Efficient Optimally Lazy Algorithms for Minimal-Interval Semantics*

Paolo Boldi Sebastiano Vigna
Dipartimento di Informatica, Università degli Studi di Milano

Abstract

Minimal-interval semantics [8] 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. In this paper we provide algorithms for computing conjunction and disjunction that are linear in the number of intervals and logarithmic in the number of operands; for additional operators, such as ordered conjunction and Brouwerian difference, we provide linear algorithms. In all cases, space is linear in the number of operands. More importantly, we define a formal property of *optimal laziness*, and either prove it, or prove its impossibility, for each algorithm. We cast our results in a general framework of finite antichains of intervals on total orders, making our algorithms directly applicable to other domains.

1 Introduction

Modern information-retrieval systems, such as web search engines, rely on *query expansion*, an automatic or semi-automatic mechanism that aims at rewriting the user intent (i.e., a set of keywords, maybe with additional context such as geographical location, past search history, etc.) as a structured query built upon a number of operators. The simplest case is that of the *Boolean model*, in which operators are just conjunction, disjunction and negation: as an example, the set of keywords provided by the user might be expanded as disjunctions of syntactically or semantically similar terms, and finally such disjunctive queries would be connected using a conjunction operator. The semantics provided by this model is simply either the value "true" or the value "false" (in the example, "true" is returned if the document contains at least one term from each disjunction).

When a document satisfies a query, however, the Boolean model fails to explain *why* and *where* the query is satisfied: this information is compressed in a single truth value. *Minimal-interval semantics* is a richer semantic model that uses *antichains*[1] *of intervals of natural numbers* to represent the semantics of a query; this is the natural framework in which operators such as ordered conjunction, proximity restriction, etc., can be defined and combined freely. Each interval is a *witness* of the satisfiability of the query, and defines a region of the

*A preliminary version of some of the results in this paper appeared in [5]. All algorithms have been significantly simplified, getting rid of the double queues of [5]. The notion of optimal laziness and all related results are entirely new. The algorithm for the AND operator has been improved during the proofs of optimality (it was not optimally lazy in the formulation of [5]).

[1] An *antichain* of a partial order is a set of elements that are pairwise incomparable.

document that satisfies the query (positions in the document are numbered starting from 0, so regions of text are identified with sets of consecutive integers, a.k.a. intervals).

Consider, for example, the document given by the start of the well-known rhyme

Pease porridge hot! Pease porridge cold!

If we query this document with the keyword "*hot*", we just get the Boolean answer "true". But a more precise answer would be "*hot* appears in the third position of the document": formally, this is described by the interval $[2\,..\,2]$. Sometimes, a query will be satisfied in multiple parts of the document; for example, the query "*pease*" has answer $\{\,[0\,..\,0], [3\,..\,3]\,\}$. If we consider *two* keywords things become more interesting: when we submit the conjunctive query "*pease* AND *porridge*" the answer will be $A = \{\,[0\,..\,1], [1\,..\,3], [3\,..\,4]\,\}$. Of course, there are more intervals containing both *pease* and *porridge*, but they are omitted because they contain (and thus they are less informative than) one of the intervals in A. The latter observation leads us to considering sets of intervals that are incomparable with respect to inclusion, that is, antichains with respect to $\subseteq$.

This approach has been defined and studied to its full extent by Clarke, Cormack and Burkowski in their seminal paper [8]. 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 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 [7].

The main feature of minimal-interval semantics is that, by its very definition, an antichain of intervals cannot contain more than w intervals, where w 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 [10] or the `sgrep` tool [13]), as the overall number of output regions is quadratic in the document size.

In this paper, we attack the problem of providing efficient lazy algorithms for the computation of a number of operators on antichains. As a subproblem, we can compute the proximity of a set of terms, and indeed we are partly inspired by previous work on proximity [19, 18]. 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), so they are a very extreme case of stream transformation algorithms [2, 12]. Moreover, our algorithms satisfy some stringent formal *laziness* properties.

Note that from a practical viewpoint laziness makes our algorithms very attractive when paired with an index structure (see, e.g., quasi-succinct indices [21]) that provides lazy I/O for reading positions. For example, it is possible to decide that a set of terms appear within a certain proximity bound *without* reading all positions: if the underlying index is lazy, our algorithms limit the I/O as much as possible. In the open-source world, the semantic engine Mímir [20] is based on MG4J [4], which contains our implementation of such algorithms.

In Section 2 we briefly introduce minimal-interval semantics, and provide some 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 [8]). The resulting structure is essentially identical to that described in the original paper [8], but our systematic approach makes good use of well-known results from order theory, making the introduction self-contained. For some mathematical background, see, for instance, [3, 9].

Another advantage of our approach is that by representing abstractly regions of text as intervals of natural numbers we can easily highlight connections with other areas of computer science: for example, antichains of intervals have been used for testing distributed computations [14]. The problem of computing operators on antichains has thus an intrinsic interest that goes beyond information retrieval. This is the reason why we cast all our results in the general framework of antichains of intervals on arbitrary (totally) ordered sets.

Finally, we present our algorithms. First we discuss algorithms based on queues, and then greedy algorithms.[2]

2 Minimal-interval semantics

Given a totally ordered set O, let us denote with $\mathscr{C}_O$ the set of intervals[3] of O that are either empty or of the form $[\ell\,..\,r] = \{ x \in O \mid \ell \leq x \leq r \}$. Our working example will always be $O = W$, where $W = \{0, 1, \ldots, w-1\}$ and w represents the number of words in a document, numbered starting from 0 (see Figure 1); elements of $\mathscr{C}_W$ can be thought of as regions of text.

Intervals are ordered by containment: when we want to order them by *reverse* containment instead, we shall write $\mathscr{C}_O^{\mathrm{op}}$ ("op" stands for "opposite"). Given intervals I and J, the interval *spanned* by I and J is the least interval containing I and J (in fact, their least upper bound in $\mathscr{C}_O$).

The idea behind minimal-interval semantics [8] is that every interval in $\mathscr{C}_W$ is a *witness* that a given query is satisfied by a document made of w words. *Smaller witnesses imply a better match, or more information*; in particular, if an interval is a witness, any interval containing it is also 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* 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 because it is not minimal (e.g., it contains $[0\,..\,2]$).

It is however more convenient to start from an algebraic viewpoint. A *lower set* X is a subset of a partial order that is closed downwards: if $y \leq x$ and $x \in X$, then $y \in X$. Given a subset A of a partial order, we denote with $\downarrow A$ the smallest lower set containing A (i.e., the set containing A and all elements smaller than an element of A).

We are interested in computing operators of the distributive lattice $\mathscr{A}_O$ whose elements are finite antichains of $\mathscr{C}_O^{\mathrm{op}}$ endowed with the following order:[4]

$$A \leq B \iff \downarrow A \subseteq \downarrow B.$$

The lattice of finite antichains[5] $\mathscr{A}_W$ thus defined is essentially the classic Clarke–Cormack–Burkowski *region algebra*, with the difference that since we allow the empty interval, we have a top element that contains only the empty interval, and which makes $\mathscr{A}_O$ a bounded lattice in the infinite case, too. For the purposes of this paper, the difference is immaterial, though.

[2] A free implementation of all algorithms described in this paper is available as a part of MG4J [4] (`http://mg4j.di.unimi.it/`) and LaMa4J (`http://lama4j.di.unimi.it/`).

[3] A subset X of O is an *interval* if $x, y \in X$ and $x \leq z \leq y$ imply $z \in X$.

[4] $\mathscr{A}_O$ is a distributive lattice because $\mathscr{C}_O^{\mathrm{op}}$ has binary greatest lower bounds [6].

[5] We remark that the construction of finite antichains (which are equivalent to finitely generated lower sets) of compact elements is the first step in the concrete construction of the *Hoare powerdomain* [1]. Thus, several formulas appearing in the rest of the paper will look familiar to readers acquainted with domain theory.

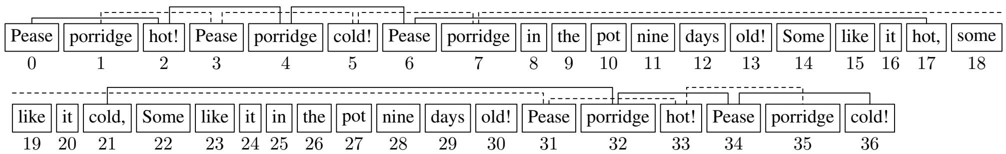

Figure 1: A sample text; the intervals corresponding to the semantics of the query "*porridge* AND *pease* AND (*hot* OR *cold*)" are shown. For easier reading, every other interval is dashed.

To make the reader grasp more easily the meaning of $\mathscr{A}_O$, we now describe in an elementary way its order and its lattice operations, which have been characterized in [6]. 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 $\vee$ 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 $\wedge$ of A and B is given by the set of all intervals spanned by pairs 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.

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 we can say that the intervals in an antichain are naturally linearly ordered by their extremes.

2.1 Examples

Consider from Figure 1 the positions of "*porridge*" $(1, 4, 7, 32, 35)$, "*pease*" $(0, 3, 6, 31, 34)$, "*hot*" $(2, 17, 33)$ and "*cold*" $(5, 21, 36)$. Queries associated with a single keyword have an easy semantics—the list of positions as singleton intervals. For example, the semantics of "*hot*" will be

$$\{\,[2\,..\,2], [17\,..\,17], [33\,..\,33]\,\}.$$

If we start combining terms disjunctively, we get simply the union of their positions. For instance, "*hot* OR *cold*" gives

$$\{\,[2\,..\,2], [5\,..\,5], [17\,..\,17], [21\,..\,21], [33\,..\,33], [36\,..\,36]\,\}.$$

If we consider the conjunction of two terms, we will start getting non-singleton intervals: the semantics of "*pease* AND *porridge*" is computed by picking all possible pairs of positions of *pease* and *porridge* and keeping the minimal intervals among those spanned by such pairs:

$$\{\,[0\,..\,1], [1\,..\,3], [3\,..\,4], [4\,..\,6], [6\,..\,7], [7\,..\,31], [31\,..\,32], [32\,..\,34], [34\,..\,35]\,\}.$$

The more complex query "(*pease* AND *porridge*) OR *hot*" is interesting because we have to take the intervals just computed, put them together with the positions of *hot*, and remove the non-minimal intervals:

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

One can see, for example, that the presence of *hot* in position 2 has eliminated the interval $[1\,..\,3]$.

