Efficient Evaluation of Set Expressions

by

Mehdi Mirzazadeh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

Abstract

In this thesis, we study the problem of evaluating set expressions over sorted sets in the comparison model. The problem arises in the context of evaluating search queries in text database systems; most text search engines maintain an inverted list, which consists of a set of documents that contain each possible word. Thus, answering a query is reduced to computing the union, the intersection, or a more complex set expression over sets of documents containing the words in the query.

At the first step, for a given expression on a number of sets and the sizes of the sets, we investigate the worst-case complexity of evaluating the expression in terms of the sizes of the sets. We prove lower bounds and provide algorithms with the matching running time up to a constant factor. We then refine the problem further and design an algorithm that computes such expressions according to the degree by which the input sets are interleaved rather than only considering sets sizes. We prove the optimality of our algorithm by way of presenting a matching lower bound sensitive to the interleaving measure.

The algorithms we present are different in the set of set operators they allow in input expressions. We provide algorithms that are worst-case optimal for inputs with union, intersection, and symmetric difference operators. One of the algorithms we provide also supports minus and complement operators and is conjectured to be optimal when an input is allowed to contain these operators as well. We also provide a worst-case optimal algorithm for the form of problem where the input may contain “threshold” operators, which generalize union and intersection operators: for a number t, a t-threshold operator selects elements that appear in at least in t of the operand sets. Finally, the adaptive algorithm we provide supports union and intersection operators.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Alex Lopez-Ortiz. He introduced the research area to me during my Master's. In the middle of my PhD he kindly accepted co-supervision, and then supervision of my thesis, as I decided to conduct my PhD research in this area. Nevertheless, I would like to acknowledge Professor Ming Li, who supervised my research at the beginning of my PhD. I also owe my thanks to my friends and coauthors, Arash Farzan and Ehsan Chiniforooshan, who were my companions in most of this study.

I would like to thank the members of my dissertation committee, Professors Timothy Chan, Lukasz Golab, Ming Li, and Norbert Zeh, for taking their valuable time to review my thesis and for providing me with their questions and comments.

I am extremely grateful to my family for their generous support during this journey. Thanks to my parents for their prayers, caring and sacrifices, and to my sisters for their love and support. I owe so much to my beloved wife Somayeh for her encouragement and quiet patience, and for her endless love. And, finally, thanks to my little angle, Leyli, for keeping me motivated and happy.

Dedication

To my beloved wife, my daughter, my father, and my mother.

Table of Contents

List of Tables **x**

List of Figures **xi**

List of Algorithms **xiii**

1 Introduction **1**

1.1 Background 1

1.2 The Thesis Structure 3

1.3 Key Contributions 4

2 Preliminaries **6**

2.1 The Problem 6

2.1.1 The Expression Tree 9

2.1.2 Elements 11

2.2 Restrictions on the Problem 11

2.2.1 Computation Model 11

2.2.2 Distinct Sets 12

2.3 Input/Output format 12

2.3.1 Sorted Array 12

2.3.2 Balanced Search Trees 13

2.3.3 Partially Expanded B-Tree 13

3 Lower Bounds **15**

3.1 The Expression Tree 15

3.1.1 Contribution of Nodes 15

3.1.2 Proof Labelings 18

3.2 Overview 19

3.2.1 Lower Bound Outline 19

3.2.2 Overview of the Proof 22

3.3 The First Lower Bound 24

3.3.1 The Binary Search Strategy 24

3.3.2 Touching all Crucial Elements 26

3.4 The Second Lower Bound 28

4 The Worst-Case Optimal Algorithm **37**

4.1 Special Cases 37

4.1.1 Special Case: Union of Sets 38

4.1.2 Special Case: Intersection with Union of Small Sets 41

4.2 The Algorithm 42

4.3 The Correctness of The Algorithm 45

4.4 Running Time 46

4.4.1 Processing Time in Union Nodes 46

4.4.2 Processing Time in Leaf Nodes 48

5 Algorithm Supporting More Operators **50**

5.1 Tools 50

5.1.1 Basic Operations on Partially Expanded B-trees 50

5.1.2 Processing Appearance Lists 51

5.2 The Algorithm 55

5.2.1 Overview of the Algorithm 55

5.2.2 Union and Delta Independent Nodes 55
5.2.3 Intersection and Minus Independent Nodes 56
5.3 Running Time 57

6 An Adaptive Approach **59**
6.1 Preliminaries 59
6.1.1 Background 59
6.1.2 Our Results 61
6.1.3 Difficulty Classes 62
6.1.4 Representative Perturbation 66
6.2 Lower Bound 70
6.2.1 First Lower Bound 74
6.2.2 Second Lower Bound 82
6.3 The Algorithm 83
6.3.1 The Sketch of the Algorithm 84
6.3.2 The "min-update" Structure 86
6.3.3 Generating Representative Perturbation 88
6.3.4 Analysis 94

7 The *t*-Threshold Problem **96**
7.1 Preliminaries 96
7.2 Lower Bound 98
7.2.1 Overview 98
7.2.2 The Number of Proof Labelings 99
7.2.3 Proving the Lower Bounds 103
7.3 The Algorithm 104
7.3.1 Trees of Height One 104
7.3.2 General Trees 106
7.4 Analysis 107

8 Further Thoughts and Conclusion **110**

8.1 Possible Extensions 110

8.1.1 Repeated Sets in Expressions 110

8.1.2 Unsorted Sets 112

8.1.3 Order-Independent Adaptive Algorithm 113

8.2 Conclusion 114

References **115**

List of Tables

2.1 How a tree can be converted to least complement form 8

List of Figures

2.1 A sample input and its least complement form 7
2.2 An example of how the result set is computed. 8
2.3 An example of sub-intersection tree and sub-union tree. 9
2.4 Redundant and tight input examples 10

3.1 Maximum sizes of result sets based on the leaf function size. 16
3.2 Maximum contribution 16
3.3 Result sets and contribution sets 17
3.4 Example tree of leaves with a-regions 23
3.5 Example of how second coordinates being set by adversary. 26
3.6 Example of unimportant leaves and backups 31
3.7 Example of a node being complete with a leaf 33
3.8 Example of violating Invariant 6 34

4.1 An example showing how universal set may help. 42

5.1 Independent nodes and their dependent trees. 56

6.1 An example of an easy input 60
6.2 Intuition of perturbations defining difficulty 61
6.3 Promotion-preserving and non-promotion-preserving perturbations 63
6.4 Illustration of partial order $\prec$ between inputs 64

6.5 Effect of a small changes in values on the $\preceq$-smallest perturbation 66
6.6 An example of min and min values 90
6.7 An example of a situation where we roll-back rounds 91

List of Algorithms

1 How to determine the second coordinates of members. 25

2 How to determine the third coordinates of members. 35

3 SimpleUnion($\mathcal{X}$) 38

4 Compute(v) 43

5 Compute(v, U) 44

6 Result evaluation for exclusive contributors 53

7 How to decide if a value is in the result 54

8 The binary-search strategy 78

9 Forcing the algorithm to touch all members in promoted groups 79

10 The Update algorithm 84

11 The main function that computes the answer. 85

12 Determining sizes of Λ-subsets. 100

13 The algorithm for threshold trees of height 1. 105

14 The Update algorithm for threshold trees 106

Chapter 1

Introduction

1.1 Background

In this thesis we study the problem of computing the result of a given set expression. The problem arises in the context of evaluating search queries in text database systems; most text search engines maintain an inverted list, which consists of a set $S(w)$, for each word w, of documents that contain w [15, 35, 43]. Thus, answering a query, such as "Database OR Search AND Engine", requires evaluation of the expression $S(\mathsf{Database}) \cup (S(\mathsf{Search}) \cap S(\mathsf{Engine}))$. Note that the queries and their corresponding expressions can become rather complicated if the queries are automatically generated [32].

Another application is column oriented stores for databases. In these databases, for each prediction in the query, one columns is scanned to collect list of rows satisfying the prediction. Then, the intersection of lists collected for different predictions is computed to obtain the list of matching rows. C-store [40] is an example of such database systems.

Initially, some search engines used inverted lists sorted by frequency of the word in the document [2]. A drawback was that the score of a page could be artificially boosted through the repetition of a word within the document arbitrarily many times. So search engines like Google started using a global ranking of pages based on links [15] and renumbering documents according to this rank. For that reason sets are usually assumed to be sorted beforehand in the preprocessing stage in the study of this problem.

We will also assume that even if two sets appearing in the expression are equal, the algorithm can only infer such equality by examining sets members, and cannot conclude that by just looking at the shape of the expression. For example, an input like $A \cap (B \cup C)$, where A and C happen to be equal, is not given to the algorithm as $A \cap (B \cup A)$, otherwise the algorithm would know the result is A before looking at actual members of any set. If we would not have this restriction, the problem would become intractable, as one could easily create instances of the problem that were equivalent to arbitrary instances of some NP-hard problems (see Section 8.1.1). The worst-case complexity in terms of the collective size of the entire input is straightforward. So we measure the running time of algorithms depending on the individual sizes of the input sets; we are interested in a worst-case optimal algorithm.

Different variations of the problem have been studied before. The simplest case is finding the intersection or union of two sets, which is equivalent to the problem of merging two ordered sets of sizes m and n. This version of the problem is also called "multiple search problem" as it can be seen as searching for occurrences of members of one set (the "queries" set) in the other set [1]. This problem for values of $m = 2$ and n, and later general values of m and n was studied by Hwang and Lin [30, 31]. They presented an algorithm that matched the information theoretic lower bound of $\left\lceil \log \binom{m+n}{n} \right\rceil$. Note that this bound does not allow for the exhaustive listing of the entire output. They chose sorted arrays as the format of the input and a list of cross pointers between arrays as the output format. Later Brown and Tarjan [16, 17] and Pugh [38] showed how data structures such as AVL-trees, B-trees, or skip-lists can be used as the format of the input and the output. The problem of computing the union of two sets was also studied for parallel algorithms [1, 42]. A variant of the problem where one of the input sets (the set of queries) is not sorted was also considered in a parallel model of computation [12].

Fernandez de la Vega, Frieze, and Santha analyzed the running time of Hwang and Ling algorithm in average case when the ratio of sizes of the two sets is any constant [27]. Later, Baeza-Yates proposed another worst-case optimal algorithm for intersection of two sets, which works better in average case [2]. Baeza-Yates and Salinger did an experimental analysis of this algorithm as well [4].

Demaine, López-Ortiz, and Munro [23] studied a more general case where the expression could involve more than two sets. The expressions they considered were limited to union or intersection of a number of sets, or one set minus the intersection of some other sets. Their algorithm is adaptive; they do not focus on the worst-case complexity of the problem. They define the *difficulty* of every possible input I as an integer $D(I)$, which measures how complicated a proof for the input I is; they focus on minimizing the maximum value of $\frac{T(I)}{D(I)}$ among all inputs I of size n, where $T(I)$ is the running time of the algorithm on I.

Long and Suel claimed that the adaptive approach does not work well with disk-resident structures [34] but, as Li et al. showed, it is beneficial for in-memory and peer-to-peer environments [33]. Demaine et al. pursued their study by comparing their adaptive algorithm with other methods proposed before [24]. This study was followed by Barbay et al. who included more variants of the adaptive algorithm in the experiments [10] In a more recent study, Barbay et al [9] considered several published deterministic and randomized algorithms, and investigated how they compare and how they can be improved.

Barbay and Kenyon studied computing the union or intersection of an arbitrary number of sets in a randomized fashion. They then derived a deterministic version of their randomized redundancy-based version of the original problem [8]. They also considered the t-threshold version of the problem which asks to report values appearing in at least t input sets [7].

In this thesis we focus on the comparison model, that is, we study algorithms that only have access to the relative values of members of the input sets. This is also the case for all references we mentioned above. Examples of works violating this restriction are the ones that use hash functions to perform operations on sets [13, 41]. Bille et al. [13] showed that we can improve worst-case running time by almost a factor of $\frac{w}{\log^2 w}$ for the case of intersection of a number of sets, where w is the size of the machine word. They also studies the problem for the case of general union-intersection trees, but their results for the general case are not as good. Tsirogiannis et al. [41] explored both sorted and unsorted (with hash techniques) multi-way intersection for modern computers (cache-aware algorithms). Barbay, Golynski, and Munro designed a succinct representation of inverted lists [5] and used that to improved previous adaptive algorithm of Barbay and Kenyon [8] slightly. This was done in a non-comparison-based model.

1.2 The Thesis Structure

In Chapter 3, we use a novel technique to develop a tight lower bound on the worst-case running time of any comparison-based algorithm for the problem when union, intersection, and delta (symmetric difference[1]) operators are used in expressions. The running time in the lower bound is based on input set sizes and the input expression. Our conjecture is that the lower bound holds even when minus and complement operators are also used in expressions.

In Chapter 4 we develop a worst-case optimal algorithm for expressions with union and intersection operators when the input sets are given in sorted array format. Then, in

[1]The *symmetric difference* of two sets A and B, denoted by $A\Delta B$, is defined as the sets of the values appearing in exactly one of A and B.

Chapter 5 we present an algorithm for a more general form of expressions when all binary and unary set operators (union, intersection, delta, minus, and complement) are allowed. The representation of input and output sets of this algorithm is a generalization of sorted arrays and binary search trees.

In Chapter 6, we design an algorithm that computes expressions according to the degree by which the input sets are interleaved. We prove the optimality of our algorithm by presenting a matching lower bound sensitive to the interleaving measure. More specifically, we group input instances into finely-sized classes possessing the same amount of interleaving. The upper and lower bounds are asymptotically tight in each such class up to an additive term that depends only on the number of sets and is independent of the set sizes. In contrast, our worst-case optimal algorithms pessimistically assume that the input sets are adversarially interleaved.

Finally, in Chapter 7, we generalize the expressions to include t-threshold operators and develop a tight lower bound and a worst-case optimal algorithm for expressions that consist of an arbitrary number of threshold operators.

1.3 Key Contributions

The most innovative technical contribution in the thesis is the lower bound we present in Chapter 3. When the algorithm starts running the algorithm, it repeatedly probes different parts of the input and get some information in response. Consider an external observer who has no access to the value of sets members in the input, but observes the information revealed to the algorithm. Roughly speaking, the idea is to show that the observer can infer the way the members of the result set are distributed among input sets using this information, and this way we obtain the lower bound using combinatorial approaches. But the challenge is that this amount of information is not sufficient for the observer to infer what it needs. So we change the model such that the external observer gets an extra bit of information on the top of every bit revealed to the algorithm. This distinction between the information revealed to the algorithm, and the information revealed to the external observer is the key technique we used to develop the lower bound.

The algorithms we propose in Chapters 4 and 5 have similar structures. The target running times in both depend only on set expression and sets sizes. As such, at the first step they both analyzes the input expression and sets sizes, and independent of actual values in sets, they decide on what order different parts of the input expression should to be evaluated. So, in this approach, the expression is broken into a number of basic sub-expressions, which are evaluated in a static order. The algorithm in Chapters 4 is the simpler one in the sense that these basic sub-expressions are pretty simple, like computing the union or the intersection of a number of sets. In Chapter 5 where more operators are allowed, however, each of the resulting basic sub-expressions can be much more complex and may contain any number of operators. We design data structures and techniques to evaluate these complex non-breakable sub-expressions.

In Chapter 6, we develop a novel frame work for adaptive algorithms. In previous attempts for developing adaptive algorithms for various problems (as examples, please refer to attempts for sorting problem [25] or sets expressions [36]), the difficulty of the problem was defined as a single number, which then appears as an extra parameter in lower bound analysis. In the approach we develop here, the inputs are divided into classes based on how values are distributed in input sets, and then the goal for the algorithm is to be worst-case optimal within each class. For the particular problem we study in this thesis, we have basically two challenges in this framework. The first one is how to define the input classes such that the easy and hard instances are distinguished, and the easiness of the inputs are reflected in tights lower bounds we prove for classes. The second challenge is that, unlike algorithm in previous chapters, the target running time depends on actual sets members and is not clear from the beginning, so the previous framework where we break the input expression to a number of basic sub-expressions and then compute the sub-expression one by one do not work. Here we need to process all inputs sets in parallel, and design efficient data structures to keep track of partial results as we proceed in inputs sets.

In Chapter 7, for the first time we support a non-constant number of different operators by allowing t-threshold operators for any value of t. This adds to the complexity of problem, as it is no longer obvious even how to compute the maximum possible size of the final result, given the expression and sets sizes. We show how we can compute the maximum possible result size and other relevant metrics, and then successfully adapt the techniques we developed in Chapter 3 to prove a lower bound. Then we use the ideas and data structures we developed in Chapter 6 to design our algorithm for the problem.

Many of the results discussed in this thesis are based on collaborations with Alejandro López-Ortiz, Arash Farzan, and Ehsan Chiniforooshan [20, 21, 22].

Chapter 2

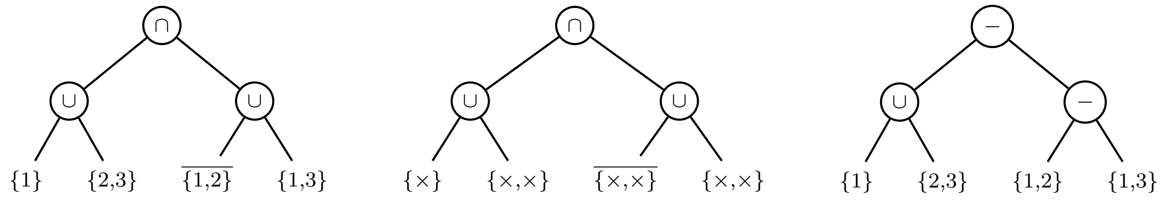
Preliminaries

2.1 The Problem

We study the problem of evaluating a set expression when the inputs are ordered sets and the output is required to be an ordered set as well.

We use set expression trees to represent an input to the problem: every internal node v of the tree corresponds to an operator in the input expression, which we denote by $\mathsf{opr}(v)$, and each leaf corresponds to an input set in the expression (an atom). When we talk about the *expression tree* (rather than the input), we are talking about the expression tree without considering actual sets associated with leaves. The *signature* of an input is the expression tree where leaves are assigned only the specific set size as opposed to actual elements forming the set. See input I and its signature in Figure 2.1 for an example. Note that many different inputs can have exactly the same signature. Given an input or signature, we use $\mathsf{size}(v)$ to denote the size of the set corresponding to a leaf v.

The set operators that internal nodes may be assigned include union, intersection, symmetric difference, difference (minus), depending on the variant of the problem we are considering. If v is a minus node and the corresponding expression is $A-B$, the root of the subtree corresponding to A is the left child of v and the subtree corresponding to B is the right child of v. Also, each node is either a *complement* or a *normal* node. For example, in the input I in Figure 2.1, aside from the third leaf from left, all other nodes are normal nodes. When we are considering a variant of the problem where the input set expression may not include complement operator, all nodes are normal nodes. When all nodes are normal nodes and the set of internal nodes only consist of union and intersection nodes, the expression tree (or the signature or input) is called a *union-intersection* one.



The input I The signature of the input I The input J

Figure 2.1: The input I representing the expression $(\{1\} \cup \{2,3\}) \cap (\overline{\{1,2\}} \cup \{1,3\})$ and its signature. The input J is the least complement form of input I.

Given an input, it is easy to see that one can change the tree by propagating the complements up to the root such that we end with an *equivalent* tree: i.e. the result of the input remains unaffected (e.g. trees corresponding to expressions $\overline{A} \cap B$, $\overline{A} - \overline{B}$, and $\overline{A \cup \overline{B}}$ are equivalent). We define two canonical forms for expression trees. A tree is in *minus-free form* if none of it internal nodes is a complement node nor is assigned a minus operator. Also, a tree is in *least complement form* if aside from its root, every node is a normal node. For example, the input I Figure 2.1 is in minus-free form, and input J in the figure is its equivalent in least complement form. If there is no complement node in the tree, it is *complement-free* (like input J in the figure).

