any interval in $\mathcal{I}(\overline{e_i e_j})$ must be in $\mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$, and vice versa.

1. Consider any interval $I \in \mathcal{I}(\overline{e_i e_j})$. We prove that I must be in $\mathcal{I}(e_i) \cup \mathcal{I}_j \setminus \mathcal{I}_i$.
   Let l and r be the left and right endpoints of I, respectively. By definition, I intersects the open segment $\overline{e_i e_j}$. Hence, $l < e_j$, implying that $I \in \mathcal{I}_j$. If $I \notin \mathcal{I}_i$, it is vacuously true that $I \in \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$. Otherwise, it must be that $l < e_i$. Since I intersects the open segment $\overline{e_i e_j}$, we can also get $r > e_i$. Therefore, it holds that $l < e_i < r$, implying that e_i is contained in the interior of I, and thus $I \in \mathcal{I}(e_i)$.
   Therefore, in any case, we obtain $I \in \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$.
2. Consider any interval $I \in \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$. We prove that I must be in $\mathcal{I}(\overline{e_i e_j})$. Let l and r be the left and right endpoints of I, respectively.
   If $I \in \mathcal{I}_j \setminus \mathcal{I}_i$, then due to $I \in \mathcal{I}_j$, we obtain $l < e_j$, and due to $I \notin \mathcal{I}_i$, we obtain $e_i \le l$. Hence, we have $e_i \le l < e_j$. Since $l < r$, I must intersect the open segment $\overline{e_i e_j}$, and thus $I \in \mathcal{I}(\overline{e_i e_j})$.
   If $I \notin \mathcal{I}_j \setminus \mathcal{I}_i$, then I must be in $\mathcal{I}(e_i)$, implying that $e_i \in (l, r)$. Therefore, I must intersect the open segment $\overline{e_i e_j}$, and $I \in \mathcal{I}(\overline{e_i e_j})$.

The above proves that $\mathcal{I}(\overline{e_i e_j}) = \mathcal{I}(e_i) \cup (\mathcal{I}_j \setminus \mathcal{I}_i)$.

Next, we show that $\mathcal{I}(e_i) \cap (\mathcal{I}_j \setminus \mathcal{I}_i) = \emptyset$. Indeed, for any interval $I = [l, r] \in \mathcal{I}_j \setminus \mathcal{I}_i$, as discussed above, it holds that $e_i \le l < e_j$, implying that e_i cannot be in the interior of I, and thus, $I \notin \mathcal{I}(e_i)$. On the other hand, for any interval $I = [l, r] \in \mathcal{I}(e_i)$, since e_i is in the interior of I, we have $l < e_i$; thus, I must be in $\mathcal{I}_i$, implying that I cannot be in $\mathcal{I}_j \setminus \mathcal{I}_i$.

The lemma thus follows. □

The preceding lemma implies the following approach for computing the value $C(\overline{e_i e_j})$. For each t with $0 \le t \le 2n+1$, let C_t be the sum of the weights of the intervals in $\mathcal{I}_t$. By Lemma 6, we can obtain $C(\overline{e_i e_j}) = C(e_i) + (C_j - C_i)$. Hence, if the values $C(e_i)$, C_j, and C_i are already known, we can compute $C(\overline{e_i e_j})$ in constant time. In the sequel, we present an algorithm that can compute $C(e_t)$ and C_t for all $t = 0, 1, \ldots, 2n+1$ in $O(n)$ time. The algorithm is similar to our decision algorithm in Section 3 (the decision algorithm can compute $C(e_t)$, but here we also need to compute C_t).

The algorithm sweeps a point x from $-\infty$ to $+\infty$. An event happens when x encounters a point, say, e_t, in E, and for processing the event, we will compute $C(e_t)$ and C_t. During the sweeping of x, we will maintain two values for x: $C(x)$, i.e., the sum of the weights of the intervals of $\mathcal{I}$ that contain x in their interior, and $C'(x)$, which is the sum of the weights of the intervals whose left endpoints are strictly to the left of x.

Initially, when $x = -\infty$, we have $C(x) = C'(x) = 0$. Consider a general step that the next event is at e_t. We assume that the values $C(x)$ and $C'(x)$ have been correctly maintained right before x arrives at e_t. Note that e_t is an endpoint of an interval of $\mathcal{I}$, and let I denote the interval (and let $w(I)$ be the weight of I). Depending on whether e_t is the left or the right endpoint of e_t, there are two cases.

If e_t is the right endpoint of I, we first set $C(e_t) = C(x) - w(I)$ and $C_t = C'(x)$. Then we update $C(x) = C(x) - w(I)$, and we do not need to change $C'(x)$. One can verify that all these values have been correctly computed. We then proceed on the next event after e_t.

If e_t is the left endpoint of I, then we set $C(e_t) = C(x)$ and $C_t = C'(x)$. We also update $C(x) = C(x) + w(I)$ and $C'(x) = C'(x) + w(I)$ because once x crosses e_t, e_t is strictly to the left of x. We proceed on the next event after e_t.

The algorithm is done once x passes the rightmost point of E. Since the points of E have already been sorted, the algorithm runs in $O(n)$ time.

As a summary, in $O(n)$ time we can compute $C(e_t)$ and C_t for all $t = 0, 1, \ldots, 2n+1$, after which, given any query (i, j) with $i < j$, we can compute $w(i, j)$ in constant time. Lemma 4 is thus proved.

Acknowledgments

Wei Cao and Jian Li were supported in part by the National Basic Research Program of China grants 2015CB358700, 2011CBA00300, 2011CBA00301, and the National NSFC grants 61202009, 61033001, 61361136003. Shimin Li and Haitao Wang were supported in part by National Science Foundation under Grant CCF-1317143. The authors wish to thank an anonymous reviewer for the detailed comments that helped improve the presentation of the paper.

References

[1] D. Chen, Y. Gu, J. Li, H. Wang, Algorithms on minimizing the maximum sensor movement for barrier coverage of a linear domain, Discrete Comput. Geom. 50 (2013) 374–408.
[2] J. Chuzhoy, R. Ostrovsky, Y. Rabani, Approximation algorithms for the job interval selection problem and related scheduling problems, Math. Oper. Res. 31 (2006) 730–738.
[3] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, third ed., MIT Press, 2009.
[4] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry—Algorithms and Applications, third ed., Springer-Verlag, Berlin, 2008.
[5] M. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.
[6] J.M. Keil, On the complexity of scheduling tasks with discrete starting times, Oper. Res. Lett. 12 (1992) 293–295.
[7] S. Khanna, S. Muthukrishnan, S. Skiena, Efficient array partitioning, in: Proc. of the 24th International Colloquium on Automata, Languages, and Programming, ICALP, 1997, pp. 616–626.
[8] J. Kleinberg, E. Tardos, Algorithm Design, Addison-Wesley, Boston, MA, USA, 2005.
[9] W. Le, F. Li, Y. Tao, R. Christensen, Optimal splitters for temporal and multi-version databases, in: Proc. of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 109–120.
[10] S. Muthukrishnan, T. Suel, Approximation algorithms for array partitioning problems, J. Algorithms 54 (2005) 85–104.
[11] K. Nakajima, S. Hakimi, Complexity results for scheduling tasks with discrete starting times, J. Algorithms 3 (1982) 344–361.
[12] K. Ross, J. Cieslewicz, Optimal splitters for database partitioning with size bounds, in: Proc. of the 12th International Conference on Database Theory, ICDT, 2009, pp. 98–110.
[13] F. Spieksma, On the approximability of an interval scheduling problem, J. Sched. 2 (1999) 215–227.