A Class of Linear Algorithms to Process Sets of Segments [*]

Gonzalo Navarro   Ricardo Baeza-Yates

Department of Computer Science
University of Chile
Blanco Encalada 2120
Santiago - Chile
{gnavarro,rbaeza}@dcc.uchile.cl

Abstract

We address the problem of efficiently performing operations on sets of segments. While current solutions to operate segments focus on single operations (e.g. insertion or searching), we are interested in set-oriented operations (e.g. union, difference and others more specific to segments). In those cases, extending the current approaches leads to $O(n \log n)$ time complexity to manipulate sets of n elements. We show that a wide class of operations can in fact be performed in $O(n)$ time, i.e. in a constant amortized cost per processed segment. We present the general framework and show a number of operations of that kind, depicting and analyzing the algorithms. Finally, we show some applications of this technique.

[*]This work has been supported in part by FONDECYT grants 1940271 and 1950622.

1 Introduction

In many practical applications the problem of manipulating segments arises under different forms. Typical examples are computational geometry [14, 11, 6, 3], temporal databases [7, 2], database models with constraints [9, 8] and structured text search [13, 10].

Because of this situation, the problem of manipulating a set of segments has been extensively studied [14, 5]. All these approaches focus on "single" operations, in which a single segment is operated against a set of segments. Examples are: searching for a segment, inserting a new segment into the set, removing a segment from the set, etc. In [5], it is shown how to achieve $O(\log n)$ behavior for these operations.

However, for some applications, that type of operations are less important, while more "set-oriented" operations are needed. This is the case, for example, of set-oriented query languages for structured text or temporal databases. Examples of the operations that are commonly performed in these applications are retrieving segments including other segments (e.g. all chapters containing at least four figures, in structured text search), or all segments from a set shortly preceding a segment from another set (e.g. all musical scenes where a given actor appears that shortly precede a colorful scene, in a temporal database describing movies).

A trivial extension of the current approaches to deal with these requirements leads to $O(n \log n)$ solutions for n elements. Our aim is to show that under quite general assumptions, much better solutions can be found.

We show that in many cases, a solution similar to list merging [1] can be applied to sets of segments, thus leading to $O(n)$ algorithms. Therefore, we extend this simple mechanism to deal with a more complex data structure, and analyze under which situations the idea works.

An important consideration is that we support nesting in the segments forming a set, since if no nesting is allowed, the segments are trivially linearly ordered, and it is easy to develop linear algorithms [4].

The main contribution of this paper is a simple technique to perform $O(n)$ set operations in $O(n)$ amortized time, while the cost of an individual operation is known to be $\Omega(\log n)$ in the worst case. An earlier but more detailed version of part of this work and its application to structured text retrieval can be found in [12, 13].

This paper is organized as follows. In section 2 we explain our model of operation. In section 3 we explain the general scheme of our solution. In section 4 we show a number of problems for which we found linear algorithms. In section 5 we show some complex operations for which we found more expensive solutions. In section 6 we show some applications using these algorithms. Finally, in section 7 we present our conclusions and future work directions.

2 Preliminaries

A *segment* is a pair $\langle x, y \rangle$, where x and y are real numbers and $x \leq y$. We define $From\langle x, y \rangle = x$ and $To\langle x, y \rangle = y$. A segment $a = \langle x, y \rangle$ is said to *contain* another one $b = \langle x', y' \rangle$ (and we denote it as $a \supseteq b$ or $b \subseteq a$) iff $x \leq x' \ \wedge \ y' \leq y$. If a segment contains another one we say that they *nest*. The segments are said to be *disjoint* iff $y < x' \ \vee \ y' < x$ (we say $a < b$ in the first case and $a > b$ in the other). If two segments do not nest and are not disjoint we say that they *overlap*. Finally, we use the equal sign between segments with the obvious meaning, and $a \subset b$ or $b \supset a$ to denote $a \subseteq b \ \wedge \ a \neq b$.