Lemma 1 *Every tree has an equivalent in minus-free form.*

Proof First we can see that any tree can be easily transformed to an equivalent tree with no minus node: we just need to replace each subtree of form $A - B$ with $A \cap \overline{B}$. Next, given a tree with no minus node, we can start from top nodes, and propagate complement operators to lower nodes: $\overline{A \cap B}$, $\overline{A \cup B}$, and $\overline{A \Delta B}$ are replaced with $\overline{A} \cup \overline{B}$, $\overline{A} \cap \overline{B}$, and $\overline{A} \Delta B$, respectively. By repeating this, we can obtain a tree with no complement in internal nodes. □

Lemma 2 *Every tree has an equivalent in least complement form.*

Proof Without loss of generality we assume each internal node of the tree has exactly two children. The lemma can be proved using induction on the height of the tree. The correctness of the lemma for just a leaf is clear. Suppose the lemma is correct for trees with height less than that of tree T. Each of the two subtrees of T can be written in complement-free form or as the complement of a complement-free expression. So T represents one of the expressions in Table 2.1, where A and B are complement-free. Note that the obvious

Expression	Least-complement equivalent
$A \cup \overline{B}$	$\overline{B - A}$
$\overline{A} \cup \overline{B}$	$\overline{A \cap B}$
$A \cap \overline{B}$	$A - B$
$\overline{A} \cap \overline{B}$	$\overline{A \cup B}$
$A \Delta \overline{B}$	$\overline{A \Delta B}$
$\overline{A} \Delta \overline{B}$	$A \Delta B$
$A - \overline{B}$	$A \cap B$
$\overline{A} - B$	$\overline{A \cup B}$
$\overline{A} - \overline{B}$	$B - A$

Table 2.1: How a tree can be converted to least complement form

cases of $A \cap B$, $A \cup B$, $A - B$, and $A \Delta B$ where T is already complement-free are not listed. As the table shows, in each case T can be converted to a least complement form. □

By Lemma 2, for every tree T we can obtain a complement-free tree which is either equivalent to T or the complement of T.

For any input I, we can propagate the leaf sets to the internal nodes of I in a natural bottom-up way. We define *the result set of* a node v in I, denoted by $\mathrm{res}_I(v)$, or simply $\mathrm{res}(v)$ when I is fixed, as follows: For a leaf v, depending on v being normal or complement, $\mathrm{res}(v)$ is the set corresponding to v or its complement. For a normal internal node v, $\mathrm{res}(v)$ is the result of the application of the operator $\mathsf{opr}(v)$ to the results of the children of v. Similarly, when v is a complement internal node, $\mathrm{res}(v)$ is defined as the complement of the result of applying $\mathsf{opr}(v)$ to the results of the children of v. By the *result* of an input I we mean the result set of the root in I. Figure 2.2 shows an example.

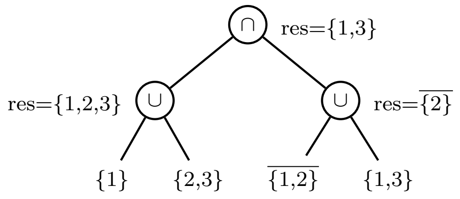
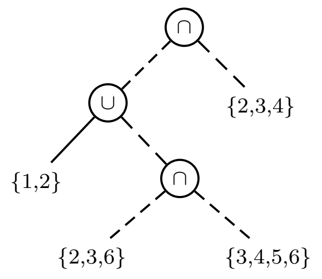
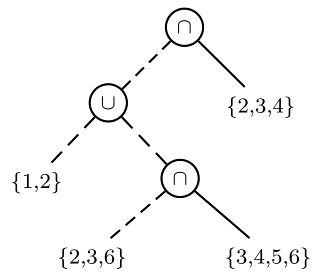


Figure 2.2: An example of how the result set is computed.



A sub-intersection tree (marked by dashes) showing 3 is in the result set as it appears in all leaves of the sub-intersection tree.



A sub-union tree (marked by dashes) showing 4 is **not** in the result set as it appears in no leaf of the sub-union tree.

Figure 2.3: An example of sub-intersection tree and sub-union tree.

2.1.1 The Expression Tree

As explained, the trees are used to represent the input set expressions. A set expression consisting of union and intersection operators can be rewritten as the union of a number of intersection terms or the intersection of a number of union terms (where the same set may appear in more than one term). The next two definitions translate this observation to the tree space.

Definition 1 *Given a complement-free expression tree T, a* sub-union tree *(a* sub-intersection tree*) is a subtree U of T with the following properties:*

1. *It contains the root of T.*
2. *If it contains a union or delta (an intersection) node, it contains all of its children.*
3. *If it contains an intersection (a union or delta) node, it contains at least one of its children.*
4. *If it contains a minus node, it contains its left child.*

U is minimal *if it contains at most one child of every intersection (union or delta) node.*

Figure 2.3 shows examples of sub-intersection and sub-union trees.

When only union and intersection operators are allowed, a tree is equivalent to the union of all of its sub-union trees, or the intersection of all of its sub-union trees. We can

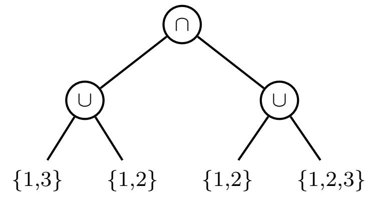


Figure 2.4: The input is redundant on 1, but is tight on 2 and 3.

think of sub-union trees as providing evidence that any given value is not in the result set, while a sub-intersection tree may provide evidence that it is in the result set. Figure 2.3 illustrates an example.

Observation 3 *Given a union-intersection input I and a value a, the following statements are equivalent:*

1. *a is in the result set of the root in I.*
2. *There is a sub-intersection tree of I in which every leaf has an element of value a.*
3. *There is no sub-union tree of I in which no leaf has an element of value a.*

A set of sub-intersection trees $U_1, \ldots, U_k$ is a *partitioning* of a sub-intersection tree T if each leaf of T appears in exactly one of the U_i's. It is easy to see that every sub-intersection tree can be partitioned into a number of "non-partitionable" sub-intersection trees (we just need to repeatedly partition partition-able sub-intersection trees into smaller ones until no further partitioning is possible).

Given a union-intersection input I and a value a in the result set of I, as mentioned, there is a sub-intersection tree of I where the set corresponding to every leaf of it contains a. If I has two such sub-intersection trees that have no leaf in common, we say I is *redundant at a*, otherwise I is *tight at a*. An input is *tight* if it is tight at every value in its result set. Figure 2.4 shows an example.

There are some leaves in tree with the property that given any member of the set associated with them, if an algorithm wants to prove that member is in the result of the expression, it has to make at least one comparison on that member. As an example, a leaf that is the child of an intersection node has this property: the algorithm has to make sure the member also appears in other siblings of that leaf. On the opposite side, in the expression tree $A - B$, A does not have this property, because if by chance the algorithm finds all members of A are less than all members of B (say by comparing the biggest member of A to the smallest member of B), then the algorithm knows all members of A are in the final result without looking at all of them. Roughly speaking, leaves in the tree that do not have this property are called "shallow leaves".

Definition 2 *A leaf v of an expression tree T in least-complement form is a* shallow leaf *if v has no intersection ancestor and has no minus ancestor u such that v is in the right subtree of u.*

Throughout the thesis, for an expression tree T, we denote the set of leaves of T by $\mathsf{leaves}(T)$. We use $T[v]$ to denote the subtree rooted at a node v.

2.1.2 Elements

For a signature S and a leaf l, we introduce $\mathsf{size}(l)$ "symbolic elements" denoted by $\mathrm{e}_1^l, \ldots, \mathrm{e}_{\mathsf{size}(l)}^l$ where e_i^l represents the ith biggest member of l. For simplicity of presentation we add two sentinel elements e_0^l and $\mathrm{e}_{\mathsf{size}(l)+1}^l$. Then, for an input I with signature S, by the *I-value* (or just *value*) of an element e_i^l, where $1 \leq i \leq \mathsf{size}(l)$, we mean the ith biggest value in the set corresponding to l. We also define the I-value of e_0^l and $\mathrm{e}_{\mathsf{size}(l)+1}^l$ as $-\infty$ and ∞, respectively. For $e = \mathrm{e}_i^l$, for some l and i, we use the expression $\mathsf{val}_I(e_i^l)$, or just $\mathsf{val}_I(e)$, to denote the I-value of the element e. Furthermore, we may write $\mathsf{val}(e)$ when the input we are talking about is clear from the context. For every i, we use $\mathsf{next}(e_i^l)$ to denote e_{i+1}^l. For a leaf l, we use $\mathsf{elements}(l)$ to denote the sequence $e_1^l, \ldots, e_{\mathsf{size}(l)}^l$. For an input I, an element e_1 is *I-smaller* (*I-bigger*, respectively) than an element e_2 if the I-value of e_1 is smaller (bigger, respectively) than that of e_2.

2.2 Restrictions on the Problem

2.2.1 Computation Model

In this thesis, we focus on *comparison-based algorithms* which are those that, for any input I, use only comparisons on the input sets to compute the result. In this model, the algorithm has oracle access to the value of the elements, which means that the algorithm reads the signature of the input and can then submit queries of the form (x, y) to the oracle, where x and y are two elements. Then, the oracle answers the algorithm with the comparative values of x and y, that is, the algorithm is told whether the value of x is less than, equal to, or greater than that of y. In such situations we say x and y are *touched* by the algorithm. In other words, an element is touched if it is compared to another element by the algorithm.

Note that in this computation model, the algorithm does not notice any differences between two inputs where elements have the same relative values.

Definition 3 *Two inputs I and J of the same signature are* equivalent *if for every two elements e_1 and e_2, e_1 is I-smaller than e_2 if and only if e_1 is J-smaller than e_2.*

Observation 4 *For any comparison based algorithm $\mathcal{A}$ and equivalent inputs I and J, $\mathcal{A}$ will make the same sequence of comparisons when run on I and J.*

In the previous works for this problem in the comparison model, it is required that each input set is given in sorted order. Then data structures, such as sorted arrays, are used to represent the input sets in which the order of members within each set is implicitly defined and the program does not need to do any further comparisons to extract them. This assumption makes complete sense as pre-sorting input sets is not expensive when multiple queries are going to be executed on the same sets.

2.2.2 Distinct Sets

In this thesis we consider only inputs where each set appears at most once in the corresponding set expression. Note that this does not mean that, for example, $\{1, 2\} \cup \{1, 2\}$ is not a valid input to the problem. We still can define the set expression $A \cup B$ and have $A = \{1, 2\}$ and $B = \{1, 2\}$. However, we do not consider expressions like $A \cup A$. In other words, there is no way for the algorithm to know two sets are equal before comparing their members.

2.3 Input/Output format

2.3.1 Sorted Array

A sorted array is the simplest data structure we can think of as an input and output format for this problem, considering the conditions explained in previous sections. Then, the sets associated with leaves will be given to the algorithm in the form of sorted arrays. However, this degrades the performance of the optimal algorithm if we require it to return the output in sorted array format. That is because in some situations the minimum effort required to decide which elements should appear in the output and in what order is asymptotically less than the number of such elements. As an example consider the case of $A \cup B$ where $|A| = 1$ and $|B| = n$. Then, an algorithm can use binary search to figure out the order in which elements appear in the output in $O(\log n)$ time. To resolve this, we will expect the algorithm to specify subintervals of input sets that appear in the result, rather than to write all elements of the result. This allows the algorithm to generate the output in sub-linear time if possible. More precisely, we define the output format below. We use $S[i]$ to denote the ith element of a sequence S.

Definition 4 *Consider an input I and a set S. A* cross reference representation *of S is a sequence of items $(v_1, b_1, b'_1), \ldots, (v_n, b_n, b'_n)$ where, for every $1 \leq i \leq n$, v_i is a leaf in I and $1 \leq b_i \leq b'_i \leq \mathsf{size}(v_i)$, and for every $1 \leq j < n$, the b'_jth element of v_j is I-smaller than b_{j+1}th element of v_{j+1}, and $S = \cup_{i=1}^{n} \cup_{j=b_i}^{b'_i} \{\mathsf{val}_I(e_j^{v_i})\}$.*

2.3.2 Balanced Search Trees

A drawback to selecting sorted arrays and cross reference arrays for input/output formats is that with this choice, the input will have a different format from the output and thus the output cannot be used directly in subsequent queries. Balanced search trees are considered as an alternative to sorted arrays as they implicitly represent the total order between elements as efficiently as in sorted arrays and they inherently support representation of 'subranges' (by selecting a number of subtrees) as in the cross-reference format.

2.3.3 Partially Expanded B-Tree

In this work, we use a slightly modified version of B-trees, which we define as follows.

Definition 5 *A* partially expanded B-tree *T is a B-tree in which for some internal nodes u, the subtree rooted at u is replaced with the sorted list of elements that are in that subtree. The* size *of T is the total number of elements in T.*

B-trees and sorted arrays are special cases of partially expanded B-trees. This choice for our algorithms' input/output format enables us to support the cases where the input sets are either sorted arrays, or B-trees. As we will discuss in Section 5.1.1, partially expanded B-trees are as efficient as regular B-trees in the basic operations we need for our algorithms.

Chapter 3

Lower Bounds

In this chapter we discuss the minimum number of comparisons an algorithm must perform in the worst case to solve the problem for an input. The worst-case analysis is done over all inputs with a given signature. The techniques we develop in this chapter will be used in analyzing adaptive running time in Chapter 6 as well.

3.1 The Expression Tree

In this section we introduce a number of functions and concepts needed to analyze different parts of a given expression tree. Given an expression tree, a *leaf function* is a function assigning integers to leaves of the tree. Recall that a signature is in fact an expression tree together with a leaf function that specifies set sizes (which we denote by size).

3.1.1 Contribution of Nodes

Given an expression tree T and a leaf function f, we use $\mathsf{cap}_f(v)$ to denote the maximum size of the result set of v in an input with expression tree T where the size of the set associated with each leaf l is $f(l)$ (in other words, the inputs for which f is the same as the function size).

Definition 6 *Given an expression tree T in least complement form and a leaf function f, for a node v of T, $\mathsf{cap}_f(v)$ is defined as follows:*

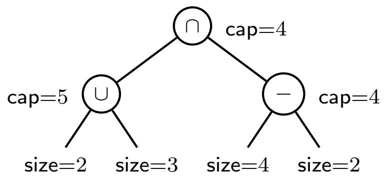


Figure 3.1: Maximum sizes of result sets based on the leaf function size.

- *For a leaf l,* $\mathsf{cap}_f(v) = f(v)$.
- *For a union or a delta node v with children $u_1, \ldots, u_k$,* $\mathsf{cap}_f(v) = \sum_{i=1}^{k} \mathsf{cap}_f(u_i)$.
- *For an intersection node v with children $u_1, \ldots, u_k$,* $\mathsf{cap}_f(v) = \min_{i=1}^{k} \mathsf{cap}_f(u_i)$.
- *For a minus node v with left child u_1 and right child u_2,* $\mathsf{cap}_f(v) = \mathsf{cap}_f(u_1)$.

When talking about a specific signature S, we may use $\mathsf{cap}(v)$ to denote $\mathsf{cap}_{\mathsf{size}}(v)$, for nodes v of the expression tree. In other words, for a given signature S and node v, $\mathsf{cap}(v)$ is the maximum size of the result set of v among all inputs with signature S. Figure 3.1 shows an example.

For a union-intersection signature S and a given node v, the result set of a node v can be as big as $\mathsf{cap}(v)$, but all these members do not necessarily contribute to the at most $\mathsf{cap}(\text{root})$ members of the result set of the whole tree. For better intuition, consider the input in Figure 3.2. Here, although the union node has a maximum result set size of 12, at

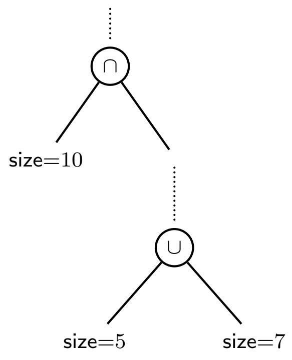


Figure 3.2: Maximum contribution

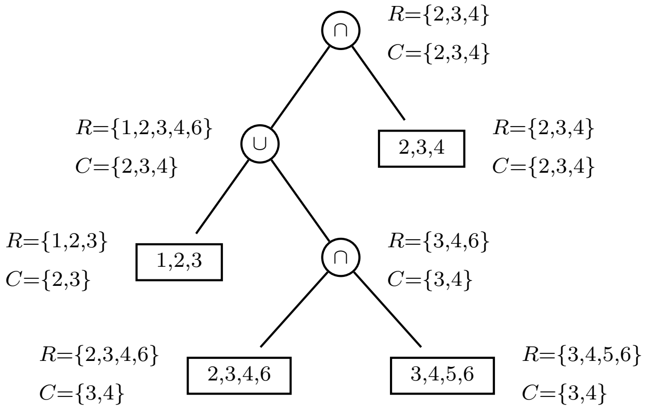


Figure 3.3: Result sets (denoted by R) and contribution sets (denoted by C) of nodes.

most 10 members out of these 12 members can appear in the result of the whole tree; the rest will be filtered out by the intersection node in the upper part of the tree. Thus, the contribution of this union node to the result of the whole tree will not be more than 10.

Definition 7 *Given a union-intersection input, the* contribution set *of a node is the intersection of the* result set*s of all of its ancestors including itself.*

This way, we track the provenance of a value in the output set by following its promotion through the expression tree. Figure 3.3 is an example. In an input an element e of a leaf l *is promoted* if the value of e is in the contribution set of l. Moreover, two elements *are promoted together* if they have the same value and they are both promoted. Next we define the maximum possible size of the contribution set of a node as the "contribution limit" of that node.

Definition 8 *Given an expression tree T and a leaf function f, the* contribution limit *of a node v, denoted by $\mathsf{share}_f(v)$, is defined as $\mathsf{share}_f(v) = \min_u \mathsf{cap}_f(u)$, where the minimum is taken over all ancestors u of v, including v itself.*

Again, when talking about a signature S, for a node v, we may use $\mathsf{share}(v)$ instead of $\mathsf{share}_{\mathsf{size}}(v)$.

Observation 5 *In a union-intersection input I, the maximum size of the contribution set of a node is the contribution limit of that node.*

Note that Observation 5 would not hold if we extended it to the trees containing minus nodes. Intuitively, in a subtree $A-B$, the contribution limit of B is the maximum number of members of result set of A that could concurrently appear in the result-set of B and so be removed from result set of $A-B$. In other words, the contribution limit of B measures how big the set of values that B causes to be removed from the contribution set of its parent can be.

We observe that the values of cap_f and share_f for all nodes of an expression tree T can be evaluated in time $O(n)$, where n is the number of nodes in the tree.

3.1.2 Proof Labelings

Consider a scenario in which a set O_T is the result set of the root of the tree. The elements of O_T stem from the leaves of the tree and move up the tree level by level according to the operation nodes. We define the notion of *proof labeling*, which captures the trace-back of nodes that had an impact on the presence of an element in the final result. A proof labeling is formally defined as follows:

Definition 9 *Consider a signature S with a complement-free expression tree. A function Λ is a* proof labeling *for S if for any internal node v with children $u_1, u_2, \ldots, u_k$ (from left to right):*

- *if v is a union or delta node, then $\bigcup_{i=1}^{k} \Lambda(u_i) = \Lambda(v)$,*
- *if v is an intersection node, then $\Lambda(u_i) = \Lambda(v)$, for $1 \leq i \leq k$, and*
- *if v is a difference node, in which case $i = 2$, then $\Lambda(u_1) = \Lambda(v)$ and $\Lambda(u_2) \subseteq \Lambda(v)$.*

A proof labeling is maximal *if for every node v, $|\Lambda(v)| = \mathsf{share}(v)$.*

It is easy to see that, if I is a union-intersection input, then the function assigning to each v the set of values in the contribution set of v is a proof labeling. We call such function the *proof labeling corresponding to the input I*. When we have minus nodes, the intuition is a little different: for the right child of a minus node, a proof labeling captures trace-back of values that could be in the result-set of the right child of the minus node, and if so, prevent the minus node from having them in its contribution set.

Observation 6 *For any proof labeling Λ for a union-intersection signature S and value o in $\Lambda(root)$, the set of nodes v with $o \in \Lambda(v)$ is a sub-intersection tree.*

Recall the definition of an input being tight at a value from Section 2.1.1 and note that if a proof labeling Λ corresponds to inputs I and J and I is tight, J is also tight. As such, we define a proof labeling to be *tight* if it corresponds to tight inputs.

Lemma 7 *Given a union-intersection signature S, any maximal proof labeling for S is tight.*

Proof Assume to the contrary that P is the maximal proof labeling corresponding to an input I that is not tight on a value a. So, there are two sub-intersection trees T_1 and T_2 of the tree, whose set of leaves are disjoint, and a appears in the result sets of all nodes of T_1 and T_2 in I. Consider a value b bigger than a, but smaller than any other value appearing in the input. We replace all occurrences of a in leaves of T_2 with b. Then, b will be in result sets (and so in contribution sets) of all nodes of T_2, while a continues to be in result sets of all noes of T_1. So, we could find an input with the same signature but with a bigger contribution set for the root. So the size of the contribution set of the root in I was less than $\mathsf{share}(\text{root})$, which contradicts P being maximal. $\square$

The reader can observe that some proof labelings can be obtained from some other ones by just applying a one-to-one monotonic function to set members; so among such a group of proof labelings we choose one as the "canonical" one: we define a proof labeling P to be *canonical* if $P(\text{root}) = \{1, \ldots, k\}$, for some integer k. Also P is *non-empty* if $P(v) \neq \emptyset$ for every node v.