Let's try something more complex: "*pease* AND *porridge* AND (*hot* OR *cold*)". We have again to pick one interval from each of the three sets associated to "*pease*", "*porridge*" and "*hot* OR *cold*", and keep the minimal intervals among those spanned by such triples (see Figure 1):

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

From this rich semantic information, a number of different outputs can be computed. 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 that proposed by Clarke and Cormack [7] would use the number and the length of these intervals to assign a score to the document with respect to the query. In a simplified setting, we can assume that each interval yields a score that is the inverse of its length. The resulting score for the query above would be

$$\frac{1}{|[0\,..\,2]|}+\frac{1}{|[1\,..\,3]|}+\cdots+\frac{1}{|[6\,..\,17]|}+\frac{1}{|[7\,..\,31]|}+\cdots\frac{1}{|[34\,..\,36]|}\\ =\frac{1}{3}+\frac{1}{3}+\cdots+\frac{1}{12}+\frac{1}{25}+\cdots\frac{1}{3}=\frac{177}{50}=3.54.$$

Clearly, documents with a large number of intervals are more relevant, and short intervals increase the document score more than long intervals. The score associated to "*hot*" would be just 3 (i.e., the number of occurrences). One can also incorporate positional information, making, for example, intervals appearing earlier in the document more important [4].

What happens if we are asked for a word or a phrase *not* appearing in a document? In this case, the natural semantics for the query turns out to be the top element of the lattice, which contains a single empty witness: this is indeed intuitively appropriate for the semantics of a query that is *not* true in the document—the only witness is located nowhere. This choice is practical, too, as queries of the form p AND NOT q are true when p is true and q is false, and their witnesses are the witnesses of p, as the top is the unit of conjunction. More generally, negation should send all non-bottom elements to bottom, and bottom to top [6], with the idea that every non-bottom element (every nonempty antichain) represents (a different degree of) the Boolean value "true", whereas the bottom is the only representation of "false". From an algorithmic viewpoint, implementing negation is trivial and will not be discussed further.

3 Operators

For the rest of the paper, we assume that we are operating on antichains based on an unknown total order O for which we just have a comparison operator. We use $\pm\infty$ to denote a special element that is strictly smaller/larger than all elements in O. Before getting to the core of the paper, however, we highlight the connection with query resolution in a search engine.

Search engines use inverted lists to index their document collections [24]. The algorithms described in this paper assume that, besides the documents in which a term appears, the index makes available the *positions* of all occurrences of a term *in increasing order* (this is a standard assumption).

Given a query, we first obtain the list of documents that could possibly satisfy the query; this is a routine process that involves merging and intersecting lists. Once we know that a certain document might satisfy the query, we want to find its witnesses, if any. To do so, we interpret the terms appearing in the query as lists of singleton intervals (the term positions), and apply in turn each operator appearing in the query. The resulting antichain represents the minimal-interval semantics (i.e., the set of witnesses) of the query with respect to the document.

For completeness, we define explicitly the operators[6] AND and OR, which are applied to a list of input antichains $A_0, A_1, \ldots, A_{m-1}$, resulting in the $\wedge$ and $\vee$, respectively, of the antichains $A_0, A_1, \ldots, A_{m-1}$. Besides, we consider other useful operators that can be defined directly on the antichain representation [8]. With this aim, let us introduce a relation $\ll$ between intervals: $I \ll J$ iff $x < y$ for all $x \in I$ and $y \in J$.

1. *("disjunction operator")* OR, given input antichains $A_0, A_1, \ldots, A_{m-1}$, returns the set of minimal intervals among those in $A_0 \cup A_1 \cup \cdots \cup A_{m-1}$.

2. *("conjunction operator")* AND, given input antichains $A_0, A_1, \ldots, A_{m-1}$, returns the set of minimal intervals among those spanned by the tuples in $A_0 \times A_1 \times \cdots \times A_{m-1}$.

3. *("phrasal operator")* BLOCK, given input antichains $A_0, A_1, \ldots, A_{m-1}$, returns the set of *intervals* of the form $I_0 \cup I_1 \cup \cdots \cup I_{m-1}$ with $I_i \in A_i$ $(0 \leq i < m)$ and $I_{i-1} \ll I_i$ $(0 < i < m)$.

4. *("ordered non-overlapping conjunction operator")* $\mathrm{AND}_<$, given input antichains $A_0, A_1, \ldots, A_{m-1}$, returns the set of minimal intervals among those spanned by the tuples $\langle I_0, I_1, \ldots, I_{m-1} \rangle \in A_0 \times A_1 \times \cdots \times A_{m-1}$ satisfying $I_{i-1} \ll I_i$.

5. *("low-pass operator")* $\mathrm{LOWPASS}_k$, given an input antichain A, returns the set of intervals from A not longer than k.

6. *("Brouwerian difference[7] operator")* Given two antichains A (the minuend) and B (the subtrahend), the difference $A - B$ is the set of intervals $I \in A$ for which there is no $J \in B$ such that $J \subseteq I$. This operator was called "not containing" in [8].

7. *(Additional containment operators)* Three more operators can be defined in the same spirit of Brouwerian difference: in [8] they were called "containing", "contained in" and "not contained in". They are defined, for a pair of antichains A and B, as the set of intervals of A that, respectively,

   - contain an interval of B;
   - are contained in an interval of B;
   - are not contained in any interval of B.

More informally, given input antichains $A_0, A_1, \ldots, A_{m-1}$, 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 $\mathrm{AND}_<$ operator is an ordered-AND operator that returns intervals spanned by intervals coming from the A_i, much like the AND operator. However, in the case of $\mathrm{AND}_<$ the left extremes of the intervals must be nondecreasing, and the intervals must be nonoverlapping. This

[6] The reader might be slightly confused by the fact that we are using $\wedge$ and AND to denote essentially the same thing (similarly for $\vee$ and OR). The difference is that $\wedge$ is a binary operator, whereas AND has variable arity. Even if the evaluation of AND could be reduced, by associativity, to a composition of $\wedge$s, from the viewpoint of the computational effort things are quite different.

[7] This operator, denoted by the minus sign, satisfies the property that $A - B \leq C$ iff $A \leq B \vee C$; it is sometimes called *pseudo-difference* [3].

operator can be used, for instance, to search for terms that must appear in a specified order. LOWPASS_k restricts the result to intervals shorter than a given threshold, and be easily combined with AND or $\text{AND}_<$ to implement searches for terms that must not be too far apart, and possibly appear in a given order. Finally, the Brouwerian difference considers the interval in the subtrahend as "poison" and returns only those intervals in the minuend that are not poisoned by any interval in the subtrahend; this operator finds useful applications, for example, in the case of passage search if the poisoning intervals are taken to be small (possibly singleton) intervals around the passage separators (e.g., end-of-paragraph, end-of-sentence, etc.). The remaining containment operators have similar applications (see [8]).[8]

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 other 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, suppose by contradiction that for two concatenations of minimal intervals we have $[\ell\,..\,r] \subset [\ell'\,..\,r']$ (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_{m-1} strictly containing one another.

4 Lazy evaluation

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. Moreover, lazy algorithms can be combined in a hierarchical way to compute lazily the result of any query.

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 [7] 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 $ms \log n$, where n is the overall number of term positions, m is the number of terms, and s is the number of results. Nonetheless, as soon as one combines several operators, the advantage of an efficient lazy implementation is again evident, in particular for automatically expanded queries, in which m can be large.

As we already remarked, in our algorithms we restrict the operations on the elements of the underlying order O to comparisons. In particular, intervals can be handled just by comparing their extremes. In this model, the logarithmic factor in the number of antichains can be easily proved to be unavoidable for the OR operator:

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

[8] We remark that in a lattice is sometimes possible to define a relative pseudo-complement operator. This operator, denoted by an arrow, is the dual of pseudo-difference, and it satisfies the property that $A \wedge B \leq C$ iff $A \leq B \to C$ [3]. However, on one hand this operator has no interpretation in information-retrieval terms; and, on the other hand, it is easy to show that $A \to B$ can be an infinite antichain even if A and B are finite [6]. For these reasons, the computation of relative pseudo-complements will not be pursued in this paper.

Proof. It is possible to sort n distinct integers by computing the OR of n antichains, each made by just one 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 the result. ▮

4.1 Minimal and optimal laziness

The term "lazy" is usually quoted informally, in particular in the context of functional or declarative programming. In this paper we consider algorithms that access input antichains under the form of lists that return the corresponding intervals in their natural order. We want to define formally a notion of laziness that makes it possible to prove optimality results.

We restrict ourselves to algorithms that read their inputs from an array of lists. Each list is accessible via a "next" function that returns the next element from the list, and when a list is empty it returns **null**. Analogously, each algorithm has a "next" function that returns the next output element (i.e., random access is not allowed), and when the output is over it returns **null**. So such algorithms can be thought of as producing an output list that can then be fed to another operator.

Given an algorithm $\mathscr{A}$, an input I (i.e., an array of lists), let us write $\rho_i^{\mathscr{A}}(I,p)$ for the number of elements (including possibly **null**) read by $\mathscr{A}$ from the i-th list of the input array I when the p-th output is produced (sometimes, we will omit $\mathscr{A}$, I or p when they are clear from the context); when writing $\rho_i^{\mathscr{A}}(I,p)$ we shall always assume that the $0 \leq i < m$ (where m is the number of input lists) and that the output of $\mathscr{A}$ on input I contains at least p intervals.

Definition 1 Two algorithms are *functionally equivalent* iff they produce the same output list when they are given the same input lists.

A first property that we would like our algorithms to feature is that there is no algorithm that uses strictly less inputs:

Definition 2 An algorithm $\mathscr{A}$ is *minimally lazy* if, for every functionally equivalent algorithm $\mathscr{B}$ such that

$$\rho_i^{\mathscr{B}}(I,p) \leq \rho_i^{\mathscr{A}}(I,p) \tag{1}$$

for all input I, and all i and p, we actually have

$$\rho_i^{\mathscr{B}}(I,p) = \rho_i^{\mathscr{A}}(I,p). \tag{2}$$

The property above is very natural, but at the same time it is very weak: the key point is that (2) must be true *only of algorithms satisfying* (1). Minimal laziness does not rule out the existence of an algorithm $\mathscr{B}$ that reads one input element more than $\mathscr{A}$ on a single input, but uses much less input elements on all other inputs. Nonetheless, minimal laziness embodies the notion that $\mathscr{A}$ cannot be improved "locally", that is, it cannot be improved for some input without making it worse on some other input.

All algorithms described in this paper will be minimally lazy. However, for most of them we will be able to prove an additional property:

Definition 3 An algorithm $\mathscr{A}$ is *k-lazy* iff for every functionally equivalent algorithm $\mathscr{B}$, and for all input I, and all i and p, we have

$$\rho_i^{\mathscr{A}}(I,p) \leq \rho_i^{\mathscr{B}}(I,p) + k. \tag{3}$$

An algorithm $\mathscr{A}$ is *optimally k-lazy* if it is k-lazy and there exists no functionally equivalent $(k-1)$-lazy algorithm. We say it is *optimally lazy* if it is optimally k-lazy for some k.

