Information and Computation 154, 1–33 (1999)
Article ID inco.1999.2811, available online at http://www.idealibrary.com on IDEAL®

Efficient Splitting and Merging Algorithms for Order Decomposable Problems[1]

Roberto Grossi[2]

Dipartimento di Sistemi e Informatica, Università di Firenze, via Lombroso 6/17, 50134 Florence, Italy

and

Giuseppe F. Italiano[3]

Dipartimento di Matematica Applicata ed Informatica, Università "Ca' Foscari" di Venezia, via Torino 155, 30173 Venice Mestre, Italy

Let S be a set whose items are sorted with respect to $d > 1$ total orders $\prec_1, \ldots, \prec_d$, and which is subject to dynamic operations, such as insertions of a single item, deletions of a single item, split and concatenate operations performed according to any chosen order $\prec_i$ $(1 \leqslant i \leqslant d)$. This generalizes to dimension $d > 1$ the notion of concatenable data structures, such as the 2-3-trees, which support splits and concatenates under a single total order. The main contribution of this paper is a general and novel technique for solving order decomposable problems on S which yields new and efficient concatenable data structures for dimension $d > 1$. By using our technique we maintain S with the time bounds: $O(\log n)$ for the insertion or the deletion of a single item, where n is the number of items currently in S; $n^{1-1/d}$ for splits and concatenates along any order, and for rectangular range queries. The space required is $O(n)$. We provide several applications of our technique. Namely, we present new multidimensional data structures implementing two-dimensional priority queues, two-dimensional search trees, and concatenable interval trees;

[1] A preliminary version of this paper appears in [14].

[2] Part of this work was done while visiting ICSI, Berkeley. Current address: Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. E-mail: grossi@di.unipi.it.

[3] Work supported in part by the Commission of the European Communities under ESPRIT LTR Project 20244 (ALCOM-IT), by a Research Grant from University of Venice "Ca' Foscari," by the Italian MURST Project "Efficienza di Algoritmi e Progetto di Strutture Informative," and by the German-Italian Program "Vigoni 1997." Part of this work was done while at University of Salerno and while visiting ICSI, Berkeley. Current address: Dipartimento di Informatica, Sistemi e Produzione, via di Tor Vergata 110, 00133 Roma, Italy. E-mail: italiano@info.uniroma2.it.

0890-5401/99 $30.00

these data structures allow us to improve many previously known results on decomposable problems under split and concatenate operations, such as membership query, minimum-weight item, range query, convex hulls, and Voronoi diagrams.

1. INTRODUCTION

Let $\mathscr{P}$ be a searching problem defined on an input set S with n items, and let $\mathscr{P}(x, S)$ denote its solution for a query item x. Problem $\mathscr{P}$ is decomposable if we can find an answer to query $\mathscr{P}(x, S)$ by first partitioning set $S = S' \cup S''$ and computing the answers to queries $\mathscr{P}(x, S')$ and $\mathscr{P}(x, S'')$ recursively, and then combining them through a suitable operator $\diamond$. Formally, $\mathscr{P}$ is said to be *$f(n)$-decomposable* if and only if $\mathscr{P}(x, S) = \diamond(\mathscr{P}(x, S'), \mathscr{P}(x, S''))$ for any partition $S = S' \cup S''$ and any query item x, where $\diamond$ is an operator whose computation requires $O(f(n))$ time. Throughout this paper, we assume that $\diamond$ is associative and commutative. Furthermore, we assume that $f(n)$ is *smooth*, namely $f(\Theta(n)) = \Theta(f(n))$, and that $f(n)$ is *strongly nondecreasing*; i.e., (i) $f(n)$ is nondecreasing, and (ii) if $f(n) = \Omega(n)$ then $f(n)/n$ is nondecreasing as well. Most of the "natural" functions that we know of are smooth and strongly nondecreasing. Some examples of $O(1)$-decomposable searching problems include: membership queries to test the existence of an item, where $\diamond$ is the logical-or function; closest point queries to find the nearest item to an input item, where $\diamond$ is the minimal distance; range queries to report the items lying in the range specified according to a linear order, where $\diamond$ is the list append operation. Convex hull searching is not decomposable as the fact that a point $x \in S$ belongs to the convex hull of S' or S'' does not necessarily imply that x belongs to the convex hull of $S = S' \cup S''$. Since the definition of decomposable *search* problems can be extended also to the decomposable *set* problems in which the query item is not specified (e.g., finding the minimum-weight item, where $\diamond$ is the minimum), we shall denote a generic solution to a decomposable problem $\mathscr{P}$ by $\mathscr{P}(S)$.

Let us introduce $d > 1$ total orders $\prec_1, \ldots, \prec_d$ defined on S, and let $\prec_i$ be a given total order, $1 \leqslant i \leqslant d$. A problem $\mathscr{P}$ is *$f(n)$-order* decomposable with respect to total order $\prec_i$ if $\mathscr{P}(S) = \diamond(\mathscr{P}(S'), \mathscr{P}(S''))$ for any *ordered partition* $S = S' \cup S''$ (i.e., $x' \prec_i x''$ for all $x' \in S'$ and $x'' \in S''$), where operator $\diamond$ takes $O(f(n))$ time. Problem $\mathscr{P}$ is *$f(n)$-order decomposable* if it is $f(n)$-order decomposable with respect to any total order $\prec_i$, $1 \leqslant i \leqslant d$. For example, performing multidimensional range queries is $O(1)$-order decomposable, convex hull searching is $O(\log n)$-order decomposable, and computing Voronoi diagrams is $O(n)$-order decomposable. Many other examples of order decomposable problems can be found in basic data structures, computational geometry, database applications, and statistics [8, 26, 30]. In the *static* case, there is an immediate divide-and-conquer algorithm for these problems on a sorted set S: split it into two equal parts S' and S'', solve the problem recursively on S' and S'', and combine their solutions in $O(f(n))$ time.

In this paper, we present a general technique for handling a *dynamic* set S with d total orders, for constant $d > 1$, under insertions of a single item, deletions of a single item, and rearrangements of any of the total orders $\prec_1, \ldots, \prec_d$ on S by

means of split and concatenate operations. Our queries involve finding the solution $\mathscr{P}(R)$ for only the items in the subset $R \subseteq S$ identified by some ranges in the orders $\prec_1, ..., \prec_d$. More formally, we introduce the following *multiordered set splitting and merging problem*:

split$(S, z, \prec_i)$ Split S into S' and S'' according to item z and the specified total order $\prec_i$ $(1 \leqslant i \leqslant d)$. That is, $x' \prec_i z$ and $z \prec_i x''$ for all $x' \in S'$ and $x'' \in S''$. S is no longer available after this operation.

concatenate$(S', S'', \prec_i', \prec_i'')$ Combine S' and S'' together according to their respective ith total orders $\prec_i'$ and $\prec_i''$ $(1 \leqslant i \leqslant d)$ into a new set $S = S' \cup S''$. The items in the resulting set S undergo the new order $\prec_i$, obtained by concatenating $\prec_i'$ and $\prec_i''$. That is, $x \prec_i y$ in S if and only if one of three conditions holds: (a) $x \prec_i' y$ and $x, y \in S'$; (b) $x \prec_i'' y$ and $x, y \in S''$; (c) $x \in S'$ and $y \in S''$. S' and S'' are no longer available after this operation.

insert(z, S) Insert item z into set S according to all orders $\prec_1, ..., \prec_d$.

delete(z, S) Delete item z from set S.

member(z, S) Check if item z belongs to set S.

range$(\langle a_1, b_1\rangle, ..., \langle a_d, b_d\rangle, S)$ Let $R = \{z \in S: a_i \prec_i z \prec_i b_i \text{ for } 1 \leqslant i \leqslant d\}$ be the region of items in S delimited by pairs $\langle a_1, b_1\rangle, ..., \langle a_d, b_d\rangle$. Find the solution $\mathscr{P}(R)$ to problem $\mathscr{P}$ restricted to the items in region R only.

We will not discuss *member*, as we can execute it with some of the instructions implementing *delete* at no cost increase. For $d = 1$, the recursive nature of order decomposable problems gives us an immediate tree structure: leaves correspond to the items of S, sorted according to $\prec_1$, and each internal node stores the solution to problem $\mathscr{P}$ restricted to the leaves descendent of that node. By using a 2-3-tree [2], each of the above operations can be simply implemented in $O(f(n) \log n)$ time, with $O(f(n))$ time per tree node. Maintaining $d > 1$ total orders on the same set S, while splitting or merging each order independently of the others, makes things much more complicated than this simple case. In the case of two or more different orders, indeed, there are some technical difficulties, which are mainly due to the interplay among different orders.

Related work. There is a great deal of work on decomposable searching problems. They were first introduced by Bentley [6] for dynamizing static data structures. The initial goal was to support insertions with low amortized times, without affecting much of the query efficiency. Other dynamization techniques were then introduced in [8, 27, 33]. Handling deletions in the dynamization of special subclasses of decomposable search problems was discussed in [20, 22, 23, 34]. The main idea behind these techniques was to partition a big data structure into a collection of small data structures, called *blocks*. An insertion or a deletion requires to rebuilding a few blocks and a query scans all the blocks in order to combine their solutions by means of operator $\diamond$. Many blocks yield fast updates and slow queries, whereas few blocks yield slow updates and fast queries. Two methods are employed to tune properly the number of blocks and obtain a good trade-off

between queries and updates: in the *equal block method* [20–23], the blocks have almost the same size and the best trade-off is between queries and deletions; in the *logarithmic method* [6, 8, 33, 34], the blocks are of exponentially increasing size and the best trade-off is between queries and insertions. Some lower bounds on the efficiency of the best possible tradeoff were given in [8, 25]. Optimal solutions were obtained by combining the equal block and the logarithmic method by means of the amortized solution in [27] and by the global rebuilding technique yielding worst-case bounds in [32, 35]. The notion of order decomposable was then introduced in [29] by generalizing the results of [31] and was independently presented in [16]. In particular, the work in [29] highlights the connection between order-decomposable problems and divide-and-conquer paradigms in the corresponding dynamic data structures. Processing a batch of insertions, deletions, and queries to be performed on an initially empty set was treated in the offline model [12, 41] and in the semi-online model [10, 41], where the deletion times are (partially) known. A different kind of characterization of "deletion"-decomposable problems was discussed in [37]. For a more detailed discussion of results on decomposable problems, we refer the interested reader to the book [30] and to the survey in [9].

Solving an order-decomposable problem only for the items contained in an input rectangular region can be done by means of range queries on quad-trees [13] and k-d trees [5]. These data structures were originally designed to support some operations for windowing problems in computer graphics, but it was difficult to keep them balanced (e.g., see [36, 38]). Many other elegant data structures for range queries were devised subsequently and we refer the reader to [9] for a comprehensive survey on this topic and a list of references. In particular, the solutions in [39, 42] combined decomposable problems and range queries together in order to add some range restrictions to the data structures supporting insertions and deletions. Split and concatenate operations were subsequently introduced in [17, 19] for a set of multidimensional points in addition to the standard operations: range queries, insertions and deletions. Specifically, divided k-d trees were presented in [17] for a set of n items, allowing for a range, a split, or a concatenate operation in $O(n^{1-1/d}\log^{1/d} n)$ time and for an insertion or a deletion in $O(\log n)$ time with $O(n)$ space. In [19], a general technique, based on the ordered equal block method, was described for solving order-decomposable problems and producing efficient concatenable data structures in $O(n)$ space. The following time bounds were obtained for a split or concatenate: $O(\sqrt{n}\log n)$ in concatenable interval trees, $O(n^{1-1/d}\log n)$ in d-dimensional 2-3-trees and $O(\sqrt{n\log n}\log n)$ in a data structure for convex hulls. The bound for insertions and deletions of items is $O(\log n)$ amortized, except for the $O(\log^2 n)$ amortized bound in the data structure for convex hulls. The range query bounds equal the split/concatenate cost plus an output sensitive cost $O(occ)$, where occ is the size of the output reported by the query. Although the range queries in [39, 42] are faster than the ones in [19], the solutions in [19] require less space and can be used to obtain an efficient dynamic version of static data structures.

Our results. In this paper, we present a novel and general technique for solving order decomposable problems on S under insertions, deletions, splits, concatenates,

and range queries, yielding new and efficient concatenable data structures for dimension $d > 1$. All these data structures are based on a new multidimensional data structure, which we call the *cross-tree*. Differently from the approach of [19], our technique is based more on simple geometric properties, rather than on underlying sophisticated data structures, and exploits the fact that some data structures can be built on sorted items more efficiently. By using our technique we maintain a set S of n items in $O(n)$ space with the following *worst-case* time bounds: $O(\log n)$ for the insertion or the deletion of a single item, and $O(n^{1-1/d})$ for splits and concatenates along any order. We use this new technique in a simple way for a wide range of applications to shave some log factors from the best known bounds [17, 19]. We obtain new multidimensional data structures implementing two-dimensional priority queues, two-dimensional search trees, and concatenable interval trees. We achieve the following time bounds for a split or concatenate: $O(\sqrt{n})$ in concatenable interval trees and $O(\sqrt{n \log n})$ for a variant of theirs, treating also the length of the intervals, $O(n^{1-1/d})$ in d-dimensional 2-3-trees (or divided k-d trees), and $O(\sqrt{n \log n})$ in a data structure for the convex hull. As a result, we improve the query bounds because they are equal to the split/concatenate cost plus an $O(occ)$ cost due to the output. Furthermore, we make the bounds for insertions and deletions of a single item worst-case, rather than amortized. The new data structures work for many other order-decomposable problems under split and concatenate operations. For example, point insertions and deletions in a planar Voronoi diagram of n points take $O(n)$ time in $O(n \log \log n)$ space [30] (a result in [1] is a semidynamic algorithm with $O(n)$ deletion time and space). We obtain an $O(n)$ cost also for range, split and concatenate operations in $O(n \log \log n)$ space (the techniques in [19, 39, 42] require more time or space). This partially solves a problem posed in [1] (i.e., given the Voronoi diagram for a set S of n points, compute the Voronoi diagram for any given subset $R \subseteq S$ in $O(n)$ time) for the special case in which R is a window defined by range queries on a dynamic set S. Splits and concatenates are useful and alternative operations to slide window R over the items in S, instead of performing many single insertions and deletions of items. Furthermore, our technique for order-decomposable problems is suitable for efficient external memory algorithms [15].

