Efficient Lazy Algorithms for Minimal-Interval Semantics

Paolo Boldi and Sebastiano Vigna

2006
Efficient Lazy Algorithms for Minimal-Interval Semantics

Paolo Boldi*
Sebastiano Vigna
Dipartimento di Scienze dell’Informazione, Università degli Studi di Milano

Abstract

Minimal-interval semantics [3] associates with each query over a document a set of intervals, called witnesses, that are incomparable with respect to inclusion (i.e., they form an antichain): witnesses define the minimal regions of the document satisfying the query. Minimal-interval semantics makes it easy to define and compute several sophisticated proximity operators, provides snippets for user presentation, and can be used to rank documents: thus, computing efficiently the antichains obtained by operations such as logic conjunction and disjunction is a basic issue. In this paper we provide the first algorithms for computing such operators that are linear in the number of intervals and logarithmic in the number of input antichains. The space used is linear in the number of antichains. Moreover, the algorithms are lazy — they do not assume random access to the input antichains. These properties make the usage of our algorithms feasible in large-scale web search engines.

1 Introduction

Search engines are a popular way to retrieve information in the web. However, the classical problem studied by the theory of information retrieval, that of answering a query by returning the set of documents that match the information provided by the user, is complicated by the huge number of documents to be taken into consideration. On the web retrieving many relevant documents is usually not a problem — the documents are simply too many already. Precision, rather than recall (in particular, precision in the first 10—20 results) is the main issue.

A first possibility for extending the user capabilities is query expansion, an automatic or semi-automatic mechanism that tries to enrich a given query, by using for example some semantics extracted from the context, or by asking directly the user what is the intended meaning of his/her query. In this case, we start from a very simple query, perhaps expressed in some natural language and finally produce a richer query that is to be submitted to the search engine.

A different approach is that of departing from the Boolean model, and provide the user with more powerful (but understandable) operators. In this paper we concentrate on minimal-interval semantics, a semantic model that uses antichains of intervals of natural numbers to represent the semantics of a query. Each interval is a witness of the satisfiability of the query, and defines a region of the document that the query satisfies (words in the document are numbered starting from 0, so regions of text are identified with intervals of integers). For instance, a query formed by the conjunction of two terms is satisfied by the minimal intervals of the document containing both terms: minimality is guaranteed by the fact that in an antichain every pair of elements is incomparable with respect to inclusion.

This approach has been defined and studied in full extent by Clarke, Cormack and Burkowski in their seminal paper [3]. They showed that antichains have a natural lattice structure that can be used to interpret conjunctions and disjunctions in queries. Moreover, it is possible to define several

*This work is partially supported by the EC Project DELIS.
additional operators (proximity, followed-by, and so on) directly on the antichains. The authors have also described families of successful ranking schemes based on the number and length of the intervals involved.

The main feature of minimal-interval semantics is that, by its very definition, an antichain of intervals cannot contain more than \( n \) intervals, where \( n \) is the number of words in the document. Thus, it is in principle possible to compute all minimal-interval operators in time linear in the document size. This is not true, for instance, if we consider different interval-semantics approaches in which all intervals are retained and indexed (e.g., the PAT system [5] or the grep tool [6]), as the overall number of regions is quadratic in the document size.

In this paper, we attack the problem of providing efficient algorithms for the computation of such operators. As a subproblem, we can compute the proximity of a set of terms, and indeed we are partly inspired by previous work on proximity [11, 9]. Our algorithms are linear in the number of input intervals. For conjunction and disjunction, there is also a multiplicative logarithmic factor in the number of input antichains, which however can be shown to be essentially unavoidable in the disjunctive case. The space used by all algorithms is linear in the number of input antichains (in fact, we need to store just one interval per antichain). Moreover, our algorithms are lazy, that is, while building their results they do not advance the input lists more than necessary.

Previously, the only attempt at linear lazy algorithms for minimal-interval region algebras we are aware of is the work of Young–Lai and Tompa on structure selection queries [13], a special type of expressions built on the primitives “contained-in”, “overlaps”, and so on, that can be evaluated lazily in linear time. Their motivations are similar to ours — application of region algebras to very large text collections. Similarly, Navarro and Baeza–Yates [8] propose a class of algorithms that using treecreavers are able to compute efficiently several operations on overlapping regions. Their motivations are efficient implementation of structured query languages that permit such regions. Indeed, some of the techniques used therein (e.g., double stacks) are similar to our double indirect priority queues. Albeit similar in spirit, they do not provide algorithms for any of the operators we consider.

We believe that the existence of (almost) linear lazy algorithms for minimal-interval semantics makes it the natural candidate for advancing web search engines beyond a purely Boolean model: in particular, the possibility of limiting the interval width has a very natural interpretation for the user in terms of proximity, and ordered conjunction has obvious applications (e.g., searching for a verse in a song when some word is missing).

Minimal intervals can also be used together with other standard information-retrieval techniques. For instance, the Indri search engine [10] expands a query into a number of subqueries, many of which are interval-based, and combines the results.

In Section 2 we briefly introduce minimal-interval semantics, referring to the original paper for examples and motivations. The presentation is rather algebraic, and uses standard terms from mathematics and order theory (e.g., “interval” instead of “extent” as in [3]). The resulting structure is essentially identical to that described in the original paper, but our systematic approach makes good use of well-known results from order theory (for instance, we do not need to prove that antichains form a lattice, as they are a well-known representation of the ideal completion of a partial order), making the introduction self-contained. For some mathematical background, see, for instance, Birkhoff’s classic [1].