With respect to Definition 2, the essential difference is that (3) must be true for *all* functionally equivalent algorithms $\mathscr{B}$. Algorithms that are k-lazy have some "looseness" in their usage of the input (the parameter k), but given that looseness, they beat every other algorithm. We have to introduce k because some algorithms contain choices that makes 0-laziness unattainable (i.e., depending on the order of the lists in the input array the algorithm will read more from an input list rather than from another).

If there is a k-lazy algorithm for a problem, there must be a minimum $\bar{k}$ for which there is such an algorithm, and $\bar{k}$-lazy algorithms will be optimally lazy. Optimally lazy algorithms cannot be improved "globally", that is, at the same time for all inputs:

Theorem 2 Let $\mathscr{A}$ be an optimally lazy algorithm. Then, there is no functionally equivalent algorithm $\mathscr{B}$ such that

$$\rho_i^{\mathscr{B}}(I,p) < \rho_i^{\mathscr{A}}(I,p) \tag{4}$$

for all input I, and all i and p.

Proof. By contradiction, suppose $\mathscr{A}$ is optimally k-lazy for some k and $\mathscr{B}$ is as in the statement. Then, for every algorithm $\mathscr{C}$

$$\rho_i^{\mathscr{B}}(I,p) < \rho_i^{\mathscr{A}}(I,p) \leq \rho_i^{\mathscr{C}}(I,p) + k,$$

which implies that $\mathscr{B}$ is $(k-1)$-lazy, contradicting the optimality of $\mathscr{A}$. ∎

Note that the converse is not true: an algorithm that cannot be improved globally might not be optimally lazy. However, since by definition there are no k-lazy algorithms when k is negative, 0-lazy algorithms are optimally lazy. Moreover, 0-laziness implies trivially minimal laziness. It is also easy to see that for no k minimal laziness implies (optimal) k-laziness. Our aim is at algorithms that are minimally *and* optimally lazy.

Another way of interpreting the notion of "being optimally lazy" is the following: let us say that $\mathscr{A}$ undergoes a loss of k on the triple $\langle I,i,p\rangle$ if there is another functionally equivalent algorithm $\mathscr{B}$ that reads k input elements less for the same triple. The global loss of $\mathscr{A}$ is the supremum of the losses on all triples $\langle I,i,p\rangle$: $\mathscr{A}$ is k-lazy if its global loss is k (or less). An algorithm is optimally lazy iff it has the smallest possible (finite) global loss.

There is a subtlety in Definitions 2 and 3 that is worth remarking. By requiring that the parameter p is never greater than the number of intervals in the output, we are not considering how many elements are read from the input lists to emit the final **null**. In principle, this choice implies that even minimally lazy algorithms may consume useless input elements to emit their final **null**. A more thorough analysis would be required to include also this case, but it would yield a further subdivision of the above taxonomy of optimality: indeed, for some problems we consider it is easy to show there is no **null**-optimal solution. We think that such an analysis would add little value to the present work, as behaving lazily on non-**null** outputs is a sufficiently strong property by itself.

5 General remarks

In the description and in the proofs of our algorithms, we use interchangeably A_i to denote the i-th input *antichain* and the *list* returning its intervals in their natural order (and, ultimately, **null**). This ambiguity should cause no difficulty to the reader.

To simplify the exposition, in the pseudocode we often test whether a list is empty. Of course, this is not allowed by our model, but in all such cases the following instruction retrieves the next interval from the same list. Thus, the test can be replaced by a call that retrieves the next interval and tests for **null**.

In all our algorithms, we do not consider the case of inputs equal to the top of the lattice (the antichain formed by the empty interval). For all our operators, the top either determines entirely the output (e.g., OR) or it is irrelevant (e.g., AND). Analogously, we do not consider the case of inputs equal to the bottom of the lattice (the empty antichain), which can be handled by a test on the first input read.

More generally, when proving optimal laziness, it is common to meet situations in which an initial check is necessary to rule out obvious outputs. The initial check can make the algorithm analysis more complicated, as its logic could be wildly different from the true algorithm behaviour. To simplify this kind of analysis, we prove the following metatheorem, which covers the cases just described; in the statement of the theorem, $\mathscr{A}$ represent the algorithm performing the initial check, whereas $\mathscr{B}$ does the real job:

Theorem 3 Let $\mathscr{B}$ be an algorithm defined on a set of inputs B, and $\mathscr{A}$ be defined on a larger set of inputs $A \supseteq B$, and such that

- on all inputs $I \in B$, $\mathscr{A}$ outputs a one-element list containing a special element, say $\perp$, and
- for all $I \in B$ and all i, $\rho_i^{\mathscr{A}}(I, 1) \leq \rho_i^{\mathscr{B}}(I, 1)$.

Then, there exists an algorithm, denoted by $\mathscr{A} \star \mathscr{B}$, such that

- $\mathscr{A} \star \mathscr{B}$ is functionally equivalent to $\mathscr{B}$ on B;
- $\mathscr{A} \star \mathscr{B}$ is functionally equivalent to $\mathscr{A}$ on $A \setminus B$;
- if $\mathscr{A}$ and $\mathscr{B}$ are (minimally) optimally lazy on $A \setminus B$ and B, respectively, then $\mathscr{A} \star \mathscr{B}$ is (minimally) optimally lazy on A.

Proof. Algorithm $\mathscr{A} \star \mathscr{B}$ simulates algorithm $\mathscr{A}$ and caches the input read so far. If $\mathscr{A}$ emits any element different from $\perp$, the simulation goes on until $\mathscr{A}$ is done, without caching the input any longer; otherwise, $\mathscr{A} \star \mathscr{B}$ starts executing $\mathscr{B}$ on the cached input and possibly on the remaining part of the input until $\mathscr{B}$ is done.

It is immediate to check that $\mathscr{A} \star \mathscr{B}$ is indeed functionally equivalent to $\mathscr{A}$ and $\mathscr{B}$ on $A \setminus B$ and B, respectively, and moreover

$$\rho_i^{\mathscr{A} \star \mathscr{B}}(I, p) = \begin{cases} \rho_i^{\mathscr{A}}(I, p) & \text{if } I \in A \setminus B \\ \rho_i^{\mathscr{B}}(I, p) & \text{if } I \in B. \end{cases}$$

Suppose now that $\mathscr{A}$ is a-lazy and $\mathscr{B}$ is b-lazy for some minimal a and b, and let $c = \max\{a, b\}$. For every algorithm $\mathscr{C}$ that is functionally equivalent to $\mathscr{A} \star \mathscr{B}$, we have that $\rho_i^{\mathscr{C}}(I, p) \leq \rho_i^{\mathscr{B}}(I, p) + b$ for all $I \in B$, and $\rho_i^{\mathscr{C}}(I, p) \leq \rho_i^{\mathscr{A}}(I, p) + a$ for all $I \in A \setminus B$. But then, using the observation above, $\rho_i^{\mathscr{C}}(I, p) \leq \rho_i^{\mathscr{A} \star \mathscr{B}}(I, p) + c$ for all $I \in A$, so $\mathscr{A} \star \mathscr{B}$ is c-lazy.

Suppose now that $\mathscr{C}$ is functionally equivalent to $\mathscr{A} \star \mathscr{B}$ but that it is $(c-1)$-lazy, and assume that $c = b$ (the other case is analogous). Then, for all $I \in B$, $\rho_i^{\mathscr{C}}(I, p) \leq \rho_i^{\mathscr{A} \star \mathscr{B}}(I, p) + c - 1 = \rho_i^{\mathscr{B}}(I, p) + b - 1$; but since $\mathscr{C}$ is also functionally equivalent to $\mathscr{B}$ on B, the latter inequality contradicts the minimality of b.

For minimal laziness, suppose that $\mathscr{C}$ is functionally equivalent to $\mathscr{A} \star \mathscr{B}$ and such that $\rho_i^{\mathscr{C}}(I, p) \leq \rho_i^{\mathscr{A} \star \mathscr{B}}(I, p)$ for all $I \in A$. In particular, this means that $\rho_i^{\mathscr{C}}(I, p) \leq \rho_i^{\mathscr{A}}(I, p)$ for all $I \in A \setminus B$, and $\rho_i^{\mathscr{C}}(I, p) \leq \rho_i^{\mathscr{B}}(I, p)$ for all $I \in B$. The minimal laziness of $\mathscr{A}$ and $\mathscr{B}$ imply that $\rho_i^{\mathscr{C}}(I, p) = \rho_i^{\mathscr{A}}(I, p)$ for all $I \in A \setminus B$ and $\rho_i^{\mathscr{C}}(I, p) = \rho_i^{\mathscr{B}}(I, p)$ for all $I \in B$, hence $\rho_i^{\mathscr{C}}(I, p) = \rho_i^{\mathscr{A} \star \mathscr{B}}(I, p)$ for all $I \in A$. ∎

Incidentally, we observe that $\mathscr{A} \star \mathscr{B}$ requires in general more space than $\mathscr{A}$ or $\mathscr{B}$, because of caching; nonetheless, in all our applications we will need to cache just one item per input list.

enqueue(Q,i)	insert item with index i in the queue
top(Q)	returns the minimum priority
dequeue(Q)	returns the index of an item of minimum priority and removes it from the queue
size(Q)	returns the number of items currently in the queue

Table 1: The operations available for a priority queue.

6 Algorithms based on queues

The algorithms we provide for AND and OR are inspired by the plane-sweeping technique used in [19] for their proximity algorithm, which is on its own right a variant of the standard sorted-list merge. The algorithms are implemented using a min-priority queue [15].

At each time, the queue contains a set of indices representing input lists from which at least one input has been read, and from which **null** has not been read yet. Initially, the queue is empty, and i can be added to the queue calling the function enqueue(Q,i). Priorities are represented by intervals. The priority of a list is given by the last interval read from it: for each algorithm, we will specify a different order between priorities.

The function dequeue(Q) removes and returns the list of minimum priority, whereas top(Q) returns the minimum priority, that is, the last interval read from a list of minimum priority; we refer to this interval as "the top interval". Table 1 summarises the operations available on a priority queue.

A trivial array-based implementation requires linear space (in the number of input lists) and has constant cost for all operations modifying the queue, whereas retrieving the top requires linear time. A better implementation uses a heap with linear space, logarithmic time complexity for all operations modifying the queue and constant-time top retrieval.