The remainder of this paper is organized as follows. In Section 2 we describe our technique for the case $d = 2$ and a single set S. In Section 3 we list some applications of this technique to concatenable data structures. General order decomposable problems of dimension $d > 2$ are considered in Section 4. Finally, in Section 5 we list some open problems and concluding remarks.

2. THE SPLITTING AND MERGING TECHNIQUE

In this section, we describe our general technique to maintain $d = 2$ total orders, which we denote by $\prec_X$ and $\prec_Y$, under split and concatenate operations. Let n be the number of items currently in S. Each item $z \in S$ is associated with a point $(X(z), Y(z))$ in the Cartesian plane, such that $X(z)$ is the rank of z in S with respect to order $\prec_X$ and $Y(z)$ is the rank of z in S with respect to $\prec_Y$, with ties broken

arbitrarily. As a result, no two items share the same coordinates. For a given item z, its coordinates $X(z)$ and $Y(z)$ are affected by our dynamic operations and so the mapping from z onto point $(X(z), Y(z))$ changes dynamically throughout the sequence of operations. Starting from n items in S, we obtain n points in the Cartesian plane, which can be stored in the form of a $n \times n$ *sparse* and *dynamic* matrix $\mathcal{M}$. We will use interchangeably both models, namely the Cartesian plane and the matrix notation. The only issue to keep in mind is that the vertical ordering in the Cartesian plane is defined bottom to top, whereas in the matrix notation it is defined top to bottom. We follow well established mathematical traditions and, thus, use the two conventions according to the model used: to switch from one model to the other, it simply suffices to conceptually rotate the points around a horizontal axis. This should not induce any confusion in the reader.

The operations in S can be simulated by a certain number of operations in $\mathcal{M}$. Operation split$(S, z, \prec_X)$ corresponds to splitting matrix $\mathcal{M}$ horizontally at a certain position $X(z)$, which is the rank of z in S with respect to $\prec_X$, while doing the same according to its order $\prec_Y$ is equivalent to handling $\mathcal{M}$ vertically at position $Y(z)$. Concatenating is analogous. Operations insert(z, S) and delete(z, S) require a new operation which sets entry $\mathcal{M}[X(z), Y(z)]$ to item z or to an empty value, respectively. Finally, solving problem $\mathcal{P}$ in the region specified by range$(\langle a_X, b_X\rangle, \langle a_Y, b_Y\rangle, S)$ can be done by solving $\mathcal{P}$ for the points contained in the rectangular part of $\mathcal{M}$ delimited by the ranks of a_X, b_X, a_Y, b_Y in their corresponding order. Based upon the above reduction, we state our multiordered set splitting and merging problem by using our sparse matrix $\mathcal{M}$, which we would like to maintain under splits, concatenates, and query operations related to problem $\mathcal{P}$. More formally for any integers h_1, h_2, v_1, v_2, such that $1 \leqslant h_1 \leqslant h_2 \leqslant n$ and $1 \leqslant v_1 \leqslant v_2 \leqslant n$, we use $\mathcal{M}[h_1, h_2; v_1, v_2]$ to denote the submatrix of $\mathcal{M}$ that contains entries $\mathcal{M}[i, j]$ with $h_1 \leqslant i \leqslant h_2$ and $v_1 \leqslant j \leqslant v_2$. We call this submatrix a *region*. We can disassemble and reassemble a single matrix $\mathcal{M}$ in many different ways by using any sequence of the following operations:

h_split$(\mathcal{M}, i)$ Split $\mathcal{M}$ horizontally at row i and obtain two new matrices $\mathcal{M}_1$ and $\mathcal{M}_2$, such that $\mathcal{M}_1 = \mathcal{M}[1, i; 1, n]$ and $\mathcal{M}_2 = \mathcal{M}[i+1, n; 1, n]$. In other words, $\mathcal{M}_1$ is given by the first i rows of $\mathcal{M}$ and $\mathcal{M}_2$ is given by the last $(n-i)$ rows of $\mathcal{M}$. Matrix $\mathcal{M}$ is no longer available after the operation.

h_concatenate$(\mathcal{M}_1, \mathcal{M}_2)$ Let $\mathcal{M}_1$ have size $m_1 \times n$ and $\mathcal{M}_2$ have size $m_2 \times n$. We meld $\mathcal{M}_1$ and $\mathcal{M}_2$ horizontally and produce a matrix $\mathcal{M}$ of size $(m_1 + m_2) \times n$, such that $\mathcal{M}[1, m_1; 1, n] = \mathcal{M}_1$ and $\mathcal{M}[m_1 + 1, m_1 + m_2; 1, n] = \mathcal{M}_2$. In other words, the first m_1 rows of $\mathcal{M}$ are given by $\mathcal{M}_1$ and the last m_2 rows of $\mathcal{M}$ are given by $\mathcal{M}_2$. This operation assumes that $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same number of columns. $\mathcal{M}_1$ and $\mathcal{M}_2$ are no longer available after the operation.

set$(i, j, z, \mathcal{M})$ Update $\mathcal{M}$ by setting $\mathcal{M}[i, j] = z$. This corresponds either to an insertion (if z is nonempty) or to a deletion (if z is empty), and causes the implicit renumbering of the horizontal and vertical rankings.

range$(h_1, h_2, v_1, v_2, \mathcal{M})$ Find the solution $\mathcal{P}(R)$ to problem $\mathcal{P}$ restricted to the nonempty entries contained in region $R = \mathcal{M}[h_1, h_2; v_1, v_2]$.

Operations v_concatenate($\mathcal{M}_1, \mathcal{M}_2$) and v_split($\mathcal{M}, j$) are analogously defined to operate vertically. Our technique works for a general matrix $\mathcal{M}$. However, in the remainder of this paper we discuss the case where each row or column of $\mathcal{M}$ contains a constant number of points. This is without loss of generality, as a row or a column with s points can be represented by a sequence of $\Theta(s)$ columns or rows with a constant number of points; this transformation does not affect the achieved bounds and can be easily maintained throughout our sequence of operations.

2.1. Data Structures

We now describe the data structures for our splitting and merging problem. We need an important notion which will be used throughout the paper. Let $X = \{x_1, x_2, ..., x_q\}$ be a sorted sequence of q elements, according to some total order $\prec$: $x_1 \prec x_2 \prec \cdots \prec x_q$. Let $I_1, ..., I_s$ be a partition of X into adjacent intervals, so that for $1 \leqslant i \leqslant s-1$ all the elements in I_i precede all the elements in I_{i+1}. For $1 \leqslant i \leqslant s$, let $|I_i|$ denote the size of interval I_i, defined as the number of elements in I_i.

DEFINITION 2.1 (Size invariant). Let $k \geqslant 1$ be an integer. The adjacent intervals $I_1, ..., I_s$ satisfy the *size invariant of order k* if two conditions are met:

(a) $|I_i| \leqslant k$ for $1 \leqslant i \leqslant s$;

(b) $|I_i| + |I_{i+1}| > k$ for $1 \leqslant i \leqslant s-1$.

The size invariant of order k in Definition 2.1 implies that the number s of intervals is $O(q/k)$. Moreover, we can easily maintain the size invariant of the adjacent intervals when an element is deleted from X or a new element is inserted into X. If an element x_j is deleted from an interval I_i, then the size of I_i decreases by one: we first check whether after the deletion of x_j condition (b) of Definition 2.1 would be violated for the two pairs of adjacent intervals $\langle I_{i-1}, I_i \rangle$ and $\langle I_i, I_{i+1} \rangle$. If this is the case, it is enough to combine I_i with either of its neighbors, according to condition (b). If a new item x_j is inserted into interval I_i, the size of I_i increases by one: we first check whether after the insertion of x_j condition (a) of Definition 2.1 would be violated for I_i; if this is the case, then I_i contains $k+1$ elements and can be split in two subintervals I_i' and I_i'', with $I_i' \prec I_i''$ so that $|I_i'|, |I_i''| \leqslant (k+1)/2 \leqslant k$. Next, it is sufficient to check whether I_i' can be combined with I_{i-1} and I_i'' can be combined with I_{i+1}, in order to satisfy rule (b). Representing each single interval with a balanced search tree yields that a size invariant of order k can be maintained dynamically in time $O(\log k)$ per operation.

We now turn back to our dynamic matrix $\mathcal{M}$ and refer to its n nonempty entries as the *points* of $\mathcal{M}$. We let k be a slack parameter, where k is an integer with $1 \leqslant k \leqslant n$. We handle the *sparse* $n \times n$ matrix $\mathcal{M}$ *as if* it were a *dense* $\Theta(n/k+k) \times \Theta(n/k+k)$ matrix. We then tune k according to the chosen problem $\mathcal{P}$ and the cost $f(n)$ of operator $\diamond$. We proceed as follows. We group adjacent rows and columns of matrix $\mathcal{M}$ into respectively *horizontal and vertical stripes*, such that the stripes satisfy the size invariant of order k (Definition 2.1), where the size of a

horizontal (resp. vertical) stripe is given by its number of rows (resp. columns). The size invariant guarantees that each stripe contains at most $O(k)$ points and that the total number of horizontal and vertical stripes is $O(n/k)$. The partition into horizontal and vertical stripes induces a partition of $\mathcal{M}$ into $O(n^2/k^2)$ squares, such that each square intersects no more than k rows and k columns. We call these the *basic squares* in $\mathcal{M}$. We maintain the solutions to $\mathcal{P}$ for each single basic square. We also store these solutions in the leaves of a two-dimensional data structure, which we call *cross-tree*, that describes recursively the partition of $\mathcal{M}$ into its basic squares. We then percolate the solutions from the cross-tree leaves toward its internal nodes in a heap-like fashion by means of operator $\diamond$, as problem $\mathcal{P}$ is decomposable.

DEFINITION 2.2. A *cross-tree* $\mathrm{CT}(T_H \times T_V)$ describes a balanced decomposition of a two-dimensional set and is the cross product of two trees T_H and T_V:

- T_H and T_V have the same height and $O(1)$ children per node. Their leaves are on the same level.
- For each pair of nodes $u \in T_H$ and $v \in T_V$ on the same level, there is a node α_{uv} in $\mathrm{CT}(T_H \times T_V)$ with $O(1)$ children (see Fig. 1).
- For each pair of edges $(u, \hat{u}) \in T_H$ and $(v, \hat{v}) \in T_V$, such that u and v are on the same level, there is an edge $(\alpha_{uv}, \alpha_{\hat{u}\hat{v}})$ in $\mathrm{CT}(T_H \times T_V)$.

An example illustrating a cross-tree and its properties is shown in Figs. 2–4. In Figs. 2 and 3, the bottom–up drawing of the cross-tree is shown in three dimensions to highlight the rationale behind its definition, and then its actual representation is shown in Fig. 4. Let us assume for now that each solution takes constant space (we will deal with the case of solutions requiring more space later on). The recursive nature of decomposable problem $\mathcal{P}$ suggests that its solution $\mathcal{P}(S) = \diamond(\mathcal{P}(S'), \mathcal{P}(S''))$ can be stored in the root (e.g., node (g10) in the cross-tree of Fig. 4) and that $\mathcal{P}(S'), \mathcal{P}(S'')$ can be stored recursively in its children (i.e., nodes f8 and f9, Fig. 4). Each internal node, therefore, stores the solution to $\mathcal{P}$ for the

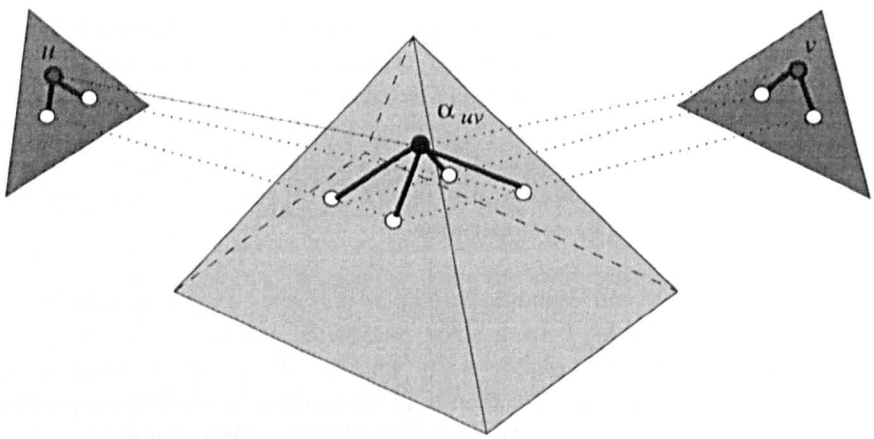

FIG. 1. Illustrating the cross-tree: nodes $u \in T_H$, $v \in T_V$ and $\alpha_{uv} \in \mathrm{CT}(T_H \times T_V)$, and their outgoing edges are shown in boldface.