Our general problem is: *given two sets of segments A and B, obtain a new set C by processing A and B in some way, in $O(|A| + |B|)$ time.*

To achieve that goal, we impose two further restrictions, one on the sets and one on the processing operations:

- Each set must form a *hierarchy*, i.e. no overlaps are allowed between any two segments of a given set. This is because the only way we could obtain linear algorithms for sets with overlapping segments was preventing nesting inside the set. In this case the segments can be trivially ordered by their first or last extreme, and the normal list merging algorithms work [4]. However, in some applications (e.g. structured text search) nesting can be more important than overlapping, so we are interested in providing nesting.

- We are interested in set-oriented operations, to which we impose the additional restriction of operating with *proximal* segments. Proximality means that the presence or absence of a given segment in the final result must be defined in terms of relatively close segments in the arguments. This is because we plan to traverse the arguments in synchronization to produce the results.

Observe that we do not need that the union of both operands forms a hierarchy, as long as the result does. That is, our algorithms work also if segments from different arguments overlap.

Our approach to the solution can be defined in general terms as follows:

- We use a tree data structure to arrange the segments of each set. Since there are no overlaps, we define the tree by the containment relations between segments.

- To obtain the solution set (which is also arranged in a tree), we are going to traverse both operand trees simultaneously, in a synchronized way, while we generate the solution tree at the same time. The idea is to generalize the list merging algorithms, to make them operate on trees and select elements under more complex criteria. To be able to generate the whole solution by traversing the operands just once, it is necessary the assumption of proximal segments. All the algorithms consist of variations of this idea.

- We are not interested in how the sets are built in the first place, and how they are finally used. Our scheme is not so efficient to build the trees by consecutive random insertions, but in the applications we are interested in, those problems are solved in an ad-hoc way (we show an example in section 6).

In what follows, we describe more in detail our data structure and algorithmic scheme.

3 A Solution Scheme

3.1 Data Structure

As said, we arrange the set of segments in a tree. The criterion to define the tree is straightforward: a segment a descends from another one b in the tree iff $a \subset b$. Although for clarity we do not allow repetitions, the algorithms are easily modified to account for this.

A formal definition of our type $Tree$ follows:

$$\begin{aligned}
&Tree = Subtree^* \\
&Subtree = Segm \times Tree \\
&Segm = \{\langle x, y\rangle / x, y \in \mathcal{R} \ \wedge x \leq y\} \\
&\forall (s, ((s_1, t_1), ..., (s_k, t_k))) \in Subtree, \forall i \in 1..k, s \supseteq s_i \\
&\forall (s_1, t_1), ..., (s_k, t_k) \in Tree, \forall i \in 1..k-1, s_i < s_{i+1}
\end{aligned}$$

where $\mathcal{R}$ is the set of real numbers. As it can be seen, the root of our tree does not have an associated segment. Our trees can be seen in fact as forests with order among their trees.

We define some functions to access this tree type:

- $node : Subtree \rightarrow Segm$ and $subtree : Subtree \rightarrow Tree$ are selectors, i.e. if $S = (s, ((s_1, t_1), ..., (s_k, t_k))) \in Subtree$, then $node(S) = s$ and $subtree(S) = ((s_1, t_1), ..., (s_k, t_k))$.
- $head : Subtree^+ \rightarrow Subtree$ returns the first element of the list, i.e. $head(\{l_1, ..., l_k\}) = l_1$.
- $tail : Subtree^+ \rightarrow Subtree^*$ eliminates the first element, i.e. $tail(\{l_1, ..., l_k\}) = \{l_2, ..., l_,l_k\}$.
- $\lambda \in Tree$ denotes an empty tree.

Although we do not consider any particular representation, the data structure for our trees must allow efficient computation of these access functions.

3.2 Algorithmic Scheme

The general form of our algorithms consists of traversing both trees, normally in pre or postorder. At each step, the current nodes of both trees are compared, and depending on the results and the operation, we move in one or both trees, going to the next node or jumping directly to the next sibling (thus skipping the current subtree). Each particular case is a variation of this general idea. We show a number of examples in the next section.