As comparison-based algorithms only work based on relative values of elements, rather than their actual values, we can define the output as a sequence of elements rather than the sequence of values in the result of the root. A *solution* is a sequence of elements, all of which are promoted and the sequence of values of these elements is the same as the sequence of the values in the result of the root in sorted order.

3.2 Overview

3.2.1 Lower Bound Outline

In this section we provide some intuition for the lower bound. A proof labeling can be seen as a proof showing that certain members are in the result set of the input. In fact, a maximal proof labeling at the same time can be a proof of the fact that no member other than the ones in the proof labeling can be in the result set of the input (as the values in the sets defined by the proof labeling leave no space for promoting any other elements).

Informally speaking, we show that for any algorithm and any maximal proof labeling P, there is an input I for which the algorithm follows the proof labeling P to figure out which members are in the result set: for any leaf l and any value v in $P(v)$, the algorithm is shown to have searched for v in l, verifying if l has an element of value v.

The lower bound we propose for the problem has the following form:

$$\underbrace{\sum_{v \in \mathsf{deep}} \log \binom{\mathsf{size}(v)}{\mathsf{share}(v)} + \mathsf{share}(v)}_{\text{the first part}} + \underbrace{\sum_{v \neq \text{root}} \log \binom{\mathsf{share}(p(v))}{\mathsf{share}(v)}}_{\text{the second part}} . \tag{3.1}$$

Here $p(v)$ is the parent of v and deep is the set of non-shallow leaves of the tree. As we prove in this section, the second part in Equation 3.1 is the logarithm of the number of canonical proof labelings, and the first part is the logarithm of the number of ways to select elements represented in the proof labeling from actual members in leaves.

The correctness of this lower bound for general expression trees remains an open conjecture, but we prove it for complement-free trees with union, intersection, and delta operators, and a possible minus at the root. In the next chapters, we provide algorithms solving the problem in a time asymptotically matching this lower bound for all trees.

Conjecture 1 *Given any signature S and any algorithm $\mathcal{A}$, there is an input I with signature S such that $\mathcal{A}$ needs the result of Equation 3.1 comparisons to evaluate the expression given by I.*

To count the number of canonical proof labelings, we define the notation of $\binom{s}{s_1,\ldots,s_n}$, where $s \leq \sum_{i=1}^{n} s_i$, as the number of ways to select sets $X_1, \ldots, X_n$ of sizes $s_1, \ldots, s_n$, respectively, such that $\cup_{i=1}^{n} X_i = \{1, 2, \ldots, s\}$. Note that this definition matches the traditional notation $\binom{s}{s_1,\ldots,s_n}$ when $s = \sum_{i=1}^{n} s_i$. The next lemma counts the number of proof labelings.

Lemma 8 *Given a signature S, the logarithm of the number of maximal canonical poof labelings is*

$$\begin{array}{lll} \displaystyle\sum_{\mathsf{opr}(v)=-} & \log \binom{\mathsf{share}(v)}{\mathsf{share}(u)} & u \text{ is the right child } v \\ + \displaystyle\sum_{\mathsf{opr}(v)\in\{\cup,\Delta\}} & \log \binom{\mathsf{share}(v)}{\mathsf{share}(u_1),\ldots,\mathsf{share}(u_k)} & u_1, \ldots, u_k \text{ are children of } v. \end{array}$$

Proof The number of canonical proof labelings for S is the number of ways to define $\Lambda(v)$, for nodes v, by distributing values in $\Lambda(\text{root}) = \{1 \dots n\}$ in the tree, all the way down to the leaves, such that for each node v conditions of Definition 9 hold and $|\Lambda(v)| = \mathsf{share}(v)$, for every node v.

Let us start from root and count the number of choices we got at each node. At an intersection node v, there is only one choice: every element appearing in $\Lambda(v)$ has to appear in $\Lambda(u)$ for all children u of v. At a union or delta node, the distribution should be such that in addition to satisfying $|\Lambda(u)| = \mathsf{share}(u)$, for children u of v, $\Lambda(v)$ is the union of $\Lambda(u)$, for children u of v. So the number of choices is $\binom{\mathsf{share}(v)}{\mathsf{share}(u_1),\dots,\mathsf{share}(u_k)}$ where the u_i's are the children of v. Finally when v is a minus node, there is only one choice for the left child of v (it has to inherit all elements in $\Lambda(v)$), but the right child u of v can be assigned any subset of size $\mathsf{share}(u)$ of $\Lambda(v)$. Hence, in total, the logarithm of number of choices is what mentioned in the lemma. $\square$

The next lemma helps to rewrite the newly defined notation $\binom{s}{s_1,\dots,s_n}$, which appears also in our lower bound, using the traditional notation $\binom{a}{b}$.

Lemma 9 *For values s and $s_1, \dots, s_n$ where $s \le \sum_{i=1}^n s_i$, we have*

$$\log \binom{s}{s_1, s_2, \dots, s_n} \ge \frac{1}{2} \sum_{i=1}^{n} \log \binom{s}{s_i}.$$

Proof Defining $t_i = \min\left\{s, \sum_{j=1}^i s_j\right\}$ and $t'_i = \min\left\{s, \sum_{j=i}^n s_j\right\}$, for $1 \le i \le n$, and $t_0 = t'_{n+1} = 0$, we prove the lemma in two steps. First we show that $\log \binom{s}{s_1,\dots,s_n} \ge \sum_{i=1}^n \log \binom{t'_i}{s_i}$ and $\log \binom{s}{s_1,\dots,s_n} \ge \sum_{i=1}^n \log \binom{t_i}{s_i}$. Therefore, $2 \log \binom{s}{s_1,\dots,s_n} \ge \sum_{i=1}^n \left(\log \binom{t_i}{s_i} + \log \binom{t'_i}{s_i}\right)$. Then we prove $\binom{t_i}{s_i}\binom{t'_i}{s_i} \ge \binom{s}{s_i}$, which implies $\log \binom{t_i}{s_i} + \log \binom{t'_i}{s_i} \ge \log \binom{s}{s_i}$. These two facts together show that $\frac{1}{2} \sum_{i=1}^n \log \binom{s}{s_i} \le \log \binom{s}{s_1,\dots,s_n}$ which concludes the lemma.

Claim 1. $\log \binom{s}{s_1,\dots,s_n} \ge \sum_{i=1}^n \log \binom{t'_i}{s_i}$ **and** $\log \binom{s}{s_1,\dots,s_n} \ge \sum_{i=1}^n \log \binom{t_i}{s_i}$**.**

We prove the first inequality; the second one can be proved similarly. Consider an arbitrary set $X = \{x_1, \dots, x_s\}$ of size s. First, we prove by induction on i that the number of ways to select subsets $X_1, \dots, X_i$ of sizes $s_1, \dots, s_i$ of X such that $|X - \cup_{j=1}^i X_j| \le t'_{i+1}$ is greater than or equal to $\prod_{j=1}^i \binom{t'_j}{s_j}$. Then, the induction hypothesis for $i = n$ proves the claim.

The base case where $i = 0$ is trivial. We show that if sets $X_1, \dots, X_{i-1}$ have been selected such that $|X_j| = s_j$ for every j, $1 \le j \le i-1$, and $|X - \cup_{j=1}^{i-1} X_j| \le t'_i$, there are at least $\binom{t'_i}{s_i}$ ways to choose the set X_i of size s_i such that $|X - \cup_{j=1}^{i} X_j| \leq t'_{i+1}$. Define $Y = X - \cup_{j=1}^{i-1} X_j$. Since $|Y| \leq t'_i$, there exists a set Y' of size t'_i such that $Y \subseteq Y' \subseteq X$. For every subset X_i of size s_i of Y', we have $|X_i \cap Y| \geq s_i - |Y' - Y| = s_i - (t'_i - |Y|)$. Therefore, $|Y - X_i| \leq t'_i - s_i \leq t'_{i+1}$ while $Y - X_i = X - \cup_{j=1}^{i} X_i$. However, X_i is an arbitrary subset of size s_i of Y' with size t'_i. Hence, there are $\binom{t'_i}{s_i}$ choices for X_i.

Claim 2. For every i, $1 \leq i \leq n$, $\binom{t_i}{s_i}\binom{t'_i}{s_i} \geq \binom{s}{s_i}$.

We consider the set $X = \{x_1, \ldots, x_s\}$, and define

$$\begin{aligned} Y &= \{x_1, \ldots, x_{t_i}\} && \text{and} \\ Y' &= \{x_1, \ldots, x_{s_i}, x_{t_i+1}, \ldots, x_s\}. \end{aligned}$$

Hence, $|Y| = t_i$ and $|Y'| \leq t'_i$. We define a *subset pair* as a pair (A, B) such that $|A| = |B| = s_i$, $A \subseteq Y$, and $B \subseteq Y'$. Clearly the number of subset pairs is at most $\binom{t_i}{s_i}\binom{t'_i}{s_i}$. We now prove that the number of subset pairs is greater than or equal to the number of subsets of X of size s_i, which is $\binom{s}{s_i}$. We define the *result* of a subset pair (A, B) as the set $(A - Y') \cup (B - Y) \cup (A \cap B)$. For every subset $S \subseteq X$ of size s_i, we can construct a subset pair $T = (A, B)$ such that the result of T is S. Since $S \subseteq Y \cup Y'$ and $|Y \cap Y'| = |S|$, $|(Y \cap Y') - S| = |S - Y'| + |S - Y|$. Therefore, there are disjoint subsets X' and X'' of $Y \cap Y'$ of sizes $|X'| = |S - Y'|$ and $|X''| = |S - Y|$, such that $X' \cup X'' = (Y \cap Y') - S$. If we define $A = ((Y \cap Y') - X') \cup (S - Y')$ and $B = ((Y \cap Y') - X'') \cup (S - Y)$, it is easy to verify that (A, B) is a subset pair and its result is S. $\square$

3.2.2 Overview of the Proof

In this section, we propose two lower bounds which, if proved, together show the correctness of Conjecture 1. We prove the first lower bound for the most general type of trees. Although proving the second lower bound seems difficult in the general case, we show its correctness on various types of expression trees. These trees consist of union, intersection and symmetric difference operator nodes with a possible difference operator as the root node, but no complement node at all. We conjecture that the second lower bound applies even for arbitrary trees.

We introduce an adversary $\mathcal{B}$ which arbitrarily chooses a maximal proof labeling Λ among all possible maximal proof labelings and answers queries according to it so that $\Lambda(\text{root})$ becomes the result of the root node. The adversary fixes the input gradually such that it is always consistent with the history of the queries it has answered.

As the algorithm has access to only the relative values of elements, we can consider whatever value set we like for the input being constructed by the adversary (see Observation 4). For convenience in this chapter, members of $\Lambda(\text{root})$ and the sets associated with leaves in the input being fixed will be triples of integers which are compared in a lexicographical manner. In particular, members of $\Lambda(\text{root})$ are of the form $(i, 0, 0)$, where $1 \leq i \leq |\Lambda(\text{root})|$ is an integer.

The adversary $\mathcal{B}$ divides the sequence of members of every leaf v of T into $|\Lambda(v)|$ consecutive *regions* of size $\left\lfloor \frac{\mathsf{size}(v)}{|\Lambda(v)|} \right\rfloor$ or $\left\lceil \frac{\mathsf{size}(v)}{|\Lambda(v)|} \right\rceil$. Each of the $|\Lambda(v)|$ regions in a leaf is named after the corresponding element in the maximal proof labeling $\Lambda(v)$: if the ith smallest member of $\Lambda(v)$ is $(a, 0, 0)$, then the ith region of v is called an *a-region*.

In any a-region $\mathcal{R}$, there is exactly one element with a value 'close' to the value $(a, 0, 0)$, more precisely, with a value $(a, 0, x)$, for some x. This element is called the *crucial member of $\mathcal{R}$*. The only role of non-crucial members of a region is to "hide" the position of the crucial element and prevent the algorithm to find it fast. The reason why the value of the crucial member of an a-region is 'close' to $(a, 0, 0)$, but not exactly equal to it, is to enable the adversary to force the algorithm to make enough comparisons between crucial members of different a-regions before making sure that if any value 'close' to $(a, 0, 0)$ is in the result set of the root. For better intuition, consider the tree in Figure 3.4. and suppose all leaves

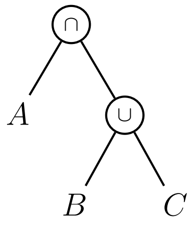


Figure 3.4: Example tree of leaves with a-regions

A, B, and C have a-regions, for some a. If the algorithm first finds the crucial members of a-regions of A and B and compares them, the adversary says they are not equal, so the algorithm needs to also find crucial member of the a-region of C and compare it to those of A and B before knowing whether any value of the form $(a, 0, x)$, for some x, is in the result set of the whole tree.

We now explain the role of each of the three coordinates in the values of elements.

First coordinate: The first coordinate of each member of an a-region is equal to a. These values, which are fixed from the beginning, are used to force every element of an a-region to be smaller than every element of a b-region when $a < b$.

Second coordinate: The second coordinate of the crucial member of an a-region is '0' and that of non-crucial members is non-zero. This way, as promised, only crucial members of a-regions have values 'close' to $(a, 0, 0)$. The adversary does not fix second coordinates from the beginning. So it is not clear from the beginning which member is crucial. The adversary fixes the second coordinate of each member of an a region when the algorithm touches it for the first time in such a way that the algorithm does not reach the crucial member of a region before touching sufficiently many non-crucial members of that region.

Third coordinate: The third coordinate is irrelevant for non-crucial members. It determines the relative values of crucial members of a-regions, for each value of a, and is the adversary's means to prevent crucial members of a-regions to be promoted in the tree before all of them are touched by the algorithm and enough comparisons are made between them.

Our first and second lower bounds are obtained by choosing appropriate strategies for determining the second and the third coordinates respectively.

3.3 The First Lower Bound

With our first lower bound, we show that given any signature, the number of comparisons needed by any algorithm for inputs with that signature exceeds the first part of Equation 3.1 in the worst case.

The strategy is to determine the second coordinates of triples of any a-region $\mathcal{R}$, for any a, in such a way that $\mathcal{A}$ does not touch the crucial member of $\mathcal{R}$ before touching $\log |\mathcal{R}|$ members of $\mathcal{R}$, where $|\mathcal{R}|$ denotes the length of $\mathcal{R}$. Also, we will show that the algorithm will be forced to touch all crucial members. These two facts together prove that the algorithm needs to touch the desired number of elements (as specified by the first part of Equation 3.1) overall.

3.3.1 The Binary Search Strategy

We first explain the strategy of determining second coordinates of the triples.

The idea is essentially similar to binary searching. For every region $\mathcal{R}$ we consider a variable $\mathcal{S}$ pointing to a subsequence in $\mathcal{R}$, showing the subsequence in which the crucial member hides. At the beginning the part pointed to by $\mathcal{S}$ is the whole of $\mathcal{R}$. After touching each element e in $\mathcal{S}$, the adversary decides to hide the crucial member in the half of $\mathcal{S}$ that does not include e, thus cutting the size of S to no less than half.

Now we explain the details. At any point, the second coordinate of every member in $\mathcal{R} \setminus \mathcal{S}$ is already fixed: the second coordinate of every member of $\mathcal{R} \setminus \mathcal{S}$ placed before members of $\mathcal{S}$ is at most $-|\mathcal{S}|$, and the second coordinate of every member of $\mathcal{R} \setminus \mathcal{S}$ placed after members of $\mathcal{S}$ is at least $|\mathcal{S}|$. Now whenever a member s of $\mathcal{S}$ is touched, if s is the only member of $\mathcal{S}$, the algorithm has found the crucial member of $\mathcal{R}$, so the adversary sets the second coordinate of s to zero, marking s as the crucial member of $\mathcal{R}$. Otherwise, depending on whether s is in the first half or in the second half of $\mathcal{S}$, $\mathcal{B}$ considers s and members of $\mathcal{S}$ placed after or before s, fixes second coordinates of these members as explained in Algorithm 1, and deletes them from $\mathcal{S}$. Figure 3.5 shows an example.

Algorithm 1: How to determine the second coordinates of members.

```
if $|\mathcal{S}| = 1$ then
    – set the second coordinate of $s$ equal to zero;
    – set $\mathcal{S}$ equal to the empty sequence;
else
    suppose $s$ is the $i$th member of $\mathcal{R}$;
    if $i < |\mathcal{S}| - i + 1$ then
        – assign values $-(|\mathcal{S}| - 1)$, $-(|\mathcal{S}| - 2)$, ..., $-(|\mathcal{S}| - i)$ to the second
        coordinates of the first $i$ members of $\mathcal{S}$;
        – Remove the first $i$ members of $\mathcal{S}$ from it;
    else
        – assign values $i - 1$, $i$, ..., $|\mathcal{S}| - 1$ to the second coordinates of the last
        $|\mathcal{S}| - i + 1$ members of $\mathcal{S}$;
        – Remove the last $|\mathcal{S}| - i + 1$ members of $\mathcal{S}$ from it;
    end
end
```

If n is the length of $\mathcal{S}$ for a region $\mathcal{R}$, by touching a member of $\mathcal{R}$ the length of $\mathcal{S}$ is reduced to at least $\lfloor \frac{n}{2} \rfloor$. Since the value of 0 is not assigned to the second coordinate of any member unless $|\mathcal{S}| = 1$, $\log |\mathcal{R}|$ members of $\mathcal{R}$ must be touched before the crucial member of $\mathcal{R}$ is touched.

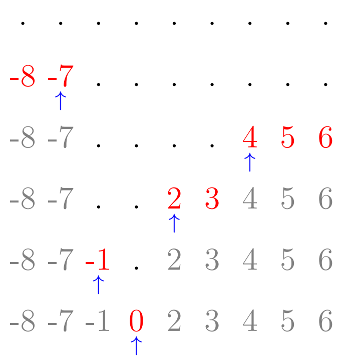


Figure 3.5: An example of how second coordinates of a region for a leaf are determined. Each row shows the second coordinates of elements after touching the element indicated by an arrow.

Whenever a member is touched for which the second coordinate has not been determined before, before attempting to answer the query, $\mathcal{B}$ determines the second coordinate according to the method we described here. A leaf l has $|\Lambda(l)|$ crucial members, each in a region of size at least $\left\lfloor \frac{\mathsf{size}(l)}{|\Lambda(l)|} \right\rfloor$. Therefore, we have the following lemma.

Lemma 10 *If all crucial members of $L \subseteq \mathsf{leaves}(T)$ are touched by an algorithm $\mathcal{A}$ in interaction with the adversary we described, then $\mathcal{A}$ has submitted at least*

$$\sum_{l \in L} |\Lambda(l)| \left(1 + \log \left\lfloor \frac{\mathsf{size}(l)}{|\Lambda(l)|} \right\rfloor\right)$$

queries.

3.3.2 Touching all Crucial Elements

In this section, we prove that the adversary can force the algorithm to touch all crucial-members of leaves by its queries. The proof we present in this section is similar to but less sophisticated than that of the second lower bound presented in Section 3.4.

As queries (x, y) arrive, in most cases the adversary can decide whether $x < y$, $x = y$, or $x > y$ only by looking at the first two coordinates of x and y (and if necessary following the aforementioned approach to determine these coordinates). The exception is when x and y both are crucial members of a-regions for some $(a, 0, 0) \in \Lambda(\text{root})$. In these cases, the first two coordinates of x and y are the same. The third coordinate of elements are set by the adversary at the first time they are touched and determine the relative values of crucial members of a-regions.

Let us fix an element $(a, 0, 0) \in \Lambda(\text{root})$ and focus on the strategy of the adversary in determining the third coordinate of crucial members of a-regions. The goal of the adversary we describe in this section is to make the algorithm touch all crucial members. The adversary will work for the most general type of trees. In the next section we will redesign the strategy of the adversary for determining third coordinates in such a way that the algorithm needs not only to touch all crucial members, but also to make sufficiently many comparisons between them. This will allow us to obtain a better lower bound (which matches the worst-case running time of our algorithm presented in Chapter 5). However, that adversary works only for a more limited type of trees.

For simplicity, we assume the tree is in minus-free form (see Lemma 1). We create a subtree of the original tree, denoted by $T^{(a)}$, that consists of nodes v for which $\Lambda(v)$ contain $(a, 0, 0)$. By the result of *result of* $T^{(a)}$ we mean the result of the input if only members of a-regions are considered.

From now on in this section and also in Section 3.4, we may use a leaf ℓ of $T^{(a)}$ and the crucial member of the a-region of ℓ (if it exists) interchangeably. For example, when talking about query (ℓ_1, ℓ_2), for leaves ℓ_1 and ℓ_2 of $T^{(a)}$, we mean the query made between crucial members of a-regions of ℓ_1 and ℓ_2, or similarly we may talk about the third coordinate of a leaf, or a leaf being touched.

As mentioned before, the goal of the adversary is to ensure the result of $T^{(a)}$ cannot be computed unless all the leaves of $T^{(a)}$ are touched. Here is the approach the adversary takes to set the third coordinate of a leaf ℓ when it is touched.