Another advantage of our presentation is that by representing abstractly regions of text as intervals of natural numbers we can easily highlight connections with other areas of computer science: antichains of intervals have been used for role-based access control [4], and for testing distributed computations [7]. The problem of computing operators on antichains has thus an intrinsic interest that goes beyond the problems of information retrieval.

Finally, we present our algorithms. First we discuss algorithms based on queues, and then greedy algorithms with backtracking. Additionally, we show how to make our algorithms slightly lazier by delaying some queue manipulation.
Figure 1: A sample text; the intervals corresponding to the semantics of the query “(hot OR cold) AND porridge AND pease” are shown. For easier reading, every other interval is dashed.

A free implementation of all algorithms described in this paper is available as a part of MG4J\(^1\).

## 2 Minimal-interval semantics

Let us denote with \(\mathcal{I}_n\) the set of intervals of \(n = \{0, 1, \ldots, n-1\}\) (a subset \(X\) of \(n\) is an interval if \(x, y \in X\) and \(x < z < y\) then \(z \in X\); note that \(\emptyset \in \mathcal{I}_n\) ordered by inclusion. By numbering words in the document starting from 0 (see Figure 1), elements of \(\mathcal{I}_n\) can be thought of as regions of text.

Given intervals \(I\) and \(J\), the interval spanned by \(I\) and \(J\) is the least interval containing \(I\) and \(J\) (in fact, the supremum in \(\mathcal{I}_n\)). Nonempty intervals will be denoted by \([\ell..r]\), where \(\ell\) is the left extreme and \(r\) is the right extreme (i.e, the smallest and largest element in the interval). Intervals are ordered by containment: when we want to order them by reverse containment instead, we shall write \(I^{\text{op}}\) (“op” stands for “opposite”).

The idea behind minimal-interval semantics \(^3\) is that every interval in \(\mathcal{I}_n\) is a witness that a given query is satisfied by a document made of \(n\) words. Smaller witnesses imply a better match, or more information; in particular, if an interval is a witness any containing interval is a witness. We also expect that more witnesses imply more information. Thus, when expressing the semantics of a query, we discard non-minimal intervals, as there are intervals that provide more relevant information. As a result, minimal-interval semantics associates with each query an antichain\(^2\) of intervals. For instance, in Figure 1 we see a short passage of text, and the antichain of intervals corresponding to a query. Note that, for instance, the interval [0..3] is not included as it is not minimal.

It is however more convenient to start from an algebraic viewpoint. An order ideal \(X\) (henceforth called just an ideal) is a subset of a partial order that is closed downwards: if \(y \leq x\) and \(x \in X\), then \(y \in X\). The ideal completion of an order \(P\) is a distributive lattice whose elements are the ideal of \(P\)

\(^1\)http://mg4j.dsi.unimi.it/

\(^2\)An antichain of a partial order is a subset of elements pairwise incomparable.
ordered by inclusion. The ideal completion of \( \mathcal{I}_n^{\text{op}} \) will be the base of our semantics:

\[
\mathcal{E}_n = \{ X \subseteq \mathcal{I}_n^{\text{op}} \mid X \text{ is an ideal} \}.
\]

It is known that (at least in the finite case) an ideal over a finite partial order is uniquely represented by the antichain of its maximal elements. Intuitively, the antichain of maximal elements is the “upper border” of the ideal. Because of this bijection, antichains of intervals are endowed with a partial order, and with the algebraic structure of a distributive lattice.

The lattice of antichains thus defined is essentially the classic Clarke–Cormack–Burkowski minimal-interval lattice, with the important difference that since we allow the empty interval, we have a top element that has the empty interval only as a witness. For the purposes of this paper, the difference is immaterial, though.

To make the reader grasp more easily the meaning of \( \mathcal{E}_n \), we now describe in an elementary way its order and its lattice operations (note that we are not giving a definition: the operations are simply the reflection on the set of antichains of those of \( \mathcal{E}_n \)). Given antichains \( A \) and \( B \), we have

\[
A \leq B \iff \forall I \in A \quad \exists J \in B \quad J \subseteq I.
\]

Intuitively, \( A \leq B \) if every witness \( I \) in \( A \) (an interval) can be substituted by a better (or equal) witness \( J \) in \( B \), where “better” means that the new witness \( J \) is contained in \( I \).

Correspondingly, the \( \lor \) of two antichains \( A \) and \( B \) is given by the union of the intervals in \( A \) and \( B \) from which non-minimal intervals have been eliminated. Finally, the \( \land \) of \( A \) and \( B \) is given by the set of all intervals spanned by a pair of intervals \( I \in A \) and \( J \in B \), from which non-minimal intervals have been eliminated. It is this very natural algebraic structure that has led to the definition of the Clarke–Cormack–Burkowski lattice.

For instance, if we consider Figure 1 the lists for “porridge” (\( \{ 1, 4, 7, 32, 35 \} \)), “pease” (\( \{ 0, 3, 6, 31, 34 \} \)) and “hot” or “cold” (\( \{ 2, 5, 17, 21, 33, 36 \} \)) give us a large number of spanned intervals, from which we keep the antichain

\[
\{ [0..2], [1..3], [2..4], [3..5], [4..6], [5..7], [6..17], [7..31],
[21..32], [31..33], [32..34], [33..35], [34..36] \}.
\]

A simple snippet extraction algorithm would compute greedily the first \( k \) smallest nonoverlapping intervals of the antichain, which would yield, for \( k = 3 \), the intervals \( [0..2], [3..5], [31..33] \), that is, “Pease porridge hot!”, “Pease porridge cold!”, and, again, “Pease porridge hot!”: A ranking scheme such as those proposed in [2] would use the number and the length of the intervals to assign a score to the document with respect to the query.