Thus we are going to describe the algorithms by marking nodes that must be selected or rejected from the final solution (almost all useful operations select elements from only one operand). This marking can be a simple boolean mark or it may have a more complex meaning. This provides a good abstraction, since we do not detail how we collect marked or delete unmarked nodes. Moreover, the marking algorithm can be used for two complementary operations, depending on whether we select or reject the marked nodes. Some algorithms can be solved without marking, though.

We describe now an important abstraction related to the way we traverse the trees. Unlike list traversal, which is trivial, we use two types of traversal operations here, one to "go right" and other to "go down" in the tree. Since several trees are traversed at the same time and it is wasteful to store pointers to parents, we keep explicit stacks to implement these two traversals.

- At the beginning of each algorithm, we initialize an empty stack for each argument. This stack holds pointers to $Trees$, and is referred to as $\langle argument\rangle.stack$. We use the normal operations $Push$, Pop, $Empty$ and Top on the stacks
- After initializing the stack, if the tree is not empty, we push the first top-level node of the tree into the stack.

- The algorithms access only the top of the stacks and terminate when a stack is empty. Argument tree names are uppercase letters. Their lowercase version denotes the top nodes of their corresponding stacks, e.g. $p = head(Top(P.stack))$. If $P.stack$ is empty, p is assumed to be $\langle\infty,\infty\rangle$ (where $\infty = \infty$, $\infty > x, \forall x \in \mathcal{R}$, and $\infty + x = \infty, \forall x \in \mathcal{R}$).
- To go down in the tree, we test whether the current top of the stack has children or not. If it does, we push its first child into the stack. If it does not, we perform a "go right" operation. In fact, going down means "process first the children and then the siblings".
- To go right, we replace the current top of the stack by its next sibling. If there is no next sibling, we pop the current top and retry the operation with the parent. We eventually empty the stack in this way.

We use a pseudocode notation for our algorithms. We include a `case`-like instruction (a big left brace). Figure 1 shows the basic algorithms for tree traversal.

```
Init($P$)

Empty($P.stack$).
If ($P \neq \lambda$) Push($P.stack, P$).

Down($P$)

If ($subtree(p) \neq \lambda$) Push($P.stack, subtree(p)$)
else Right($P$).

Right($P$)

While ($\neg Empty(P.stack) \ \wedge \ tail(Top(P.stack)) = \lambda$) Pop($P.stack$).
If ($\neg Empty(P.stack)$) $Top(P.stack) \leftarrow tail(Top(P.stack))$.
```

Figure 1: Basic operations for tree traversal.

We use the following numbers in the analysis: n_X is the size of the set corresponding to operand X, h_X is the height of its tree (in the worst case it can be n_X) and d_X is the maximum degree of its tree (it can also be n_X in a flat tree). We also use n, d and h as the maximum corresponding value between all operands (there are normally two operands).

Two observations about the analysis:

- It should be clear that either collecting marked or deleting unmarked nodes is $O(n)$ time, where n is the number of nodes of the tree.
- Although a particular operation of tree traversal can work up to $O(h)$, we note that the whole traversal, even by using `Down`, is $O(n)$. This is because no edge of the tree is traversed more than twice. So, the amortized cost of tree traversals is always linear with the size of the tree.

We are now in position to describe a number of example algorithms. Their description is very simple with these primitives.

4 Linear Operations

There are a number of interesting problems that admit a linear implementation, for example: set manipulation, segments including or included in others, segments after or before others, etc. In this section we explain in detail a couple of them and their analysis.

4.1 Set Difference and Intersection

Set difference and intersection are complementary versions of a single marking algorithm. The idea is that the trees are unmarked at the first place, and we traverse both trees in synchronization, marking all nodes of the first argument that are also in the second one. Thus, we later implement set difference by collecting unmarked nodes and set intersection by collecting marked nodes.