1. Let v be the lowest ancestor of ℓ that has at least another descendant leaf other than ℓ in $T^{(a)}$, that is not touched. If there is no such ancestor, then ℓ is the only leaf of $T^{(a)}$ that is not touched. In this case,
   (a) If ℓ is a normal node, the adversary sets the third coordinate of ℓ to zero.
   (b) Otherwise, if ℓ is a complement node, the third coordinate is set to a unique positive integer.
2. If the ancestor v exists and one of the following holds:
   (a) v is an intersection node and ℓ is a normal node, or
   (b) v is a union or delta node and ℓ is a complement node,

then the adversary sets the third coordinate of ℓ to zero.

3. Otherwise (i.e. in cases not covered above), the adversary sets the third coordinate of ℓ to a unique positive integer.

Suppose queries are answered in this manner, and v is an internal node with at least one untouched node, and u is a child of v whose all leaves are touched. We can see that if v is an intersection node, $(a, 0, 0)$ is in the result set of u; otherwise $(a, 0, 0)$ is not in the result set of u. Therefore, for each internal node v with at least one untouched node, it is not determined yet if $(a, 0, 0)$ is in the result set of v or not. Thus, assuming that $T^{(a)}$ has at least two leaves, the adversary guarantees at any point that the algorithm does not have enough knowledge to ensure $(a, 0, 0)$ is in the root result, unless all the leaves of $T^{(a)}$ are touched.

Lemma 11 *There is an adversary that can force $\mathcal{A}$ to touch all crucial members of all regions of non-shallow leaves of T.*

Lemmas 10 and 11 prove our first lower bound, as expressed in the next theorem.

Theorem 12 (First Lower Bound) *There is an adversary that forces $\mathcal{A}$ to submit at least $\mathcal{L}_1 = \sum_{l \in \mathsf{deep}} |\Lambda(l)| \left(1 + \log \left\lfloor \frac{\mathsf{size}(l)}{|\Lambda(l)|} \right\rfloor\right)$ queries, where* deep *is the set of non-shallow leaves of the tree.*

3.4 The Second Lower Bound

Given a signature S, our goal in this section is to show that the adversary can select any maximal proof labeling of S at the beginning, and act in such a way that the algorithm is forced to collect enough information to be able to discover S. We then will use this result to obtain information theoretic lower bounds on the number of queries submitted by the algorithm.

In order to achieve this goal we need to make our computational model a little bit more generous to the algorithm and, for each query the algorithm submits, let the algorithm get some more information than just the regular answer to its query. We should note that for each query submitted by the algorithm, we will give only one extra bit of information, in addition to the regular answer to the query, and so it will affect the resulting lower bound only by a constant factor. We will discuss this in a formal way later on.

Here is the explanation of the extra information given to the algorithm. We define two crucial members to be *similar* if they both belong to a-regions for some $1 \leq a \leq |\Lambda(\text{root})|$. Whenever a query (a, b) is submitted, in addition to the regular answer to the query, the algorithm is given one extra bit of information which is 1 only if a and b are two similar crucial elements.

Next we show how the algorithm will be able to infer the maximal proof labeling from information revealed to it. We create a graph $\mathcal{G}$ on all elements where there is an edge between two elements if they are compared by the algorithm. The next definition specifies how the adversary will make the proof labeling computable for the algorithm.

Definition 10 *At the end of the interaction, we say* the adversary is in a winning state *if the next two conditions hold.*

1. *The algorithm has submitted enough queries such that for any a, $1 \leq a \leq |\Lambda(root)|$, the subgraph induced by crucial members belonging to all a-regions is connected.*
2. *All members of $\Lambda(root)$ are in the result set of the root.*

The next lemma shows why the definition satisfies our goal.

Lemma 13 *At the end of the interaction between an adversary and algorithm, if the adversary is in a winning state, one can determine the maximal proof labeling using the information revealed to the algorithm.*

Proof By definition of the winning state, for every crucial element e, the algorithm knows the set of all crucial elements similar to e. Also, at the end, the algorithm has computed a solution for the instance, which means the algorithm knows a sequence of promoted elements whose sequence of values is $\Lambda(\text{root})$ in increasing order. So, for each a where $(a, 0, 0) \in \Lambda(\text{root})$, the algorithm knows all leaves l where $(a, 0, 0) \in \Lambda(l)$, that is, the algorithm knows $\Lambda(l)$ for all leaves l. Therefore, the algorithm can compute the function Λ for any node. □

Similar to Section 3.3, the adversary $\mathcal{B}$ knows how to answer queries (x, y) if either of x or y is not a crucial member or if x and y are not similar, based on the first two coordinates of x and y. Our adversary in this section differs from the adversary in Section 3.3 in the way it sets the third coordinate of crucial members. We again focus on a particular value a, $1 \leq a \leq |\Lambda(\text{root})|$, and discuss queries (x, y) where x and y are crucial members of a-regions. Also, as mentioned we use a leaf l of $T^{(a)}$ and the crucial member of the a-region of l interchangeably.

Each condition in Definition 10 can be broken down into $|\Lambda(\mathrm{root})|$ independent conditions, one for each member of $\Lambda(\mathrm{root})$. For each value a, $1 \leq a \leq |\Lambda(\mathrm{root})|$, we monitor the interaction between the algorithm and the adversary on crucial members of a-regions independently, and try to ensure that the adversary satisfies conditions 1 and 2 for that particular value of a. So let us fix a value a, $1 \leq a \leq |\Lambda(\mathrm{root})|$, consider the tree $T^{(a)}$, and focus on the queries made between crucial members of a-regions of leaves of $T^{(a)}$. We also only consider the subgraph of $\mathcal{G}$ induced by crucial members of a-regions of leaves of $T^{(a)}$, and denote it by $\mathcal{G}_a$. Moreover, the label of the edge between members a and b is '$<$', '$=$', or '$>$', if $\mathcal{B}$'s answer to the query (a, b) is $<$, $=$, or $>$, respectively.

$\mathcal{A}$ and $\mathcal{B}$ play the following game on $T^{(a)}$: the algorithm submits queries comparing two leaves and $\mathcal{B}$ responds in a non-contradictory manner. The game finishes when $\mathcal{A}$ has a proof that the result of $T^{(a)}$ is empty or not empty. $\mathcal{B}$ aims to avoid ending of game before $\mathcal{G}_a$ becomes connected, and also to make sure the result of $T^{(a)}$ is non empty at the end.

We use the following general schema for setting the third coordinates of leaves (i.e. of the crucial members of a-regions of leaves). We never set the third coordinate of a leaf to a negative value, and we may set it to zero only at the very end of the game. Moreover, there is a global variable called *eliminator*, which is initialized to the number of leaves of the tree. This variable is used whenever the adversary wants to set the third coordinate of a leaf ℓ to a positive value: it sets that coordinate to the current value of eliminator, and eliminator is decreased by one. From that point, we say ℓ is *eliminated*.

Observation 14 *If a leaf ℓ_1 is eliminated before a leaf ℓ_2, the third coordinate of ℓ_1 is greater than that of ℓ_2.*

Since $\mathcal{G}_a$ is connected for $n = 1$, we can assume that $n > 1$, and hence, for every leaf ℓ of $T^{(a)}$, ℓ is a non-shallow leaf.

Let $(\ell_1^{(i)}, \ell_2^{(i)})$ be the ith query between crucial members of a-regions of leaves of the tree submitted by $\mathcal{A}$. In order to respond to this query, $\mathcal{B}$ looks at the third coordinates of $\ell_1^{(i)}$ and $\ell_2^{(i)}$. If both $\ell_1^{(i)}$ and $\ell_2^{(i)}$ are eliminated, then $\mathcal{B}$ can answer the query without doing anything further. If only one of them, say $\ell_1^{(i)}$, is eliminated, then $\mathcal{B}$ answers $\ell_1^{(i)} > \ell_2^{(i)}$, without setting the third coordinate of $\ell_2^{(i)}$. Due to Observation 14, this is consistent with future queries. The final case, which is more involved, is the case in which none of $\ell_1^{(i)}$ and $\ell_2^{(i)}$ are eliminated. In the rest of this section we explain the strategy for dealing with this case.

We introduce a number of invariants that, due to the way the adversary responds to the query, as we will prove, hold during the game. The first invariant comes from the fact that a leaf eliminated from the tree will not be equal to every non-eliminated leaf.

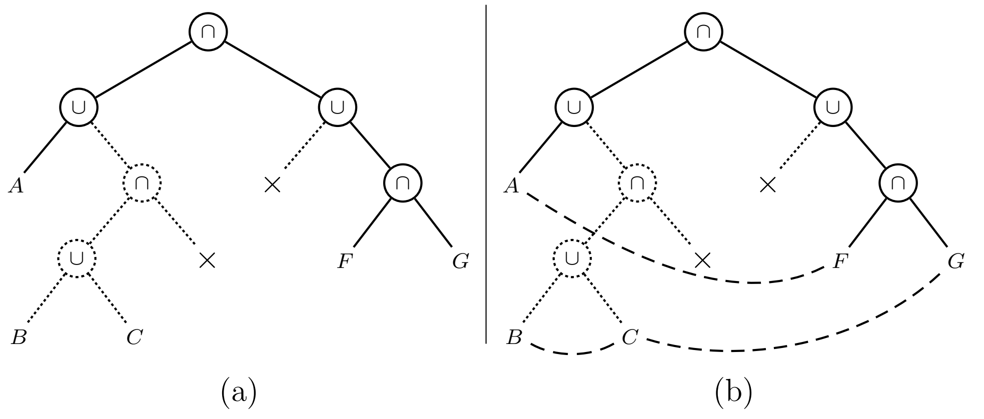


Figure 3.6: (a): Example of how eliminated leaves (indicated by '×') make parts of tree "unimportant" (marked by dashes). (b): Example of the graph $\mathcal{G}_a^=$ where Invariant 2 holds. The connections between leaves of the tree are edges of $\mathcal{G}_a^=$.

Invariant 1 *The crucial member of the a-region of an eliminated leaf is not promoted.*

Inspired by this fact, we say a node v of $T^{(a)}$ is *unimportant* if any of the following conditions holds:

1. v is an eliminated leaf.
2. $\mathsf{opr}(v) \in \{\cup, \Delta\}$ and all the children of v are unimportant.
3. $\mathsf{opr}(v) = \cap$ and at least one of its children is unimportant.
4. v is a descendant of an unimportant node.

All other nodes are called *important* nodes. Figure 3.6(a) is an example. Due to Invariant 1, it is easy to see that the crucial member of the a-region of an unimportant leaf cannot be promoted.

We say two leaves are *united* if they are proven to have equal third coordinates. More formally, they are united if they are in the same connected component of $\mathcal{G}_a^=$, where $\mathcal{G}_a^=$ is the result of removing all edges from $\mathcal{G}_a$ excepts those with label '='. An important leaf that is united with a leaf ℓ is called a *backup* for ℓ. Note that every leaf is united with itself, and so every important leaf is a backup for itself.

Invariant 2 *At any point, every non-eliminated leaf has a backup.*

Invariant 2 guarantees that any unimportant leaf, which can be ignored by the algorithm as it has no influence on the result of the tree, is connected to an important one, which prevents the tree from being disconnected when something is promoted. See Figure 3.6 for an example.

Given a node v and a leaf ℓ of $T^{(a)}[v]$[1], ℓ is *v-restricted* if v is not united with any important leaf outside $T^{(a)}[v]$. As an example, in Figure 3.6(b), for v the right child of the root, the leaf G is v-restricted. A node v with a v-restricted important leaf in its subtree is *incomplete*. For example, in the tree of Figure 3.6(b), the right child of the root is an incomplete node while the left child of the root is complete. The next invariant guarantees that if v is an incomplete node, by eliminating its v-restricted nodes, we can make the result of v empty, which intuitively means the result set of v cannot be determined to be empty without looking at values of v-restricted leaves.

Invariant 3 *For any incomplete node v, any sub-intersection tree of v has a v-restricted leaf.*

For a union node v, we do not want a situation where v has a complete important child u_1, and at the same time, an incomplete important child u_2 To see why, note that leaves of u_1 are unified with other leaves and we may not be able to eliminated them in the future. Hence, this situation may enable the algorithm to finish the game before making u_2 complete. So we will ensure that the next invariant also holds.

Invariant 4 *For every incomplete union node v, every important child of v is incomplete.*

Furthermore, once an important union node v gets complete, we want that only one child of v remains important. This will ensure that at the end at most one child of a union node has a non-empty contribution set, which means a union node will behave the same as a delta node. So we can treat delta nodes the exact same way we treat union nodes, which simplifies the problem.

Invariant 5 *An important complete union node v has exactly one important child.*

Lemma 15 *If Invariants 4, and 5 hold, Invariant 3 also holds.*

[1]For a tree T and a node v, $T[v]$ is the subtree of T that includes v and all of its descendants in T.

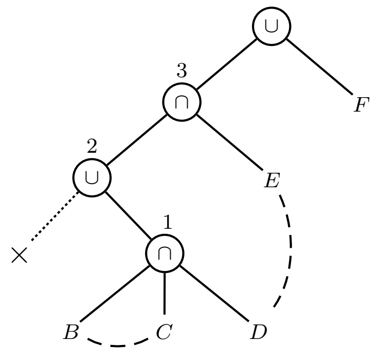


Figure 3.7: The ancestors number 1 and 2 of C are complete with C. But the ancestor number 3 of C is not complete with C and so it is the effective ancestor of C.

Proof We use induction on the height of a node v to show the correctness of Invariant 3 for v and any sub-intersection tree U of v. The invariant clearly is true for leaves v. If v is an incomplete union node and u is a child of v in U, by Invariant 4, u is incomplete and so by induction U has a u-restricted leaf, which is also v-restricted. Next suppose v is an incomplete intersection node, that is, there is an important v-restricted leaf l in $T[v]$. Consider the lowest ancestor u of l in U. Clearly u is a union node. By induction u is not incomplete. The node u has at least two important children: an ancestor of l and a node in U. This is a contradiction with Invariant 5. □

When all important v-restricted leaves of a node v are united with a leaf ℓ of $T[v]$, we say v is *complete with* ℓ. We define the *effective ancestor* of a leaf ℓ to be the first (lowest) ancestor v of ℓ that is not complete with ℓ. The example in Figure 3.7 illustrates the concept. If v is the effective ancestor of ℓ, the result of any node on the path connecting v to ℓ, excluding v, depends on ℓ: if ℓ is eliminated, nothing from their subtree may be promoted, but if ℓ is not eliminated, this might be the case.

A leaf ℓ with effective ancestor v is *eliminable* if v is a $\cup$ or Δ node, or v is a minus node and l is in its right subtree. When an eliminable node l is compared to another node, the strategy is roughly to eliminate l so that we can keep Invariant 4 true. In order to do that, we need to avoid situations where leaves l_1 and l_2 are united and are descendants of two different children of a union node v and the effective ancestor of l_1 is v, because then by eliminating l_1, the l_2 also gets eliminated, and this way, a subtree containing l_2 may get unimportant before it gets complete. See Figure 3.8 for an example. So, we make sure the following invariant holds during the game.

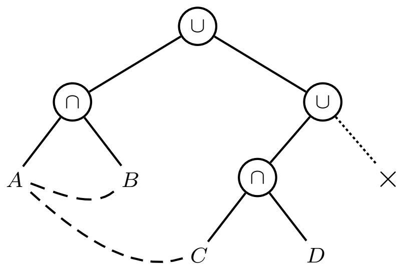


Figure 3.8: In this example, the effective ancestor of the leaf A is a union node (the root, in this example), and so A is eliminable. A is united with C which is a descendant of a different child of the union node, Now if A gets eliminated, C and D both get unimportant, while D has no backup.

Invariant 6 *For an eliminable leaf l with an effective ancestor v, there is no important leaf l' united with l unless l and l' are descendants of the same child of v.*

Lemma 16 *Suppose all invariants 1 to 6 hold at some point during the game, the algorithm submits a query(ℓ_1, ℓ_2), and the adversary responds according to Algorithm 2. Then, after the adversary response, invariants 1 to 6 still hold.*

Proof We consider two cases based on any of ℓ_1 and ℓ_2 having an eliminable backup. First suppose one of them, say ℓ_1, has an eliminable backup and the adversary eliminates ℓ_1 by his response. Suppose v is the effective ancestor of ℓ_1 and u is the child of v such that ℓ_1 is in $T[u]$. By Invariant 6, every backup of ℓ_1 is in $T[u]$, all being eliminated by this response. So, by eliminating ℓ_1, only elements in $T[u]$ may become unimportant. Also, there is no node not united with ℓ_1 that is u-restricted. So every u-restricted node finds a new backup ℓ_2. Thus after the operation Invariant 2 remains true. Invariants 4 and 5 also remain true because the whole subtree $T[u]$ becomes incomplete and importance and completeness of any other node does not change. So by Lemma 15, Invariant 3 will also hold. Invariant 6 also will be true because no new node becomes eliminable.

Next suppose both ℓ_1 and ℓ_2 are non-eliminable and hence the response will be equality of ℓ_1 and ℓ_2. Suppose v_1 and v_2 are effective ancestors of ℓ_1 and ℓ_2, respectively, and u_1 and u_2 are children of v_1 and v_2, respectively, such that ℓ_1 is in $T[u_1]$ and ℓ_2 is in $T[u_2]$. It is easy to see that after the response v_1 and v_2 remain incomplete and so no node outside $T[u_1]$ and $T[u_2]$ becomes complete, while after the response u_1 and u_2, which are children of an intersection node, will be complete. So Invariants 4 and 5 remain true. Moreover, if

Algorithm 2: How to determine the third coordinates of members.

```
if $\ell^{(i)}{}_1$ is united with $\ell^{(i)}{}_2$ then
    answer '=';
else
    if $\ell^{(i)}{}_1$ has an eliminable backup $l$ then
        eliminate $l$ and return $\ell^{(i)}{}_1 > \ell^{(i)}{}_2$.
    else
        if $\ell^{(i)}{}_2$ has an eliminable backup $l$ then
            eliminate $l$ and return $\ell^{(i)}{}_2 > \ell^{(i)}{}_1$.
        else
            answer '=';
        end
    end
end
```

any leaf l becomes an eliminable node by this response, it is in $T[v_1]$ and is v_1-restricted, or it is in $T[v_2]$ and is v_2-restricted. So Invariant 6 will also hold. □

Now we can see why the strategy we designed for the adversary works. All we need to show is that at the end all non-eliminated nodes are connected. If the root of the tree is a minus node, the right child of the root should be eliminated, otherwise its result set could be non-empty which would mean the result of the whole tree is empty. When the game finishes the algorithm knows a sub-intersection tree all of whose leaves are important and unified. But from Invariant 5 we know a complete node has only one-intersection tree in which all leaves are important. So this means at the end all important leaves are unified and thus by Definition 10 the adversary is in a winning state. Therefore, by Lemma 13, using the information revealed to the algorithm, one can compute the proof labeling.

We showed that for each canonical proof labelings, there is an input where one can find the proof labeling by looking at the input signature and the interaction between the algorithm and the adversary. Suppose there are N canonical proof labelings for a signature S, and an algorithm $\mathcal{A}$ performs at most c comparisons on inputs with signature S. Since on each comparisons made by the algorithm, there are six possibilities for the response we get from the adversary (the result of the comparison, and the extra bit the adversary reveals), there are totally at most 6^c distinct sequence of responses we may see from the adversary, and hence, the number of distinct canonical proof labelings for the signature may not exceed 6^c. Thus, c is at least a constant factor of $\log N$ which by Lemma 8 is a constant factor of the second part in Equation 3.1. This proves the following theorem.

Theorem 17 *For a signature S, given that there is no complement node in the tree and aside from root, every node is a union or intersection node, Conjecture 1 holds.*

Chapter 4

The Worst-Case Optimal Algorithm

In this chapter we present our first algorithm which:

- works on sorted arrays as input format and the output is in cross reference format,
- the input is a union-intersection input, and
- the algorithm is worst-case optimal.

Throughout this chapter we assume the parent of every intersection node is a union node and vice versa. Obviously this does not reduce the generality of the problem as one can always merge an internal node with its parent if they have the same operator.

To present the algorithm, we need to first explain two special cases of the problem in Section 4.1; these two sub-problems will be used in the algorithm for the general problem, which is presented in Section 4.2. Then in Sections 4.3 and 4.4 we investigate the correctness and analyze the running time of the algorithm and show it matches the lower bound.

4.1 Special Cases

First we study two special cases separately; these special problems will be used in solving the general problem.

4.1.1 Special Case: Union of Sets

The first special case involves evaluating the union of a series of sets: $X_1 \cup X_2 \cup \ldots \cup X_k$. Hwang and Lin [31] studied this problem for $k = 2$. They showed how to compute $A \cup B$ with tight lower and upper bounds of $\Theta\left(\log \binom{|A|+|B|}{|A|}\right)$.