Finally, we remark that the intervals in an antichain can be ordered in principle either by left or by right extreme, but these orders can be easily shown to be the same, so the intervals in an antichain are naturally linearly ordered by their extremes.

### 3 Operators

We shall not give a formal definition of query: the syntax is implied by our choice of operators. As a guide, the reader must consider that the semantics of a query containing a single term is the antichain of singleton intervals corresponding to the positions in which the term appears.

For completeness, we define explicitly the operators \( \land \) and \( \lor \), which are applied to a list of input antichains \( A_0, A_1, \ldots, A_{n-1} \), resulting in the \( \land \) and \( \lor \), respectively, of the antichains \( A_0, A_1, \ldots, A_{n-1} \). There are other useful operators that can be defined directly on the antichain representation [3]:

1. **AND** (\( \land \))
2. **OR** (\( \lor \))
1. **BLOCK**, given input antichains $A_0, A_1, \ldots, A_{n-1}$, returns the set of intervals of the form $[\ell_0 \ldots r_0] \cup [\ell_1 \ldots r_1] \cup \cdots \cup [\ell_{n-1} \ldots r_{n-1}]$ for which $[\ell_i \ldots r_i] \in A_i \ (0 \leq i < n)$ and $r_{i-1} + 1 = l_i \ (0 < i < n)$.

2. **AND$_\leq$**, given input antichains $A_0, A_1, \ldots, A_{n-1}$, returns the set of minimal intervals among those spanned by a set of intervals $[\ell_i \ldots r_i] \in A_i \ (0 \leq i < n)$ satisfying $l_{i-1} \leq l_i \ (0 < i < n)$.

3. **AND$_<$**, given input antichains $A_0, A_1, \ldots, A_{n-1}$, returns the set of minimal intervals among those spanned by a set of intervals $[\ell_i \ldots r_i] \in A_i \ (0 \leq i < n)$ satisfying $r_{i-1} < l_i \ (0 < i < n)$.

4. **LOWPASS$_k$**, given an input antichain $A$, returns the set of intervals from $A$ not longer than $k$.

More informally, given input antichains $A_0, A_1, \ldots, A_{n-1}$, the operator **BLOCK** builds sequences of consecutive intervals, each of which is taken from a different antichain, in the given order. It can be used, for instance, to implement a phrase operator. The **AND$_\leq$** and **AND$_<$** operators are ordered-AND operators which return intervals containing intervals from all of the $A_i$, much like the **AND** operator. However, in the case of **AND$_\leq$** and **AND$_<$** the left extremes of the intervals must be nondecreasing, and in the case of **AND$_<$** the intervals must be nonoverlapping. These operators (in particular **AND$_<$**) can be used, for instance, to search for terms that must appear in a specified order. Finally, **LOWPASS** restricts the result to intervals shorter than a given threshold, and can be easily combined with **AND** or **AND$_<$** to implement searches for terms that must not be too far apart, and possibly appear in a given order.

Note that the natural lattice operators **AND** and **OR** cannot return the empty antichain when all their inputs are nonempty. This is not true of the above operators: for instance, **BLOCK** might fail to find a sequence of consecutive intervals even if all its inputs are nonempty.

Finally, we remark that all intervals satisfying the definition of the **BLOCK** operator are minimal. Indeed, assume by contradiction that for two concatenations of minimal intervals we have $[\ell \ldots r] \subset [\ell' \ldots r']$ (which implies either $\ell' < \ell$ or $r < r'$). Assume that $\ell' < \ell'$ (the case $r < r'$ is similar), and note that removing the first component interval from both concatenations we still get intervals strictly containing one another. We iterate the process, obtaining two intervals of $A_{n-1} \ (A_0$, respectively) strictly containing one another.

## 4 Lazy evaluation of query operators

Most search engines use inverted files to index their document collections [12]. Usually, inverted indices are to be scanned in a sequential, left-to-right manner. Thus, given a document containing a term $t$, we assume that it is possible to obtain a list $L_t$ containing the positions of $t$ in the document in increasing order. Each call to next($L_t$) returns a new position, and, when no more positions are available, **null** is returned. We identify position $k$ with the singleton interval $[k \ldots k]$, so that $L_t$ can be viewed as an antichain of intervals. More generally, the list $L_t$ will return the intervals of the input antichain $A_t$.

The main point of this paper is that algorithms for computing operators on antichain of intervals should be always lazy and linear in the input intervals: if an algorithm is lazy, when only a small number of intervals is needed (e.g., for presenting snippets) the computational cost is significantly reduced. Linearity in the input intervals is the best possible result for a lazy algorithm, as input must be read at some point. All algorithms described in this paper satisfy this property, albeit in the case of **AND** and **OR** there is also a logarithmic factor in the number of input antichains.

Note that if the inverted index provides random-access lists of term positions, algorithms such as those proposed in [3] might be more appropriate for first-level operators (e.g., logical operators computed directly on lists of term positions), as by accessing directly the term positions they achieve...
complexity proportional to $ns \log m$, where $m$ is the overall number of intervals in the input antichains, $n$ is the number of antichains, and $s$ is the number of results. Nonetheless, as soon as one combines several operators, the advantage of a lazy linear implementation is again evident. Moreover, $s$ is in principle bounded only by $m$, and the estimate above hides the fact that somehow the input antichains must have been computed, with a cost proportional to $m$, and space occupancy proportional to $m$.

The logarithmic factor in the number of antichains can be easily proved to be unavoidable for the OR operator in a model in which intervals can be handled just by comparing their extremes:

**Theorem 1** Every algorithm to compute OR that is only allowed to compare interval extremes requires $\Omega(n \log n)$ comparisons.