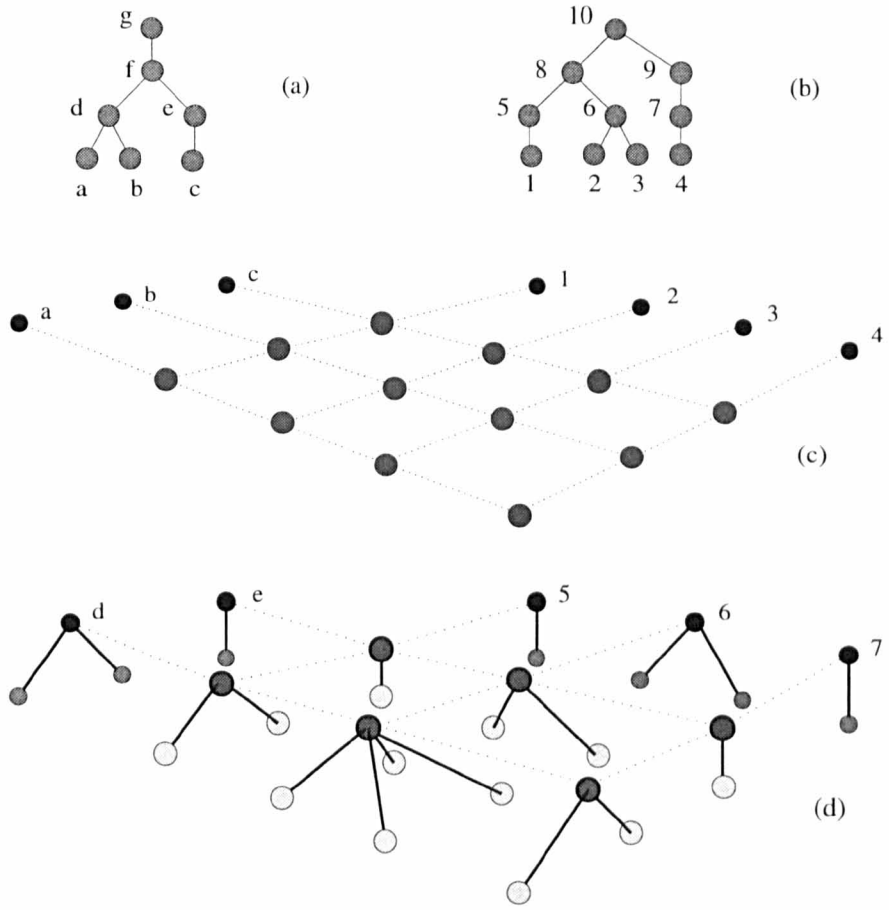

FIG. 2. The bottom–up drawing of cross-tree $\mathrm{CT}(T_H \times T_V)$ for the two trees T_H and T_V shown in (a) and (b), respectively. $\mathrm{CT}(T_H \times T_V)$ is shown in three dimensions to give its intuitive representation, by assuming that T_H is on the xz-plane and T_V is on the yz-plane (the cross-tree leaves are on the xy-plane). The level-0 nodes are shown in (c), and the level-1 nodes are shown in (d).

items in S corresponding to its descendent leaves. We exploit this simple observation in our basic data structure, which indeed comprises matrix $\mathcal{M}$ and cross-tree $\mathrm{CT}(T_H \times T_V)$. We wish to point out that the leaves of T_H and T_V are in one-to-one correspondence to the horizontal and vertical stripes in matrix $\mathcal{M}$, respectively. Consequently, the leaves of $\mathrm{CT}(T_H \times T_V)$ are in one-to-one correspondence to the basic squares in $\mathcal{M}$. As T_H and T_V have $O(n/k)$ leaves, one for each stripe of $\mathcal{M}$, and a total of $O(n/k)$ nodes, the resulting cross-tree $\mathrm{CT}(T_H \times T_V)$ has $O(n^2/k^2)$ leaves, one for each basic square of $\mathcal{M}$, and a total of $O(n^2/k^2)$ nodes. Specifically, our data structure has the following features (see Fig. 5):

1. For each nonempty basic square of $\mathcal{M}$, we keep its points sorted according to a *total order* $\prec_P$ (which is not necessarily equal to $\prec_X$ or $\prec_Y$) by means of a *threaded* binary search tree, whose nodes are linked together in symmetrical order. Searching, inserting, and deleting a point takes $O(\log k)$ time. We can scan the points in a basic square in their $\prec_P$-order and take constant time per scanned

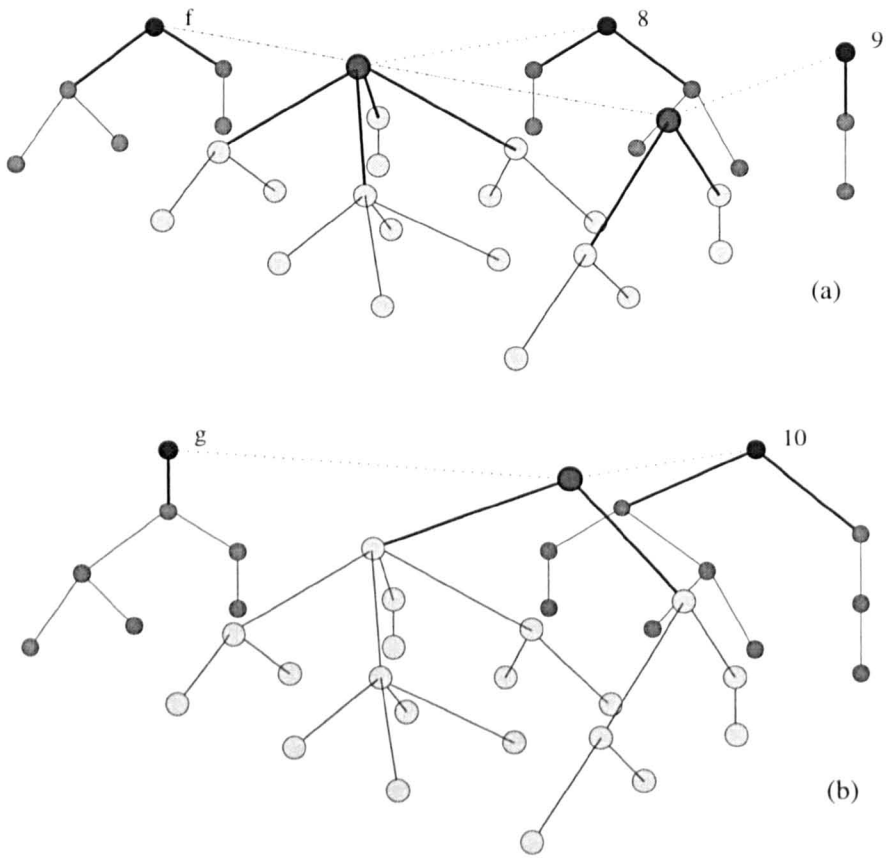

FIG. 3. The bottom–up drawing of cross-tree $CT(T_H \times T_V)$ (continued), where level-2 nodes are shown in (a), and level-3 nodes and the whole cross-tree are shown in (b).

point. It is worth noting that we introduce order $\prec_P$ because some data structures can be (re)built more efficiently on a sorted set of points.

2. Each cross-tree node corresponds to a region of matrix $\mathcal{M}$. The cross-tree leaves correspond to the basic squares. An internal node α corresponds to a *region* $R = \mathcal{M}[h_1, h_2; v_1, v_2]$ and has $O(1)$ children $\alpha_1, ..., \alpha_j$ corresponding to an orthogonal partition of $\mathcal{M}[h_1, h_2; v_1, v_2]$ into smaller regions (if a child is empty, then its corresponding region is empty). For example, let us assume that α has $j = 4$ children: α_1 corresponds to $\mathcal{M}[h_3, h_2; v_1, v_3]$, α_2 corresponds to $\mathcal{M}[h_3, h_2; v_3, v_2]$, α_3 corresponds to $\mathcal{M}[h_1, h_3; v_3, v_2]$ and α_4 corresponds to $\mathcal{M}[h_1, h_3; v_1, v_3]$ for some $h_1 \leqslant h_3 \leqslant h_2$ and some $v_1 \leqslant v_3 \leqslant v_2$. In other words, the smaller (nonoverlapping) regions corresponding to $\alpha_1, ..., \alpha_j$ can be united to produce the region R corresponding to α.

3. For each nonempty basic square of $\mathcal{M}$, we examine its points and store their solution to problem $\mathcal{P}$ into their corresponding cross-tree leaves. We then percolate this information from the leaves towards the cross-tree root in a heap-like fashion. Namely, let α be an internal node of the cross-tree, and let $s_1, ..., s_j$ be the solutions stored in the j children of α, where $j = O(1)$. As $\mathcal{P}$ is a decomposable problem, the

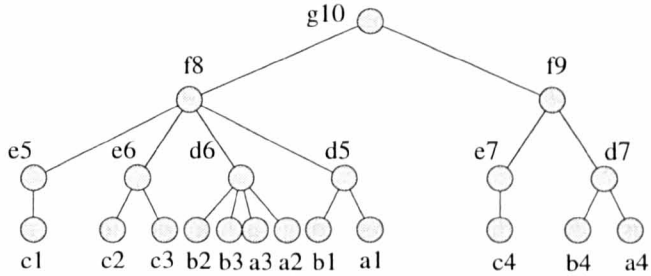

FIG. 4. The bottom–up drawing of cross-tree $CT(T_H \times T_V)$ (continued). The actual representation of the cross-tree is shown, where its nodes α_{uv} are labeled by uv, where $u = a, b, ..., g$ and $v = 1, 2, ..., 10$. Note that the cross-tree part identified by nodes b1, b2, b3, b4, d5, d6, d7, f8, f9, g10 is isomorphic to T_V and corresponds to leaf-to-root path $\Pi = \{b, d, f, g\}$ in T_H, shown in Fig. 2. Analogously, the part identified by a3, b3, c3, d6, e6, f8, g10 is isomorphic to T_H and corresponds to path $\Pi = \{3, 6, 8, 10\}$ in T_V.

internal node α stores its solution $\diamond(s_1, ..., s_j)$ for the points in its corresponding region R (as $\diamond$ is associative, it is well defined also for an arbitrary number of arguments.) This solution is stored in an efficient way, which we will specify later, according to the problem $\mathscr{P}$ considered. Note that if $j = 1$ this is trivial, as $\diamond(s_1) = s_1$.

A comment regarding point 3 is in order at this point. When the solutions require more than constant space each, node α stores *implicitly* its corresponding

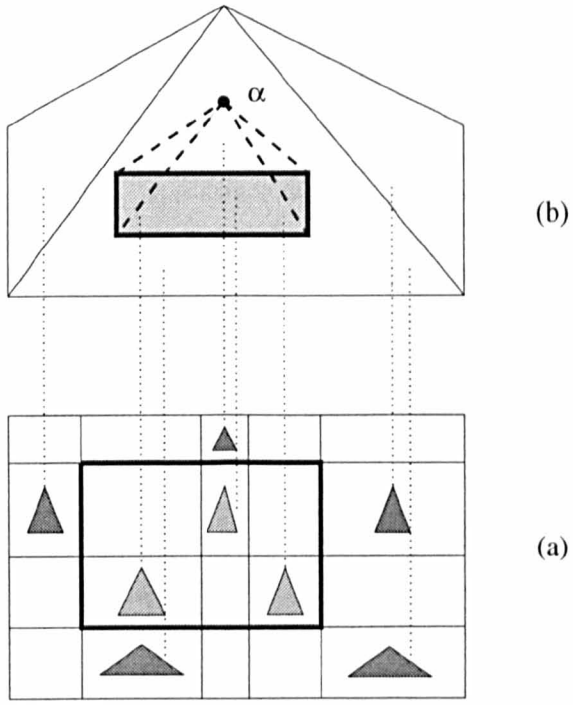

FIG. 5. (a) A partition of matrix $\mathscr{M}$ into its basic squares, where each nonempty square contains a *threaded* binary search tree storing its points in $\prec_p$ order. (b) The cross-tree with its leaves storing the basic squares in (a). The boldface region in (a) corresponds to a cross-tree node α in (b).

solution $\mathscr{P}(R)$. Let us assume without loss of generality that α has two children storing solutions $\mathscr{P}(R')$ and $\mathscr{P}(R'')$, respectively. Node α stores the $O(f(n))$ actions taken to compute $\mathscr{P}(R) = \diamond(\mathscr{P}(R'), \mathscr{P}(R''))$ and its children store only the leftover pieces of $\mathscr{P}(R')$, $\mathscr{P}(R'')$ that are not employed to build $\mathscr{P}(R)$. We illustrate this by an example. If $\mathscr{P}(R)$ requires computing the distinct items in R and $\diamond$ is the union of sets, the leftover pieces are the duplicated items, namely the items in R' belonging to its intersection with R''. By repeating such a trick for all the nodes in the cross-tree, it turns out that a full solution is only available at the root. In order to obtain the solution in the other nodes, we can use a procedure called DOWN by [29, 30] to obtain $\mathscr{P}(R')$, $\mathscr{P}(R'')$ from $\mathscr{P}(R)$ recursively along a downward path, and a procedure UP to obtain $\mathscr{P}(R)$ from $\mathscr{P}(R')$, $\mathscr{P}(R'')$ along an upward path. This motivates the following definition.