When using heaps, all algorithms based on priority queues have time complexity $O(n \log m)$ if the input is formed by m antichains containing n intervals overall, and use $O(m)$ space. This is immediate, as all loops contain exactly one queue advancement. The worst-case complexity of an array-based implementation is instead $O(nm)$. One should consider carefully which implementation to use, however, as in the case of a very small arity (e.g., three input lists) the array-based implementation turns out to be significantly faster in practice.

6.1 Basic comparators

Our algorithms will be based on two priority orders. The first one, denoted by $\trianglelefteq$, is defined by

$$[\ell \mathinner{.\,.} r] \trianglelefteq [\ell' \mathinner{.\,.} r'] \iff r < r' \text{ or } (r = r' \text{ and } \ell \geq \ell').$$

In other words, $[\ell \mathinner{.\,.} r] \trianglelefteq [\ell' \mathinner{.\,.} r']$ if $[\ell \mathinner{.\,.} r]$ ends before or is a suffix of $[\ell' \mathinner{.\,.} r']$. Note in particular that (somewhat counterintuitively) $[\ell \mathinner{.\,.} r] \trianglelefteq [\ell' \mathinner{.\,.} r]$ iff $\ell \geq \ell'$.

The second order, denoted by $\preceq$, is defined by

$$[\ell \mathinner{.\,.} r] \preceq [\ell' \mathinner{.\,.} r'] \iff \ell < \ell' \text{ or } (\ell = \ell' \text{ and } r \geq r').$$

In other words, $[\ell \mathinner{.\,.} r] \preceq [\ell' \mathinner{.\,.} r']$ if $[\ell \mathinner{.\,.} r]$ starts before or prolongs $[\ell' \mathinner{.\,.} r']$. Note in particular that $[\ell \mathinner{.\,.} r] \preceq [\ell \mathinner{.\,.} r']$ iff $r \geq r'$, and that the following implication holds:

$$[\ell \mathinner{.\,.} r] \subseteq [\ell' \mathinner{.\,.} r'] \implies [\ell \mathinner{.\,.} r] \trianglelefteq [\ell' \mathinner{.\,.} r'] \text{ and } [\ell' \mathinner{.\,.} r'] \preceq [\ell \mathinner{.\,.} r]$$

The algorithms for AND/OR use a priority queue with priority order $\preceq$ or $\trianglelefteq$. In the initialisation phase, we read an interval from each list, and the queue contains all lists.

To simplify the description, we define a procedure advance(Q) that updates with the next interval a list of minimum priority. If the update cannot be performed because the list is empty, the list is dequeued. The function is described in pseudocode in Algorithm 1.

Algorithm 1 The advance function.

procedure advance(Q) **begin**
  $i \leftarrow$ dequeue(Q);
  if A_i is not empty **then**
    next(A_i);
    enqueue(Q,i)
  end;
end;

6.2 The OR operator

We start with the simplest nontrivial operator. To compute the OR of the antichains A_0, A_1, $\dots$, A_{m-1}, we merge them using a priority queue Q with priority order $\trianglelefteq$.

We keep track of the last interval c returned (initially, $c = [-\infty \,..\, -\infty]$). When we want to compute the next interval, we advance Q as long as the top interval contains c, and then if the queue is not empty we return the top interval. The algorithm[9] is described in pseudocode in Algorithm 2.

Theorem 4 Algorithm 2 for OR is correct.

Proof. First of all, note that all intervals in A_0, A_1, $\dots$, A_{m-1} are assigned to c at some point, unless they contain a previously returned interval. Thus, we just have to prove that only minimal intervals are returned.

Let $[\ell \,..\, r]$ be a non-minimal element of $A_0 \cup A_1 \cup \dots \cup A_{m-1}$, and $[\ell' \,..\, r']$ the *largest* (according to $\trianglelefteq$) minimal interval contained in $[\ell \,..\, r]$. After returning $[\ell' \,..\, r']$ (which certainly appears at the top of the queue *before* $[\ell \,..\, r]$ due to the fact that $\subseteq$ implies $\trianglelefteq$), all intervals in the queue have a right extreme larger than or equal to r'. When we advance the queue, and until we get past $[\ell \,..\, r]$, the top interval will always contain $[\ell' \,..\, r']$, for otherwise there would be a minimal interval with right extreme between r' and r, and $[\ell' \,..\, r']$ would not be largest. Thus, the while loop will eventually remove $[\ell \,..\, r]$.

To prove that all returned intervals are unique, we just have to note that when I is returned, each other copy of I is the last interval read from some list. Thus, at the next call the while loop will be repeated until all remaining copies are discarded. ∎

Theorem 5 Algorithm 2 for OR is 0-lazy (and thus optimally and minimally lazy).

[9] Note that this algorithm, as discussed in Section 8, can be derived from the dominance algorithms presented in [16].

Algorithm 2 The algorithm for the OR operator. Note that the second part of the while condition is actually equivalent to "left(top(Q)) $\leq$ left(c)" due to the monotonicity of the top-interval right extreme.

```
Initially c ← [−∞ .. −∞] and Q contains one interval from each A_i.
function next begin
   while Q is not empty and c ⊆ top(Q) do
      advance(Q)
   end;
   if Q is empty then return null;
   c ← top(Q);
   return c
end;
```

Proof. The first output of the algorithm (let us call it $\mathscr{A}$) requires reading exactly one interval from each list. No correct algorithm can emit the first output without this data.

Suppose now that for an algorithm $\mathscr{A}^*$ it happens that

$$\rho_i^{\mathscr{A}^*}(I,p) < \rho_i^{\mathscr{A}}(I,p)$$

for some input I and some i and p. Upon returning the p-th output $[\ell\,..\,r]$ we have just read from each list the least interval (w.r.t. $\trianglelefteq$) after $[\ell\,..\,r]$; hence, $\mathscr{A}^*$ emits $[\ell\,..\,r]$ having read from the i-th input list an interval $[\ell'\,..\,r']$ strictly smaller than $[\ell\,..\,r]$ according to $\trianglelefteq$; this means that either $r' < r$, or $r' = r$ and $\ell < \ell'$, but the latter case is ruled out by minimality of $[\ell\,..\,r]$. Thus, $r' < r$, and $\mathscr{A}^*$ would return an incorrect output if the i-th input list would return $[s\,..\,s]$ as next input, with $r' < s < r$. ∎

6.3 The AND operator

The AND operator is more challenging. The priority order of Q is $\preceq$, and additionally the queue keeps track of the largest right extreme of any interval ever read, which we will call *the right extreme of* Q (we just need a variable that is maximised with the right extreme of each new input interval). We say that Q is *full* if it contains exactly m indices, where again m is the number of input antichains.

At any time, the interval *spanned* by Q is the interval defined by the left extreme of the top interval and the right extreme of Q: it will be denoted by span(Q). Clearly, it is the minimum interval containing all intervals currently in the queue.

We keep track of the last interval c returned (initially, $c = [-\infty\,..\,-\infty]$). When we want to compute the next interval, we first advance Q until the spanned interval does not contain c, and in case Q is no longer full we return **null**. Then, we store the interval $[\ell\,..\,r]$ currently spanned by Q as a candidate and advance Q. If the new interval spanned by Q is included in $[\ell\,..\,r]$ we repeat the operation, updating the candidate. Otherwise (or if Q is no longer full) we just return the candidate. The algorithm is described in pseudocode in Algorithm 3.

Theorem 6 Algorithm 3 for AND is correct.

Proof. We say that a queue configuration is *complete* if it contains all copies of the top interval from all lists that contain it. Now observe that *every complete configuration of a*

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

```
Initially c ← [−∞ . . −∞] and Q contains one interval from every A_i.
function next begin
   while Q is full and c ⊆ span(Q) do
      advance(Q)
   end;
   if Q is not full then return null;
   do
      c ← span(Q);
      if c = top(Q) then return c;
      advance(Q)
   while Q is full and span(Q) ⊆ c;
   return c
end;
```

priority queue is entirely defined by its 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 (according to $\preceq$). 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 when it has been dequeued the top must have become 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 configuration of the queue to be complete.

We now show that for every minimal interval $[\ell \,..\, r]$ in the AND of A_0, A_1, $\ldots$, A_{m-1} there is a complete configuration of Q spanning $[\ell \,..\, r]$. Consider for each i the set C_i of intervals of A_i contained in $[\ell \,..\, r]$. At least one of these sets must contain a (necessarily unique) *right delimiter*, that is, an interval of the form $[\ell' \,..\, r]$ (see Figure 2). 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: the maximum of such right extremes, say $r' < r$, would define a spanned interval $[\ell \,..\, r']$ showing that $[\ell \,..\, r]$ was not minimal. We conclude that at least one C_i, say $C_{\bar{\imath}}$, is a singleton containing a right delimiter.

Let I_i be the leftmost interval in each C_i; these intervals are a complete configuration of Q: if $I_i = [\ell \,..\, r']$ is the $\preceq$-smallest among such intervals 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 $[\ell \,..\, r]$ (because the right extreme of $I_{\bar{\imath}}$ is r, and the left extreme of the $\preceq$-least interval I_i is ℓ). We conclude that all minimal intervals in the output are eventually spanned by Q.

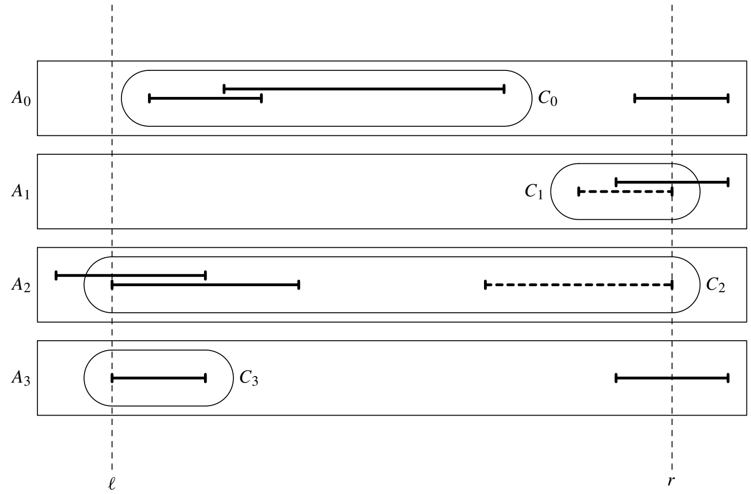

Figure 2: A sample configuration found in the proofs of Theorems 6 and 7. The dashed intervals are right delimiters. The first two input lists are in the inner set; the last two input lists are in the conflict set; the last input list is also in the resolution set.