Algorithm 3 is a solution we suggest for the more general case where k can have values greater than two.[1] It has some similarities with the Huffman algorithm [29]: each time it selects two of the smallest sets and replaces them with their union. We use this similarity in analysis of the algorithm.

```
Algorithm 3: SimpleUnion($\mathcal{X}$)
  Data: $\mathcal{X}$: collection of input sets
  Result: The union of the sets inside $\mathcal{X}$
  while there is more than 1 set in $\mathcal{X}$ do
      Select the two smallest sets in $\mathcal{X}$, call them $X_a$ and $X_b$;
      Compute $X_a \cup X_b$, using the trivial merge algorithm in $O(|X_a| + |X_b|)$;
      Replace $X_a$ and $X_b$ in $\mathcal{X}$ with $X_a \cup X_b$;
  end
  Report the only set in $\mathcal{X}$ as the result;
```

Fixing the number n of the sets in $\mathcal{X}$ and their sizes, we first show that the worst-case scenario of Algorithm 3 happens when all sets in $\mathcal{X}$ are disjoint. To show this intuitive fact, we use the following notation. For a given sequence $A = a_1, \ldots, a_n$ of integers, we use $\mathsf{sorted}(A)$ to denote the sequence of elements of A in sorted order (which is of the same length n). Also, for two sequences $A = a_1, \ldots, a_n$ and $B = b_1, \ldots, b_n$ of the same length, we say $A \leq B$ if $a_i \leq b_i$ for all $1 \leq i \leq n$.

Lemma 18 *For sequences A and B of the same length, if $A \leq B$ then $\mathsf{sorted}(A) \leq \mathsf{sorted}(B)$.*

Proof We use induction on the length n of A and B to prove the lemma. For $n = 1$ the lemma is obvious. Suppose $n > 1$ and $A = a_1 \ldots, a_n$, and $B = b_1 \ldots, b_n$. Also suppose a_i and b_j are the smallest elements of A and B, respectively. Due to the choices of i and j

[1]After submitting this algorithm as a paper, we noticed this technique (for this special case) was published before by Burge [18].

and the fact that $A \leq B$, we have $a_i \leq a_j \leq b_j \leq b_i$. Thus, for the sequence A' obtained by swapping a_i and a_j in A, we have $A' \leq B$, while in each of A' and B the smallest member is at index j. Since $A' \leq B$, if we remove the elements at index j from both of them, we obtain sequences A^* and B^* with $A^* \leq B^*$ and so, by induction, $\mathsf{sorted}(A^*) \leq \mathsf{sorted}(B^*)$. Now it is easy to see that $\mathsf{sorted}(A)$ and $\mathsf{sorted}(B)$ can be obtained by attaching a_i and b_j to the beginning $\mathsf{sorted}(A^*)$ and $\mathsf{sorted}(B^*)$. Thus, as $a_i \leq b_j$, $\mathsf{sorted}(A) \leq \mathsf{sorted}(B)$. $\square$

Now considering two inputs $\mathcal{X}$ and $\mathcal{X}^*$ where the sets in both inputs have the same sequence of sizes $(s_1, \ldots, s_n)$, and in $\mathcal{X}^*$ all sets are disjoint, we show that the time the algorithms spends for $\mathcal{X}$ is no more than the time it spends for $\mathcal{X}^*$. Define $S_0 = S_0^* = (s_1, \ldots, s_n)$, and for $i \geq 1$, define S_i (S_i^*, respectively) to be the sequence of sizes of sets after the ith round of the algorithm on input $\mathcal{X}$ (on input X^*, respectively) in sorted order. We use induction on i to show that $S_i \leq S_i^*$. For $i = 0$, this is true as $S_i = S_i^*$.

To prove the induction hypothesis for $i > 0$, consider the set Y created in round i of the algorithm when run on $\mathcal{X}$, and similarly the set Y^* created in round i of the algorithm when run on $\mathcal{X}^*$. We define S_i' to be the sequence obtained from S_{i-1} by replacing the two smallest members of S_{i-1} with the size of Y, and similarly $S_i^{*'}$ to be the sequence obtained by replacing the two smallest members of S_{i-1}^* with the size of Y^*. Now it is easy to see that $S_i = \mathsf{sorted}(S_i')$, $S_i^* = \mathsf{sorted}(S_i^{*'})$, and because $S_{i-1} \leq S_{i-1}^*$, $S_i' \leq S_i^{*'}$. So, by Lemma 18, $S_i = \mathsf{sorted}(S_i') \leq \mathsf{sorted}(S_i^{*'}) = S_i^*$. Thus, in all rounds, the sizes of the sets chosen by the algorithm when run on $\mathcal{X}$ are smaller than or equal to the sizes of the sets chosen when run on $\mathcal{X}^*$. Therefore the claim is true.

We proved the worst-case number of comparisons happens for cases when all sets in $\mathcal{X}$ are disjoint. In such cases, the aforementioned algorithm works similarly to Huffman coding; consider each set X_i as a symbol appearing $|X_i|$ times in a text to be encoded and construct the Huffman tree for encoding such a text. In each step of Algorithm 3, we select the two smallest elements and replace them with a new element aggregating them, so following exactly the Huffman algorithm. Therefore, the depth of a set X_i in the corresponding Huffman tree shows the number of times that X_i or a superset of X_i is selected in step 1 of our algorithm.

The overall number of comparisons is $\sum_{i=1}^n |X_i| h(X_i)$, where $h(X_i)$ is the depth of X_i in the corresponding Huffman tree. On the other hand, we know $\sum_{i=1}^n |X_i| h(X_i)$ is the total length of the bits needed to encode the aforementioned text using Huffman tree, which due to the optimality of Huffman trees for such purposes is at most

$$\sum_{j=1}^{n} |X_j| \left(\log\left(\sum_{i=1}^{n} |X_i| / |X_j|\right) + 1 \right) = \sum_{i=1}^{n} s_i \left(\log(s/s_i) + 1 \right).$$

Defining $S_{\max} = \max_{i=0}^{n} S_i$, we can write:

$$\begin{aligned}
\sum_{i=1}^{n} s_i \left(1 + \log \frac{s}{s_i}\right) &= s + \sum_{i=1}^{n} s_i \log \frac{s}{s_i} \\
&\leq 2 \sum_{i=1}^{n} s_i \log \frac{s + s_i}{s_i} \\
&\leq 2 \sum_{i=1}^{n} s_i \log \frac{s + s_{\max}}{s_i} \\
&\leq 2 \sum_{i=1}^{n} \log \binom{s + s_{\max}}{s_i}.
\end{aligned}$$

This proves the following lemma.

Lemma 19 *Suppose $\mathcal{X}$ is a collection of n sets $X_1, X_2, \ldots, X_n$ of sizes $s_1, s_2, \ldots, s_n$ respectively, $s = \sum_{i=1}^{n} s_i$, and $s_{\max} = \max_{i=1}^{n} s_i$. Then, $\bigcup_{i=1}^{n} X_i$ can be computed in time $O\left(\sum_{i=1}^{n} \log \binom{s+s_{\max}}{s_i}\right)$.* □

The running time obtained from Lemma 19 is not optimal if one set is much bigger than the rest. In such cases an optimal algorithm may need to generate the cross reference output in time sublinear to the size of the biggest set, and so it may need to avoid walking though every single element of the biggest set. So, we separate out the largest set (say X_1), and use Algorithm 3 to compute $X_{2,n} = X_2 \cup \ldots \cup X_n$ in $O\left(\sum_{i=2}^{n} \log \binom{s'+s'_{\max}}{s_i}\right)$, where $s', s'_{\max}$ are the sum and the maximum of the sizes of sets $X_2, \ldots, X_n$. Since $s' + s'_{\max} \leq s$, the time is $O\left(\sum_{i=2}^{n} \log \binom{s}{s_i}\right)$. We use the algorithm of Hwang and Lin [31] to compute the union of the largest set and the remaining sets $(X_1 \cup X_{2,k})$ in $O(\log \binom{s}{s_1})$ time. Therefore, the overall time is $O(\sum_{i=1}^{n} \log \binom{s}{s_i})$.

Lemma 20 *Suppose $\mathcal{X}$ is a collection of n sets $X_1, X_2, \ldots, X_n$ of sizes $s_1, s_2, \ldots, s_n$ respectively and $s = \sum_{i=1}^{n} s_i$. Then $\bigcup_{i=1}^{n} X_i$ can be computed in time $O\left(\sum_{i=1}^{k} \log \binom{s}{s_i}\right)$ in cross reference format.* □

Lemma 20 and Lemma 9 together imply the following corollary:

Corollary 21 *A cross reference representation of the union of sets $X_1, X_2, \ldots, X_k$ can be computed in time $O\left(\log \binom{s}{s_1, \ldots, s_k}\right)$, where $s_i = |X_i|$ and $s = \sum_{i=1}^{k} s_i$.* □

This shows that the algorithm takes optimal time matching the lower bound we proved in the previous chapter (see Theorem 17).

To obtain the sorted array representation rather than a cross reference representation, one can expand the ranges of the output to have the union in the sorted list format again. The time this takes is proportional to the size of the output, which is at most $O(\sum_{i=1}^{k} |X_k|)$:

Corollary 22 *A sorted array representation of the union of sets $X_1, X_2, \ldots, X_n$ can be computed in time $O\left(s + \log \binom{s}{s_1,\ldots,s_k}\right)$, where $s_i = |X_i|$ and $s = \sum_{i=1}^{k} |X_i|$.* $\square$

4.1.2 Special Case: Intersection with Union of Small Sets

The second special case has the form $Y \cap (X_1 \cup X_2 \cup \ldots \cup X_k)$, where $|Y| \geq |X_i|$ for all i. For the case where $k = 1$ (i.e. computing $Y \cap X$), This problem has been studied and tight lower and upper bounds of $\Theta(|X| \log \frac{|X|+|Y|}{|X|})$ already exist [31].

To solve the problem for $k > 1$, we first create a boolean array $\mathcal{B}$ of size $|Y|$, so that each element y in Y has an associated element in the array, namely $\mathcal{B}[y]$, which will become true if y appears in one of the X_i's. We then follow these steps:

1. We initialize all the elements in array $\mathcal{B}$ to false.
2. We compute the intersection of each X_i with Y separately ($Y_i = Y \cap X_i$) in
$$O\left(\sum_{i=1}^{k} |X_i| \log \frac{|X_i| + |Y|}{|X_i|}\right)$$
time using Hwang and Lin's algorithm [31].
3. For each Y_i, $1 \leq i \leq k$, and every $y \in Y_i$, we set $\mathcal{B}[y] = \mathit{true}$. Note that, in order for the algorithm to be able to set $\mathcal{B}[y]$ to true in constant time, it needs to know the index of y in Y. This is possible if we keep track of that index for each element of Y during the execution of Hwang and Lin's algorithm in step 2.
4. Finally, we scan array $\mathcal{B}$ and return as output each element b such that $\mathcal{B}[b]$ is true.

It is clear that going through all Y_i's will take $\sum_{i=1}^{k} |Y \cap X_i|$ which is less than the time consumed for computing the Y_i's. Also creating B in the beginning and scanning it in the end takes time $O(|Y|)$; therefore:

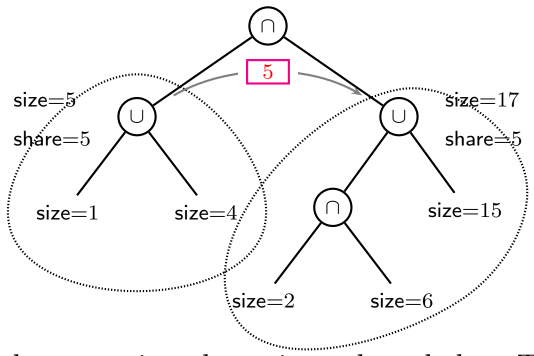


Figure 4.1: An example where passing the universal set helps. The intersection node root in this figure has two children, with maximum result sizes of 5 and 17. The maximum result set of the right child is 17, but its contribution limit is only 5 because its contribution set is a subset of the result set of the left child of the root. So, there is no reason to compute all 17 members of its result set, we can pass the result set of left child (which is of size at most 5) to the function computing result set of right child, and only focus on this 5 members.

Lemma 23 *The result set of $Y \cap (X_1 \cup X_2 \cup \ldots \cup X_k)$ can be computed in*

$$O\left(|Y| + \sum_{i=1}^{k} |X_i| \log \frac{|X_i| + |Y|}{|X_i|}\right)$$

time. □

4.2 The Algorithm

We now turn to the general case and describe the algorithm. We generalize the problem and define two types of problems: In the first type of the problem we are simply asked to compute $\text{res}(v)$ for a node v. In the second type, the universe set is restricted and we are given a set U as the restricted universe set. In this type we are asked to compute $\text{res}(v) \cap U$. In these cases, U is a superset of the contribution set of v and so we somehow know no element with a value outside U may be promoted from leaves of the subtree rooted at v. The procedures COMPUTE(v) and COMPUTE (v, U) in Algorithms 4 and 5 are designed to solve these two types of problems.

The intuition behind the universe set U in COMPUTE (v, U) is the following: consider an intersection node v with its children $u_1, \ldots, u_k$. Suppose we have processed the subtree rooted at u_i for some i, and have obtained res(u_i). It makes perfect sense to pass res(u_i) as a universe set to the subtrees rooted at children of v other than u_i so that they only report back elements that are also in the universe set and ignore those that do not appear in the universe set. Figure 4.1 is an example.

As for COMPUTE(v), it turns out that, for some nodes v, the size of the possible result of a node is smaller than any universe set we can possibly provide in advance. In these cases, we do not provide any universe set, as it will not save any computation time. These are nodes for which we compute the results first. We use their result sets as universe sets passed to other nodes.

Algorithm 4: Compute(v)

```
  // precondition: cap($v$) = share($v$).
  begin
      switch type of node $v$ do
1         case Leaf: return res($v$) ;
          case Union:
              foreach child $u_i$ of $v$ do
2                 $X_i \longleftarrow Compute(u_i)$
              end
3             return $X_1 \cup X_2 \cup \cdots \cup X_k$
          endsw
          case Intersection:
              $j \longleftarrow minindex(\mathsf{cap}(u_i))$
4             $X \longleftarrow Compute(u_j)$
              foreach child $u_i$ of $v$ do
                  if $i \neq j$ then
5                     $X \longleftarrow Compute(u_i, X)$
                  end
              end
              return X
          endsw
  end
```

Algorithm 5: Compute(v, U)

```
  // precondition: $|U| \leq \mathsf{share}(v)$.
  begin
      switch type of node $v$ do
1         case Leaf:  return $\mathrm{res}(v) \cap U$ ;
          case Union:
              foreach child $u_i$ of $v$ do
                  if $\mathsf{cap}(u_i) < |U|$ then
2                     $X_i \longleftarrow Compute(u_i)$
                  else
3                     $X_i \longleftarrow Compute(u_i, U)$
                  end
              end
4             return $U \cap (X_1 \cup X_2 \cup \cdots \cup X_k)$
          endsw
          case Intersection:
              $X \longleftarrow U$
              foreach child $u_i$ of $v$ do
5                 $X \longleftarrow Compute(u_i, X)$
              end
              return $X$
          endsw

  end
```

4.3 The Correctness of The Algorithm

In this section, we investigate the correctness of the algorithm and preconditions mentioned for the two procedures. We show that:

- The precondition mentioned for Algorithms 5 and 4 always hold whenever they are called.
- The procedures correctly compute the results they are supposed to compute when they are called.

The fact that the procedures produce the right output is trivial to show by using induction on the height of the tree. So let us validate the preconditions.

In Algorithm 5, we have three recursive calls. The first one is in line 2: since $\mathsf{cap}(u_i) < |U|$ and, by the algorithm's precondition, $|U| \leq \mathsf{share}(v)$, we have $\mathsf{cap}(u_i) < \mathsf{share}(v)$. By the definition of $\mathsf{share}(u_i)$, the latter inequality implies that $\mathsf{share}(u_i) = \mathsf{cap}(u_i)$, which means the precondition of Algorithm 4 holds in the recursive call. The second recursive call occurs in Line 3; we know that $\mathsf{cap}(u_i) \geq |U|$ and since $\mathsf{share}(v) \geq |U|$, we can deduce $\mathsf{share}(u_i) \geq |U|$. Thus the precondition of Algorithm 5 holds in the recursive call. The last recursive call occurs in line 5; since v is an intersection node, $\mathsf{cap}(u_i) \geq \mathsf{cap}(v)$ for each i. By definition, $\mathsf{cap}(v) \geq \mathsf{share}(v)$; hence $\mathsf{cap}(u_i) \geq \mathsf{share}(v)$. Since $\mathsf{share}(u_i) = \min\{\mathsf{cap}(u_i), \mathsf{share}(v)\}$, this implies $\mathsf{share}(u_i) = \mathsf{share}(v)$. By the precondition of Algorithm 5, $\mathsf{share}(v) \geq |U|$, so $\mathsf{share}(u_i) \geq |U|$. As $|U| \geq |X|$, the precondition of Algorithm 5 holds in the recursive call.

Similarly in Algorithm 4, there are three recursive calls. The first one is in line 2; since v is a union node, $\mathsf{cap}(u_i) \leq \mathsf{cap}(v)$, and by precondition of Algorithm 4, $\mathsf{cap}(v) = \mathsf{share}(v)$. Thus, $\mathsf{cap}(u_i) = \mathsf{share}(u_i)$, which implies the precondition of Algorithm 4 holds. The second recursive call occurs in line 4; since v is an intersection node, $\mathsf{cap}(v) = \mathsf{cap}(u_j)$ for $j = \mathit{minindex}(\mathsf{cap}(u_i))$, where the u_i's are the children of v. By the precondition of Algorithm 4, $\mathsf{cap}(v) = \mathsf{share}(v)$. So $\mathsf{share}(u_j) = \min\{\mathsf{cap}(u_j), \mathsf{share}(v)\} = \min\{\mathsf{cap}(u_j), \mathsf{cap}(v)\} = \mathsf{cap}(u_j)$, and hence the precondition of Algorithm 4 holds. The third and last recursive call occurs in line 5. we know that

$$\begin{aligned} \mathsf{share}(u_i) &= \min\{\mathsf{cap}(u_i), \mathsf{share}(v)\} \\ &= \min\{\mathsf{cap}(u_i), \mathsf{cap}(v)\} \\ &= \mathsf{cap}(u_j), \end{aligned}$$

and $|X| \leq \mathsf{cap}(u_j)$, so $|X| \leq \mathsf{share}(u_i)$.

4.4 Running Time

We analyze the running times of the procedures by evaluating the time we spend at each node v of the tree, not taking into account the time we spend in recursive calls. The total running time of the algorithm is the sum of the running times for the individual nodes.

The algorithms are carefully tailored such that when node v is an intersection node, no processing time (other than iterating over children) is spent. So the processing time in an intersection node v is $O(k)$, where k is the number of children of v, which is negligible. In the next sections we discuss the running times for union nodes and leaves.

4.4.1 Processing Time in Union Nodes

The only lines we spend time for union nodes are line 4 in Algorithm 5 and line 3 in Algorithm 4. These two are exactly the special cases we studied in the beginning of this section. We prove the following lemma.

Lemma 24 *Processing a union node v takes time*

$$O\left(\sum_{i=1}^{k} \mathit{share}(u_i) + \log\binom{\mathit{share}(v)}{\mathit{share}(u_1),\ldots,\mathit{share}(u_k)}\right),$$

where $u_1,\ldots,u_k$ are children of v.