DEFINITION 2.3. Given $\mathscr{P}(R) = \diamond(\mathscr{P}(R'), \mathscr{P}(R''))$, let $\diamond^{-1}(\mathscr{P}(R)) = \{\mathscr{P}(R'), \mathscr{P}(R'')\}$. Operator $\diamond$ is *invertible* if we can keep $O(f(n))$ bookkeeping information associated with any solution $\mathscr{P}(R)$ so that we can compute $\diamond^{-1}(\mathscr{P}(R))$ in $O(f(n))$ time.

Operator $\diamond$ corresponds to executing procedure UP while the inverse operator $\diamond^{-1}$ corresponds to DOWN. For example, if $\diamond$ is the destructive list append with cost $f(n) = O(1)$, we can simply keep a pointer to the last item in the appended lists to "de-append" them in $O(1)$ time by means of $\diamond^{-1}$. Operators $\diamond$ and $\diamond^{-1}$ are useful to save space in our technique, as they avoid data duplication. Specifically, in point 3 above, if $\diamond$ is invertible (Definition 2.3) and solution $\diamond(s_1, ..., s_j)$ is not of constant size, we store the $O(f(n))$ actions needed to compute $\diamond(s_1, ..., s_j)$ into α itself and only the leftover of solutions $s_1, ..., s_j$ into the children of α. We can, indeed, recover $s_1, ..., s_j$ in $O(f(n))$ time by applying $\diamond^{-1}$ to the solution in α, which is itself recursively computed from the parent of α. This way, *only the cross-tree root* needs to store the solution explicitly.

As far as the supported operations are concerned, we have to handle efficiently the split and concatenate operations on cross-tree $\mathrm{CT}(T_H \times T_V)$. We, therefore, choose to use two *weight-balanced B-trees* T_H and T_V; they are a weight-balanced variant of 2-3-trees, introduced by Arge and Vitter [4]. We actually need a simpler version of them. Let the weight $w(u)$ of a node u be the number of its descendent leaves. A weight-balanced B-tree T with branching parameter $a > 4$ satisfies the constraints:

- All the leaves have the same depth and are on level 0.
- An internal node u on level ℓ has weight $(1/2)a^\ell < w(u) < 2a^\ell$.
- The root has at least two children and weight less than $2a^h$, where h is its level.

Tree T has height $h = O(\log_a |T|)$ and between $a/4$ and $4a$ children per node, except the root. Among others, T satisfies an interesting property (typical of BB[α]-trees [28]).

FACT 2.4. *Given a leaf-to-root path Π in T, the sum of the weights of the nodes in Π is geometrically decreasing*: $\sum_{u \in \Pi} w(u) = O(|T|)$.

Tree T allows for insertions and deletions in logarithmic time and can be also split and concatenated like 2-3-trees [3].

THEOREM 2.5 (Arge and Vitter [3, 4]). *A weight-balanced B-tree T supports leaf insertions, leaf deletions, splits, and concatenates in $O(\log_a |T|)$ time per operation. Each operation only involves the nodes in a leaf-to-root path and their children.*

In our case, the two *weight-balanced B-trees* trees T_H and T_V in the cross-tree have parameter $a = O(1)$ and $O(n/k)$ leaves, one for each stripe of $\mathcal{M}$, with a total of $O(n/k)$ nodes. If T_H and T_V have different heights, we replace the root of the lower tree with a chain of unary nodes so that they both have the same height. Consequently, the cross-tree $\mathrm{CT}(T_H \times T_V)$ is balanced, and its height is $O(\log(n/k))$. We now give some bounds on our basic data structure, comprising $\mathcal{M}$ and the cross-tree.

LEMMA 2.6. *Apart from the time and space bounds needed for computing the solutions to decomposable problem $\mathcal{P}$, our basic data structure takes the following bounds for its construction and occupied space*:

- $O(f(n)\log(n/k) + (n^2/k^2))$ *when* $f(n) = \Omega(n)$;
- $O(f(n)(n^2/k^2))$ *otherwise.*

Proof. We call a cross-tree node α nonempty if its corresponding region in $\mathcal{M}$ contains at least one point, and we call it empty otherwise. Apart from the bounds needed to compute the solutions to $\mathcal{P}$ in the basic squares, the contribution to the total cost is $O(f(n))$ time and space for each nonempty node, and $O(1)$ time and space for each empty node. The total contribution of the empty nodes is clearly bounded by $O(n^2/k^2)$, and thus, we need to focus only on nonempty nodes.

Let N_ℓ denote the set of nonempty nodes on level ℓ of the cross-tree, where the leaves are on level 0 and the root is on level $h = O(\log(n/k))$. The total contribution of nonempty nodes can be expressed as

$$O\left(\sum_{\ell=0}^{h} \sum_{\alpha \in N_\ell} f(r_\alpha)\right),$$

where r_α denotes the number of points in the region corresponding to α, $1 \leqslant r_\alpha \leqslant n$. We bound this sum according to the two cases given by the function $f(n)$. As $f(n)$ is nondecreasing, $f(r_\alpha) \leqslant f(n)$, and so

$$O\left(\sum_{\ell=0}^{h} \sum_{\alpha \in N_\ell} f(r_\alpha)\right) = O\left(\sum_{\ell=0}^{h} \sum_{\alpha \in N_\ell} f(n)\right)$$

$$= O\left(\sum_{\ell=0}^{h} f(n)\,|N_\ell|\right) = O\left(f(n) \sum_{\ell=0}^{h} |N_\ell|\right) = O(f(n)(n^2/k^2)).$$

When $f(n)=\Omega(n)$, the fact that $f(n)$ is strongly nondecreasing implies that $f(a)+f(b)\leqslant f(a+b)$ (see [8]). This holds for any number of additive terms by induction, and thus,

$$\sum_{\alpha\in N_\ell} f(r_\alpha)\leqslant f\left(\sum_{\alpha\in N_\ell} r_\alpha\right)$$

for any given level ℓ. Furthermore,

$$\sum_{\alpha\in N_\ell} r_\alpha = n,$$

as the regions corresponding to the nonempty nodes on each level ℓ form a partition of the input set of points. Consequently, we can get a better estimate of the upper bound:

$$O\left(\sum_{\ell=0}^{h}\sum_{\alpha\in N_\ell} f(r_\alpha)\right)=O\left(\sum_{\ell=0}^{h} f\left(\sum_{\alpha\in N_\ell} r_\alpha\right)\right)=O\left(\sum_{\ell=0}^{h} f(n)\right)=O(f(n)\log(n/k)).$$

This completes the proof. ∎

We now discuss some basic operations to modify cross-tree $\mathrm{CT}(T_H\times T_V)$ efficiently. They are needed later on because we have to modify T_H or T_V while handling the stripes in our dynamic matrix $\mathcal{M}$. Here we describe why this happens. When we need to divide a horizontal stripe σ of matrix $\mathcal{M}$ into two stripes σ_1 and σ_2, we have to transform the corresponding leaf $w\in T_H$ into two leaves w_1 and w_2. We then rebalance T_H by splitting the "full" nodes in the path from w to the root. These steps are equivalent to executing a leaf insertion in T_H. Analogously, when we want to merge two adjacent stripes σ_1 and σ_2 of $\mathcal{M}$ into a single stripe σ, we need to replace two adjacent leaves w_1 and w_2 in T_H by a single leaf w. These operations are equivalent to executing a leaf deletion from T_H. Finally, when splitting the whole $\mathcal{M}$ into $\mathcal{M}_1$ and $\mathcal{M}_2$, or merging them, we have to split T_H into two trees T_1 and T_2, or concatenate them. Performing the above operations on the vertical stripes of $\mathcal{M}$ involves T_V in a similar way. Each of these operations on T_H and T_V can be implemented in $O(\log(n/k))$ time by Theorem 2.5 and must be reflected on the cross-tree $\mathrm{CT}(T_H\times T_V)$. We therefore discuss how to update the cross-tree accordingly.

We examine the case when T_H is split into T_1 and T_2. We can obtain cross-trees $\mathrm{CT}(T_1\times T_V)$ and $\mathrm{CT}(T_2\times T_V)$ from cross-tree $\mathrm{CT}(T_H\times T_V)$ as follows: We examine the nodes in T_H involved by its split. They form a path Π that leads from a leaf to the root in T_H by Theorem 2.5. Given a node $u\in\Pi$ to be split on level ℓ, we identify the corresponding nodes α_{uv} in the cross-tree for all $v\in T_V$ that are on level ℓ, according to the cross-tree definition, and reorganize these nodes suitably. By repeating this for all nodes in Π, we traverse a part of $\mathrm{CT}(T_H\times T_V)$ that is isomorphic to T_V. We then recompute the solutions to $\mathcal{P}$ in the modified cross-tree nodes by running two steps. In the first step, we traverse the involved nodes downward and apply operator $\diamond^{-1}$ to them to "pull down" their solutions. We execute this step only if $\diamond$ is invertible and the solutions do not have constant size. In the second step, we

apply operator $\diamond$ to the traversed nodes to update and "push up" their solutions in an upward traversal. We wish to point out that when executing the other operations (concatenate, inserte, and delete), we proceed along the same lines because these operations need to reorganize the nodes corresponding to a path in T_H by Theorem 2.5. We therefore have

THEOREM 2.7. *We can insert in, delete from, concatenate, and split T_H or T_V in order to update cross-tree* $\mathrm{CT}(T_H \times T_V)$ *in time*:

- $O(f(n) + (n/k))$ *when* $f(n) = \Omega(n)$;
- $O(f(n)(n/k))$ *otherwise.*

Proof. We only analyze the complexity of splitting as the other operations have an analogous analysis. Splitting T_H takes $O(\log(n/k))$ time by Theorem 2.5, and the cost of splitting cross-tree $\mathrm{CT}(T_H \times T_V)$ accordingly consists of two tasks: (a) updating the cross-tree topology and (b) recomputing the solutions to $\mathscr{P}$ by means of operators $\diamond$ and $\diamond^{-1}$.

Let n_ℓ be the number of nodes of T_V on level ℓ, where $0 \leqslant \ell \leqslant h = O(\log(n/k))$. In task (a), we examine $O(n_\ell)$ nodes per level in the cross-tree and execute $O(n_\ell)$ work on them. By summing over all the levels ℓ, we obtain a total cost of

$$\sum_{\ell=0}^{h} O(n_\ell) = O(|T_V|) = O(n/k).$$

To analyze task (b), we proceed as in the proof of Lemma 2.6. Since we traverse $O(n/k)$ empty nodes by the analysis of task (a), we have to bound the contribution of nonempty nodes and take their cost $f(n)$ into account. We define I_ℓ to be the set of nonempty nodes involved, on level ℓ, in the splitting operation. Their total contribution can be expressed as

$$O\left(\sum_{\ell=0}^{h} \sum_{\alpha \in I_\ell} f(r_\alpha)\right),$$

where again r_α denotes the number of points in the region corresponding to α, with $1 \leqslant r_\alpha \leqslant n$. We still have two cases. For any $f(n)$, we can use the simple upper bound:

$$O\left(\sum_{\ell=0}^{h} \sum_{\alpha \in I_\ell} f(n)\right) = O\left(\sum_{\ell=0}^{h} f(n)\,|I_\ell|\right) = O\left(f(n) \sum_{\ell=0}^{h} |I_\ell|\right) = O(f(n)(n/k)),$$

as $|I_\ell| = O(n_\ell)$ by the split operation. Otherwise, if $f(n) = \Omega(n)$,

$$\sum_{\alpha \in I_\ell} f(r_\alpha) \leqslant f\left(\sum_{\alpha \in I_\ell} r_\alpha\right),$$

and we exploit the fact that T_H is weight-balanced to give a better bound. Let $w(u)$ be the number of descendent leaves of a node u in T_H and examine the nodes involved by the split along its path Π, where

$$\sum_{u \in \Pi} w(u) = O(n/k)$$

by Fact 2.4. For any given level ℓ, the regions corresponding to the nodes $\alpha \in I_\ell$ are disjoint and contained in the $w(u)$ stripes associated with the leaves descendent of u, where u is the node in Π on level ℓ. Furthermore, as at most k points are in each stripe by the size invariant (Definition 2.1), we have

$$\sum_{\alpha \in I_\ell} r_\alpha = O(w(u) \cdot k)$$

points in these regions. Consequently, we bound

$$O\left(\sum_{\ell=0}^{h} \sum_{\alpha \in I_\ell} f(r_\alpha)\right) = O\left(\sum_{\ell=0}^{h} f\left(\sum_{\alpha \in I_\ell} r_\alpha\right)\right) = O\left(\sum_{u \in \Pi} f(w(u) \cdot k)\right) = O(f(n))$$

as

$$\sum_{u \in \Pi} f(w(u) \cdot k) \leqslant f\left(\sum_{u \in \Pi} w(u) \cdot k\right) = f\left(k \cdot \sum_{u \in \Pi} w(u)\right) \leqslant f(k \cdot O(n/k)) = f(O(n))$$

and f is strongly nondecreasing and smooth. Putting the analysis of tasks (a) and (b) together and adding the $O(\log(n/k))$ splitting cost of T_H, we obtain the claimed bounds. ∎

2.2. *Supported Operations*

We now show how to implement the operations in our splitting and merging problem. We adopt the data structure introduced in Section 2.1. In order to keep our algorithm analysis general, we introduce some smooth functions that help us to state the bounds achieved by our technique:

$P(k)$ = The cost of preprocessing an $O(k)$-point stripe to solve problem $\mathscr{P}$ for every basic square in the stripe. We expect to exploit the $\prec_P$-order inside each basic square when determining $P(k)$. We assume that $P(k) = \Omega(k)$ and that it is a strongly nondecreasing function, i.e., $P(a)/a \leqslant P(b)/b$ for $a \leqslant b$.