However, no minimal interval can be spanned during the first while loop, unless it has been already returned, as all intervals spanned in that loop contain a previously returned interval (notice that at the first call the loop is skipped altogether). Finally, if an interval is spanned in the second 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 second while loop, and are thus returned.

We are left to prove that if an interval is returned, it is guaranteed to be minimal. If we exit the loop using the check on the top interval, the returned interval is indeed guaranteed to be minimal. Otherwise, assume that the interval $[\ell \mathinner{.\,.} r]$ spanned by Q at the start of the second while loop is not minimal, so $[\bar{\ell} \mathinner{.\,.} r] \subset [\ell \mathinner{.\,.} r]$, for some minimal interval $[\bar{\ell} \mathinner{.\,.} r]$ that will be spanned later (as we already proved that all minimal intervals are returned). Since the right extreme of Q is nondecreasing, the second while loop will pass through intervals of the form $[\ell' \mathinner{.\,.} r]$, with $\ell < \ell' < \bar{\ell}$, until we exit the loop.

Finally, we remark the uniqueness of all returned intervals is guaranteed by the first while loop. ∎

Note that our algorithm for AND *cannot* be 0-lazy, because the choices made by the queue for equal intervals cause different behaviours. For instance, on the input lists $\{[0 \mathinner{.\,.} 0], [2 \mathinner{.\,.} 2]\}$, $\{[1 \mathinner{.\,.} 1]\}$, $\{[0 \mathinner{.\,.} 0], [2 \mathinner{.\,.} 2]\}$ the algorithm advances the last list before returning $[0 \mathinner{.\,.} 1]$, but there is a variant of the same algorithm that keeps intervals sorted lexicographically by $\preceq$ and by input list index, and this variant would advance the first list instead.

Nonetheless:

Theorem 7 Algorithm 3 for AND is minimally lazy and optimally 1-lazy.

Proof. We denote Algorithm 3 with $\mathscr{A}$, and let $\mathscr{A}^*$ be a functionally equivalent algorithm. Let us number the intervals appearing in a certain input $I = A_0, A_1, \ldots, A_{m-1}$: in particular, let $\left[\ell_i^j \mathinner{.\,.} r_i^j\right]$ be the j-th interval appearing in A_i. For sake of simplicity, let us identify the **null** returned as last element by the input lists with the interval $[\infty \mathinner{.\,.} \infty]$ (it is immediate to see that $\mathscr{A}$ behaves identically). Let us write ρ_i (respectively, ρ_i^*) for $\rho_i^{\mathscr{A}}(I, p)$ (respectively,

$\rho_i^{\mathscr{A}^*}(I,p)$), and $[\ell\,..\,r]$ be the p-th output interval; let also s_i be the index of the first interval in list A_i that is included in $[\ell\,..\,r]$.

We divide the indices of the input lists in two sets: the *inner set* is the set of indices i for which $\ell < \ell_i^{s_i}$ (that is, the first interval of A_i included in $[\ell\,..\,r]$ has left extreme larger than ℓ); the *conflict set* is the set of indices i for which $\ell = \ell_i^{s_i}$ (that is, the first interval of A_i included in $[\ell\,..\,r]$ has left extreme equal to ℓ). Finally, the *resolution set* is a subset of the conflict set containing those indices i for which $r_i^{s_i+1} > r$ (that is, the successor of the first interval of A_i included in $[\ell\,..\,r]$ is no longer contained in $[\ell\,..\,r]$). Note that the resolution set is always nonempty, or otherwise $[\ell\,..\,r]$ would not be minimal (recall that we substituted **null** with $[\infty\,..\,\infty]$). The situation is depicted in Figure 2.

We remark the following facts:

(i). for all i, $\rho_i^* \geq s_i$; that is, when $\mathscr{A}^*$ outputs $[\ell\,..\,r]$ it has read at least the first interval of the antichain with left extreme larger than or equal to ℓ; otherwise, $\mathscr{A}^*$ would emit $[\ell\,..\,r]$ even on a modified input in which A_i has no intervals contained in $[\ell\,..\,r]$ (such intervals have index equal to or greater than s_i, so they have not been seen by $\mathscr{A}^*$, yet);

(ii). for all i in the inner set, $\rho_i = s_i \leq \rho_i^*$;

(iii). for all i in the conflict set, $\rho_i \in \{s_i, s_i+1\}$; that is, in the case an antichain *does* contain an interval J with left extreme ℓ, either the last interval read by $\mathscr{A}$ when $[\ell\,..\,r]$ is output is exactly J, or it is the interval just after J;

(iv). if for some i we have $\left[\ell_i^{s_i}\,..\,r_i^{s_i}\right] = [\ell\,..\,r]$, then $\rho_j = s_j$ for all j, because we exit the second while loop at line 8;

(v). otherwise, there is a unique index $\bar\imath$ in the resolution set such that $\rho_{\bar\imath} = s_{\bar\imath}+1$ (i.e., $r_{\bar\imath}^{\rho_{\bar\imath}} > r$), and for all other resolution indices i we have $\rho_i = s_i$ (i.e., $r_i^{\rho_i} \leq r$); this happens because we interrupt the second while loop when we see the first interval whose right extreme exceeds r (at line 10).

Let us first prove that $\mathscr{A}$ is 1-lazy by showing that $\rho_i \leq \rho_i^*+1$: this is true for all indices in the inner set because of (ii), and for all indices in the conflict set because of $\rho_i \leq s_i+1 \leq \rho_i^*+1$ (by (iii) and (i)).

Now, let us show that $\mathscr{A}^*$ cannot be 0-lazy. Suppose it is such; then, in particular, $\rho_i^* \leq \rho_i$ for all indices i, and we can assume w.l.o.g. that $\rho_i^* < \rho_i$ for some i (if for all inputs, all output prefixes and all i we had $\rho_i^* = \rho_i$, then we would conclude that $\mathscr{A}$ is 0-lazy as well, contradicting the observation made before this theorem).

Note that we can also assume w.l.o.g. not to be in case (iv) (as in that case $\rho_i = \rho_i^*$ for all i), which also implies that $\ell \neq r$. Thus, the unique index $\bar\imath$ of (v) is also the only index in the resolution set such that $\rho_{\bar\imath}^* = s_{\bar\imath}+1$ ($\mathscr{A}^*$ must advance some list in the resolution set, or it would emit a wrong output on a modified input in which the (s_i+1)-th interval of A_i is $[r\,..\,r]$ for all i in the conflict set).

Let i_0, i_1, $\ldots$, i_{t-1} be the indices in the conflict set for which $\rho_{i_p} = s_{i_p}+1$, in the order in which they are accessed from the corresponding lists by $\mathscr{A}$: clearly $i_{t-1} = \bar\imath$ is the only resolution index in this sequence, by (v). Let j_0, j_1, $\ldots$, j_{u-1} be the indices in the conflict set for which $\rho_{j_p}^* = s_{j_p}+1$, in the order in which they are accessed from the corresponding lists by $\mathscr{A}^*$. Necessarily, $\{j_0, j_1, \ldots, j_{u-1}\} \subseteq \{i_0, i_1, \ldots, i_{t-1}\}$ (because $s_{j_p}+1 = \rho_{j_p}^* \leq \rho_{j_p} \leq s_{j_p}+1$, hence $\rho_{j_p} = s_{j_p}+1$) and inclusion is strict (because, for some index i, $\rho_i^* < \rho_i$, hence $s_i \leq \rho_i^* < \rho_i \leq s_i+1$, which implies that $i = i_v$ for some v, whereas $i \neq j_v$ for all v).

Let p be the first position that $\mathscr{A}$ and $\mathscr{A}^*$ choose differently, that is, $i_p \neq j_p$ (this happens at least at the position of $j_0, j_1, \ldots, j_{u-1}$ where $\bar{\imath}$ appears). We build a new input similar to $A_0, A_1, \ldots, A_{m-1}$, except for A_{i_p} and A_{j_p}, which are identical up to their interval of left extreme ℓ; then, A_{i_p} continues with $[r' \,..\, r']$ for some $r' > r$ (so i_p is in the resolution set), whereas A_{j_p} continues with $[r \,..\, r]$ (so j_p is in the inner set). On this input, to output $[\ell \,..\, r]$ $\mathscr{A}$ advances the input list A_{j_p} strictly less than $\mathscr{A}^*$, which contradicts the assumption on $\mathscr{A}^*$. ▮

7 Greedy algorithms

The remaining operators admit greedy algorithms: they advance the input lists until some condition becomes true. The case of $\mathrm{LOWPASS}_k$ is of course trivial, and the algorithm for BLOCK is essentially a restatement in terms of intervals of the folklore algorithm for phrasal queries. They are minimally and optimally lazy. The case of $\mathrm{AND}_<$ and Brouwerian difference are more interesting: $\mathrm{AND}_<$ is the only algorithm for which we prove the impossibility of an optimally lazy implementation in the general case.

All greedy algorithms have time complexity $O(n)$ if the input is formed by m antichains containing n intervals overall, and use $O(m)$ space. This is immediate, as all loops advance at least one input list.

7.1 The BLOCK operator

The BLOCK operator is the only one that can be implemented exclusively if the underlying total order is locally finite,[10] that is, if it admits a notion of successor. In discussing this algorithm, we shall assume that every element $x \in O$ has a successor, denoted by $x + 1$, satisfying $x < x + 1$ and $x \leq y \leq x + 1 \implies x = y$ or $y = x + 1$.

We keep track of a current interval for all lists $A_0, A_1, \ldots, A_{m-1}$; initially, these intervals are set to $[-\infty \,..\, -\infty]$. 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 A_i until the returned interval has left extreme larger than the right extreme of the current interval for A_{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 A_{m-1} we return the interval spanned by all current intervals. The algorithm is described in pseudocode in Algorithm 4.

Theorem 8 Algorithm 4 for BLOCK is correct.

Proof. At the start of an iteration of the external while loop (line 5) 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 of the loop we certainly return a correct interval.

To complete the proof, we start by proving the following invariant property: at line 5, for all $0 < j < m$ there are no intervals in A_j with left extreme in $[r_{j-1} + 1 \,..\, \ell_j - 1]$. In other words, the j-th current interval $[\ell_j \,..\, r_j]$ 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 $[\ell_0 \,..\, r_0]$ cannot change this fact. We are left to prove that the execution of the internal while loop (line 6) cannot either.

During the execution of the loop at line 6, only $[\ell_i \,..\, r_i]$ can change. This affects the invariant because it modifies the intervals $[r_{i-1} + 1 \,..\, \ell_i - 1]$ and $[r_i + 1 \,..\, \ell_{i+1} - 1]$, but in

[10] A partially ordered set is *locally finite* if all intervals of the form $[x \,..\, y]$ are finite.