**Proof.** It is possible to sort $n$ distinct integers by computing the OR of $n$ antichains, each containing a single singleton interval containing one of the integers to be sorted. The resulting antichain is exactly the list of sorted integers. By an application of the $\Omega(n \log n)$ lower bound for sorting in this model, we get to the result. \qed

5. Algorithms based on double indirect queues

The algorithms we provide for AND and OR are inspired by the plane-sweeping technique used in [11] for their proximity algorithm, which is on its own right a variant of the standard sorted-list merge. The algorithms are implemented using a double indirect priority queue.

A double indirect priority queue $Q$ is a data structure based on an array (called the reference array), which is managed outside the queue itself, and two priority orders that compare items from the reference array: these two orders are called primary and secondary. At each time, the queue contains a set of indices into the reference array (initially, a specified set, possibly empty). An array index $x$ can be added to the queue calling the function `enqueue(Q, x)`. The index of the least item in the reference array with respect to the primary (secondary) priority order can be accessed by invoking the function `topIndex(Q)` (secondaryTopIndex(Q), resp.). The index of the least item with respect to the primary priority order is also returned by `dequeue(Q)`, which also removes the index from $Q$. Analogously, `top(Q)` (secondaryTop(Q), resp.) return the least item in the reference array with respect to the primary (secondary) priority order.

The data structure assumes that the only item of the reference array that might change its value is the top item. Such a change must be communicated immediately to the queue by calling the function `change(Q)`. Table 1 summarises the operations available on a double indirect priority queue.

A double indirect priority queue can be easily and efficiently implemented using two priority queues: a primary semi-indirect queue and a secondary indirect queue$^3$. Note that we need the secondary queue to be fully indirect, as the primary queue must be able to adapt just to changes of its top item, but the secondary queue must be able to adapt to changes of any item (as it must be able to reflect changes in the top of the primary queue).

A trivial array-based implementation requires linear space and has constant cost for all operations modifying the queue, whereas retrieving the (secondary) top requires $O(n)$ time. A better implementation uses a priority queue (e.g., based on a heap) with linear space and logarithmic time complexity for all operations modifying the queue, and constant-time access to the (secondary) top. Sophisticated heaps with linear costs for several operations do not modify significantly the overall behaviour, as each time the queue is advanced the interval corresponding to the top index becomes greater: there

---

$^3$A **semi-indirect** queue has a change operation that allows to restore the correct state after a change in the value associated to the top item. An **indirect** queue has a change operation that restores the correct state after a change in the value associated to any index.
enqueue\((Q, x)\)
insert item with index \(x\) in the queue

topIndex\((Q)\)
returns the index of the top item w.r.t. the primary order

secondaryTopIndex\((Q)\)
returns the index of the top item w.r.t. the secondary order

top\((Q)\)
returns the top item w.r.t. the primary order, and deletes it from the queue

secondaryTop\((Q)\)
returns the top item w.r.t. the secondary order

dequeue\((Q)\)
returns the top item w.r.t. the primary order, and deletes it from the queue

change\((Q)\)
signals that the top item has changed

size\((Q)\)
returns the number of indices currently in the queue

Table 1: The operations available for a double indirect priority queue.

are data structures that make it possible to decrease in constant time the top, but not in increase it (or we could sort in linear time by comparison).

All algorithms based on double priority queues have complexity \(O(m \log n)\) if the input is formed by \(n\) antichains containing \(m\) intervals overall. This is immediate, as all loops contain exactly one queue advancement.

5.1 Basic comparators

To describe our algorithms we will use two main priority orders. The first one, denoted by \(\leq\), is defined by

\[
[\ell \ldots r] \leq [\ell' \ldots r'] \iff \ell < \ell' \text{ or } \ell = \ell' \text{ and } r \geq r'.
\]

In other words, \([\ell \ldots r] \leq [\ell' \ldots r']\) if \([\ell \ldots r]\) starts before or prolongs \([\ell' \ldots r']\). Note in particular that (somewhat counterintuitively) \([\ell \ldots r] \leq [\ell' \ldots r']\) iff \(r \geq r'\). This order will always be used as a primary order in a queue.

The second order, denoted by \(\preceq\), is easier: it compares the right extremes according to their natural order:

\[
[\ell \ldots r] \preceq [\ell' \ldots r'] \iff r \leq r'.
\]

We remark that in a queue using \(\leq\) as primary order the left extreme of the top interval is nondecreasing.

The algorithms for AND/OR use a double indirect priority queue with primary priority order \(\leq\). The reference array underlying the queue contains one interval per input antichain, which we assume without loss of generality nonempty (in the case of AND, an empty list implies an empty result, and in the case of OR empty lists can be simply dropped). In the initialisation phase, the reference array is filled with the first interval from each antichain, and the queue contains all indices.

To simplify the description, we define a function advance\((Q)\) that stores temporarily the current top interval, updates with the next interval the list associated with the top index, notifies the queue of the change, and finally returns the stored interval. If the update cannot be performed because the list is empty, the top index is dequeued. The function is described in pseudocode in Algorithm 1, where we assume that \([\ell_i \ldots r_i]\) is the interval in the reference array for list \(i\).

5.2 The OR operator

We start with the simplest nontrivial operator. To compute the interval antichain corresponding to the OR of the antichains \(A_0, A_1, \ldots, A_{n-1}\) we create a double indirect priority queue \(Q\) with primary priority order \(\leq\) and secondary priority order \(\preceq\). As a consequence, the right extreme of the secondary
Algorithm 1 The advance function.