Proof In Algorithm 5, the only line that we spend some computing time is line 4. Since $|X_i| \le |U|$, the computation can be done in time

$$O\left(|U| + \sum_{i=1}^{k} |X_i| \log \frac{|X_i| + |U|}{|X_i|}\right),$$

by Lemma 23. As $|U| < \mathsf{share}(u_i)$, by the precondition, and $|X_i| \le |U|$, $|X_i| < \mathsf{share}(u_i)$. Given that $|U| < \mathsf{share}(v)$ and $|X_i| \le \mathsf{share}(u_i)$, the time we spend in Algorithm 5 is in

$$O\left(\mathsf{share}(v) + \sum_{i=1}^{k} \mathsf{share}(u_i) \log \frac{\mathsf{share}(u_i) + \mathsf{share}(v)}{\mathsf{share}(u_i)}\right).$$

Finally, since $\mathsf{share}(v) \leq \sum_{i=1}^{k} \mathsf{share}(u_i)$ and the term $\log \frac{\mathsf{share}(u_i)+\mathsf{share}(v)}{\mathsf{share}(u_i)}$ is not less than one, we can eliminate the term $\mathsf{share}(v)$. Thus, the processing time of line 4 in Algorithm 5 is

$$\begin{aligned} & O\left(\sum_{i=1}^{k} \mathsf{share}(u_i) \log \frac{\mathsf{share}(u_i)+\mathsf{share}(v)}{\mathsf{share}(u_i)}\right) \\ = \; & O\left(\sum_{i=1}^{k} \mathsf{share}(u_i) + \sum_{i=1}^{k} \log \binom{\mathsf{share}(v)}{\mathsf{share}(u_i)}\right) \end{aligned}$$

which is, by Lemma 9,

$$O\left(\sum_{i=1}^{k} \mathsf{share}(u_i) + \log \binom{\mathsf{share}(v)}{\mathsf{share}(u_1), \ldots, \mathsf{share}(u_k)}\right).$$

In Algorithm 4, only line 3 is important. Due to Corollary 22, the result can be computed in $O(s + \log \binom{s}{|X_1|+\ldots+|X_k|})$, where $s = \sum_{i=1}^{k} |X_i|$. By the precondition of this procedure, $\mathsf{share}(v) = \mathsf{cap}(v)$, and for each child u_i of v, $\mathsf{share}(u_i) = \min\{\mathsf{cap}(u_i), \mathsf{share}(v)\} = \min\{\mathsf{cap}(u_i), \mathsf{cap}(v)\} = \mathsf{cap}(u_i)$. Therefore,

$$\mathsf{share}(v) = \mathsf{cap}(v) = \sum_{i=1}^{k} \mathsf{cap}(u_i) = \sum_{i=1}^{k} \mathsf{share}(u_i).$$

Also, $|X_i| \leq \mathsf{cap}(u_i) = \mathsf{share}(u_i)$ for every i. Thus,

$$\begin{aligned} \binom{\sum_{i=1}^{k} |X_i|}{|X_1|, \ldots, |X_k|} & \leq \binom{\sum_{i=1}^{k} \mathsf{share}(u_i)}{\mathsf{share}(u_1), \ldots, \mathsf{share}(u_k)} \\ & = \binom{\mathsf{share}(v)}{\mathsf{share}(u_1), \ldots, \mathsf{share}(u_k)}. \end{aligned}$$

Hence, since $s \leq \sum_{i=1}^{k} \mathsf{share}(u_i)$, the running time is in

$$O\left(\sum_{i=1}^{k} \mathsf{share}(u_i) + \log \binom{\mathsf{share}(v)}{\mathsf{share}(u_1), \ldots, \mathsf{share}(u_k)}\right).$$

□

We make a slight change in the algorithm to save time: in the case when the root of the whole tree is a union node, we take union using the algorithm in Corollary 21 instead of that in Corollary 22 in the root. That is, we do not expand the ranges in the result and we keep it in the cross reference format; then, in the case when v is the root and is a union node, we can get a better result than Lemma 24.

Lemma 25 *If the root is a union node, the processing time in the root is*

$$O\left(\log\left(\frac{\mathsf{share}(root)}{\mathsf{share}(u_1),\ldots,\mathsf{share}(u_k)}\right)\right),$$

where $u_1,\ldots,u_k$ are children of the root. □

Here we claim that the term $\sum_{i=1}^{k}\mathsf{share}(u_i)$ in Lemma 24 is negligible when it is summed over all union nodes. In the sum, share of all the children of union nodes are added together, which means the sum is over all the intersection nodes and leaves.

Now we argue that if S is the set of all intersection nodes of T, we have $\sum_{v\in S}\mathsf{share}(v)\le\sum_{v\in L}\mathsf{share}(v)$ where L is the set of non-shallow leaves. To verify this claim, one can see that for every intersection node v with children $u_1,\ldots,u_k$, $\mathsf{share}(v)\le\frac{1}{k}\sum_i\mathsf{share}(u_i)\le\frac{1}{2}\sum_i\mathsf{share}(u_i)$, and for every union node with children $u_1,\ldots,u_k$, $\mathsf{share}(v)\le\sum_i\mathsf{share}(u_i)$. Therefore, for every intersection node with leaves $l_1,\ldots,l_k$ in its subtree, where there are n_i intersection nodes in the path connecting u to l_i, $\mathsf{share}(v)\le\sum_i\frac{1}{2^{n_i}}\mathsf{share}(l_i)$. So $\sum_{v\in S}\le\sum_{l\in L}(\frac{1}{2}+\frac{1}{4}+\ldots)\mathsf{share}(l)\le\sum_{l\in L}\mathsf{share}(l)$.

Now consider the term $\sum_{v\in\mathsf{leaves}(T)}\mathsf{share}(v)$; by making the modification, the sum of share over all leaves and intersection nodes in Lemma 24 is less than twice the sum over non-shallow leaves:

Lemma 26 *Processing in union nodes and leaves takes time*

$$O\left(t+\textstyle\sum_{v\in L}\mathsf{share}(v)+\displaystyle\sum_{\text{union node } v}\log\left(\tfrac{\mathsf{share}(v)}{\mathsf{share}(u_1),\mathsf{share}(u_2),\ldots,\mathsf{share}(u_k)}\right)\right),$$

where L is the set of non-shallow leaves and t is the time we spend in non-shallow leaves. □

4.4.2 Processing Time in Leaf Nodes

In this part we compute the time consumed for a leaf v in line 1 of Algorithm 4, or line 1 in Algorithm 5.

In line 1 in Algorithm 5, we compute the intersection of $\text{res}(v)$, which is the set associated with v, and U. By the precondition of the procedure, $|U| < \mathsf{share}(v)$. Also by definition, $\mathsf{share}(v) \leq \mathsf{cap}(v) = \mathsf{size}(v)$. Thus, $|U| < \mathsf{size}(v)$. In the second special case, Lemma 23, it is shown how to compute the intersection in time $O(|U| \log \frac{|U|+\mathsf{size}(v)}{|U|})$. Since $|U| < \mathsf{share}(v) \leq \mathsf{size}(v)$, the processing time is in $O\left(\mathsf{share}(v) \log \frac{\mathsf{share}(v)+\mathsf{size}(v)}{\mathsf{share}(v)}\right)$.

In line 1 of Algorithm 4, we simply return $\text{res}(v)$ which, by the algorithm's precondition, has size $\mathsf{share}(v)$. In case v is a shallow leaf by the argument mentioned in Lemma 26, we use a slightly different method to take the union at the root, and therefore we do not spend any time in the shallow leaves (we do spend, however, some time in the root for computing the union, which has been accounted for in Lemma 26).

We conclude the time we spent in a non-shallow leaf is

$$O\left(\mathsf{share}(v) \log(\frac{\mathsf{size}(v)}{\mathsf{share}(v)} + 1)\right) = O\left(\log \binom{\mathsf{size}(v)}{\mathsf{share}(v)} + \mathsf{share}(v)\right)$$

and we spend no time in shallow leaves. So the next lemma is proved.

Lemma 27 *The total time spent on leaves is* $\sum_{v \in \mathit{deep}} \log \binom{\mathit{size}(v)}{\mathit{share}(v)} + \mathit{share}(v)$ *where deep is the set of non-shallow leaves of the tree.* □

Lemmas 26 and 27 show that the total running time matches Equation 3.1. Thus, due to Theorem 17, our algorithm is optimal.

Chapter 5

Algorithm Supporting More Operators

In this chapter, we present a more sophisticated algorithm that accepts trees with more varieties of operators: union, delta, intersection, minus, and complement. We use partially expanded trees as the format for input and output data, which is a generalization of both sorted arrays and B-trees. Without loss of generality, we allow a node in the tree to have the same operator as its parent and we assume that all the internal nodes of T have exactly two children.

5.1 Tools

5.1.1 Basic Operations on Partially Expanded B-trees

We first explain how we can apply the basic operations union, intersection, delta, and minus to two partially expanded B-trees in optimal time. Note that, as long as the algorithm uses edges of a partially expanded B-tree to reach other nodes of the tree for the first time rather than trying to "jump" to new nodes, a partially expanded B-tree essentially behaves analogously to a B-tree and operations can be performed in the same asymptotic cost. The reason is that once the algorithm reaches any non-expanded node for the first time, it can expand the node by creating its children (but not expanding them) in constant time. So, we explain the base algorithms for B-trees.

Lemma 28 ([23]) *Evaluating union or intersection of two B-trees with n and m leaves, respectively, can be performed in time $O(m \log \frac{n+m}{m})$.* $\square$

To apply the minus operation on B-trees, one can find the common members between two subtrees using the intersection operation and then use the following lemma.

Lemma 29 *Given a B-tree T with n nodes and a set of m leaves on it, the cost of deleting those nodes is $O\left(m \log \frac{n+m}{m}\right)$.*

Proof We first observe that the time needed to delete these m leaves is proportional to the number of nodes in the tree that are ancestors of the designated m nodes. This is clear once we consider the deletion algorithm from the B-tree as deleting a child (or a set of children) is done by rearranging the children of the node and its siblings, and if necessary recursively doing the operation on the parent node.

Considering a set M of m leaves of T such that the size of the set $A(M)$ of all ancestors of leaves in M is maximized, we show that $|A(M)| \leq n \log{(n+m)/m}$. For any subtree U of T, define $f(U)$ to be the number of leaves of U that are in M. It can be seen that for any two subtrees T_1 and T_2, if $f(T_1) = 0$ and $f(T_2) > 1$, then the height of T_2 is greater than the height of T_1; otherwise we could increase $|A(M)|$ by replacing one leaf of T_2 in M with the deepest leaf of T_1. Therefore, for each subtree T_v rooted at a node v of depth $d = \lceil \log m \rceil$, $f(T_v) \leq 1$; otherwise since there are at least m nodes at depth d, there is another node u at the same depth such that for the subtree T_u rooted at u, $f(T_u) = 0$, contradicting what we proved. Hence, for each node v at depth d, at most $\log n - \log m$ descendants of v are in $A(M)$, so $A(M)$ has at most $m \log \frac{n}{m}$ members of depth at least d. Moreover, there are less than $2^d \leq 2m$ nodes of depth less than d in total. Thus, M has no more than $O(m + m \log \frac{n}{m}) = O(m \log \frac{n+m}{m})$ members. $\square$

5.1.2 Processing Appearance Lists

In this part we discuss a special case of the problem when for one member e of the sets, we are interested in knowing if e appears in the result set of the whole tree. As the input, in addition to the tree, we are given the list of all leaves of the tree, from left to right in order, that have e in their corresponding sets, which is called *the appearance list of e in T*.

In order to decide on an element based on its appearance list in a tree T, we need to preprocess the tree in advance and prepare some data structures first. The first data structure is used to find the lowest common ancestor of any given two nodes n_1 and n_2 of the tree, denoted by $\text{lca}(n_1, n_2)$.

Lemma 30 (Harel and Tarjan [28]) *After preprocessing a tree T in linear time, it is possible to find the lowest common ancestor of any two nodes n_1 and n_2 of T in constant time.*

The second data structure can be used to find the ancestors of a node u that may filter out members of the result set of u. More precisely, consider an ancestor v of u with two children w_1 and w_2, where u is a descendant of say w_1. Then, we look for the situation where in order for a member of the result set of u to be in the result set of v, it needs to be in the result set of w_2. Next definition formulates this situation.

Definition 11 *An ancestor v of a node u is a* dominant ancestor *of u if $u \neq v$ and*

- *v is an intersection node, or*
- *v is a minus node and u is in the right subtree of v.*

For an ancestor w of a node u, we say u is hidden *to w, if there is a dominant ancestor v of u where w is an ancestor of v and $w \neq v$.*

Our second data structure can be used to determine, for a node n and an ancestor a of n, whether n is hidden to a.

Lemma 31 *By a linear-time preprocessing of the tree one can build a data structure H_T that, for a given node v and an ancestor w of v determines if v is hidden to w in constant time.*

Proof It is sufficient, for any node v, to precompute the depth l_v of its lowest dominant ancestor, if there is any, and also the depth d_v of v. Then, given node v and an ancestor w of v, v is hidden to w if and only if $l_v < d_v$. $\square$

Recall the definition of the contribution set of a node u as the set of values appearing in results sets of u and all of its ancestors all the way to the root. We extend this definition here.

Definition 12 *Given a node u and an ancestor v of u, the* contribution *of u to v is the set of values appearing in result sets of all the nodes in the path connecting u to v, including u and v.*

Algorithm 6: How to decide if e is in result set of lca(n_1, n_2) when n_1 and n_2 are its exclusive contributors.

```
cnt=number of nodes in $\{n_2, n_3\}$ not hidden to lca$(n_1, n_2)$;
switch $opr(lca(n_1, n_2))$ do
    case $\cap$: return true iff cnt=2;
    case $\cup$: return true iff cnt$\neq 0$;
    case $\Delta$: return true iff cnt=1;
    case $-$:
        Suppose $n_1$ is in the right subtree and $n_2$ is in the left subtree of lca$(n_1, n_2)$;
        return true iff $n_1$ is hidden to lca$(n_1, n_2)$, but $n_2$ is not;
    end
endsw
```

Clearly the contribution of a node to the root is the contribution set of that node.

Before explaining how we can determine a value e is in the result, we explain a special case of the problem, where for two nodes n_1 and n_2, we want to find out if e is in the result set of lca(n_1, n_2) given that:

- e is in the result sets of both n_1 and n_2.
- e is not in the contribution of any node u to lca(n_1, n_2) unless u is a descendant or an ancestor of n_1 or n_2.

In other words, e is not in the contribution of any leaf to lca(n_1, n_2) that is not a descendant of n_1 or n_2. In this situation we say n_1 and n_2 are *exclusive contributors of* e. Algorithm 6 shows how we can find if e is in the result set of lca(n_1, n_2) in this situation in constant time using the data structures we created.

Algorithm 7 shows the whole procedure for deciding if an element e is in the result. In this algorithm, we create an empty stack $\mathcal{S}$ and push leaves of $\mathcal{L}$ in $\mathcal{S}$, in order from left to right. Before pushing each leaf l, we repeat the following procedure until less than two nodes remain in the stack or no more changes are made. Suppose n_1, n_2 are the top two nodes in $\mathcal{S}$ in order (n_1 is the top one), and lca$(n_1, n_2) = v$. If lca$(l, n_1) \neq v$, we know there will be no more descendant of v to be processed; so we pop n_1 and n_2 from $\mathcal{S}$, and push v into $\mathcal{S}$ if and only if e will be in the result of v, which can be determined using Algorithm 6. We then push l into $\mathcal{S}$. The running-time is linear in $\mathcal{L}$.

Algorithm 7: How to decide if a value is in the result

```
for $i$=1 ... $n + 1$ do
    NotDoneWithPreviousSubTree = true;
    while $|S| > 2$ AND NotDoneWithPreviousSubTree do
        $n_2$=pop($S$);
        $n_3$=pop($S$);
        NotDoneWithPreviousSubTree = $i > n$ OR lca($l_i, n_3$) $\neq$ lca($n_2, n_3$);
        if NotDoneWithPreviousSubTree then
            Use Algorithm 6 to decide if the value is in result set of lca($n_1, n_2$);
            if Algorithm 6 returns true then
                push($S$, lca($n_1, n_2$))
            end
        else
            push($S, n_2$);
            push($S, n_1$);
        end
    end
    if $i \leq n$ then
        push($S, l_i$);
    end
end
```

Lemma 32 *After a linear-time one-time preprocessing on a set expression tree T, there is an algorithm that given as input an element e and node v and the list $\mathcal{L}$ of all leaves of T_v containing e in their results in order from left to right, it can determine if e is in the result of T_v in time linear to the size of $\mathcal{L}$.* □

5.2 The Algorithm

5.2.1 Overview of the Algorithm

The algorithm in this section is based on the same idea as the algorithm in the previous chapter. In that algorithm, as the precondition in Algorithm 4 states, we call Compute(v) only if $\mathsf{cap}(v) = \mathsf{share}(v)$, otherwise, we need a universe set, computed from other parts of the tree. In other words, the result of the subtree rooted at a node v is calculated independently of the rest of the tree only if $\mathsf{cap}(v) = \mathsf{share}(v)$.

The algorithm we describe here generalizes the same idea: it first specifies some nodes, called "independent nodes" in the tree. A node v is defined to be *independent* if $\mathsf{cap}(v) = \mathsf{share}(v)$; otherwise, v is *dependent*. The result of an independent node is evaluated entirely within the subtree rooted at it, and without using anything outside this subtree.

For an independent node v, the maximal subtree of $T[v]$ that includes v but has no other independent node as an internal node is called the *dependent tree of* v, and is denoted by T_v. Thus, each leaf of T_v is either a leaf of T or an independent node. This way the tree is broken into a number of dependent trees. See Figure 5.1 for an example.

We consider independent nodes and compute their results in a bottom-up fashion. Thus, when computing the result of an independent node v, the results of all leaves of the dependent tree of v have been previously computed and the algorithm has access to the corresponding partially expanded B-trees.

In the following sections we discuss how the result of such a node v can be computed.

5.2.2 Union and Delta Independent Nodes

As the next lemma shows, the dependent tree of an independent union or delta node v consists of only v and two children of v, and so is easy to process.

Lemma 33 *Given a union or delta independent node v, both children u_1 and u_2 of v are independent.*

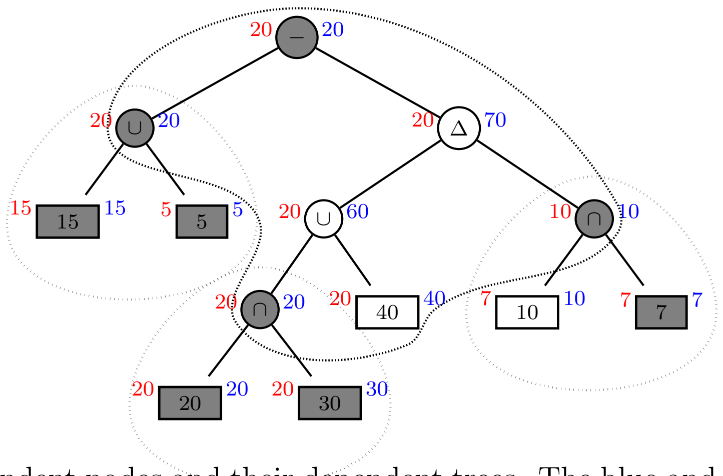


Figure 5.1: Independent nodes and their dependent trees. The blue and red labels are sizes of result sets and contribution sets, respectively

Proof It suffices to show that u_1 and u_2 are independent. If v is a union or delta node, then $\mathsf{cap}(v) = \mathsf{cap}(u_1) + \mathsf{cap}(u_2)$ and so as v is independent, $\mathsf{share}(v) = \mathsf{cap}(v) \geq \mathsf{cap}(u_1)$. Thus, $\mathsf{share}(u_1) = \min\{\mathsf{cap}(u_1), \mathsf{share}(v)\} = \mathsf{cap}(u_1)$, which means u_1 is independent. Independency of u_2 is proved similarly. □

So, the problem for such a node v reduces to computing the union or delta of two partially expanded B-trees, which was discussed in Section 5.1.1.

5.2.3 Intersection and Minus Independent Nodes

The basic idea for finding the result set of an intersection or minus independent node v is that we first select a child of v as the *key leaf* with the following properties: it is an independent node and so a leaf in T_v, and the result set of v is a subset of the result set of its key leaf. Then for each member a of that set, we find what leaves of T_v have a in their result sets and we collect all this information to decide if a is in the result set of v using Algorithm 7.

Definition 13 *For an intersection or minus independent node v we define the* key leaf *of T_v as*

- *the child u of v with the minimum value of $\mathsf{share}(u)$ if u is an intersection node,*

- *the left child of v if u is a minus node.*

We denote the key leaf of T_v by k_v.

It is easy to see that "the key leaf" of a dependent tree is independent:

$$\begin{aligned}\mathsf{share}(u) &= \min(\mathsf{cap}(u), \mathsf{share}(v)) && \text{by definition}\\ &= \min(\mathsf{cap}(u), \mathsf{cap}(v)) && \text{as } v \text{ is independent}\\ &= \min(\mathsf{cap}(u), \mathsf{cap}(u)) && \text{due to the choice of } u\\ &= \mathsf{cap}(u).\end{aligned}$$

Also due to the choice of u, the result set of v is a subset of the result set of k_v and $\mathsf{share}(v) = \mathsf{share}(k_v)$.

Lemma 34 *For an intersection or minus independent node v, k_v is a leaf in T_v such that the result set of v is a subset of the result set of k_v and $\mathsf{share}(v) = \mathsf{share}(k_v)$.* □

Denoting the result set of k_v by K_v, if we define K'_v to be the set of elements in K_v that also appear in the result of at least another leaf of T_v, the result of v is a subset of K'_v if $\mathsf{opr}(v) = \cap$ and is K_v minus a subset of K'_v if $\mathsf{opr}(v) = -$.