$U(k)$ = The cost of updating the solution to problem $\mathscr{P}$ for a basic square in an $O(k)$-point stripe after its preprocessing. We assume that $U(k) = \Omega(\log k)$, since we have to update at least the threaded search tree in the basic square.

$S(k)$ = The space occupied after preprocessing an $O(k)$-point stripe. We also assume that $S(k) = \Omega(k)$ is a strongly nondecreasing function.

In most of our applications, we will have $P(k) = O(k)$, $S(k) = O(k)$, and $U(k) = O(f(k) \log k)$. We now show how to preprocess the n points. We sort them in lexicographic order $(\prec_X; \prec_Y)$ by first computing their ranks in $\prec_X$ and $\prec_Y$ and then by sorting the resulting rank pairs. We then take their corresponding basic squares in row major order, i.e., the ones in the first horizontal stripe from left to right, then the ones in the second horizontal stripe from left to right, and so on. We distribute the points over these basic squares, such that globally $\lceil (k+1)/2 \rceil \leqslant k$ rows (columns) are in each horizontal (vertical) stripe, except for the last one, to which some dummy rows (columns) are added. For each nonempty basic square,

we sort its points in their $\prec_P$-order and build a threaded search tree on them. We also find their solution to $\mathscr{P}$. We finally build the cross-tree $\mathrm{CT}(T_H \times T_V)$ and store the proper solutions in its nodes by using operator $\diamond$.

LEMMA 2.8. *The total preprocessing time is*:

- $O(n \log n + P(n) + (n^2/k^2) + f(n) \log(n/k))$ *when* $f(n) = \Omega(n)$;
- $O(n \log n + P(n) + f(n)(n^2/k^2))$ *otherwise.*

The total memory space is:

- $O(S(n) + (n^2/k^2) + f(n) \log(n/k))$ *when* $f(n) = \Omega(n)$;
- $O(S(n) + f(n)(n^2/k^2))$ *otherwise.*

Proof. Sorting the points takes $O(n \log n)$ time and distributing them over the basic squares takes $O(n)$ time. Building the threaded search trees takes linear time because the points are already sorted in $\prec_P$-order. Computing the solutions in the basic squares takes $P(k)$ time per stripe and so $O(n/k \cdot P(k)) = O(n \cdot P(k)/k) = O(P(n))$ time for all the $O(n/k)$ stripes, as $P(k)/k \leqslant P(n)/n$. The remaining terms in the preprocessing time follow from Lemma 2.6, which gives the cost of building the cross-tree.

As each of the $O(n/k)$ stripes occupies $S(k)$ space, the space required by the basic squares and their solutions is $O(n/k \cdot S(k)) = O(S(n))$ space. We need also $O(S(n))$ space to store the solutions in the cross-tree. The remaining terms take into account the actual space usage of the cross tree, and follow again from Lemma 2.6. ∎

We remark that the term $n \log n$ in the preprocessing bound in Lemma 2.8 becomes n when the set S of items is already sorted.

Operations h_split *and* v_split. We perform v_split($\mathscr{M}$, j) as follows. Column j might fall inside a vertical stripe σ, which must necessarily be split. We examine the basic squares of σ. For each such basic square, we scan its points according to their $\prec_P$-order and produce two $\prec_P$-*ordered* lists: one list contains all the points whose second coordinate is smaller than or equal to j and the other list contains the remaining points, i.e. the points whose second coordinate is larger than j. We split each basic square into two squares and build two threaded search trees for them by using the two $\prec_P$-ordered lists. This creates $O(n/k)$ smaller squares and splits stripe σ into new stripes σ_1 and σ_2, such that σ_1 contains all the points of σ before and including column j, and σ_2 contains all the points of σ after column j. We check to see if we can combine σ_1 and σ_2 with their neighbor stripes to maintain the size invariant of order k. For any two such stripes to be merged, we examine their basic squares in pairs (a square per stripe), such that the two squares are on the same horizontal stripe. We take their two $\prec_P$-ordered lists of points and merge them in order to build a threaded search tree on the resulting list. It is worth noting that splitting and merging stripes preserve the order of their presorted points. Next, we determine the solutions for the basic squares in the $O(1)$ stripes involved and update the cross-tree $\mathrm{CT}(T_H \times T_V)$ to reflect the split operation on the vertical stripes: We have to split tree T_V at the leaf w corresponding to stripe σ. We split w into two new leaves w_1 and w_2, corresponding to the split of σ into the new

stripes σ_1 and σ_2. If σ_1 or σ_2 are combined with their neighbor stripes, we should do the same on w_1 and w_2 and their neighbor leaves. Globally, we create no more than $O(n/k)$ leaves corresponding to the new basic squares in $O(1)$ stripes and we traverse and reorganize their ancestor nodes all the way up to the cross-tree root as shown in Theorem 2.7. The implementation of h_split is completely analogous.

LEMMA 2.9. *Both operations* h_split *and* v_split *take time*:

- $O(P(k)+f(n)+(n/k))$ *when* $f(n)=\Omega(n)$;
- $O(P(k)+f(n)(n/k))$ *otherwise.*

Proof. Splitting and merging the $\prec_P$-ordered lists and rebuilding the threaded search trees takes linear time because the points are presorted. This implies that we can process σ, σ_1, σ_2 and their $O(1)$ neighboring stripes in $O(n/k+k)$ time as any stripe consists of $O(n/k)$ basic squares and contains $O(k)$ points. Recomputing the solutions in their basic squares needs $O(P(k))$ time, namely, $P(k)$ time per stripe. The resulting $O(n/k+k+P(k))=O(n/k+P(k))$ cost must be added to the cost of updating the cross-tree given in Theorem 2.7. ∎

Operations h_concatenate *and* v_concatenate. They are the inverse of h_split and v_split, respectively. They involve recombining two cross-trees whose corresponding points undergo an *identical* (horizontal or vertical) stripe partition and they cause the pairwise merging of $O(n/k)$ cross-tree leaves in the worst case. Again, this requires recomputing the solutions in the traversed nodes. We have

LEMMA 2.10. *Both operations* h_concatenate *and* v_concatenate *take time*:

- $O(P(k)+f(n)+(n/k))$ *when* $f(n)=\Omega(n)$;
- $O(P(k)+f(n)(n/k))$ *otherwise.*

Operation set$(i, j, z, \mathcal{M})$. We use it to perform either an insertion (if w is nonempty) or a deletion (if w is empty). Let us assume first that we only have insertions. We locate the basic square corresponding to position (i, j) by traversing a path τ from the root to the relative leaf in the cross-tree. If necessary, we insert (i, j) into the threaded search tree of the basic square (we use the $\prec_P$-order). We then update the solution in the basic square and link it to its relative cross-tree leaf if the basic square becomes nonempty. We propagate the new solution along path τ in two steps. In the first step, we traverse τ downward and apply operator $\diamond^{-1}$ to the traversed nodes. We run this step only if $\diamond$ is invertible (Definition 2.3) and the solutions do not have constant size. In the second step, we apply operator $\diamond$ to the nodes in τ to recompute their solutions in an upward traversal. As setting $\mathcal{M}[i, j]=z$ can cause the insertion of a new row at position i and a new column at position j, the horizontal stripe and the vertical stripe whose intersection gives the updated basic square can violate the size invariant of order k. Since we are treating insertions, this means that each such stripe σ has increased its number of rows (columns) from $\lceil (k+1)/2 \rceil$ to $k+1$. To maintain the size invariant, we split σ into two stripes σ_1 and σ_2 of at most $\lceil (k+1)/2 \rceil$ rows (columns) each, and then update the cross-tree accordingly. In order to handle also item deletions, we simply mark the item as logically deleted and update the solution in its basic square (this

is the so-called "weak" deletion and does not violate the size invariant). We then percolate the updated solution along its corresponding path τ in the cross-tree.

LEMMA 2.11. *Inserting and deleting a point with operation set takes time*:

- $O(U(k)+f(n)(1+n/k^2)+P(k)/k)$ *when* $f(n)=\Omega(n)$;
- $O(U(k)+f(n)(\log(n/k)+n/k^2)+P(k)/k)$ *otherwise*

Proof. As the cost of (weak) deletions does not exceed the cost of insertions, we examine the cost of an insertion. Traversing path τ in the cross-tree takes $O(\log(n/k))$ time, as the cross-tree height is $O(\log(n/k))$. It takes $O(\log k)$ time to update the threaded search tree and $U(k)$ time to update the solution in the basic square. The cost of percolating the updated solution along path τ depends on $f(n)$: if $f(n)=\Omega(n)$, the cost is a geometrically decreasing sum equal to $O(f(n))$, as a consequence of Fact 2.4 on weight-balanced B-trees; otherwise, the cost is $O(f(n)\log(n/k))$. The resulting cost is $O(\log(n/k)+\log k+U(k)+f(n))=O(U(k)+f(n))$ in the former case and $O(\log(n/k)+\log k+U(k)+f(n)\log(n/k))=O(U(k)+f(n)\log(n/k))$ in the latter case (as $U(k)\geqslant\log k$).

We now have to add the splitting cost for a full stripe σ (with $k+1$ rows or columns in it) into σ_1 and σ_2 to maintain the size invariant. It does not exceed the cost of a split or concatenate operation (Theorem 2.7 and Lemmas 2.9 and 2.10) and depends on $f(n)$. This gives an immediate amortized bound over the next $\Theta(k)$ insertions in either σ_1 and σ_2. In order to obtain good worst-case bounds for the single insertion, we spread this cost for σ among the subsequent operations. Essentially, we use Overmars' global rebuilding technique [30] by adding an extra worst-case time $O(P(k)/k+f(n)(n/k^2))$ to each insertion, such that $\Theta(k)$ insertions are sufficient to cover the splitting cost for σ, which does not exceed $O(P(k)+f(n)(n/k))$ due to Lemmas 2.9 and 2.10. ∎

Operation range$(h_1,h_2,v_1,v_2,\mathcal{M})$. We perform it by executing an h_split at row (h_1-1) followed by another h_split at row h_2. We let r be the number of rows of $\mathcal{M}$. These two horizontal splits produce three matrices $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$, such that $\mathcal{M}_1$ consists of the first (h_1-1) rows of $\mathcal{M}$, $\mathcal{M}_2$ contains the rows of $\mathcal{M}$ between h_1 and h_2; and $\mathcal{M}_3$ contains the last $(r-h_2+1)$ rows of $\mathcal{M}$. We produce submatrix $\mathcal{M}[h_1,h_2;v_1,v_2]$ by carrying out two additional vertical splits on $\mathcal{M}_2$ at columns (v_1-1) and v_2, respectively. We give the solution that is stored in the root of the resulting cross-tree, which corresponds to region $\mathcal{M}[h_1,h_2;v_1,v_2]$, and then assemble back the pieces by a proper sequence of four concatenate operations. Since this involves a constant number of h_split, v_split, h_concatenate, and v_concatenate operations, we have

LEMMA 2.12. *A range query takes time*:

- $O(P(k)+f(n)+(n/k))$ *when* $f(n)=\Omega(n)$;
- $O(P(k)+f(n)(n/k))$ *otherwise*

The worst-case time bounds for the operations in Lemmas 2.9–2.12 are stated in terms of the slack parameter k. It is worth noting that k depends on n and so the choice of k makes things more difficult for us, as n can vary due to the *set* operations

and k must be adapted to the changes in n. This is not a problem, however, as we might guess a value for n and then compute the data structure for the guessed value of n: whenever the actual value gets twice as large or twice as small as the guessed value, we update our guess to its actual value and reinitialize the data structure accordingly. This would make our time bounds amortized. However, we can use again standard techniques [30] to make these bounds worst-case, as summarized below.

THEOREM 2.13. *The splitting and merging problem on n points can be solved with the following bounds for a slack parameter k* $(1 \leqslant k \leqslant n)$ *and an operator cost* $f(n)$:

(a) *Operations* range, h_split, v_split, h_concatenate, v_concatenate *take time*:

- $O(f(n) + (n/k) + P(k) + P(n)/n)$ *when* $f(n) = \Omega(n)$;
- $O(f(n)(n/k) + P(k) + P(n)/n)$ *otherwise.*

(b) *Inserting and deleting a point with operation* set *takes time*:

- $O(U(k) + f(n)(1 + n/k^2) + P(n)/n)$ *when* $f(n) = \Omega(n)$;
- $O(U(k) + f(n)(\log(n/k) + n/k^2) + P(n)/n)$ *otherwise.*

(c) *The total memory space is*:

- $O(S(n) + (n^2/k^2) + f(n)\log(n/k))$ *when* $f(n) = \Omega(n)$;
- $O(S(n) + f(n)(n^2/k^2))$ *otherwise.*

(d) *The total preprocessing time is*:

- $O(n \log n + P(n) + (n^2/k^2) + f(n)\log(n/k))$ *when* $f(n) = \Omega(n)$;
- $O(n \log n + P(n) + f(n)(n^2/k^2))$ *otherwise.*

Proof. The operations described in Lemmas 2.9–2.12 are "weak"; i.e., there exists a constant $0 < \beta < 1$, such that, after $\beta \cdot n$ weak operations, the time and space bounds do not increase asymptotically. Overmars and van Leeuwen's global rebuilding technique [30, 35] allows us to obtain worst-case time bounds for an *arbitrary number* of weak operations. We keep two copies of the data structures and switch among them every other $\beta \cdot n_0$ operations, where n_0 is the number of items after the last switch. In this way, the worst-case bounds stated in Lemmas 2.9–2.12 increase by an additive term given by the preprocessing cost in Lemma 2.8 divided by the number n of points. Since the points are presorted, this additive term is $O(P(n)/n + (n/k^2) + (f(n)/n)\log(n/k))$ when $f(n) = \Omega(n)$ and $O(P(n)/n + f(n)(n/k^2))$ otherwise. As a result, we obtain the claimed bounds. ∎

2.3. *Some Examples with Operator Cost* $f(n)$

We now show how to use Theorem 2.13 with some choices of the operator cost $f(n)$. We first have to find the costs $P(k)$, $U(k)$, and $S(k)$ for the points in a stripe. We apply the following theorem to each basic square in the stripe.