0 function advance(Q) begin
1    i ← topIndex(Q);
2    c ← [ℓ_i..r_i];
3    if the input list i is not empty then
4        [ℓ_i..r_i] ← next(L_i);
5        change(Q)
6    else
7        dequeue(Q)
8    end;
9    return c
10 end;

top interval is nondecreasing, because every time we advance the queue we either eliminate an interval or substitute it with one that has a larger right extreme.

When we want to compute the next interval, we advance Q and store the returned interval [ℓ .. r] (which is, essentially, the leftmost largest remaining interval) as a candidate. We repeat the process until Q is empty or [ℓ .. r] does not contain the secondary-top interval. The algorithm is described in pseudocode in Algorithm 2.

Theorem 2 The algorithm for OR is correct.

Proof. First of all, note that all intervals in A_0, A_1, . . . , A_n−1 are assigned to c at some point, and if c contains a minimal interval, we certainly exit the loop (more precisely, we exit when c is the last instance of a given minimal interval to appear in the queue top). Thus, we only have to prove is that only minimal intervals are returned.

Assume that at the start of the while loop [ℓ .. r] is the primary-top interval, and, after advancing the queue, let [ℓ′ .. r′] be the primary-top interval and [ℓ″ .. r″] the secondary-top interval. If [ℓ .. r] is not minimal, then it must contain some smaller interval, say [ℓ .. r] ⊂ [ℓ .. r] coming from the i-th list. We can assume without loss of generality that [ℓ .. r] is actually one of the intervals currently in Q, as if this is not true the interval in the reference array at index i has smaller extremes but left extreme larger than or equal to l, so it is a fortiori included in [ℓ .. r] (note that this fact strictly depends on the definition of ≤).

Since [ℓ .. r] is in the queue, we have [ℓ″ .. r″] ⊆ [ℓ .. r] ≤ [ℓ .. r] ⊆ [ℓ .. r], hence r″ ≤ r, so [ℓ .. r] ⊎ [ℓ″ .. r″] (by monotonicity of the top-interval left extreme ℓ ≤ ℓ′ ≤ ℓ″) and we conclude that only minimal intervals are returned.

To prove that all returned intervals are unique, we just have to note that if several copies of the interval I are present in the input antichains, then as soon as the first copy of I becomes the top, all other copies of I are in the reference array (or there would be intervals in the reference array with left extreme smaller than I). Thus, the while loop will be repeated until all copies are discarded. At that point, I will be returned only if it is minimal. }

5.3 The AND operator

Then AND operator is much more subtle. The primary comparator of Q is ≤, whereas the secondary comparator is ≥ (note the inversion). At any time, the interval spanned by Q is the interval defined by the left extreme of the primary-top interval and the right extreme of the secondary-top interval; it
Algorithm 2 The algorithm for the OR operator. Note that the second part of the while condition is actually equivalent to “right(secondaryTop(Q)) ≤ right(c)” due to the monotonicity of the top-interval left extreme.

```
function next begin
  if Q is empty return null;
  do
    c ← advance(Q)
    while ¬ (Q is empty) and secondaryTop(Q) ⊆ c ;
  return c
end;
```

will be denoted by span(Q). Clearly, it is the minimum interval containing all intervals currently in the queue. Note that the right extreme of the secondary top cannot decrease while Q is full, that is, the size of Q is n.

When we want to compute the next interval, we store the interval $[\ell \ldots r]$ currently spanned by Q as a candidate and advance Q. If the new interval spanned by Q is included in $[\ell \ldots r]$ we repeat the operation, updating the candidate. Then, before returning $[\ell \ldots r]$ we advance Q until the spanned interval does not contain $[\ell \ldots r]$. If at any time Q is no longer full, we just return the candidate. The algorithm is described in pseudocode in Algorithm 3.

Algorithm 3 The algorithm for the AND operator. Note that the second part of the first while condition can be substituted with “right(c) = right(secondaryTop(Q))” and that the second part of the second while condition can be substituted with “left(top(Q)) = left(c)” because of the monotonicity of the top-interval left extreme and of the secondary-top interval right extreme.

```
function next begin
  if ¬ (Q is full) then return null;
  do
    c ← span(Q);
    advance(Q)
    while Q is full and span(Q) ⊆ c ;
  while Q is full and c ⊆ span(Q) do
    advance(Q)
  end;
  return c
end;
```

Theorem 3 The algorithm for AND is correct.

Proof. We say that a queue configuration is complete if it contains all copies of the primary top interval from all lists that contain it. Now observe that every complete configuration of a double indirect priority queue is entirely defined by its primary top interval. More precisely, if the top is an interval I from list i, then for every other list j the corresponding interval J in the queue is the
minimum interval in $A_j$ larger than or equal to $I$ (following $\preceq$). Indeed, suppose by contradiction that there is another interval $K$ from $A_j$ satisfying

$$I \preceq K \prec J.$$ 

Then, at some point $K$ must have entered the queue, and must have been dequeued when the top was some interval $I' \preceq I$, so we get

$$K \preceq I' \preceq I \preceq K,$$

which yields $K = I$: a contradiction, as we assumed the state of the queue to be complete.

We now show that for every minimal interval $[\ell \ldots r]$ in the AND of $A_0, A_1, \ldots, A_{n-1}$ there is a complete state of $Q$ spanning $[\ell \ldots r]$. Consider for each $i$ the set $C_i$ of intervals of $A_i$ contained in $[\ell \ldots r]$. At least one of these sets must contain a (necessarily unique) right delimiter, that is, an interval of the form $[\ell' \ldots r]$. Moreover, at least one of the sets containing a delimiter must be a singleton. Indeed, if every $C_i$ containing a right delimiter would also contain some other interval, the right extreme of that interval would clearly be smaller than $r$: removing all right delimiters from the $C_i$’s, we would span a strictly smaller new interval showing that $[\ell \ldots r]$ was not minimal. We conclude that at least one $C_i$, say $C_i$, is a singleton containing a right delimiter.