For every element e in K'_v, the algorithm makes a list of all leaves of T_v having e in their results in the following way: considering leaves of T_v, except k_v, from left to right, it computes the intersection of the result of each leaf ℓ of T_v with K_v and for every element e in the intersection, the algorithm adds ℓ to the corresponding list of e. Then, for any element e of K_v, the algorithm decides whether e is in the result of v or not, using Algorithm 7.

Then, if $\mathsf{opr}(v) = \cap$, the algorithm builds a fully-expanded B-tree on the elements of K'_v that are in the result of v and returns it. If $\mathsf{opr}(v) = -$, the algorithm removes the elements of K'_v that are not in the result of v from K and returns the resulting partially expanded B-tree.

5.3 Running Time

The following lemma shows the running time of the algorithm.

Lemma 35 *The total processing time to evaluate an expression tree is*

$$\sum_{v \in \mathit{leaves}} \mathit{share}(v) \log \left(\frac{\mathit{cap}(v)}{\mathit{share}(v)} + 1 \right) + \sum_{v \in \mathit{independents}} \mathit{share}(v) \log \left(\frac{\mathit{share}(p(v))}{\mathit{share}(v)} + 1 \right), \quad (5.1)$$

where ***leaves*** *is the set of all leaves but the leftmost leaf of T,* ***independents*** *is the set of independent nodes.*

Proof Considering any independent node v, we prove that the time consumed to evaluate the result of T_v can be written as the sum of the contribution of the leaves of T_v to Equation 5.1. Since every leaf and independent node of the main tree is a leaf of the dependent tree of exactly one independent node, this suffices for the proof.

First suppose v is a union or delta node with children u_1 and u_2, which are by Lemma 33 leaves in T_v. As v is independent, $\mathsf{share}(v) = \mathsf{cap}(v) = \mathsf{share}(u_1) + \mathsf{share}(u_2)$ and for each $i = 1, 2$, $\mathsf{cap}(u_i) = \mathsf{share}(u_i)$. As u_1 and u_2 are independent nodes, the cost of computation for all such nodes u is bounded by $\sum_{u \in \mathsf{independent}} \mathsf{share}(u) \log \left\lceil \frac{\mathsf{share}(p(u))}{\mathsf{share}(u)} + 1 \right\rceil$.

Now consider the case when v is a minus or an intersection node. By Lemma 34, $\mathsf{share}(v) = \mathsf{share}(u_k)$ for u_k the key leaf of T_v. Thus, since every internal node of T_v other than v is dependent, for every leaf u of T_v we have $\mathsf{share}(p(u)) = \mathsf{share}(v)$. Therefore, the cost for every non-key independent leaf of T_v is the cost of taking an intersection between the result u, which is of size at most $\mathsf{cap}(u) = \mathsf{share}(u)$ and a B-tree of size at most $\mathsf{share}(u_k) = \mathsf{share}(v) = \mathsf{share}(p(u))$. For non-key non-independent leaves of T_v also, the cost will be the cost of an intersection between a B-tree of size $\mathsf{cap}(u)$ and another one of size $\mathsf{share}(u_k) = \mathsf{share}(p(u)) = \mathsf{share}(u)$. So in all cases the time we spend for T_v equals the sum of the contribution of the leaves of T_v to Equation 5.1. Hence, the lemma is true. □

We proved our running time matches Equation 3.1 and so for signatures where Conjecture 1 holds, the running time of the algorithm is worst-case optimal.

Chapter 6

An Adaptive Approach

In Chapters 3 and 4, we showed matching worst-case lower and upper bounds for the worst-case. In this chapter, we consider a more refined framework in which the instances of the problem are grouped into "small" difficulty classes. We present a comparison-based algorithm for the problem, and for every difficulty class, we give a matching lower bound to demonstrate the algorithm is optimal within every class. The running time of the algorithm for any instance is never (asymptotically) more than that of the previous worst-case optimal algorithm.

6.1 Preliminaries

6.1.1 Background

As explained in Chapter 1, Demaine et al. [23] presented adaptive algorithms for computing the union, intersection, and difference of an arbitrary number of sets. As the number of sets in the expression exceeds two, specially in the case of intersections, the definition of "difficulty" goes beyond a simple number of alternations between sets, as one does not really need to obtain a full ordering of all members in order to compute the solution. For example, consider the example in Figure 6.1). As shown in the example, a small amount of interleaved-ness in any of two sets results in an easy input, no matter how other sets are interleaved. To define the measure of difficulty of an input instance, they used the number of bits required to encode a *proof structure* of a given input instance and developed an algorithm sensitive to this measure. Here, a proof structure is essentially a set

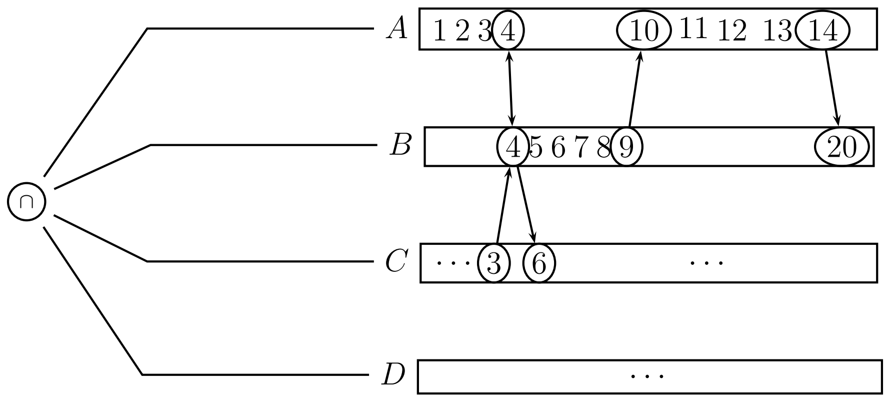


Figure 6.1: As A and B are not too interleaved, with just three comparisons we can show that the only member of $A \cap B$ is 4. So we only need two more comparisons to show that 4 does not belong to the set C and we may then ignore all other members of C and D.

of comparisons that an algorithm may make to derive (or just verify the correctness of) the result. An obvious lower bound for the number of comparisons required is the information-theoretic bound for encoding a proof structure. They show that for the evaluation of the union of a number of sets, the lower bound obtained in this fashion is indeed tight by giving an algorithm matching the bound. However, for the evaluation of the intersection of a number of sets, they give a stronger lower bound using an adversary argument. They also give an algorithm matching this bound.

In earlier parts of the thesis, we proposed an optimal algorithm for evaluating general set expressions that can have both union and intersection operations in any combination. The complexity was analyzed as a function of only the expression and sizes of the input sets, and as a result, the worst-case complexity is assumed, where input sets are maximally interleaved. Our contribution in this chapter is to give an *adaptive* algorithm to calculate such expressions. As discussed by Mirzazadeh [36], there is no unique way of defining difficulty of an instance for the general form of the union-intersection problem: we are given a non-symmetric expression tree and depending upon the weights assigned to comparisons at different parts of the tree, we may end up with an infinite number of different definitions for difficulty of an instance. As such, we think for such complex set of inputs, using a single number as difficulty of an input is not expressive enough; we need a finer expression.

In this chapter, we develop a novel framework for adaptive algorithms. Instead of representing the hardness of an instance by a single number, we partition the input space into classes of "similar structure" where inputs within each class are considered as a consequence to be of the same level of difficulty. Then, we look for algorithms that are worst-case

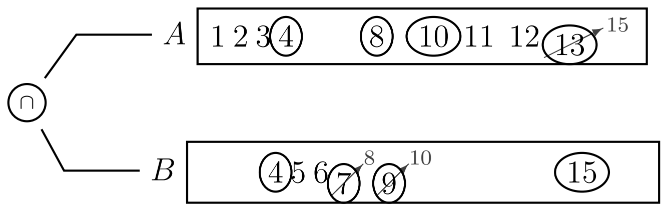


Figure 6.2: The perturbation defining critical members (which are indicated by circles). A "small change" in the member 4 in set B will exclude it from being in the result set, while a "small change" in member 7 or 9 will include them in the result set.

optimal for inputs within each class. For the case of our problem, structural similarity between inputs means a shared expression tree, same set sizes and at the same time, same level of "interleaved-ness" between input sets in the tree.

6.1.2 Our Results

In order to give a high-level overview of the results of this chapter, we first need to define what we mean by difficulty of an instance. Our goal is to define a function ϕ which assigns a number to each leaf measuring the "difficulty" of processing said leaf based on how it is interleaved with other sets in the tree. If the input sets are fully interleaved, then the function size (defining the size of each set) alone is indeed a good measure of difficulty. On the other hand, there are instances where the difficulty is strictly less than size for some of the leaves. The difficulty measure we define aims to capture the number of "critical" members in each of the input sets.

The intuition behind the definition of critical is as follows. Suppose we increase the value of an element in a set slightly[1]. If this change results in a new inclusion or exclusion in the result set, we call this element critical. A collective set of such changes is called a "perturbation". The difficulty is the size of the perturbation that results in the maximum number of critical elements being in the result set (see Figure 6.2).

Let us compare the complexity of the adaptive algorithm we propose here with that of the worst-case optimal algorithm. Consider a leaf function f and suppose for each leaf l, only $f(l)$ members of leaf l are marked as "important" and the rest can be ignored. Then, for each node v, $\mathsf{share}_f(v)$ represents the maximum number of members in the subtree

[1] To be more precise, as we will explain later, here by "slight increasing the value of an element" we mean increasing the value of the element such that the new value does not reach or exceed the value of the next biggest element in that set.

rooted at v that may advance all the way to the result of the whole expression if we ignore all "non-important" members. We define A_f and B_f as follows:

$$A_f = \sum_{\ell \in \mathsf{leaves}} \log \binom{\mathsf{size}(\ell)}{\mathsf{share}_f(\ell)}, \text{ and } B_f = \sum_{u \in \text{union}} \sum_{v \in \text{children}(u)} \log \binom{\mathsf{share}_f(u)}{\mathsf{share}_f(v)}.$$

For our worst-case optimal algorithms, the analysis is performed under the pessimistic assumption that the input is fully interleaved and gives an optimal algorithm using (the maximal) size as the function to analyze the complexity. Namely, the optimal algorithm has asymptotic complexity $\Theta(A_{\mathsf{size}} + B_{\mathsf{size}})$. In this part, we define a finer adaptive measure ϕ as a parameter to both the complexity of the algorithm and the definition of the complexity class containing the input. We then present an algorithm with running time of roughly $\Theta(A_\phi + B_\phi)$ (which means the function f above is ϕ) and establish a matching lower bound in *each* class as defined by ϕ. This is in contrast to previous adaptive algorithms [19, 23, 6, 7, 8], which use a single value as the measure of difficulty of a given instance (in addition to set sizes).

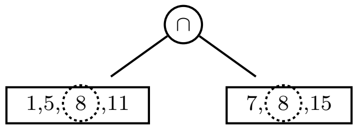
6.1.3 Difficulty Classes

As mentioned, in this chapter, we give an algorithm that is worst-case optimal within individual difficulty classes that partition the set of instances according to their level of interleaving difficulty. Roughly speaking, instances in each difficulty class differ from each other by small "perturbations" of the elements.

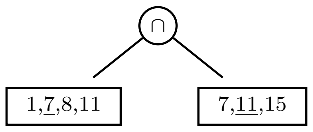
Definition 14 *An input J is a* perturbation *of an input I if they have the same signature and for every element e, $\mathsf{val}_I(e) \le \mathsf{val}_J(e) < \mathsf{val}_I(\mathsf{next}(e))$. J is a* promotion-preserving perturbation of I *if for every element e promoted in I, $\mathsf{val}_I(e) = \mathsf{val}_J(e)$.*

Figure 6.3 shows an example. The intuition is that the existence of an input perturbation causing many new elements to be promoted is evidence of a higher level of interleaving of the input, and thus it requires more time to solve. Loosely speaking, we will look for a "maximal" perturbation for the given input to capture its difficulty:

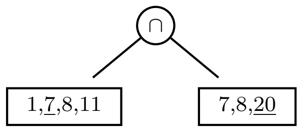
Definition 15 *An input I is* aligned *if there is no promotion-preserving perturbation J of I and element e such that e is promoted in J but not in I.*



Input I: Elements that are promoted are specified with circles



Input J: It is a perturbation of I because $5 \leq \underline{7} < 8$ and $8 \leq \underline{11} < 15$.



Input K: It is a perturbation of I because $5 \leq \underline{7} < 8$ and $15 \leq \underline{20}$.

Figure 6.3: Perturbation examples. Elements of perturbations whose value has changed are underlined. Consider the element of the right leaf whose value is 8 in I. This element was already promoted in I, but it has changed its value to 11 in J. So, J is **not** a promotion-preserving perturbation of I. But, K is a promotion-preserving perturbation of I because all elements promoted in I retain their value in K.

As an example, the input I in Figure 6.3 is a non-aligned input, as it has a promotion-preserving perturbation (the input K on the right) in which more elements are promoted. The input J in the figure is aligned because no promotion-preserving perturbation of J causes any additional element to be promoted. The intuition behind aligned inputs is that, if an input I is aligned, knowing the values of elements promoted in I is enough to conclude that no other element is promoted in I. That's roughly because by definition of aligned inputs, there is no way to perturb values of other elements and make any of them being promoted.

The idea is that, given an input I, we consider a perturbation of I that converts I to an aligned input. The number of elements promoted from each leaf l in this input is $\phi(l)$ and so defines the cluster of inputs containing I. We will then design an algorithm that is worst-case optimal within each and every difficulty class. One problem here is that there could be several perturbations of I that result in aligned inputs. As an example, in Figure 6.4, both J and K are aligned perturbations of I. So which one to use?

We identify a unique perturbation of the input as the "representative perturbation". The representative perturbation will be an aligned input, and as it takes into account all corner cases, its precise definition is quite technical and is presented separately in

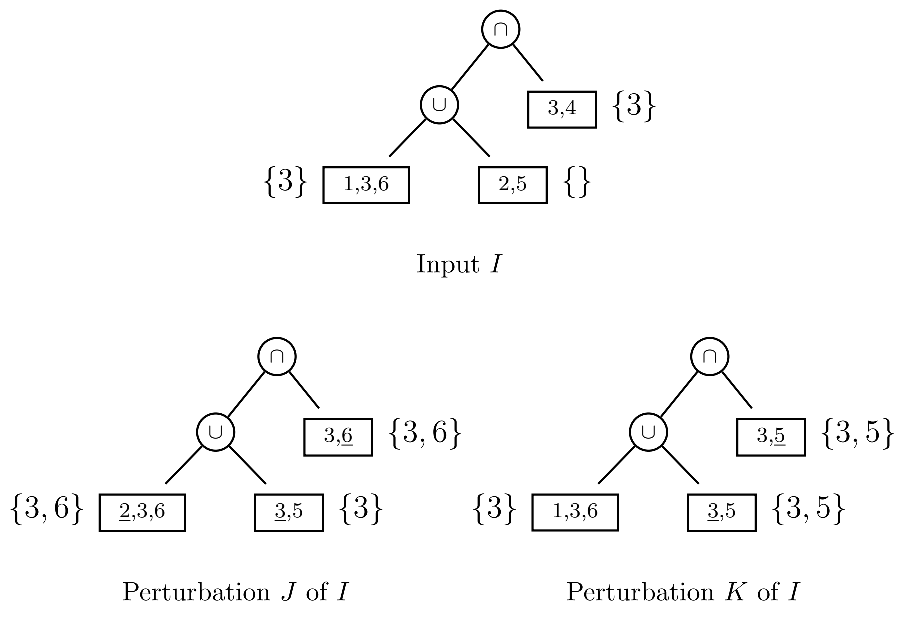


Figure 6.4: The set next to each leaf is the contribution set of that leaf. Contribution sets show that $\mathsf{diff}(I, J) = \mathsf{diff}(I, K) = 3$, $\mathsf{diff}(J, K) = 5$, and $K \prec J \prec I$.

Section 6.1.4.

Roughly speaking, the representative perturbation is the "lexicographically first" perturbation of the input that satisfies certain conditions including being an aligned input. More formally we choose the first "qualified" perturbation according to $\preceq$ which is defined as follows:

Definition 16 *We define the partial order $\preceq$ between instances I_1 and I_2 with the same signature as follows. Consider the smallest value a, denoted by $\mathsf{diff}(I_1, I_2)$, such that the sets S_1 and S_2 of nodes from which a is promoted in instances I_1 and I_2, respectively, are not the same. If such a value does not exist, we define $\mathsf{diff}(I_1, I_2) = \infty$ and we write $I_1 \equiv I_2$; otherwise, if $S_2 \subset S_1$, we define $I_1 \prec I_2$. We write $I_1 \preceq I_2$ if $I_1 \equiv I_2$ or $I_1 \prec I_2$.*

Figure 6.4 shows an example of the concepts defined in Definition 16.

Definition 17 *A perturbation J of an instance I is* maximal at a point *b if for any perturbation K of I with $\mathsf{diff}(J, K) = b$, $J \preceq K$.*

As an example, for perturbations J and K of instance I in Figure 6.4, J is clearly maximal at 3 (because 3 is already in contribution sets of all leaves in J), but it is not maximal at 5 because for K we have $\mathsf{diff}(J, K) = 5$ and $K \prec J$.

The following observations will be useful throughout the rest of the chapter.

Observation 36 *Consider inputs I, J_1, and J_2 of the same signature where $\mathsf{diff}(I, J_1) < \mathsf{diff}(J_1, J_2)$. Then:*

- $\mathsf{diff}(I, J_1) = \mathsf{diff}(I, J_2)$,
- $I \prec J_1$ *if and only if* $I \prec J_2$, *and*
- $J_1 \prec I$ *if and only if* $J_2 \prec I$.

Observation 37 *Consider perturbations J and K of an input I where J is maximal at a point b. Then one of the following is true:*

- *Every element promoted with value b in K is also promoted with value b in J.*
- *There is an element promoted with value b in K that is promoted with a value less than b in J.*

For a given input I and a node v, we use $\phi(v)$ to denote the size of the contribution set of v in the representative perturbation of I. We use the function ϕ to define input difficulty classes.

Definition 18 *For a signature S and a function f,* the input class of f *contains all inputs with signature S in which $\phi(\ell) = f(\ell)$ for all leaves ℓ.*

As mentioned, we prove a tight bound of roughly $\Theta(A_\phi + B_\phi)$ for the worst-case complexity of any algorithm within each difficulty class. We now present the precise definition share_f used in the formulation of A_f and B_f.

6.1.4 Representative Perturbation[2]

As explained, an input may have several perturbations making it an aligned input, thus we need to uniquely identify one as the representative perturbation for defining the input class. We choose the lexicographically first aligned perturbation satisfying certain extra conditions.

Let us give some intuition why we may need these "extra conditions". Suppose we select a first-rank perturbation according to the order of Definition 16 among aligned promotion-preserving perturbations (with no extra conditions) as the representative perturbation. The problem is that this way we end up with a non-robust definition for the concept of representative perturbation in the sense that, given an aligned input, a small change in the value of an input element may result in an input with a completely different representative perturbation, and so belonging to a completely different difficulty class.[3] Figure 6.5 shows an example.

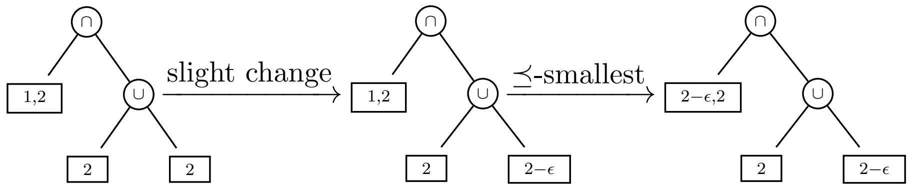


Figure 6.5: Effect of small change in the value of an element on $\preceq$-smallest aligned promotion-preserving perturbation: In the left tree (which is already aligned and thus the only, and so the $\preceq$-smallest, promotion-preserving perturbation of itself), if we make a small change in the value of an element (and obtain the input in the middle), its $\preceq$-smallest promotion-preserving perturbation will change to the input on the right, which belongs to a completely different class (as different numbers of elements are promoted from its leaves).

To address this issue, we introduce the following additional condition for selecting the "representative perturbation": when adjusting values of elements of input I to obtain the representative perturbation of I, if for an element e which is not promoted in I, values can be adjusted so that e is promoted with a value v already in the result set of the root in

[2]The reader may skip Sections 6.1.4 and 6.3.3 if only interested in an overview of the algorithm.

[3] The reader may still wonder why non-robustness, in the way explained here, is a problem. Recall the way we obtained our lower bound in Section 3.4. In order to obtain an input in the class realizing the lower bound we want to prove, the adversary needs to make 'small adjustments" to values of elements during its interaction with the algorithm, and still resulting input should remain in the class.

I, we should not make adjustments so that e is promoted with a value u "slightly smaller than v". In the next definition, conditions 2, 3, and 4 formulate such situations; specifically conditions 2 and 3 specify precisely when a value u is "slightly smaller than v".