Algorithm 4 The algorithm for the BLOCK operator.

Initially $[\ell_k \mathinner{..} r_k] \leftarrow [-\infty \mathinner{..} -\infty]$ for all $0 \le k < m$.
function next **begin**
  if A_0 is empty **then return null**;
  $[\ell_0 \mathinner{..} r_0] \leftarrow \text{next}(A_0)$;
  $i \leftarrow 1$;
  while $i < m$ **do**
    while $\ell_i \le r_{i-1}$ **do**
      if A_i is empty **then return null**;
      $[\ell_i \mathinner{..} r_i] \leftarrow \text{next}(A_i)$
    end;
    if $\ell_i = r_{i-1} + 1$ **then** $i \leftarrow i + 1$
    else begin
      if A_0 is empty **then return null**;
      $[\ell_0 \mathinner{..} r_0] \leftarrow \text{next}(A_0)$;
      $i \leftarrow 1$
    end
  end;
  return $[\ell_0 \mathinner{..} r_{m-1}]$
end;

the second case 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 $r_{i-1} + 1 \le \ell_i - 1$, 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 $[r_{i-1} + 1 \mathinner{..} \ell_i - 1]$ is empty, and the loop will advance $[\ell_i \mathinner{..} r_i]$ up to the first interval in A_i with a left extreme larger than r_{i-1}, making again the invariant true.

Suppose now that there are $[\bar\ell_0 \mathinner{..} \bar r_0]$, $[\bar\ell_1 \mathinner{..} \bar r_1]$, $\ldots$, $[\bar\ell_k \mathinner{..} \bar r_k]$ satisfying $\bar r_i + 1 = \bar\ell_{i+1}$ for some $k > 0$ and $0 \le i < k$. 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 $[\ell_j \mathinner{..} r_j] = [\bar\ell_j \mathinner{..} \bar r_j]$ 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 $\ell_j = \bar\ell_j$, $r_j = \bar r_j$ for $j = 0, 1, \ldots k - 1$. Because of the invariant, either $[\ell_k \mathinner{..} r_k] = [\bar\ell_k \mathinner{..} \bar r_k]$ or $[\ell_k \mathinner{..} r_k]$ will be advanced by the execution of the internal while loop up to $[\bar\ell_k \mathinner{..} \bar r_k]$. 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_{m-1} are returned.

We note that all intervals returned are unique (minimality has been already discussed in Section 3), as $[\ell_0 \mathinner{..} r_0]$ is advanced at each call, so a duplicate returned interval would imply the existence of two comparable intervals in A_0. ∎

Theorem 9 Algorithm 4 for BLOCK is 0-lazy (and thus optimally and minimally lazy).

Proof. The algorithm is trivially minimally lazy, as all outputs are uniquely determined by a tuple of intervals from the inputs. An algorithm $\mathscr{A}^*$ advancing an input list A_i less than Algorithm 4 for some output $[\ell \mathinner{..} r]$ would emit $[\ell \mathinner{..} r]$ even if we truncated A_i after the last interval read by $\mathscr{A}^*$. ∎

Practical remarks. In the case of intervals of integers, the advancement of the first list at the end of the outer loop can actually be iterated until $r_0 \geq \ell_i - i$. This change does not affect the complexity of the algorithm, but it may reduce the number of iterations of the outer loop. In case the input antichains are entirely formed by singletons[11], a folklore algorithm aligns the singletons circularly rather than starting from the first one (since they are singletons, once the position of an interval is fixed all the remaining ones are, too). The main advantage is that of avoiding to resolve several alignments if the first few terms appear often consecutively, but not followed by the remaining ones.

7.2 The $\mathrm{AND}_<$ operator

The algorithm for computing this operator is 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 then our current candidate $[\ell' \mathinner{..} r']$; as in the case of BLOCK, there is no queue and the lists A_0, A_1, …, A_{m-1} are advanced greedily. Again, we keep track of a current interval $[\ell_i \mathinner{..} r_i]$ for every list A_i; initially, these intervals are $[-\infty \mathinner{..} -\infty]$, except for the first one, which is taken from the first list. The algorithm is described in pseudocode in Algorithm 5; an informal description follows.

Algorithm 5 The algorithm for the $\mathrm{AND}_<$ operator. For sake of simplicity, we use the convention that returning $[\infty \mathinner{..} \infty]$ means returning **null**, and that if one of the input lists is exhausted the function returns **null**.

0 **Initially** $[\ell_0 \mathinner{..} r_0] \leftarrow \mathrm{next}(A_0)$, $[\ell_k \mathinner{..} r_k] \leftarrow [-\infty \mathinner{..} -\infty]$ for all $0 < k < m$ and $i \leftarrow 1$.
1 **function** next **begin**
2   $[\ell' \mathinner{..} r'] \leftarrow [\infty \mathinner{..} \infty]$;
3   $b \leftarrow \infty$;
4   **forever**
5   **forever**
6   **if** $r_{i-1} \geq b$ **then return** $[\ell' \mathinner{..} r']$;
7   **if** $i = m$ **or** $\ell_i > r_{i-1}$ **then break**;
8   **do**
9   **if** $r_i \geq b$ **or** A_i is empty **then return** $[\ell' \mathinner{..} r']$;
10         $[\ell_i \mathinner{..} r_i] \leftarrow \mathrm{next}(A_i)$
11   **while** $\ell_i \leq r_{i-1}$;
12       $i \leftarrow i + 1$;
13   **end**;
14     $[\ell' \mathinner{..} r'] \leftarrow [\ell_0 \mathinner{..} r_{m-1}]$;
15     $b \leftarrow \ell_{m-1}$;
16     $i \leftarrow 1$;
17   **if** A_0 is empty **then return** $[\ell' \mathinner{..} r']$;
18     $[\ell_0 \mathinner{..} r_0] \leftarrow \mathrm{next}(A_0)$
19   **end**;
20 **end**;

The core of the algorithm is in the loop starting at line 8: this loop tries to align the i-th

[11] We emphasise this case because this is what happens with phrasal queries all of whose subqueries are simple terms; implementation may treat this special case differently to obtain further optimisation, for instance using *ad hoc* indices [22].

interval, that is, advance it until $[\ell_{i-1} \mathinner{.\,.} r_{i-1}] \ll [\ell_i \mathinner{.\,.} r_i]$. The loop starting at line 5 aims at aligning all intervals; note that we assume as an invariant that, after the first execution, every time we discover that the i-th interval is already aligned we can conclude that also the remaining intervals (the ones with index larger than i) are aligned as well (second condition at line 7).

The loop at line 5 can be interrupted as soon as, trying to align the i-th interval, we exhaust the i-th list or we find an interval whose right extremes exceeds b, the left extreme of the $(m-1)$-th interval forming the current candidate alignment. If any such condition is satisfied, the current candidate is certainly minimal and can thus be returned.

Upon a successful alignment (line 14), we have a new candidate: note that either this is the first candidate (i.e., $[\ell' \mathinner{.\,.} r'] = [\infty \mathinner{.\,.} \infty]$ before the assignment), or its right extreme coincides with the one of the previous candidate (i.e., $r' = r_{m-1}$ before the assignment), whereas its left extreme is certainly strictly larger. In either case, we try to see if we can advance the first interval and find a new, smaller candidate with a new alignment: this should explain the outer loop.

Theorem 10 Algorithm 5 for $\mathrm{AND}_<$ is correct.

Proof. Let us say that a sequence $[\ell'_h \mathinner{.\,.} r'_h] \ll [\ell'_{h+1} \mathinner{.\,.} r'_{h+1}] \ll \cdots \ll [\ell'_{k-1} \mathinner{.\,.} r'_{k-1}]$ of intervals ($h < k \leq m$), one from each list $A_h, A_{h+1}, \ldots, A_{k-1}$, is *leftmost* if, for all $h < j < k$, there are no intervals in A_j with left extreme in $(r'_{j-1} \mathinner{.\,.} \ell'_j)$: such a sequence is uniquely determined by k and by $[\ell'_h \mathinner{.\,.} r'_h]$. Let $[\bar\ell \mathinner{.\,.} \bar r]$ be the interval returned at the last call (initially, $[\bar\ell \mathinner{.\,.} \bar r] = [\infty \mathinner{.\,.} \infty]$). Then, the following invariant holds at the start of the loop at line 5:

1. $[\ell_0 \mathinner{.\,.} r_0] \ll [\ell_1 \mathinner{.\,.} r_1] \ll \cdots \ll [\ell_{i-1} \mathinner{.\,.} r_{i-1}]$ is leftmost;
2. if $\ell_i \neq -\infty$ also $[\ell_i \mathinner{.\,.} r_i] \ll [\ell_{i+1} \mathinner{.\,.} r_{i+1}] \ll \cdots \ll [\ell_{m-1} \mathinner{.\,.} r_{m-1}]$ is leftmost;
3. if $[\ell_{i-1} \mathinner{.\,.} r_{i-1}] \ll [\ell_i \mathinner{.\,.} r_i]$ then this pair is leftmost.

The fact that this invariant holds is easy to check; in particular, see the inner while loop at line 8 and the exit at line 7.

We now show that each output interval $[\bar\ell \mathinner{.\,.} \bar r]$ is at some time assigned to $[\ell' \mathinner{.\,.} r']$. Note that $i > 0$ at all times, so $[\ell_0 \mathinner{.\,.} r_0]$ is assigned only at the end of the infinite loop. This means that $[\ell_0 \mathinner{.\,.} r_0]$ runs through the whole first input list.

Thus, as soon as $\ell_0 = \bar\ell$ the inner loop will either compute the leftmost representation of $[\bar\ell \mathinner{.\,.} \bar r]$, or exit prematurely. In the second case, the function will necessarily complete the leftmost representation at the next call. We conclude that leftmost representations of all output intervals are assigned to $[\ell' \mathinner{.\,.} r']$ eventually: since $[\bar\ell \mathinner{.\,.} \bar r]$ is minimal, it will be emitted before $[\ell' \mathinner{.\,.} r']$ is assigned again. Uniqueness follows by uniqueness of leftmost representations. ∎

It is not difficult to see that there is *no algorithm* for $\mathrm{AND}_<$ that is k-lazy for any k, except for the case $m = 2$; indeed:

Theorem 11 If $m > 2$, there exist no optimally lazy algorithm for $\mathrm{AND}_<$.