Consider now for each $C_i$ the leftmost (in the sense of $\preceq$) interval $I_i$. The resulting set of intervals defines a complete configuration of $Q$: if $i$ is such that $I_i = [\ell \ldots r']$ and if $I_i \in A_j$ necessarily $I_i = I_j$, because $A_j$ cannot contain two intervals with the same left extreme. The set of intervals also spans $[\ell \ldots r]$ (because the right extreme of $I_i$ is $r$, and the left extreme of the $\preceq$-least interval $I_i$ is $\ell$). We conclude that all minimal intervals are sooner or later spanned by $Q$.

However, no minimal interval can be spanned during the second while loop. All intervals spanned in that loop contain the candidate interval, which makes them nonminimal (independently of the minimality of the candidate) or copies of the minimal candidate we are going to return. Finally, if an interval is spanned in the first while loop and we do not get out of the loop, the next candidate interval will be smaller or equal. We conclude that sooner or later all minimal intervals cause an interruption of the first while loop, and are thus returned.

We are left to prove that if an interval is returned, it is necessary minimal. We prove at the same time the following invariant: no interval containing a previously returned interval will be ever spanned by $Q$ (this is trivially true at the first call). Assume now that the interval $[\ell \ldots r]$ spanned by $Q$ at the start of the first while loop is not minimal, so $[\hat{\ell} \ldots \hat{r}] \subset [\ell \ldots r]$, for some minimal interval $[\hat{\ell} \ldots \hat{r}]$ that will be necessarily spanned later (because of the invariant, as we already proved that all minimal intervals are returned). Then, letting $[\ell'' \ldots r'']$ be the secondary-top interval after we advanced $Q$, we have $r'' \leq \hat{r}$ by monotonicity of the secondary-top right extreme. On the other hand, always by monotonicity of the secondary-top right extreme, $r \leq r''$, and since $[\hat{\ell} \ldots \hat{r}] \subset [\ell \ldots r]$, $\hat{r} \leq r$. We conclude $r = r'' = \hat{r}$. By monotonicity of the primary-top left extreme, the interval spanned by $Q$ is contained in $[\ell \ldots r]$, so we will not exit the first while loop.

We must prove that the invariant is true at the end of the call. However, this is trivial, as the second while loop advances $Q$, eliminating all intervals that could contain $c$. By monotonicity of the primary-top interval left extreme, after the second while loop the left extreme of the interval spanned by $Q$ will be larger than that of $c$: thus, no following intervals spanned by $Q$ will be able to contain $c$.

Finally, we remark that the invariant yields immediately that all returned intervals are unique. $\blacksquare$

6 Greedy algorithms

The algorithms for BLOCK, AND$_{\preceq}$ and AND$_{<}$ are much simpler: they are just a greedy enumeration procedure with backtracking (in the latter two cases, borrowing also some ideas from queue-based
algorithms). In some cases they are part of the folklore, at least when applied to list of term positions. Nonetheless, a thorough correctness proof for the case of interval antichains is not completely obvious. All algorithms have trivial complexity $O(m)$, where $m$ is the number of intervals in the input antichains, as all loop bodies advance at least one of the input lists.

6.1 The BLOCK operator

We keep track of a current interval for all lists $L_0, L_1, \ldots, L_{n-1}$; initially, these intervals are set to $[-1 \ldots -1]$. When we want to compute the next interval, we update the interval associated to the first list. Then, we try to fix index $i$ (initially, $i = 1$). To do so, we advance the list $L_i$ until the returned interval has left extreme larger than the right extreme of the current interval for $L_{i-1}$. If we go too far, we just advance the first list, reset $i$ to 1 and restart the process, otherwise we increment $i$. When we find an interval for $L_{n-1}$ we return the interval spanned by all current intervals. The algorithm is described in pseudocode in Algorithm 4.

Algorithm 4 The algorithm for the BLOCK operator.

```plaintext
0 function next begin
1   if $L_0$ is empty then return null;
2   $[\ell_0 \ldots r_0] \leftarrow \text{next}(L_0)$;
3   $i \leftarrow 1$;
4   while $i < n$ do
5     while $(L_i$ is empty) and $\ell_i \leq r_{i-1}$ do
6       $[\ell_i \ldots r_i] \leftarrow \text{next}(L_i)$
7       end;
8     if $\ell_i \leq r_{i-1}$ then return null
9     else if $\ell_i = r_{i-1} + 1$ then $i \leftarrow i + 1$
10    else begin
11       if $L_0$ is empty then return null;
12       $[\ell_0 \ldots r_0] \leftarrow \text{next}(L_0)$;
13       $i \leftarrow 1$
14    end
15   end;
16   return $[\ell_0 \ldots r_{n-1}]$
17 end;
```

Theorem 4 The algorithm for BLOCK is correct.

Proof. At the start of an iteration of the external while loop with a certain index $i$ we clearly have $r_k + 1 = \ell_{k+1}$ for $k = 0, 1, \ldots, i - 2$. Thus, if we complete the execution we certainly return a correct interval.

To complete the proof, we start by proving the following invariant property: at the start of the external while loop, for all $0 < j < n$ there are no intervals in $A_j$ with left extreme in $[r_{j-1} + 1 \ldots \ell_j - 1]$. In other words, the $j$-th current interval $[\ell_j \ldots r_j]$ has either left extreme smaller than or equal to $r_{j-1}$, or it is the first interval in $A_j$ whose left extreme is larger than $r_{j-1}$. The property is trivially true at the beginning, and advancing $[\ell_0 \ldots r_0]$ cannot change this fact. We are left to prove that the execution of the internal while loop cannot either.