Figure 2 shows the algorithm to mark the tree. P and Q are the arguments.

```
Init($P$).  Init($Q$).
While $max(To(p), To(q)) < \infty$
    $p < q$ :       Right($P$).
    $p > q$ :       Right($Q$).
    $p = q$ :       Mark $p$.  Down($P$).
    $p \subset q$ : Down($Q$).
    $else$ :        Down($P$).
```

Figure 2: Marking algorithm for set difference or intersection.

This algorithm is linear, since a single traversal is done on each argument (that traversal has been already shown linear), and we work $O(1)$ at each step. Therefore, we have $O(n_P + n_Q) = O(n)$ time. It is also $\Omega(n)$ in the worst case.

4.2 Segments Included in Others

Another interesting operation is `In(`P, Q`)`, that selects elements from P that are included in a segment of Q.

The algorithm to solve `In(`P, Q`)` is presented in Figure 3. We traverse the top levels of P and Q, in synchronism. When a node of P is included in a top-level node of Q, that subtree of P is marked. Otherwise we discard that P node and continue with its children.

To avoid too much marking overhead, we state that a mark in a node means that not only that node but also its whole subtree is considered marked. The collection algorithm must be modified to account for this.

This algorithm is $O(n_P + n_Q) = O(n)$, by the same arguments as before. In this case, we can refine this analysis as follows: each time we do a `Down(`P`)` is because it contains an element of Q or because it overlaps with an element of Q, thus each extreme of each segment of (the top-level of) Q is compared, at most, with a complete path of P (length h_P). That is because once we descend the first level in P, the relevant list from the top-level of Q has only one element (the original q). At each level of this path, the operation can take us d_P comparisons, thus the cost is $O(d_Q h_P d_P) = O(d^2 h)$. That means, for example, that in an application with constant d and balanced trees the operation takes $O(\log n)$ (at least to do the marking).

```
Init($P$).  Init($Q$).
While $max(To(p), To(q)) < \infty$
    $p < q$ :    Right($P$).
    $p > q$ :    Right($Q$).
    $p \subset q$ :    Mark $p$.  Right($P$).
    $else$ :    Down($P$).
```

Figure 3: Marking algorithm for segments included in others.

Then, the complexity of this operator is $O(\min(n_P + n_Q, d_Q h_P d_P)) = O(\min(n, d^2 h))$.

5 Complex Operations

Although most operations can be implemented in linear time, there are more complex operations that seem not to have a linear implementation: segments included in others with positions, segments including k segments, and complex versions of "after" and "before" are some examples. We explain in detail the first one.

5.1 Segments Included in Others with Positions

An operation that happens to be useful for applications is to select segments from P that are included in some segment q of Q *at a given position*. The position is defined in terms of the other segments of P that are included under the same q (e.g. the third section of each chapter). Since there may be segments included in q that nest in P (e.g. sections inside sections), we define "position" by only considering the maximal segments of P included in q, for each Q. If a segment of P overlaps with q, the segment and its descendants in P are not considered (see Figure 4). Other criteria produce slightly different algorithms.

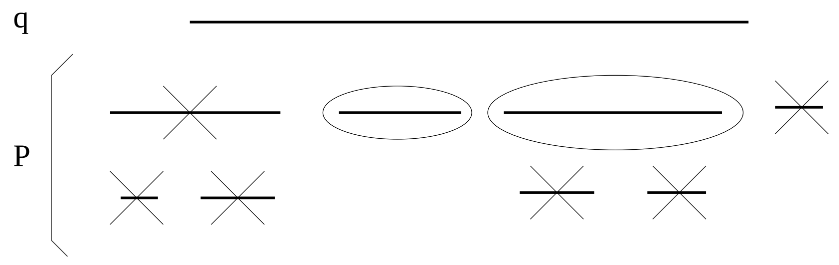

Figure 4: Criterion to participate in inclusion with positions. Ellipses indicate selected segments.

Another concern is which language will we use to denote the allowed positions of an included node. We can use, e.g. first, k-th, last, last$-k$, prime positions, etc. For this algorithm we use a generic predicate to avoid any restriction: s is an expression denoting the allowed positions.