THEOREM 2.14 (Overmars [29, 30]). *Given an* $f(k)$*-decomposable problem* $\mathcal{P}$ *on a set R of k items, there exists a dynamic data structure for maintaining its solution* $\mathcal{P}(R)$ *with preprocessing, update, and memory space bounds*:

$$P(k) = \begin{cases} O(k \log k) & \text{when } f(k) = O(k^{\varepsilon}) \text{ for some } \varepsilon < 1; \\ O(f(k)) & \text{when } f(k) = \Omega(k^{1+\varepsilon}) \text{ for some } \varepsilon > 0; \\ O(f(k) \log k + k \log k) & \text{otherwise;} \end{cases}$$

$$U(k) = \begin{cases} O(f(k)) & \text{when } f(k) = \Omega(k^{\varepsilon}) \text{ for some } \varepsilon > 0; \\ O(f(k) \log k) & \text{otherwise;} \end{cases}$$

$$S(k) = \begin{cases} O(k) & \text{when } f(k) = O(k^{\varepsilon}) \text{ for some } \varepsilon < 1; \\ O(f(k)) & \text{when } f(k) = \Omega(k^{1+\varepsilon}) \text{ for some } \varepsilon > 0; \\ O(f(k) \log \log k) & \text{when } f(k) \text{ is at least linear;} \\ O(f(k) \log k + k \log k) & \text{otherwise.} \end{cases}$$

The term $k \log k$ in the preprocessing bound $P(k)$ in Theorem 2.14 becomes k when the set R of items is already sorted. Our first result is for our $f(n)$-decomposable problem $\mathscr{P}$ on a set S of n items when $f(n) = O(\log n)$; in this case we apply Theorem 2.14 to obtain $U(k) = O(f(k) \log k)$, $S(k) = O(k)$, and $P(k) = O(k)$ as we maintain the points in $\prec_P$-order in the basic squares of each stripe. The resulting cost for range, split, and concatenate operations in Theorem 2.13 is minimized when k satisfies asymptotically: $(n/k) f(n) = P(k) + P(n)/n$; i.e., $k = \lceil \sqrt{n f(n)} \rceil$:

THEOREM 2.15. *The splitting and merging problem on n items can be solved with time bounds when the cost of operator* $\diamond$ *is* $f(n) = O(\log n)$:

- range, h_split, v_split, h_concatenate, v_concatenate: $O(\sqrt{n f(n)})$;
- set: $O(f(n) \log n)$;

The space required is $O(n)$ *and the preprocessing time is* $O(n \log n)$.

When $f(n) = \Theta(n)$, we can use Theorem 2.14 to obtain $U(k) = O(k)$, $S(k) = O(k \log \log k)$, and $P(k) = O(k \log k)$. The resulting cost for range, split, and concatenate operations in Theorem 2.13 is minimized for $k = \lceil n/\log n \rceil$.

THEOREM 2.16. *The splitting and merging problem on n items can be solved with time bounds when* $f(n) = \Theta(n)$:

- set, range, h_split, v_split, h_concatenate, v_concatenate: $O(n)$.

The space required is $O(n \log \log n)$ *and the preprocessing time is* $O(n \log n)$.

3. SOME APPLICATIONS OF SPLITTING AND MERGING ORDER DECOMPOSABLE PROBLEMS

3.1. Two-Dimensional Priority Queues

A first and simple example of order-decomposable problem deals with maintaining a priority queue for a set S of weighted items subjected to two orders $\prec_X$ and $\prec_Y$. Each item e has a positive real weight $w(e)$ associated with it, which is called its *priority*. Without loss of generality, we assume that all the weights are distinct:

if this is not the case, we can make use of the pair $(w(e), e)$ to break the ties. Problem $\mathscr{P}$ consists of finding the minimum-weight item in the priority queue restricted to an input region, where the order of the items changes by splitting and concatenating them according to either $\prec_X$ or $\prec_Y$.

THEOREM 3.1. *A two-dimensional priority queue for a set of n items can be maintained in time bounds*:

- *an item insertion or deletion*: $O(\log n)$;
- *a split or concatenate of one order*: $O(\sqrt{n})$;
- *a minimum-weight query in a region*: $O(\sqrt{n})$.

The space required is $O(n)$ *and the preprocessing time is* $O(n \log n)$.

Proof. We define the $\prec_P$-order inside each basic square to be the lexicographic order $(\prec_X; \prec_Y)$ and build the corresponding threaded search trees. We augment each search tree by storing, in each of its internal nodes, the minimum-weight item in its descendent nodes. This requires a total of $O(n \log n)$ preprocessing time and $S(n) = O(n)$ space. It takes $U(k) = O(\log k)$ time to insert or delete an item in a basic square and to update the minimum-weight item by using the search tree. When splitting or merging stripes, we have to split or concatenate the basic squares. Since the search tree in a basic square can be split and concatenated in time linear with its size, we have a total of $P(k) = O(k)$ stripe processing time. We then store, in each cross-tree node α, the minimum-weight item lying in the region of α and we choose operator $\diamond(e_1, e_2)$ to return the minimum-weight item between e_1 and e_2, with an empty solution being represented by $+\infty$. Note that we do not need to use $\diamond^{-1}$, as the solutions here have constant size. We obtain the claimed time bounds by setting $f(n) = O(1)$ in Theorem 2.15. ∎

3.2. *Two-Dimensional* 2-3-*Trees and Range Searching*

A two-dimensional 2-3-tree stores a collection of points in the Cartesian plane, under the following operations: insert or delete a point, and split or concatenate all the points along one of their coordinates. Furthermore, we wish to search for the points that range inside a rectangular axis parallel region.

A vast literature deals with the two-dimensional range searching problem but only a few solutions handle splitting and concatenating along both the coordinates (see [9]). The best previously known time bounds for n points are obtained with the divided k-d tree of van Kreveld and Overmars [17]: $O(\log n)$ for an insertion or a deletion; $O(\sqrt{n \log n})$ for a split or concatenate; $O(\sqrt{n \log n} + occ)$ for a range search, where occ is the number of points retrieved by the search. With our technique, we shave the $\sqrt{\log n}$ factor from these bounds.

THEOREM 3.2. *A two-dimensional* 2-3-*tree storing n points can be maintained with bounds*:

- *a point insertion or deletion*: $O(\log n)$;
- *a split or concatenate along one coordinate*: $O(\sqrt{n})$;

- *a range search*: $O(\sqrt{n} + occ)$, *where occ is the number of points reported by the search.*

The space required is $O(n)$ *and the preprocessing time is* $O(n \log n)$.

Proof. We take $\prec_P$ as the lexicographic order $(\prec_X; \prec_Y)$ and take $O(n \log n)$ preprocessing time. We have $U(k) = O(\log k)$ for updating the search tree in a basic square. We require $P(k) = O(k)$ stripe processing time when splitting or merging stripes. The solution stored in a node is now the list of points lying in its corresponding region of the Cartesian plane. In order to get a total $S(n) = O(n)$ space, instead of $O(n \log n)$, due to the list duplication in the cross-tree nodes, we use a "destructive" list append as operator $\diamond$ in $f(n) = O(1)$ time. Its inverse operator $\diamond^{-1}$ can be implemented in $O(1)$ time if we record the point in which we append the two lists by means of $\diamond$. This way, we store explicitly all the points in the cross-tree root only, and we store them implicitly in the remaining cross-tree nodes. The inverse operator $\diamond^{-1}$ will enable us to build the explicit list of points if needed (Definition 2.3). We obtain the claimed time bounds by plugging $f(n) = O(1)$ in the bounds of Theorem 2.15. ∎

3.3. Concatenable Interval Trees

An interval tree stores a set I of some intervals having real endpoints. This is useful for answering stabbing queries, which are defined as follows [11]: given a real number r, we want to find all the intervals in I that contain r. A fast, nearly logarithmic amortized solution to splitting and concatenating interval trees is given by van Kreveld and Overmars [18], in which they assume that intervals are indivisible; i.e., the real number on which we want to split cannot stab any intervals and, when concatenating two interval trees, the interval endpoints in one tree have to be all smaller than the ones in the other tree. We consider a more general problem by relaxing this assumption at an increased cost for splitting, concatenating, and stabbing queries. Namely, let I be the interval set. We wish to split as follows:

left_split(I, r) We obtain two interval trees by splitting on the left endpoints, where we allow r to stab any interval in I. The two resulting trees store the intervals in $\{[a, b] \in I : a \leqslant r\}$ and $\{[a, b] \in I : r < a\}$, respectively.

right_split(I, r) We obtain two interval trees by splitting on the right endpoints, where we allow r to stab any interval in I. The two resulting trees store the intervals in $\{[a, b] \in I : b \leqslant r\}$ and $\{[a, b] \in I : r < b\}$, respectively.

Concatenate operations are the inverse of *split* operations and can be applied to I with respect to either its left or right endpoints. We only require that, when concatenating two interval sets I_1 and I_2, the left (resp., right) endpoints in I_1 are all smaller than or equal to the ones in I_2. Note that we put no constraints on the other endpoints.

THEOREM 3.3. *An interval tree that stores n intervals can be maintained with bounds*:

- *an interval insertion or deletion*: $O(\log n)$;
- *a split or concatenate according to the intervals' endpoints*: $O(\sqrt{n})$;

• *a stabbing query (i.e., find all the intervals containing a given point)*: $O(\sqrt{n} + occ)$, *where occ is the number of intervals reported by the query.*

The space required is $O(n)$ *and the preprocessing time is* $O(n \log n)$.

Proof. We use a well-known transformation of interval stabbing into a particular case of the two-dimensional range query called corner query; each interval $[a, b] \in I$ is mapped into a point (a, b) in the Cartesian plane. A stabbing query for a real number r amounts to retrieving the points (x, y) such that $x \leqslant r \leqslant y$. Consequently, we can search for the points contained in the region obtained by intersecting two halfplanes $x \leqslant r$ and $y \geqslant r$ (see Figs. 6a, 6b). A left split (or concatenate) in the intervals corresponds to a horizontal split (or concatenate) on the points in the plane; analogously, a right split (or concatenate) corresponds to a vertical split (or concatenate). We can, therefore, use Theorem 3.2 to achieve the claimed bounds. ∎

In another paper of theirs [19], van Kreveld and Overmars present a solution that retrieves the stabbed intervals with their lengths bounded by two input values. They obtain $O(\log n)$ *amortized* insertion and deletion time and $O(\sqrt{n} \log n + occ)$ stabbing query time with $O(n)$ space, where n is the number of intervals and occ is the output size. We achieve better *worst-case* bounds:

THEOREM 3.4. *An interval tree that stores n intervals can be maintained with worst-case time bounds*:

• *an interval insertion or deletion*: $O(\log n)$;

• *a split or concatenate according to the intervals' endpoints*: $O(\sqrt{n \log n})$;

• *a stabbing query, retrieving only the intervals whose lengths are between two input values* $\ell_1 \cdots \ell_2$: $O(\sqrt{n \log n} + occ)$, *where occ is the number of intervals reported by the query.*

The space required is $O(n)$ *and the preprocessing time is* $O(n \log n)$.

Proof. We let (r, ℓ_1, ℓ_2) be the stabbing query, where r is the stabbing number and the lengths of the stabbed intervals must all be in $\ell_1 \cdots \ell_2$. We use the same

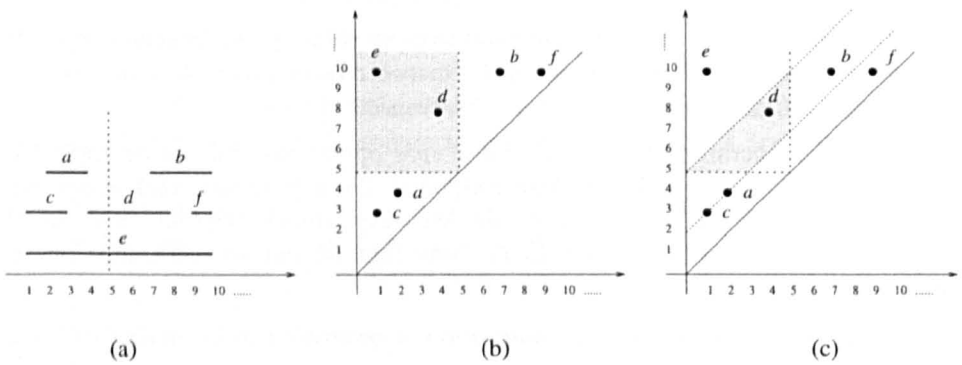

FIG. 6. A well-known transformation of the intervals in (a) into the two-dimensional points in (b). A stabbing query for $r = 5$ is illustrated in (a) and (b). A stabbing query for $r = 5$ and for the interval lengths bounded by $\ell_1 = 2$ and $\ell_2 = 5$ is shown in (c).

range searching reduction as in the proof of Theorem 3.3. Consequently, we wish to report the points (x, y), such that $x \leqslant r \leqslant y$ and $\ell_1 \leqslant y - x \leqslant \ell_2$. The corresponding region in the Cartesian plane is shown in Fig. 6c and is obtained by intersecting four halfplanes: $x \leqslant r$, $y \geqslant r$, $y - x \geqslant \ell_1$ and $y - x \leqslant \ell_2$. We adapt the solution to two-dimensional 2-3-trees (Theorem 3.2) in order to answer our query. We replace the $\prec_P$-order in which the points are stored in the threaded search tree of each basic square by a different order, where $(x_1, y_1) \prec_P (x_2, y_2)$ if and only if either $(y_1 - x_1 < y_2 - x_2)$ or $(y_1 - x_1 = y_2 - x_2$ and $x_1 < x_2)$. This new order $\prec_P$ arranges the points in a basic square according to the diagonals, as illustrated in Fig. 7a. We now discuss how to find the intervals satisfying the stabbing query (in the example of Fig. 6, they correspond to the points in the region shown in Fig. 6c. We identify five groups of basic squares (see Fig. 7b):

- V: The basic squares in which the vertical line $x = r$ falls.
- H: The basic squares in which the horizontal line $y = r$ falls.
- D_1: The basic squares in which the diagonal line $y - x = \ell_1$ falls.
- D_2: The basic squares in which the diagonal line $y - x = \ell_2$ falls.
- R: All the remaining basic squares in the region bounded by the above four lines (shown in boldface in Fig. 7b).