During the execution of the internal loop, only $[\ell_i \ldots r_i]$ can change. This affects the invariant because it modifies the intervals $[r_{j-1} + 1 \ldots \ell_i - 1]$ and $[r_i + 1 \ldots \ell_{i+1} - 1]$, but in the second case
Algorithm 5 The algorithm for the $\text{AND}_{\leq}$ / $\text{AND}_{<}$ operator. For the $\text{AND}_{<}$ operator, the test $\ell_i < \ell_{i-1}$ must be substituted with $\ell_i \leq r_{i-1}$.

0 function next begin
1     if some $L_i$ is empty then return null;
2     do
3       $c \leftarrow [\min_{j \in n} \ell_j \ldots \max_{j \in n} r_j]$;
4       if $L_0$ is empty then return $c$;
5       $[\ell_0 \ldots r_0] \leftarrow \text{next}(L_0)$;
6       $i \leftarrow 1$;
7       while $i < n$ do
8         while ($L_i$ is empty) and $\ell_i < \ell_{i-1}$ do
9           $[\ell_i \ldots r_i] \leftarrow \text{next}(L_i)$
10          end;
11         if $\ell_i < \ell_{i-1}$ then return null;
12         $i \leftarrow i + 1$;
13       end
14       while $[\min_{j \in n} \ell_j \ldots \max_{j \in n} r_j] \subseteq c$;
15       return $c$
16 end

the interval is made smaller, so the invariant is a fortiori true. In the first case, at the beginning of the execution of the internal while loop either $r_{i-1} + 1 \leq \ell_i - 1$, that is, $r_{i-1} < \ell_i$, so the loop is not executed at all and the invariant cannot change, or $r_{i-1} + 1 > \ell_i - 1$, which means that the interval $[r_{i-1} + 1 \ldots \ell_i - 1]$ is empty, and the loop will advance $[\ell_i \ldots r_i]$ up to the first interval in $A_i$ with a left extreme larger than or equal to $r_{i-1}$, making again the invariant true.

Suppose now that there are $[\tilde{\ell}_0 \ldots \tilde{r}_0]$, $[\tilde{\ell}_1 \ldots \tilde{r}_1]$, $[\tilde{\ell}_k \ldots \tilde{r}_k]$ satisfying $r_i + 1 = \ell_{i+1}$ for some $k > 0$ and $0 \leq i < k$. We prove by induction on $k$ that at some point during the execution of the algorithm we will be at the start of the external while loop with $i = k$ and $[\ell_j \ldots r_j] = [\tilde{\ell}_j \ldots \tilde{r}_j]$ for $j = 0, 1, \ldots, k$. The thesis is trivially true for $k = 0$. Assume the thesis for $k - 1$, so we are at the start of the external while loop with $i = k - 1$ and $l_j = \tilde{l}_j$, $r_j = \tilde{r}_j$ for $j = 0, 1, \ldots, k - 1$. Because of the invariant, either $[l_k \ldots r_k] = [\tilde{l}_k \ldots \tilde{r}_k]$ or $[l_k \ldots r_k]$ will be advanced by the execution of the internal while loop up to $[\tilde{l}_k \ldots \tilde{r}_k]$. Thus, at the end of the external while loop the thesis will be true for $k$. We conclude that all concatenations of intervals from $A_0, A_1, \ldots, A_{n-1}$ are returned.

We note that all intervals returned are unique (minimal has been already discussed in Section 3), as $[0 \ldots r_0]$ is advanced at each call, so a duplicate returned interval would imply the existence of two comparable intervals in $A_0$.

6.2 The $\text{AND}_{\leq}$ and $\text{AND}_{<}$ operators

The algorithms for computing these operators are a medley of the algorithms for AND and for BLOCK. As in the case of AND, we must check that future intervals are not smaller than our current candidate. As in the case of BLOCK, there is no queue and the lists $L_0, L_1, \ldots, L_{n-1}$ are advanced greedily. Again, we keep track of a current interval $[\ell_j \ldots r_j]$ for all lists $L_0, L_1, \ldots, L_{n-1}$; initially, these intervals are $[-1 \ldots -1]$. At any time, the spanned interval is $[\min_{j \in n} \ell_j \ldots \max_{j \in n} r_j]$.

When we want to return a new interval, we store the interval currently spanned. Then, we update the interval associated to the first list and try to fix index $i$ (initially, $i = 1$). To do so, in the case of $\text{AND}_{\leq}$ we advance $L_i$ until the returned interval has left extreme larger than or equal to the left
extreme of the current interval; in the case of \( \text{AND}_\leq \) we advance \( L_i \) until the returned interval has left extreme larger than the right extreme of the current interval. If we can find a suitable interval, we increment \( i \) and continue. When we find an interval for \( A_{n-1} \) we check whether it is contained in the candidate, in which case we choose it as a new candidate, and restart the process, otherwise we return the candidate. The algorithm is described in pseudocode in Algorithm 5.

**Theorem 5** The algorithm for \( \text{AND}_\leq/\text{AND}_\ll \) is correct.

**Proof.** We discuss the correctness of the algorithm for \( \text{AND}_\leq \); the case of \( \text{AND}_\ll \) is completely analogous.