Proof. By contradiction, let $\mathscr{B}$ be k-lazy, and observe that, on any given input I, every algorithm for $\mathrm{AND}_<$, before emitting its p-th output $[\ell \mathinner{.\,.} r]$, must have reached at least the leftmost sequence $[\ell'_0 \mathinner{.\,.} r'_0] \ll [\ell'_1 \mathinner{.\,.} r'_1] \ll \cdots \ll [\ell'_{m-1} \mathinner{.\,.} r'_{m-1}]$ spanning it. Now, choose any $x \in (r'_{m-2} \mathinner{.\,.} \ell'_{m-1})$ and, for all $i = 0, \ldots, m-2$, take an arbitrary sequence $u_i^0 < u_i^1 < u_i^2 < \cdots < u_i^{k+1} \in (r'_i \mathinner{.\,.} \min\{\ell'_{i+1}, x\})$; also choose an arbitrary sequence $v^0 < v^1 < v^2 < \cdots < v^{k+1} \in (x \mathinner{.\,.} \ell'_{m-1})$. Run $\mathscr{B}$ on a different input J, obtained as follows: whenever $\mathscr{B}$ asks for

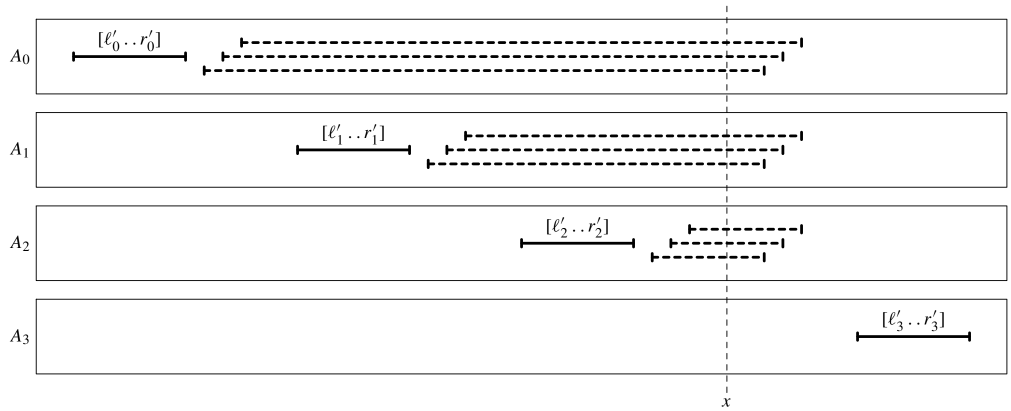

Figure 3: A sample configuration found in the proof of Theorem 11. In this case, $m = 4$ and $k = 1$. The dashed intervals are those of the form $\left[u_i^j \,..\, v^j\right]$: while reading such intervals it is impossible to decide whether the continuous intervals span an element of the output.

an input from list $i < m-1$, we use the original intervals from I only up to $[\ell'_i \,..\, r'_i]$, and then we do the following: if $i < m-1$, we start offering $[u_i^0 \,..\, v^0]$, $[u_i^1 \,..\, v^1]$ and so on; as far as the last input list is concerned, we do not make any change. An example of this construction is given in Figure 3.

Note that the intervals are chosen so that $\mathscr{B}$ cannot yet emit $[\ell \,..\, r]$, because there is always some chance for it not to be minimal. We stop testing $\mathscr{B}$ as soon as, for some $\bar{\imath}$, $\mathscr{B}$ has read at least $k+2$ inputs after $[\ell'_{\bar{\imath}} \,..\, r'_{\bar{\imath}}]$ from the $\bar{\imath}$-th list for some $\bar{\imath} < m-1$; let J' be the portion of J read by $\mathscr{B}$ so far, let $\bar{\jmath}$ any index different from $\bar{\imath}$ and from $m-1$, and let $\mathscr{A}$ be an algorithm for $\mathrm{AND}_<$ obtained from $\mathscr{B}$ by modifying its behaviour on the input as follows: when faced with an input that coincides with I up to $[\ell'_0 \,..\, r'_0] \ll [\ell'_1 \,..\, r'_1] \ll \cdots \ll [\ell'_{m-1} \,..\, r'_{m-1}]$ inclusive, it then reads one more interval for each list and, if all these intervals contain any common point, say z, it starts reading from list $\bar{\jmath}$ until an interval not including z is reached, or until the $\bar{\jmath}$-th list ends, in which case it emits $[\ell \,..\, r]$. Note that this modification does not harm the correctness of the algorithm, but now $\rho_{\bar{\imath}}^{\mathscr{A}}(J', p) + k + 1 = \rho_{\bar{\imath}}^{\mathscr{B}}(J', p)$ which contradicts the k-laziness of $\mathscr{B}$. ∎

Hence, for $\mathrm{AND}_<$, there is no hope for our algorithm to be optimally lazy in the general case; yet, it enjoys three interesting properties:

Theorem 12 Let $\mathscr{A}$ be Algorithm 5 for $\mathrm{AND}_<$.

1. $\mathscr{A}$ is minimally lazy;
2. $\mathscr{A}$ is 0-lazy (and thus optimally and minimally lazy) when $m = 2$;
3. for any functionally equivalent algorithm $\mathscr{B}$, $\rho_i^{\mathscr{A}}(I, p) \leq \rho_i^{\mathscr{B}}(I, p+1)$; that is, our algorithm, to produce any output, never reads more input than $\mathscr{B}$ needs to produce its next output.

Proof. (1) Suppose that $\mathscr{B}$ is functionally equivalent to $\mathscr{A}$ and $\rho_j^{\mathscr{B}}(I, p) \leq \rho_j^{\mathscr{A}}(I, p)$ for every j, I and p, and $\rho_{\bar{\jmath}}^{\mathscr{B}}(\bar{I}, \bar{p}) < \rho_{\bar{\jmath}}^{\mathscr{A}}(\bar{I}, \bar{p})$ for some specific $\bar{\jmath}$, $\bar{I}$ and $\bar{p}$. Let $[\ell \,..\, r]$ be the $\bar{p}$-th output on input $\bar{I}$, and $[\ell'_0 \,..\, r'_0] \ll \ldots \ll [\ell'_{m-1} \,..\, r'_{m-1}]$ be its leftmost spanning sequence

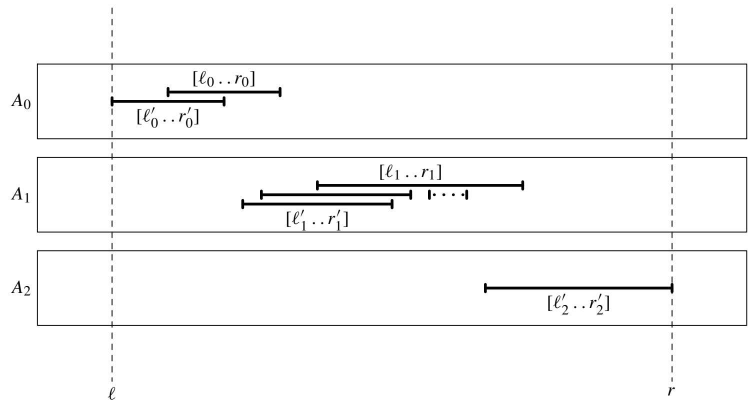

Figure 4: A sample configuration found in the proof of Theorem 12. The intervals $[\ell'_i\,..\,r_i]$ form a leftmost spanning sequence, and $i = 1$, so $\bar{\jmath} = 0$. Note that no algorithm can avoid reading $[\ell_1\,..\,r_1]$, or it would fail if we replaced it with the dotted interval.

(Figure 4 displays an example); when $\mathscr{A}$ outputs $[\ell\,..\,r]$, we have that $[\ell_j\,..\,r_j] = [\ell'_j\,..\,r'_j]$ for all $j > i$, whereas the i-th list is over or is such that $r_i \geq \ell'_{m-1}$ (with leftmost r_i), $[\ell_0\,..\,r_0] \ll \ldots \ll [\ell_{i-1}\,..\,r_{i-1}]$ is leftmost and $\ell < \ell_0$. Since no correct algorithm can emit $[\ell\,..\,r]$ before scanning its input up to the leftmost spanning sequence, necessarily $\bar{\jmath} \leq i$.

Moreover, necessarily $\bar{\jmath} \neq i$: otherwise, we could modify the inputs by substituting the unread intervals of the lists $A_i, A_{i+1}, \ldots, A_{m-2}$ with a suitable sequence of aligned intervals which, together with the remaining ones, would span $[\ell_0\,..\,r]$; this would make $[\ell\,..\,r]$ non minimal.

Now, suppose that J is an input equal to $\bar{I}$ but modified so that the $\bar{\jmath}$-th list ends immediately after the last interval read by $\mathscr{B}$: on input J, algorithm $\mathscr{A}$ does not read a single interval from list i beyond $[\ell'_i\,..\,r'_i]$, because it emits $[\ell\,..\,r]$ as soon as the test for emptiness of $A_{\bar{\jmath}}$ is performed. So $\rho_i^{\mathscr{A}}(J, \bar{p}) < \rho_i^{\mathscr{B}}(J, \bar{p})$, a contradiction.

(2) We prove that $\mathscr{A}$ is 0-lazy in that case. Indeed, when a certain output $[\ell\,..\,r]$ is ready to be produced, $\mathscr{A}$ tries to read one more interval $[\ell_0\,..\,r_0]$ from the first list, and this is unavoidable (any other algorithm must do this, or otherwise we might modify the next interval so that $[\ell\,..\,r]$ is not minimal). This interval has a right extreme larger than or equal to ℓ_1, or otherwise $[\ell\,..\,r]$ would not be minimal: $\mathscr{A}$ exits at this point, so it is 0-lazy.

(3) This is trivial: when $\mathscr{A}$ outputs an interval, it has not yet reached (or, it has just reached) the leftmost sequence spanning the following output, and no correct algorithm could ever emit the next output before that point. ▮

Practical remarks. In the case of intervals of integers, the check for $r_i \geq b$ can replaced by $r_i \geq b - (m - i - 2)$, and the check for $r_{i-1} \geq b$ by $r_i \geq b - (m - i - 1)$, obtaining in some case faster detection of minimality. If the input antichains are entirely formed by singletons, the check $r_i \geq b$ can be removed altogether, because in that case we know that $r_i = \ell_i \leq r_{i-1} < b$.

7.3 Brouwerian difference

The Brouwerian difference $M - S$ between antichains M (the minuend) and S (the subtrahend) can be computed by searching greedily, for each interval $[\ell \mathinner{..} r]$ in M, the first interval $[\ell' \mathinner{..} r']$ in S for which $\ell' \geq \ell$ or $r' \geq r$. We keep track of the last interval $[\ell' \mathinner{..} r']$ read from the input list S (initially, $[\ell' \mathinner{..} r'] = [-\infty \mathinner{..} -\infty]$) and update it until $\ell' \geq \ell$ or $r' \geq r$. At that point, if we did not exhaust S and $[\ell' \mathinner{..} r'] \subseteq [\ell \mathinner{..} r]$ (in which case $[\ell \mathinner{..} r]$ should not be output) we continue scanning M; otherwise, we return $[\ell \mathinner{..} r]$. The algorithm is described in pseudocode in Algorithm 6.