We want to retrieve some points in V, H, D_1, D_2 and all the points in R. Since V and H are two stripes and contain $O(k)$ points by the size invariant on the stripes, we can read these points and report the ones that are inside the region in $O(k + n/k)$ time. We can also traverse the cross-tree part that corresponds to R and report all of its points in $O(n/k + occ)$ time by means of operators $\diamond$ and $\diamond^{-1}$, where $\diamond$ is the list-append with $f(n) = O(1)$ as in the proof of Theorem 3.2. Unfortunately, we cannot examine all the points in D_1 and D_2 because they are too many. We only know that D_1 and D_2 consist of $O(n/k)$ basic squares and, in the worst case, we could scan $O(k)$ points per basic square: indeed the size invariant only holds for the stripes while it does not necessarily hold for D_1 and D_2. Although D_1

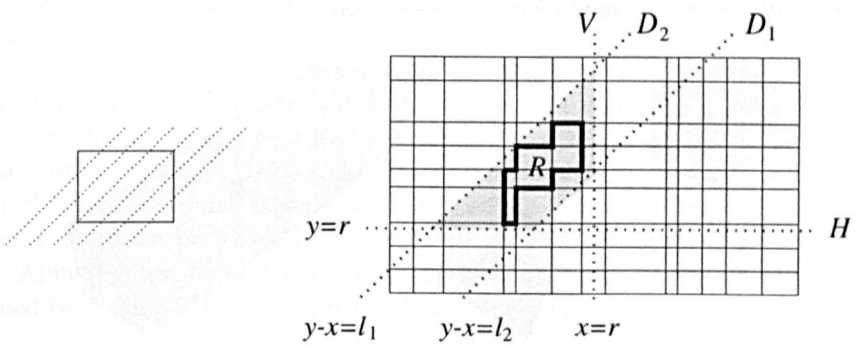

FIG. 7. (a) The $\prec_P$ order inside a basic square for the problem of stabbing intervals of bounded lengths. (b) An arbitrary query identifies the region obtained by intersecting four delimiting halfplanes.

and D_2 can contain $O(n)$ points, we process them in $O((n/k)\log k + occ)$ total time by examining their basic squares as follows: Given a basic square, we scan its $\prec_P$-ordered point list, say, L, stored in its search tree. If the basic square is in D_1, we report all points (x, y), such that $y - x \geqslant \ell_1$, by first searching in L and then scanning it from the largest to the smallest point in $\prec_P$-order. If the basic square is in D_2, we report all points (x, y), such that $y - x \leqslant \ell_2$, by searching and scanning L from the smallest to the largest point in $\prec_P$-order: note that the points in the four "corner" squares are examined in H and V (see Fig. 7b). We can perform the above searches and scans in $O(\log k)$ time plus a linear cost with respect to the number of retrieved occurrences. This accumulates to $O((n/k)\log k + occ)$ time because we repeat it $O(n/k)$ times. To complete the proof, it is enough to take $f(n) = O(1)$ and $k = \lceil \sqrt{n \log n} \rceil$. ∎

3.4. *Convex Hulls in the Plane*

The convex hull for a set of n points in the Cartesian plane is the smallest convex polygon containing all the points. It can be computed in $O(n)$ time *after* sorting the points [24]. Computing the convex hull is a $(\log n)$-order decomposable problem since it is possible to split the plane into two halfplanes, compute recursively the convex hull for each halfplane, and then combine the two convex hulls together in time $O(\log n)$. Overmars and van Leeuwen [31] show that two disjoint convex hulls can be combined together in logarithmic time by finding their two common tangents called supporting lines (see Fig. 8). This fact is exploited in van Kreveld and Overmars [19] to give the following time bounds on the convex hull: $O(\log^2 n)$ *amortized* for an insertion or a deletion; $O(\sqrt{n \log n \log n})$ for a split or a concatenate. Our results can be summarized.

THEOREM 3.5. *The convex hull for n points in the Cartesian plane can be maintained with worst-case time bounds*:

- *a point insertion or deletion*: $O(\log^2 n)$;
- *a split or concatenate along one coordinate*: $O(\sqrt{n \log n})$;

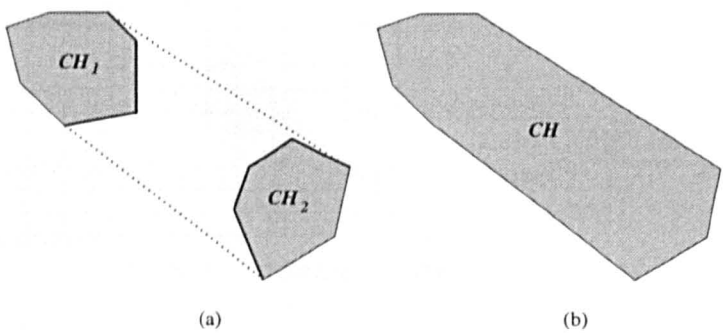

FIG. 8. (a) The supporting lines (dotted) for combining two convex hulls CH_1 and CH_2 together and (b) the resulting convex hull CH. If CH is associated with a node, then the bold parts in CH_1 and CH_2 are the leftovers associated with its two children, respectively. A bridging operation obtains CH from CH_1 and CH_2 while a de-bridging operation obtains CH_1 and CH_2 from CH.

- *a query checking if a point is inside or outside the convex hull*: $O(\log n)$;
- *a query reporting the convex hull for the points in an input region*: $O(\sqrt{n \log n} + h)$, *where* h *is the output size.*

The space required is $O(n)$ *and the preprocessing time is* $O(n \log n)$.

Proof. We use the dynamic technique in [31] for maintaining the convex hull of n points under point insertions and deletions in $O(\log^2 n)$ time per operation. A convex hull is represented as a balanced search tree storing the convex-hull points in clockwise order. Combining two convex hulls in two disjoint halfplanes by means of their supporting lines (Fig. 8) takes $O(\log n)$ time. This operation is called *bridging* and its inverse operation, called *debridging*, also takes $O(\log n)$ time [9]. In [31], a balanced binary tree is kept, such that its leaves store the points sorted by their lexicographic order and each internal node α stores the convex hull CH_α of the points stored in the leaves descendent of α (CH_α is stored as a balanced tree in the secondary structure of α). Actually, α only stores the points in CH_α that are not in CH_σ for any ancestor node σ of α and the total space remains $O(n)$ because each point is stored $O(1)$ times in the tree. The root stores the whole convex hull CH_{root} for the point set and querying if a point is inside or outside a convex hull requires logarithmic time by traversing CH_{root} (e.g., see Ref. [9]).

We turn to our splitting and merging data structure. For each basic square, we set $\prec_P$ to be the lexicographic order $(\prec_X, \prec_Y)$ for its points. We compute their convex hull and maintain the balanced binary tree for them in clockwise order by following the dynamic technique of [31]. This allows for insertions and deletions of a point in a basic square in $U(k) = O(\log^2 k)$ time and requires a total of $O(n \log n)$ preprocessing time and $S(n) = O(n)$ space. When splitting or merging a basic square we can maintain its internal order $\prec_P$ and rebuild the balanced binary trees for its convex hull in linear time. This is possible because the cost $c(k')$ of rebuilding this tree for its k' *sorted* points is given by $c(k') = 2c(k'/2) + O(\log k')$, which is $O(k')$. Consequently, processing a stripe takes $P(k) = O(k)$ time.

Analogously to the balanced binary trees in [31], each cross-tree node α stores, as its secondary structure, the convex hull CH_α of the points in the region corresponding to α (i.e., only the points in CH_α that are not in CH_σ for any ancestor node σ of α). The cross-tree root stores the whole convex hull CH_{root} and the cross-tree leaves store the convex hulls of the points in their corresponding basic squares. We let $\diamond$ be the bridging operation and $\diamond^{-1}$ the debridging operation described above, both having cost $f(n) = O(\log n)$. Since an internal node in the cross-tree has $O(1)$ children, $\diamond$ must recombine $O(1)$ convex hulls. This is not a problem, however, as it can be solved by a simple variation of the original bridging operation. Applying the inverse debridging operations is analogous. We obtain our claimed time bounds by setting $f(n) = O(\log n)$ in Theorem 2.15. ∎

3.5. Planar Voronoi Diagrams

The planar Voronoi diagram for a set of n points is the partition of the plane into n Voronoi polygons, where each point δ induces the Voronoi polygon given by the (possibly unbounded) convex region formed by the points in the plane that are

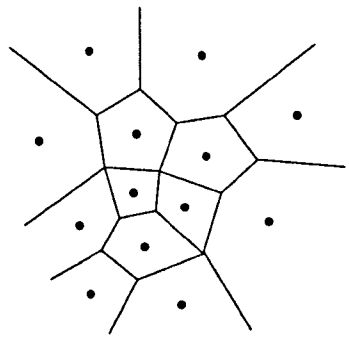

FIG. 9. An example of planar Voronoi diagram.

closer to δ (see Fig. 9). The Voronoi diagram can be computed in $O(n \log n)$ time and $O(n)$ space (e.g., see [26]). Computing the Voronoi diagram is an $O(n)$-order decomposable problem, as the plane can be split into two halfplanes in order to compute the Voronoi diagram in each halfplane recursively, and the two resulting Voronoi diagrams can then be merged in time $O(n)$ [40]. Point insertions and deletions in a planar Voronoi diagram of n points take $O(n)$ time in $O(n \log \log n)$ space [30] (a result in [1] is a semidynamic algorithm with $O(n)$ deletion time and space). We obtain an $O(n)$ cost also for range, split, and concatenate operations in $O(n \log \log n)$ space (the techniques in [19, 39, 42] require more time or space). This gives an answer to a question posed in [1] (i.e., given the Voronoi diagram for a set S of n points, compute the Voronoi diagram for any given subset $R \subseteq S$ in $O(n)$ time) for the special case in which R is defined by range queries on a dynamic set S.

THEOREM 3.6. *The Voronoi diagram for n points in the Cartesian plane can be maintained with worst-case time bounds:*

- *a point insertion or deletion:* $O(n)$;
- *a split or concatenate along one coordinate:* $O(n)$;
- *a query reporting the Voronoi diagram for the points in an input region:* $O(n)$.

The space required is $O(n \log \log n)$ *and the preprocessing time is* $O(n \log n)$.

Proof. It follows directly from Theorem 2.16. ∎

Once the Voronoi diagram for a region R is available, we can solve nearest neighbor and closest point problems and find the minimum Euclidean spanning tree, the triangulation, and the largest empty circle for the points in R efficiently. We refer to [30] for more discussion.

4. MAINTAINING $d > 2$ TOTAL ORDERS

In this section, we describe our technique for the general case of $d > 2$ total orders $\prec_1, \ldots, \prec_d$ defined on a set S containing n items. We reduce our problem to a matrix problem in which M is a d-dimensional matrix and item $x \in S$ is a d-dimensional point $(x_1, \ldots, x_d)$, such that x_i is the rank of x in S according to order

$\prec_i$ ($i=1, ..., d$). We begin by defining the cross-trees for dimension $d>2$. A cross-tree $\mathrm{CT}(T_1 \times T_2 \times \cdots \times T_d)$ of d weight-balanced B-trees $T_1, T_2, ..., T_d$ is defined:

- $T_1, T_2, ..., T_d$ have the same height and $O(1)$ children per node. Their leaves are on the same level.
- For each choice of nodes $u_1, u_2, ..., u_d$ on the same level, such that $u_i \in T_i$ for $1 \leqslant i \leqslant d$, there is a node $\alpha_{u_1 u_2 .. u_d}$ in $\mathrm{CT}(T_1 \times T_2 \times \cdots \times T_d)$.
- For each choice of edges $(u_1, \hat{u}_1), (u_2, \hat{u}_2), ..., (u_d, \hat{u}_d)$, such that $u_1, u_2, ..., u_d$ are on the same level and $(u_i, \hat{u}_i) \in T_i$ for $1 \leqslant i \leqslant d$, there is an edge $(\alpha_{u_1 u_2 .. u_d}, \alpha_{\hat{u}_1 \hat{u}_2 .. \hat{u}_d})$ in $\mathrm{CT}(T_1 \times T_2 \times \cdots \times T_d)$.