The first part of the proof is very similar to that for the BLOCK operator. Similarly to that case, one proves that the following invariant property is true at the start of the internal while loop: for all \( 0 < j < n \) there are no intervals in \( A_j \) with left extreme in \([\ell_{j-1} + 1 \ldots \ell_j - 1]\). In other words, the \( j \)-th current interval \([\ell_j \ldots r_j]\) has either left extreme smaller than to \( \ell_{j-1} \), or it is the first interval in \( A_{j+1} \) whose left extreme is larger than or equal to \( \ell_{j-1} \). Let us say that a sequence \([\ell_i \ldots r_i], i = 0, 1, \ldots, n\) of intervals, one from each list, with nondecreasing left extremes is *leftmost* if for all \( 0 < j < n \) there are no intervals in \( A_j \) with left extreme in \([\ell_{j-1} + 1 \ldots \ell_j - 1]\). Then, it is immediate to show that all sequences of leftmost intervals appear at some point at the start of the external loop.

We now note that every minimal interval \([\ell \ldots r]\) spanned by minimal intervals from the \( A_i \)'s has a unique leftmost representation. To obtain it from a generic set of interval, substitute iteratively the interval for list \( i > 0 \) with the interval with smallest left extreme satisfying \( \ell_i \geq \ell_{i-1} \). Note that the interval for \( A_{n-1} \) cannot change, for \([\ell \ldots r]\) was assumed to be minimal, so the resulting set of intervals still spans \([\ell \ldots r]\). We conclude that sooner or later all minimal intervals of the result are spanned, and thus returned.

We are left to prove that only minimal intervals are returned. As in the proof for the AND operator, we prove at the same time the following invariant: no interval ever spanned in the future will contain a previously returned interval (the invariant is trivially true at the start). Suppose the current candidate \([\ell \ldots r]\) is not minimal. This means that there is an interval \([\tilde{\ell} \ldots \tilde{r}] \subset [\ell \ldots r]\) that will be spanned later (because of the invariant). By monotonicity of the extremes of spanned intervals, this implies that after advancing the interval set the new spanned interval must be contained in \([\ell \ldots r]\), so we will not get out of the external while loop.

We must show that the invariant holds at the end of a call. But if the candidate does not contain the currently spanned interval, this means that both extremes are larger than those of the candidate (the left extreme is increased each time the external while loop is executed, and the right extreme must be larger then that of the candidate, or the loop would repeat). We conclude that no spanned intervals will ever contain the candidate.

Finally, as in the proof of Theorem 3 we remark that the invariant yields immediately that all returned intervals are unique. \( \blacksquare \)

### 7 Lazier algorithms

The algorithms presented in the previous sections are very efficient, but they are not always as lazy and they could. In this section we describe lazier algorithms, which advance less the underlying lists, by tweaking slightly the algorithms we described. The tweaked version is less elegant and less clear, so we prefer to describe it separately in this section. We can in any case reuse our correctness proofs, as the differences are mild, so we can prove correctness by showing that that returned intervals are the same.

It should be noted that in some programming languages (most notably, Java) the suggested iterator implementation is only partially lazy, as there is a method that returns whether more elements can be returned by the iterator. In that case, the algorithms described in Section 5 turn out to be particularly
useful, as after returning a result it is possible to know in constant time whether another one will be returned by a following call (note that the only way to return \texttt{null} is to fail the test at the start of the algorithms).

### 7.1 The OR operator

A lazier version of the algorithm for the OR operator is shown in Algorithm 6. We assume that the value of the candidate \( c \) is maintained between call to the function \texttt{next}, and that it is initialised to \([-1 \ldots 1]\).

As it is easy to see, the only difference in behaviour between Algorithm 2 and Algorithm 6 is that in case the secondary top is \textit{equal} to the candidate we return it immediately, delaying the advancement of \( Q \) to the next call. Note that by definition of \( \preceq \) if the candidate is equal to the secondary top advancing the queue can only introduce copies of the candidate or intervals incomparable with the candidate. Thus, when the candidate is equal to the secondary top we can safely return it. The only advantage of the new algorithm is that if multiple copies of an interval are found, we return the interval when we see the first copy, rather than the last.

**Algorithm 6** A lazier algorithm for the OR operator.

```plaintext
function next begin
  while \neg (Q is empty) and secondaryTop(Q) = c
    advance(Q)
  end;
  if Q is empty return null;
  do
    c ← advance(Q)
    while \neg (Q is empty) and secondaryTop(Q) \subset c
      return c
  end;
end:
```

### 7.2 The AND operator

A lazier version of the algorithm for the AND operator is shown in Algorithm 7. We assume again that the value of the candidate \( c \) is maintained between call to the function \texttt{next}, and that it is initialised to \([-1 \ldots 1]\). In essence, all we do is to move the second while loop before the first one. Since at the first call \( c \) is \([-1 \ldots -1]\), the first loop is never executed. After the first call, the algorithm behaves exactly like the original one, but advancement will stop as soon as the result is found, rather than going on until \texttt{span}(Q) does not contain the result.

### 8 Conclusions

We have provided efficient algorithms for the computation of several operators based on minimal-interval semantics. In particular, the algorithm for OR has been proved to be optimal in a comparison-based model. Moreover, the algorithms are lazy and use space linear in the number of input antichains. This compares favourably with the previously known algorithms [3], which in particular required an eager computation of all component antichains (albeit it should be noted that the two bounds, \( O(ns \log m) \) and \( O(m \log n) \), are in general incomparable).
Algorithm 7 A lazier algorithm for the AND operator.

0 function next begin
1 while Q is full and $c \subseteq \text{span}(Q)$ do
2 advance($Q$)
3 end;
4 if $\neg (Q$ is full) then return null;
5 do
6 $c \leftarrow \text{span}(Q)$;
7 advance($Q$)
8 while $Q$ is full and $\text{span}(Q) \subseteq c$;
9 return $c$
10 end;

An interesting open problem is that of providing a matching lower bound for the AND operator, at least for a comparison-based computational model.

References