Algorithm 6 The algorithm for Brouwerian difference (a.k.a. "not containing").

```
0   Initially [ℓ'..r'] ← [−∞..−∞].
1   function next begin
2      while M is not empty do
3         [ℓ..r] ← next(M);
4         while S is not empty and ℓ' < ℓ and r' < r do
5            [ℓ'..r'] ← next(S)
6         end;
7         if S is empty or [ℓ'..r'] ⊈ [ℓ..r] then return [ℓ..r]
8      end;
9      return null
10  end;
```

Theorem 13 Algorithm 6 for Brouwerian difference is correct.

Proof. Note that at the start of the inner while loop (line 4) $[\ell' \mathinner{..} r']$ contains either the leftmost interval of S such that $\ell' \geq \ell$ or $r' \geq r$, or some interval preceding it. This is certainly true at the first call, and remains true after the execution of the inner while loop because of the first part of its exit condition (line 4). Finally, advancing the list of M cannot make the invariant false.

Given the invariant, at the end of the inner loop $[\ell' \mathinner{..} r']$ contains the leftmost interval of S such that $\ell' \geq \ell$ or $r' \geq r$, if such an interval exists. Note that if $[\ell' \mathinner{..} r']$ is not contained in $[\ell \mathinner{..} r]$, then no other interval of S is. Indeed, if $\ell' < \ell$ this means that $r' \geq r$, so all preceding intervals have too small left extremes, and all following intervals have too large right extremes (the same happens *a fortiori* if $\ell' \geq \ell$). Thus, the test at line 7 will emit $[\ell \mathinner{..} r]$ if and only if it belongs to the output. ∎

Theorem 14 Algorithm 6 for Brouwerian difference is 0-lazy (and thus optimally and minimally lazy).

Proof. When Algorithm 6 outputs $[\ell \mathinner{..} r]$, it has read just just $[\ell \mathinner{..} r]$ from M and the first element $[\ell' \mathinner{..} r']$ of S such that $\ell' \geq \ell$ or $r' \geq r$. If either interval has not been read by some other algorithm $\mathscr{A}$, $\mathscr{A}$ would fail if we removed altogether $[\ell \mathinner{..} r]$ from M or if we substituted $[\ell' \mathinner{..} r']$ with $[\ell \mathinner{..} r]$ and deleted all following intervals in S. ∎

7.4 Other containment operators

The three remaining containment operators have greedy, minimally lazy algorithms similar to Algorithm 6, and are shown as Algorithm 7, 8 and 9. The correctness and 0-laziness of the algorithms can be easily derived along the lines of the proofs for Brouwerian difference.

Algorithm 7 The algorithm for the "containing" operator.

Initially $[\ell' \mathinner{..} r'] \leftarrow [-\infty \mathinner{..} -\infty]$.
function next **begin**
    while A is not empty **do**
        $[\ell \mathinner{..} r] \leftarrow \mathrm{next}(A)$;
        while B is not empty **and** $\ell' < \ell$ **and** $r' < r$ **do**
            $[\ell' \mathinner{..} r'] \leftarrow \mathrm{next}(B)$
        end;
        if B is empty **then return null**;
        if $[\ell' \mathinner{..} r'] \subseteq [\ell \mathinner{..} r]$ **then return** $[\ell \mathinner{..} r]$
    end;
    return null
end;

Algorithm 8 The algorithm for the "is contained" operator.

Initially $[\ell' \mathinner{..} r'] \leftarrow [-\infty \mathinner{..} -\infty]$.
function next **begin**
    while A is not empty **do**
        $[\ell \mathinner{..} r] \leftarrow \mathrm{next}(A)$;
        while B is not empty **and** $r' < r$ **do**
            $[\ell' \mathinner{..} r'] \leftarrow \mathrm{next}(B)$
        end;
        if B is empty **then return null**;
        if $\ell' \leq \ell$ **then return** $[\ell \mathinner{..} r]$
    end;
    return null
end;

8 Previous work

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* [23], 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 [17] propose a class of

Algorithm 9 The algorithm for the "is not contained" operator.

```
Initially [ℓ′ . . r′] ← [−∞ . . −∞].
function next begin
  while A is not empty do
    [ℓ . . r] ← next(A);
    while B is not empty and r′ < r do
      [ℓ′ . . r′] ← next(B)
    end;
    if B is empty or ℓ < ℓ′ then return [ℓ . . r]
  end;
  return null
end;
```

algorithms that using tree-traversals are able to compute efficiently several operations on overlapping regions. Their motivations are efficient implementation of structured query languages that permit such regions. Albeit similar in spirit, they do not provide algorithms for any of the operators we consider, and they do not provide a formal proof of laziness.

The manipulation of antichain of intervals can be translated into manipulation of points in the plane compared by *dominance*—coordinatewise ordering. Indeed, $[\ell \mathinner{..} r] \supseteq [\ell' \mathinner{..} r']$ iff the point $(\ell, -r)$ is dominated by the point $(\ell', -r')$. Dominance problems have been studied for a long time in computational geometry: for instance, [16] presents an algorithm to compute the maximal elements w.r.t. dominance. This method can be turned into an algorithm for antichains of intervals by coupling it with a simple (right-extreme based) merge to produce an algorithm for the OR operator. One has just to notice that since dominance is symmetric in the extremes, the mapping $[\ell \mathinner{..} r] \mapsto (-r, \ell)$ turns minimal intervals (by containment) into maximal points (by dominance). The algorithm described in [16] assume a decreasing first-coordinate order of the points, which however is an *increasing* ordering by right extreme on the original intervals. After some cleanup, the algorithm turns out to be identical to our algorithm for OR (albeit the authors do not study its laziness).

The other operators have no significant geometric meaning, and to the best of our knowledge there is no algorithm in computational geometry that computes them.

Lazy evaluation is a by-now classical topic in the theory of computation, dating back to the mid-70s [11], originally introduced for expressing the semantics of call-by-need in functional languages. However, the notion of lazy optimality used in this paper is new, and we believe that it captures as precisely as possible the idea of optimality in accessing sequentially multiple lists of inputs in a lazy fashion.

9 Conclusions

We have provided efficient lazy algorithms for the computation of several operators on the lattice of interval antichains. The algorithms for lattice operations require time $O(n \log m)$ for m input antichains containing n intervals overall, whereas the remaining algorithms are linear in n. In particular, the algorithm for OR has been proved to be optimal in a comparison-based model. Moreover, the algorithms are minimally and optimally lazy (with the exception of $\mathrm{AND}_<$ when $m > 2$, in which case we prove an impossibility result) and use space linear in the number of input antichains.

We remark that, in principle, input antichains need not be finite. As long as the underlying order is locally finite and the "next" operator returns more intervals that form an antichain (ordered by their extremes), the algorithms described in this paper will return more results. In this sense, they can be thought as algorithms that transform infinite input streams into infinite output streams.

An interesting open problem is that of providing a matching lower bound for the AND operator (in the comparison-based computational model).

References

[1] Samson Abramsky and Achim Jung. Domain theory. In Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, Oxford, 1994.

[2] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[3] Garrett Birkhoff. *Lattice Theory*, volume XXV of *AMS Colloquium Publications*. American Mathematical Society, third (new) edition, 1970.

[4] Paolo Boldi and Sebastiano Vigna. MG4J at TREC 2005. In Ellen M. Voorhees and Lori P. Buckland, editors, *The Fourteenth Text REtrieval Conference (TREC 2005) Proceedings*, number SP 500-266 in Special Publications. NIST, 2005. `http://mg4j.di.unimi.it/`.

[5] Paolo Boldi and Sebastiano Vigna. Efficient lazy algorithms for minimal-interval semantics. In Fabio Crestani, Paolo Ferragina, and Mark Sanderson, editors, *Proc. SPIRE 2006*, number 4209 in Lecture Notes in Computer Science, pages 134–149. Springer–Verlag, 2006.

[6] Paolo Boldi and Sebastiano Vigna. On the lattice of antichains of finite intervals. *CoRR*, abs/1510.03675, 2016. To appear in *Order*.

[7] Charles L. A. Clarke and Gordon V. Cormack. Shortest-substring retrieval and ranking. *ACM Trans. Inf. Syst*, 18(1):44–78, 2000.

[8] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *Comput. J.*, 38(1):43–56, 1995.

[9] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[10] G. H. Gonnet. PAT 3.1: An efficient text searching system. User's manual. Technical report, Center for the New Oxford English Dictionary. University of Waterloo, Waterloo, Canada, 1987.

[11] Peter Henderson and Jr. James H. Morris. A lazy evaluator. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 95–103, New York, NY, USA, 1976. ACM Press.

[12] Monika Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams, 1998.

[13] Jani Jaakkola and Pekka Kilpeläinen. Nested text-region algebra. Technical Report C-1999-2, Department of Computer Science, University of Helsinki, 1999.

[14] Guy-Vincent Jourdan, Jean-Xavier Rampon, and Claude Jard. Computing on-line the lattice of maximal antichains of posets. *Order*, 11(3):197–210, 1994.

[15] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[16] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On finding the maxima of a set of vectors. *J. Assoc. Comput. Mach.*, 22(4):469–476, 1975.

[17] Gonzalo Navarro and Ricardo Baeza-Yates. A class of linear algorithms to process sets of segments. In *Proc. CLEI'96*, volume 2, pages 671–682, 1996.

[18] Jürg Nievergelt and Franco P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, 1982.

[19] Kunihiko Sadakane and Hiroshi Imai. Fast algorithms for k-word proximity search. *IEICE Trans. Fundamentals*, E84-A(9), September 2001.

[20] Valentin Tablan, Kalina Bontcheva, Ian Roberts, and Hamish Cunningham. Mímir: An open-source semantic search framework for interactive information seeking and discovery. *Web Semantics: Science, Services and Agents on the World Wide Web*, 30(0):52 – 68, 2015. Semantic Search.

[21] Sebastiano Vigna. Quasi-succinct indices. In Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis, editors, *Proceedings of the 6th ACM International Conference on Web Search and Data Mining, WSDM'13*, pages 83–92. ACM, 2013.

[22] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[23] Matthew Young-Lai and Frank Wm. Tompa. One-pass evaluation of region algebra expressions. *Inf. Syst.*, 28(3):159–168, 2003.

[24] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.