We define the stripes in $\mathcal{M}$ by following the approach described in Section 2.1. In dimension two, a stripe is composed of $O(k)$ rows or columns, which can be seen as matrices of dimension one. We therefore define the stripes as the groups of the $O(k)$ adjacent matrices of dimension $(d-1)$, each such matrix containing $O(1)$ items. There are $O(n/k)$ stripes defined for each coordinate of $\mathcal{M}$ (i.e., we have a total of d sets of $O(n/k)$ stripes each) and they satisfy the size invariant of order k (Definition 2.1) on their number of matrices of dimension $(d-1)$. In this way, a stripe contains still $O(k)$ points. By intersecting these d sets of stripes, we obtain a partition of $\mathcal{M}$ into $O((n/k)^d)$ basic squares, such that each basic square overlaps no more than k matrices of dimension $(d-1)$ in each direction. We keep the items in a basic square sorted by a $\prec_p$-order to be specified and we build a cross-tree $\mathrm{CT}(T_1 \times T_2 \times \cdots \times T_d)$ on the top of these basic squares, where T_i is a weight-balanced B-tree whose leaves are in one-to-one correspondence to the stripes for the ith coordinate ($1 \leqslant i \leqslant d$). We then store the solutions in the cross-tree nodes like in Section 2.1. As $|T_i| = O(n/k)$, we can use an argument analogous to the one presented in Theorem 2.7 to obtain the following lemma.

LEMMA 4.1. *We can insert in, delete from, concatenate, and split any tree T_i in order to update cross-tree* $\mathrm{CT}(T_1 \times T_2 \times \cdots \times T_d)$ *in the time bounds*:

- $O(f(n) + (n/k)^{d-1})$ *when* $f(n) = \Omega(n)$;
- $O(f(n)(n/k)^{d-1})$ *otherwise.*

Theorem 2.13 for two orders can be smoothly generalized to d orders.

THEOREM 4.2. *The splitting and merging problem on n items undergoing $d>1$ total orders can be solved with time bounds for a slack parameter k ($1 \leqslant k \leqslant n$) and an operator cost $f(n)$*:

(a) *Operations* range, h_split, v_split, h_concatenate, v_concatenate *take time*:

- $O(f(n) + (n/k)^{d-1} + P(k) + P(n)/n)$ *when* $f(n) = \Omega(n)$;
- $O(f(n)(n/k)^{d-1} + P(k) + P(n)/n)$ *otherwise.*

(b) *Inserting and deleting a point with operation* set *takes time*:

- $O(U(k) + f(n)(1 + n^{d-1}/k^d) + P(n)/n)$ *when* $f(n) = \Omega(n)$;
- $O(U(k) + f(n)(\log(n/k) + n^{d-1}/k^d) + P(n)/n)$ *otherwise.*

(c) *The total memory space is*:

- $O(S(n) + (n^d/k^d) + f(n)\log(n/k))$ *when* $f(n) = \Omega(n)$;
- $O(S(n) + f(n)(n^d/k^d))$ *otherwise.*

(d) *The total preprocessing time is*:

- $O(n \log n + P(n) + (n^d/k^d) + f(n)\log(n/k))$ *when* $f(n) = \Omega(n)$;
- $O(n \log n + P(n) + f(n)(n^d/k^d))$ *otherwise.*

Proof. Splitting and merging the cross-tree along one coordinate take $O(f(n) + (n/k)^{d-1})$ time when $f(n) = \Omega(n)$ and takes $O(f(n)(n/k)^{d-1})$ time otherwise by Lemma 4.1. Scanning a stripe takes $O(k)$ time for retrieving its $O(k)$ items plus $O((n/k)^{d-1})$ time for traversing the cross-tree. Processing a stripe therefore costs $O(f(n) + (n/k)^{d-1} + P(k))$ time when $f(n) = \Omega(n)$ and takes $O(f(n)(n/k)^{d-1} + P(k))$ time otherwise. We use these facts to analyze the operations stated in the theorem and note that they are implemented by the algorithms described in Section 2.2. For example, the term $(n^{d-1}/k^d) f(n)$ in the complexity for *set* is due to a stripe overflow or underflow which requires splitting and merging the cross-tree every $\Theta(k)$ insert operation. Consequently, we can analyze the d-dimensional case by following the same method employed for the two-dimensional case. ∎

Theorem 4.2 finds applications to the d-dimensional versions of some order-decomposable problems. For example, we examine the multidimensional 2-3-trees that generalize those presented in Section 3.2. In [17], van Kreveld and Overmars provide the following time bounds: $O(\log n)$ for an insertion or a deletion; $O(n^{1-1/d} \log^{1/d} n)$ for a split or concatenate; $O(n^{1-1/d} \log^{1/d} n + occ)$ for a range search, where *occ* is the number of points retrieved by the search. By setting $k = \lceil n^{1-1/d} \rceil$ in Theorem 4.2, we obtain slightly better bounds.

THEOREM 4.3. *A d-dimensional 2-3-tree storing n d-dimensional points can be maintained with the bounds*:

- *a point insertion or deletion*: $O(\log n)$;
- *a split or a concatenate along one coordinate*: $O(n^{1-1/d})$;
- *a range search*: $O(n^{1-1/d} + occ)$, *where occ is the number of points reported by the search.*

The space required is $O(n)$ and the preprocessing time is $O(n \log n)$.

5. CONCLUSIONS

In this paper, we presented a general technique for solving some order-decomposable problems which generalize to dimension $d > 1$ the notion of concatenable data structures such as 2-3-trees. With our technique we are able to improve several previously known results on order-decomposable problems under splits and concatenates. A large body of order-decomposable problems have been investigated in several areas, such as basic problems (e.g., member searching, predecessor, ranking), computational geometry (e.g., neighbor queries, union and intersection queries,

visibility queries), database applications (e.g., partial match queries, range queries), and statistics (e.g., maxima queries). The reader is referred to Appendix I in [8], Chap. II in [30], and Chap. VII of [26] for a discussion of decomposable problems, where our technique is also applicable. One interesting question is whether our technique could be extended so as to handle copy-and-paste operations, where a rectangular region of items can be copied from one part to another of the input space.

ACKNOWLEDGMENTS

We are indebted to Amnon Nissenzweig and to Giuseppe Persiano for many delightful conversations at the beginning of this research. We are grateful to Lars Arge, Paolo Ferragina, and Renzo Sprugnoli for helpful technical discussions and to Marc van Kreveld and Mark Overmars for sending us a copy of [19]. We wish to thank Peter van Emde Boas and the anonymous referees for spotting some inaccuracies in an earlier draft and for their suggestions which greatly improved the presentation of this paper.

Received April 22, 1997; final manuscript received June 8, 1999

REFERENCES

1. Aggarwal, A., Guibas, L., Saxe, J., and Shor, P. W. (1989), A linear-time algorithm for computing the Voronoi diagram of a convex polygon, *Discrete Comput. Geom.* **4**, 591–604.
2. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, MA.
3. Arge, L. (1997), personal communication.
4. Arge, L., and Vitter, J. S. (1996), Optimal dynamic interval management in external memory, *in* "37th IEEE Symp. on Foundations of Computer Science."
5. Bentley, J. L. (1975), Multidimensional binary search trees used for associated searching, *Commun. ACM* **19**, 509–517.
6. Bentley, J. L. (1979), Decomposable searching problems, *Inform. Process. Lett.* **8**, 244–251.
7. Bentley, J. L. (1980), Multidimensional divide-and-conquer, *Commun. ACM* **23**, 214–229.
8. Bentley, J. L., and Saxe, J. L. (1980), Decomposable searching problems I. Static-to-dynamic transformation, *J. Algorithms* **1**, 301–358.
9. Chiang, Y.-J., and Tamassia, R. (1992), Dynamic algorithms in computational geometry, *Proc. IEEE* (special issue on computational geometry, G. Toussaint, Ed.) **80**, 1412–1434.
10. Dobkin, D., and Suri, S. (1991), Maintenance of geometric extrema, *J. Assoc. Comput. Mach.* **38**, 275–298.
11. Edelsbrunner, H. (1983), A new approach to rectangle intersection, Part I, *Int. J. Comput. Math.* **13**, 209–219.
12. Edelsbrunner, H., and Overmars, M. H. (1985), Batched dynamic solutions to decomposable searching problems, *J. Algorithms* **6**, 515–542.
13. Finkel, R. A., and Bentley, J. L. (1974), Quad-trees: A data structure for retrieval of composite keys, *Acta Inform.* **4**, 1–9.
14. Grossi, R., and Italiano, G. F. (1997), Efficient splitting and merging algorithms for order decomposable problems, *in* "Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP 97), Bologna, Italy, July 7–11," pp. 605–615, Lecture Notes in Computer Science, Springer-Verlag, Berlin.
15. Grossi, R., and Italiano, G. F. (1999), Efficient cross-trees for external memory, *in* "External Memory Algorithms and Visualization" (J. Abello and J. S. Vitter, Eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS.

16. Gowda, I. G., and Kirkpatrick, G. F. (1980), Exploiting linear merging and extra storage in the maintenance of fully dynamic geometric data structures, *in* "Proc. 19th Allerton Conference on Communication, Control and Computing," pp. 1–10.

17. van Kreveld, M. J., and Overmars, M. H. (1991), Divided k-d trees, *Algorithmica* **6**, 840–858.

18. van Kreveld, M. J., and Overmars, M. H. (1993), Union-copy structures and dynamic segment trees, *J. Assoc. Comput. Mach.* **40**, 635–652.

19. van Kreveld, M. J., and Overmars, M. H. (1994), Concatenable structures for decomposable problems, *Inform. Comput.* **110**, 130–148.

10. van Leeuwen, J., and Maurer, H. A. (1980), "Dynamic Systems of Static Data Structures," Technical Report 42, Technical University Graz.

21. van Leeuwen, J., and Overmars, M. H. (1981), The art of dynamizing, *in* "Proc. 10th Mathematical Foundations of Computer Science," Lecture Notes in Comput. Sci., Vol. 118, pp. 121–131, Springer-Verlag, New York/Berlin.

22. van Leeuwen, J., and Wood, D. (1980), Dynamization of decomposable searching problems, *Inform. Process. Lett.* **10**, 51–56.

23. Maurer, H. A., and Ottmann, T. A. (1979), Dynamic solutions of decomposable searching problems, *in* "Discrete Structures and Algorithms" (U. Pape, Ed.), pp. 17–24, Hanser Verlag, Vienna.

24. McCallum, D., and Avis, D. (1979), A linear algorithm for finding the convex hull of a simple polygon, *Inform. Process. Lett.* **9**, 201–206.

25. Mehlhorn, K. (1981), Lowerbounds on the efficiency of transforming static data structures into dynamic structures, *Math. Systems Theory* **15**, 1–16.

26. Mehlhorn, K. (1984), "Multi-Dimensional Searching and Computational Geometry," EATCS Monographs on Theoretical Computer Science, Vol. 3, Springer-Verlag, Berlin/New York.

27. Mehlhorn, K., and Overmars, M. H. (1981), Optimal dynamization of decomposable searching problems, *Inform. Process. Lett.* **12**, 93–98.

28. Nievergelt, J., and Reingold, E. M. (1973), Binary search trees of bounded balance, *SIAM J. Comput.* **2**, 33–43.

29. Overmars, M. H. (1973), Dynamization of order decomposable set problems, *J. Algorithms* **2**, 245–260.

30. Overmars, M. H. (1983), "The Design of Dynamic Data Structures," Lect. Notes in Comput. Sci., Vol. 156, Springer-Verlag, Berlin/New York.

31. Overmars, M. H., and van Leeuwen, J. (1981), Maintenance of configurations in the plane, *J. Comput. System Sci.* **23**, 166–204.

32. Overmars, M. H., and van Leeuwen, J. (1981), Dynamization of decomposable searching problems yielding good worst-case bounds, *in* "Proc. 5th GI Conference on Theoretical Computer Science," Lect. Notes in Comput. Sci., Vol. 104, pp. 224–233, Springer-Verlag, New York/Berlin.

33. Overmars, M. H., and van Leeuwen, J. (1981), Some principles for dynamizing decomposable searching problems, *Inform. Process. Lett.* **12**, 49–53.

34. Overmars, M. H., and van Leeuwen, J. (1981), Two general methods for dynamizing decomposable searching problems, *Computing* **26**, 155–166.

35. Overmars, M., and van Leeuwen, J. (1981), Worst-case optimal insertion and deletion methods for decomposable searching problems, *Inform. Process. Lett.* **12**, 168–173.

36. Overmars, M., and van Leeuwen, J. (1982), Dynamic multi-dimensional data structures based on quad- and k-d trees, *Acta Inform.* **17**, 267–285.

37. Rao, N.S.V., Vaishnavi, V. K., and Iyengar, S. S. (1988), On the dynamization of data structures, *BIT* **28**, 37–53.

38. Samet, H., Bibliography on quad-trees and related hierarchical data structures, *in* "Data Structures for Raster Graphics" (L. Kessenaar, F. Peters, and M. van Lierop, Eds.), pp. 181–201, Springer-Verlag, Berlin.

39. Scholten, H. W., and Overmars, M. H. (1989), General methods for adding range restrictions to decomposable searching problems, *J. Symbolic Comput.* **7**, 1–10.

40. Shamos, M. I., and Hoey, D. (1975), Closest-point problems, *in* "16th IEEE Symp. Foundations of Computer Science," pp. 151–162.

41. Smid, M. (1990), Algorithms for semi-online updates on decomposable problems, *in* "Proc. 2nd Canadian Conference in Computational Geometry," pp. 347–350.

42. Willard, D. E., and Lueker, G. S. (1985), Adding range restriction capability to dynamic data structures, *J. Assoc. Comput. Mach.* **32**, 597–617.