The algorithm requires simple boolean marking. We traverse both trees in synchronization. When we find a set of nodes of P included in one of Q, we mark the s-th nodes, and then we pass again over the included P-nodes, this time comparing them with the subtree of the Q-node. If, instead, the node of P includes one of Q, we follow the children of the P-node. Figure 5 shows the algorithm.

```
Init($P$).  Init($Q$).
While $max(To(p), To(q)) < \infty$
    $p < q$ :       Right($P$).
    $p > q$ :       Right($Q$).
    $p \subset q$ : $lp \leftarrow Top(P.stack)$.   $pos \leftarrow 1$.
                    While $head(lp) \subset q$
                        If ($pos \in s$) Mark $head(lp)$.
                        $lp \leftarrow tail(lp)$.   $pos \leftarrow pos + 1$.
                    Down($Q$).
    $else$ :        Down($P$).
```

Figure 5: Marking algorithm for segments included in others at a given position.

To analyze this algorithm, consider that we can traverse both P and Q completely, and that the final collection of nodes is $O(n_P)$. Observe that each element of Q is deleted from the problem by doing at most $O(d_P)$ work (when $p \subset q$), thus the algorithm is $O(n_P + n_Q d_P)$. But also observe that each element of P can be worked on by at most a complete path of Q, thus the algorithm is also $O(n_Q + n_P h_Q)$. Then, the algorithm has the best from both complexities, namely $O(\min(n_P + n_Q d_P, n_Q + n_P h_Q)) = O(n \min(d, h))$.

To see that it is also $\Omega(n^2)$, consider the following example: In($\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}, \{\langle 1,n\rangle, \langle 2,n\rangle, ..., \langle n,n\rangle\}, s$).

This non-linear complexity may be surprising, since the requirement of proximality seems to be met. But observe that in order to determine whether a P node is to be marked or not, we have to consider many nodes of Q, not just one, and there seems to be no better way to do that. If the language of positions (s) is restricted we can obtain better complexities, e.g. if we allow to express only ranges we can implement it in $O(n \min(h, k \log d))$ where k is the number of ranges we have.

6 Some Applications

Our technique applies to a number of dissimilar applications. In this section we briefly discuss a temporal database and explain in detail a structured text search application. Our aim is not only to expose real situations where the problem arises, but also to show the practical performance of our algorithms.

6.1 Temporal Databases

Temporal databases [7, 2] can manipulate information with temporal validity. Their data is based on events that occur at a given point or interval in time. If we have a set-oriented query language we will be interested in questions such as "give me all the events (and their times) that satisfy some constraint" (an example is given in the introduction).

In fact, temporal databases are much more complex, since they also deal with uncertainty in time (e.g. A happened before B, without knowing when). We just want to show that if a database has to manage a number of quantitative facts (i.e. events that are known to have happened at known times), it may be interesting for its back-end to perform set-oriented operations on these facts. These algorithms would be a part of a more general inference engine, being an efficient way to select relevant facts to work on.

In most temporal databases, the time intervals can overlap and nest. As explained earlier, we cannot handle results (final or intermediate) having segment overlapping. Therefore, this is a restriction we impose on the applicability of our model to temporal databases.

This does not mean that there cannot be overlaps in the database. We can divide the knowledge into a set of sequential processes, and subsets of different processes can be combined into operations, although each answer and intermediate result is a subset of some sequential process. Those sequential processes can be hierarchically structured.

6.2 Structured Text Search

Modern textual databases are a good example of hierarchical structuring. We describe here a specific model to query structured text databases, called Proximal Nodes [13, 12]. The development of this model has motivated this work on efficient algorithms to evaluate queries, although some operations differ slightly from those exposed here.

In this model, a textual database is seen as a text (a sequence of symbols) plus a number of independent hierarchical structures built on the text. Each hierarchy is strict, although there may be overlaps between different hierarchies. The nodes of the hierarchies are the structural components of the database (e.g. chapters, pages, etc.). Examples of hierarchical views of the text are: chapter/section/paragraph and fascicle/page/line. In this case, it makes full sense for answers to be a subset of a given hierarchy.

The leaves of the query syntax trees are names of structural components and text pattern-matching expressions. Each structural component (e.g. chapters, figures, pages) is preindexed, so the set of all elements of a given type can be retrieved in a time proportional to the size of the retrieved set. Pattern-matching uses another index to return a list of segments of the text that matched the pattern. These lists are considered to be part of a special hierarchy. This is how the sets are built in the first place. All internal nodes of query syntax trees correspond to operations between segments of the operands, and they are further restricted to operate only on proximal segments.

It is shown that the query language obtained under these restrictions has good expressivity, while the implementation (using the algorithms we expose here) is very efficient. The model constitutes a good tradeoff between the models which achieve high efficiency by strongly restricting the structuring possibilities and the query language of the database and those which have rich structuring and querying facilities but a high execution overhead.

Some examples of queries are:

- I want text paragraphs in italics which are before (but in the same page) a figure that says something about the earth.
- I want a paragraph preceding another paragraph where the word "Computer" appears before (at 10 symbols or less) the word "Science". Both paragraphs must be in the same page.
- Give me all references to Knuth's books in chapters 2-4.

- I want all sections with mathematical formulas that are not appendices.

We also implemented a lazy version of the algorithms, that only processes the nodes that are needed for the final result (thus saving a lot of work when processing the intermediate results). This mechanism can also be used to implement a navigational interface, in which the user sees the top-level of the result tree and asks to expand only some nodes. In this case, we save the work of computing nodes that are not to be seen. Although the worst-case complexity of the lazy operations is worse than that of "full" operations, the real times are better, since less nodes are processed in practice.

A prototype has been implemented for this model, to test the average performance of the algorithms. We used a database of C programs and Latex documents, whose structuring is quite different. We tested each operation on different operands of sizes (i.e. number of nodes) ranging from 100 to 10000. We also tested a number of more complex queries, to compare the full and lazy versions on real queries.

Figure 6 shows a typical example of the times of an algorithm (i.e. a single operation). They correspond to a Sun SparcClassic of approximately SpecMark 26 and 16 Mb of RAM. From the tests we extract the following conclusions (they may differ for other applications):

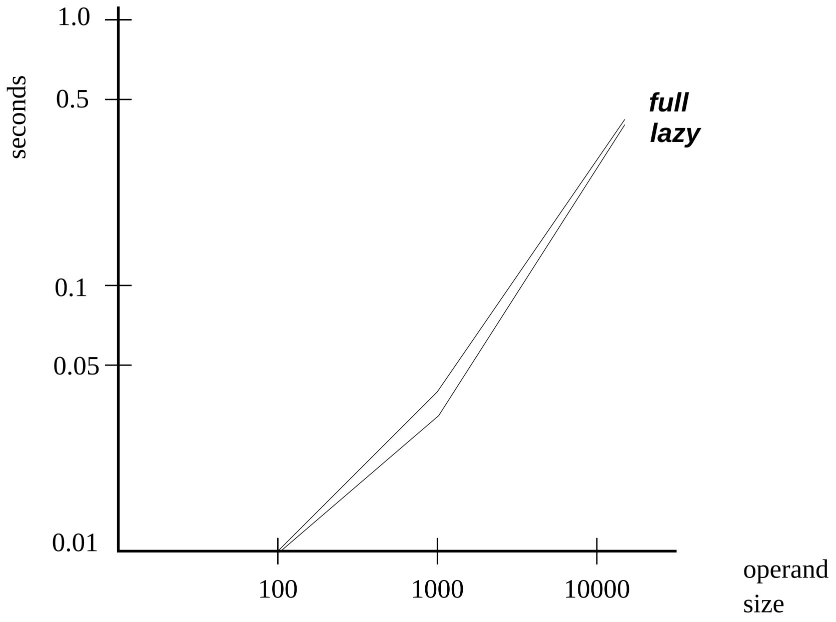

Figure 6: Typical times for equal-sized operands. Observe that we use a logarithmic scale.

- The full versions of the algorithms are all linear in practice, since the situations under which they are not are very unlikely to occur in structured text databases (e.g. a very deep tree).

- The full versions have very low variance, being their times highly predictable, proportional to the sum of the sizes of all intermediate results. The constant for our machine is approximately 50.000 nodes processed per second per operator.

- The lazy version is normally better than the full one in practice, despite the worse complexities, especially for complex queries. This is because less nodes are expanded. Lazy algorithms have very large variance, though.

7 Conclusions and Future Work

We analyzed the problem of manipulating a set of segments when we are interested in set-oriented operations. Classical solutions provide simple operations with $O(\log n)$ complexity, which leads to $O(n \log n)$ solutions for set-oriented operations.

However, we show that, under some general assumptions, set-oriented operations can be performed in linear time. These assumptions are: the operands and the result must not have overlapping segments inside, and the operations must work on proximal segments.

We developed a framework oriented to tree traversal that generalizes the idea of list traversal for merge-like operations, and applied the framework to solve in linear time a number of set-oriented example operations on segments. The aim was to show that the framework is flexible enough to be adapted to a number of apparently dissimilar problems. This technique works well for set-oriented operations because of the amortized cost, i.e. its performance for single operations is not good.

Finally, we presented some applications to which this technique could be applied. We also include average times of a real implementation of the algorithms.

There are a number of future work directions related to this work. The most important are:

- Extend these ideas to allow overlaps into a set, since this will open a wealth of new possibilities to apply this technique.
- Find a technique that keeps these good results while improving the performance for single operations.
- Search for more operations that can be implemented in linear time by using this framework, specially by looking at applications that can benefit from this work.
- Study a disk-based implementation, trying to minimize seek times. Some work on this direction has already been done.
- Extend the framework to account for more dimensions, e.g. manipulating hypercubes instead of just one-dimensional segments. This may open a number of opportunities of applications in computational geometry and in map processing (e.g. in geographic information systems).
- Study a parallel implementation, since our algorithms seem to be highly parallelizable.

Acknowledgments

We thank Claudia Medeiros and Nina Edelweiss for their help on temporal databases. We also thank the helpful comments of the referees.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning.* Morgan Kaufmann, 1990.

[3] J. Bentley. Algorithms for Klee's rectangle problems. Dept. of Computer Science, Carnegie-Mellon Univ. Unpublished notes., 1977.

[4] C. Clarke, G. Cormack, and F. Burkowski. Schema-independent retrieval from heterogeneous structured text. In *Procs. of the 4th Annual Symposium on Document Analysis and Information Retrieval*, Apr. 1995.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[6] H. Edelsbrunner. Dynamic data structures for orthogonal intersection. Technical Report F59, Tech. Univ. Graz, Institute für Informationsverarbeitung, 1980.

[7] A. T. et al. *Temporal Databases: Theory, Design and Implementation.* Benjamin Cummings, 1993.

[8] P. Kanellakis and D. Goldin. Constraint programming and database query languages. Technical Report CS-94-31, Dept. of Computer Science, Brown University, June 1994.

[9] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. Vitter. Indexing for data models with constraints and classes. Technical Report CS-93-21, Dept. of Computer Science, Brown University, May 1993.

[10] A. Loeffen. Text databases: A survey of text models and systems. *ACM SIGMOD Conference. ACM SIGMOD RECORD*, 23(1):97–106, Mar. 1994.

[11] E. McCreight. Priority search trees. Technical Report CSL-81-5, Xerox PARC, 1981.

[12] G. Navarro. A language for queries on structure and contents of textual databases. Master's thesis, Dept. of Computer Science, Univ. of Chile, Apr. 1995. `ftp://sunsite.dcc.uchile.cl/-pub/users/gnavarro/thesis95.ps.gz`.

[13] G. Navarro and R. Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proc. ACM SIGIR'95*, pages 93–101, 1995. `ftp://sunsite.dcc.uchile.cl/-pub/users/gnavarro/sigir95.ps.gz`.

[14] F. Preparata and M. Shamos. *Computational Geometry.* Springer-Verlag, 2nd edition, 1988.