Definition 19 *Consider a perturbation J of an input I. For a perturbation K of I, we say K* compresses *J if for $u = \mathit{diff}(J, K)$ and some value v, $v > u$, the following conditions hold.*

1. *u is in the result set of J and v is in the result set of I.*
2. *neither the result set of I nor the result set of K has a value in the range $[u, v)$.*
3. *For any element e promoted with a value in the range $[u, v)$ in J, $\mathit{val}_I(\mathit{next}(\mathit{next}(e))) > v$.*
4. *Every leaf containing an element promoted with a value in the range $[u, v]$ in J contains an element that is promoted with value v in K.*
5. *K is tight at v (recall the definition from Section 2.1.1).*

So when J is compressed by a perturbation K, it means some elements are being promoted in J with value $u = \mathsf{diff}(J, K)$ (which is not in the result set of I), while they could be promoted with a value "slightly bigger than u", namely v in Definition 19, which is already in the result set of I. Figure 6.5 shows an example of Definition 19: it suffices to label inputs as K, I, and J in order and set $v = 2$ and $u = 2 - \epsilon$.

Observation 38 *Consider perturbations P_1, P_2, and Q of an input I and suppose there is a value a in the result set of I such that $\mathit{diff}(Q, P_1) < a < \mathit{diff}(P_1, P_2)$. Then:*

- *P_1 compresses Q if and only if P_2 compresses Q.*
- *Q compresses P_1 if and only if Q compresses P_2.*

The main restriction that we will add to the definition of representative perturbation is not to be compressed by any aligned input.

Definition 20 *A perturbation J of an instance I is* compact *if there is no aligned perturbation of I that compresses J.*

A perturbation J of an input I is *minimal* if every element has the same value in I and in J unless it is promoted in J (in other words, no element has changed value without side effects). We define a "representative perturbation" to be aligned, minimal (to keep it unique), maximal at any point (see Definition 16), compact, and prior to any other perturbation (according to the order $\preceq$) except when compressing a perturbation:

Definition 21 *A representative perturbation of an instance I is a minimal aligned compact perturbation P satisfying the following conditions.*

- *For every value a in the result set of I, P is maximal at a.*
- *For all perturbations P' of I, either $P \preceq P'$, or there is another perturbation P'' with $P'' \preceq P'$ such that P compresses P'' but P' does not compress P''.*

The existence of a representative perturbation follows from Lemma 49 that we will prove on page 92. The next lemma proves its uniqueness.

Lemma 39 *The representative perturbation of an input is unique.*

Proof We assume to the contrary that there are two representative perturbations P_1 and P_2 of an input I. As P_1 and P_2 are minimal and distinct, $P_1 \equiv P_2$ cannot be true, so $\mathsf{diff}(P_1, P_2) \neq \infty$. Let a be the biggest value in the result set of I that is smaller than $\mathsf{diff}(P_1, P_2)$ ($-\infty$ if no such value exists) and let b be the smallest value in the result set of I that is not smaller than $\mathsf{diff}(P_1, P_2)$ (∞ if no such value exists). By Definition 21, if $b \neq \infty$, then P_1 and P_2 are maximal at b. Therefore, if $\mathsf{diff}(P_1, P_2) = b$, by Definition 17, $P_1 \preceq P_2$ and $P_2 \preceq P_1$, which implies $P_1 \equiv P_2$, a contradiction. So $\mathsf{diff}(P_1, P_2) \neq b$.

Consider the set $\mathcal{P}$ of perturbations P such that $P_1 \not\preceq P$, $P_2 \not\preceq P$, $a < \mathsf{diff}(P, P_1) \leq b$, $a < \mathsf{diff}(P, P_2) \leq b$, and at least one of P_1 or $P2$ compresses P. We first show that $\mathcal{P}$ is not empty. Since $P_1 \not\equiv P_2$, at least one of $P_1 \not\preceq P_2$ or $P_2 \not\preceq P_1$ is true; suppose the former is true. Then, according to Definition 21 there is a perturbation P, $P \preceq P_2$, where P_1 compresses P while P_2 does not. We prove that $P \in \mathcal{P}$. If $P_1 \preceq P$, as $P \preceq P_2$, then $P_1 \preceq P_2$ which is a contradiction; thus $P_1 \not\preceq P$. Also the condition $P_2 \not\preceq P$ is true because otherwise, $P \equiv P_2$ (because $P \preceq P_2$), which means P_1 compresses P_2 and so P_2 is not compact, contradicting the choice of P_2. Since only one of P_1 and P_2 compresses P, by Observation 38 $a \leq \mathsf{diff}(P_1, P)$. As P_1 compresses P, $\mathsf{diff}(P_1, P)$ is not in the result set of I and thus, $\mathsf{diff}(P_1, P) \neq a$; so, $a < \mathsf{diff}(P_1, P)$. As a result, $a < \mathsf{diff}(P_2, P)$ because otherwise $\mathsf{diff}(P_2, P) \leq a < \mathsf{diff}(P_1, P)$ and so by Observation 36 $\mathsf{diff}(P_2, P) = \mathsf{diff}(P_1, P_2) \leq a$ which contradicts the choice of a. Moreover, $\mathsf{diff}(P_1, P_2) < b$;

so if $\mathsf{diff}(P, P_1) > b$, then, since $P \preceq P_2$, by Observation 36 $P_1 \preceq P_2$, which contradicts our assumption; therefore, $\mathsf{diff}(P, P_1) \leq b$. Also $\mathsf{diff}(P, P_2) \leq b$ because otherwise as $\mathsf{diff}(P_1, P) \leq b$ and P_1 compresses P, by Observation 38 P_1 also compresses P_2 which means P_2 is not compact and so contradicts Definition 21. Hence $P \in \mathcal{P}$.

Considering a first-rank member P of $\mathcal{P}$ (according to the order $\preceq$), we prove that both P_1 and P_2 compress P. As $P \in \mathcal{P}$, one of P_1 or P_2, say P_1, compresses P. Because $P_2 \not\preceq P$, by Definition 21 there is a perturbation Q where $Q \preceq P$ and P_2 compresses Q while P does not; thus by Observation 38 $a < \mathsf{diff}(Q, P_2)$ and so, $a < \mathsf{diff}(Q, P_1)$ (because otherwise by Observation 36 $\mathsf{diff}(Q, P_1) = \mathsf{diff}(P_1, P_2) \leq a$ which is not true). We show that $Q \in \mathcal{P}$. The condition $P_1 \not\preceq Q$ holds because otherwise $P_1 \preceq Q \preceq P$ while we know $P_1 \not\preceq P$ (since $P \in \mathcal{P}$). Similarly, $P_2 \not\preceq Q$. In addition, $\mathsf{diff}(P, P_1) \leq b$; so if $\mathsf{diff}(Q, P_1) > b$, then by Observation 36 since $Q \preceq P$, $P_1 \preceq P$, which is false; therefore, $\mathsf{diff}(Q, P_1) \leq b$ and similarly $\mathsf{diff}(Q, P_2) \leq b$. Hence, $Q \in \mathcal{P}$ and so, as $Q \preceq P$, due to the choice of P, $Q \equiv P$. This means P_2 compresses P.

We showed that P_1 and P_2 compress P while $\mathsf{diff}(P, P_1)$ and $\mathsf{diff}(P, P_2)$ both are in the range $(a, b]$. First suppose $\mathsf{diff}(P, P_1) = \mathsf{diff}(P, P_2)$. Then, $\mathsf{diff}(P_1, P_2) \geq \mathsf{diff}(P, P_1)$ because otherwise by Observation 36 $\mathsf{diff}(P_1, P_2) = \mathsf{diff}(P, P_2) = \mathsf{diff}(P, P_1)$. On the other hand, as P_1 and P_2 compress P, neither have any value in their result set in the range $[\mathsf{diff}(P, P_1), b)$; so $\mathsf{diff}(P_1, P_2) \geq b$. Therefore $\mathsf{diff}(P_1, P_2) = b$, which as we proved is impossible.

Now consider the case where $\mathsf{diff}(P, P_1) < \mathsf{diff}(P, P_2)$ (the case $\mathsf{diff}(P, P_2) < \mathsf{diff}(P, P_1)$ will be similar). By Observation 36, $\mathsf{diff}(P_2, P_1) = \mathsf{diff}(P, P_1)$. P_1 compresses P which means all four conditions of Definition 19 hold for $K = P_1$, $J = P$, $u = \mathsf{diff}(P, P_1)$, and $v = b$. We use this fact to prove these four conditions still hold if we change J to P_2, which means P_1 compresses P_2. By Observation 37, all elements promoted with value b in P_2 are also promoted with value b in P_1 because none of them could have been promoted in P_1 with a value less than $\mathsf{diff}(P_2, P_1)$ (by the Definition of $\mathsf{diff}(P_1, P_2)$), nor with a value in the range $[\mathsf{diff}(P_2, P_1), b)$ (as P_1 compresses P), and by definition, P_1 is maximal at b. Also, every element promoted with a value in the range $[\mathsf{diff}(P_1, P_2), \mathsf{diff}(P, P_2))$ in P_2 is promoted with the same value in P and no element is promoted with a value in the range $[\mathsf{diff}(P, P_2), b]$ in P_2. Thus, all four conditions of Definition 19 still hold, so P_1 compresses P_2. This means P_2 is not compact: a contradiction with P_2 being a representative perturbation. □ □

The next lemma proves an intuitive property about values of elements in the representative perturbation that are different from values of those elements in the original input, which will be handy.

Lemma 40 *Suppose J is a representative perturbation for an input I, and e is an element* *promoted in J. Let S be the set of elements promoted in J with the same J-value as e, then $\mathsf{val}_J(e) = \max_{f \in S} \mathsf{val}_I(f)$.*

Proof Let us assume by way of contradiction that the claim is not true and e is the I-biggest element for which the lemma does not hold. Then every element in the set S has an I-value less than its J-value. We define c to be the biggest value in the range $(\mathsf{val}_I(e), \mathsf{val}_J(e))$ in the value set of the root in I if such a value exists; otherwise we define $c = \mathsf{val}_I(e)$. In either case, $\mathsf{val}_I(e) \leq c < \mathsf{val}_J(e)$ and there is no value in the range $(c, \mathsf{val}_J(e))$ in the result set of the root in I.

We obtain[4] a perturbation K of I from J by changing the value of any element in S to c. Then, clearly $K \prec J$ and $\mathsf{diff}(K, J) = c$. So, by Definition 21, c is not in the value set of the root in I. Moreover, by Definition 21, there is a perturbation L of I where $L \preceq K$ and J compresses L while K does not. As $L \preceq K \prec J$, by Observation 36, $\mathsf{diff}(L, J) \leq \mathsf{diff}(K, J) = c$. For b the biggest value in the result set of I smaller than c ($-\infty$ if no such value exists), by Observation 38, $b < \mathsf{diff}(L, J)$ because only one of J and K compresses L. This means there is no value in the value set of the root in I in the range $[\mathsf{diff}(L, J), \mathsf{val}_J(e))$. So, as e is promoted in J and J compresses L, $\mathsf{val}_J(e)$ is in the value set of the root in I, which means an element f of I-value $\mathsf{val}_J(e)$ is promoted in I. Since J is a promotion-preserving perturbations of I, f is promoted with value $\mathsf{val}_J(f) = \mathsf{val}_I(f)$ in J and so $f \in S$, contradicting the fact that the I-value of every element in S is less than its J-value. □

6.2 Lower Bound

Recall the definition of ϕ and that we partition input instances into classes according to the value of ϕ at the leaves. Let $\mathcal{C}$ be an arbitrary class. Our goal is to prove that for every comparison-based algorithm $\mathcal{A}$, there are input instances $I_1, I_2 \in \mathcal{C}$ such that $\mathcal{A}$ needs roughly A_ϕ comparisons to find a solution for I_2 (the first lower bound) and B_ϕ comparisons to find a solution for I_1 (the second lower bound); this implies that our algorithm is worst-case optimal in every class $\mathcal{C}$ of inputs.

Proofs in this section are mostly adapted from Chapter 3. In our arguments here, we manipulate inputs to obtain inputs within the same class for which the algorithm needs

[4]In this work, when we say we obtain a perturbation K of I from J, it means that K and J are both perturbations of I, and each element, except from those mentioned explicitly, has the same J-value and K-value.

a certain number of comparisons. The next definition describes one situation where two inputs are in the same class.

Definition 22 *For inputs I and J with the same signature, J is a* reduction of I *if there is a sequence S of elements that is a solution for both I and J, and for any two elements e_1 and e_2:*

- *If e_1 is I-smaller than e_2, e_1 is also J-smaller than e_2.*
- *If e_1 is in S and e_2 is I-equal to e_1, e_1 is not J-smaller than e_2.*

Lemma 41 *For any reduction J of a tight aligned input I, I and J are in the same class.*

Proof Consider a solution $s_1, \ldots, s_k$ satisfying the conditions in Definition 22, and define s_0 and s_{k+1} to be some elements with values $-\infty$ and ∞ respectively (as defined in Section 2.1.2). For any element e promoted with s_i,[5] for some i, in I, $\mathsf{val}_I(e) = \mathsf{val}_I(s_i) < \mathsf{val}_I(\mathsf{next}(e))$ and so, by Definition 22 $\mathsf{val}_J(e) \leq \mathsf{val}_J(s_i) < \mathsf{val}_J(\mathsf{next}(e))$. Thus, we can define the perturbation M of J as one in which the value of every element promoted in I with s_i, for some i, has changed to $\mathsf{val}_J(s_i)$. Then, $s_1, \ldots, s_k$ is also a solution for M and, moreover, for every element e and value i, $1 \leq i \leq k$, e is promoted in I with s_i if and only if it is promoted in M with s_i Thus, M and I are in the same class. We show that M is the representative perturbation of J and so the lemma is proved. In order to that, we first show that M is aligned, and then assuming it is not the representative perturbation of J, we show there is a contradiction.

Claim 1 *The perturbation M is aligned.*

Proof If not, consider the representative perturbation N of M. Then $M \not\preceq N$ because otherwise $M \equiv N$ and so as N is aligned, M is also aligned. We define $a = \mathsf{diff}(N, M)$ and S as the set of elements promoted with value a in N. Consider the M-biggest member f of S; if there are multiple choices for f, then we choose the J-biggest one among them; in case there still are ties, if s_i is one of choices, for some i, we select s_i; otherwise we make an arbitrary choice. Then by Lemma 40 $\mathsf{val}_N(f) = \mathsf{val}_M(f) = a$.

We show now that f is also a J-biggest element of S and its J-value is a. First suppose a is not in the result set of J and so is not in the result set of M either. Then, for every element e of S, the J-value of e equals its M-value because otherwise e is promoted in M

[5]Recall the definition of two elements being promoted together from Page 17.

and so has not changed its value in N, that is, it is promoted in M with the value a, which is a contradiction. Therefore, the claim is true. Next suppose a is in the result set of J and hence an element $s_i \in S$, for some i, is promoted in J with value a. As no element in S may have an M-value or J-value of more than $a = \mathsf{val}_J(s_i) = \mathsf{val}_M(s_i)$ and $s_i \in S$, f has to be s_i. Thus, again the claim is true.

Now, we define i as an integer, $1 \le i \le k+1$, such that $\mathsf{val}_I(s_{i-1}) < \mathsf{val}_I(f) \le \mathsf{val}_I(s_i)$, and we consider two cases. The first case is when $i \le k$ and there is at least one leaf not containing any element promoted with value s_i in I, but containing an element in S; let L be the set of all such leaves. As $\mathsf{val}_I(f) \le \mathsf{val}_I(s_i)$, for every element e in S, $\mathsf{val}_J(e) \le \mathsf{val}_J(f) \le \mathsf{val}_J(s_i)$ and so $\mathsf{val}_I(e) \le \mathsf{val}_I(s_i)$. Also, for any such element e, neither e nor any element after e in the same leaf as e is promoted with s_j in I, for $j < i$, because otherwise that element would be promoted with s_j in M and in N and so e could not be promoted with value a (which is bigger than $\mathsf{val}_J(s_j) = \mathsf{val}_N(s_j)$) in N. So, for any $e \in S$, neither e nor any element after e in the same leaf as e is promoted with a value less than s_i in I. Let's use g_e to denote the I-biggest element of the leaf containing e that has an I-value smaller than or equal to the I-value of s_i, for $e \in S$ (note that we proved $\mathsf{val}_I(e) \le \mathsf{val}_I(s_i)$ for $e \in S$). As mentioned, g_e is not promoted in I with a value less than s_i. Thus, if we create a new perturbation I' of I from I by changing the value of g_e to $\mathsf{val}_I(s_i)$, for all $e \in S$, we get a promotion-preserving perturbation of I in which every leaf in L will have an element promoted with s_i, a contradiction with I being aligned.

Next, we consider the other case where either $i = k+1$, or every leaf containing an element in S contains an element promoted with s_i in I and so in M. In case $i < k+1$, $\mathsf{val}_M(f) \ne \mathsf{val}_M(s_i)$ because we assumed there is a leaf containing an element in S but not containing any element promoted with f in M. Also, since $\mathsf{val}_I(f) \le \mathsf{val}_I(s_i)$, $\mathsf{val}_M(f) = a = \mathsf{val}_J(f) \le \mathsf{val}_J(s_i) = \mathsf{val}_M(s_i)$. Therefore, in any case $\mathsf{val}_M(f) < \mathsf{val}_J(s_i)$ and thus, f is not promoted in M nor in I. For every element e in S, $\mathsf{val}_M(e) \le a = \mathsf{val}_J(f) < \mathsf{val}_M(\mathsf{next}(e))$. If $\mathsf{val}_M(\mathsf{next}(e)) \ne \mathsf{val}_J(\mathsf{next}(e))$, $\mathsf{next}(e)$ is promoted in M and in I with s_j, for some $j \ge i$, and so $\mathsf{val}_I(\mathsf{next}(e)) \ge \mathsf{val}_I(s_i) \ge \mathsf{val}_I(f)$. Otherwise, i.e. when $\mathsf{val}_J(\mathsf{next}(e)) = \mathsf{val}_M(\mathsf{next}(e)) = \mathsf{val}_N(\mathsf{next}(e)) = a > \mathsf{val}_J(f)$, $\mathsf{val}_I(f) \le \mathsf{val}_I(\mathsf{next}(e))$; so in any case $\mathsf{val}_I(f) \le \mathsf{val}_I(\mathsf{next}(e))$. Also, $\mathsf{val}_J(e) \le \mathsf{val}_M(e) \le \mathsf{val}_M(f) = \mathsf{val}_J(f)$ and so $\mathsf{val}_I(e) \le \mathsf{val}_I(f) \le \mathsf{val}_I(\mathsf{next}(e))$. Furthermore, if e is promoted in I with some s_j, it is promoted in M and in N with s_j and hence $s_i = f$ and $j = i$ (because $\mathsf{val}_N(f) = \mathsf{val}_J(f) > \mathsf{val}_J(s_{i-1}) = \mathsf{val}_N(s_{i-1})$). Therefore, we can create a promotion-preserving perturbation I' of I from I by changing the value of every element e in S where $\mathsf{val}_I(\mathsf{next}(e)) \ne \mathsf{val}_I(f)$ to $\mathsf{val}_I(f)$, and have all leaves with some element in S having an element promoted with f. As I is aligned, all such elements are promoted in I with f as well which means they are promoted in M with f, contradicting the choice of S. □

Let K be the representative perturbation of J, assume $M \neq K$, and define $a = \mathsf{diff}(M, K)$. We select i, $1 \leq i \leq k + 1$, such that $\mathsf{val}_J(s_{i-1}) < a \leq \mathsf{val}_J(s_i)$, and define S as the set of all elements e promoted in K with a value in the range $(\mathsf{val}_J(s_{i-1}), \mathsf{val}_J(s_i)]$ where $\mathsf{val}_K(e) = \mathsf{val}_J(e)$. Then, as $\mathsf{val}_J(s_{i-1}) < \mathsf{val}_J(e) \leq \mathsf{val}_J(s_i)$, $\mathsf{val}_I(s_{i-1}) < \mathsf{val}_I(e) \leq \mathsf{val}_I(s_i)$, for all $e \in S$.

We prove that $i \leq k$ and every member of S is promoted in K and in I with s_i. For this purpose, for every $e \in S$, we define T_e as the set of elements promoted with e in K. For any $f \in T_e$, we define $A_e(f)$ to be $\mathsf{next}(f)$ if $\mathsf{next}(f)$ has the same I-value as e and to be f otherwise. Then, clearly $A_e(e) = e$.

We show that for every $f \in T_e$,

$$\mathsf{val}_I(A_e(f)) \leq \mathsf{val}_I(e) < \mathsf{val}_I(\mathsf{next}(A_e(f))). \tag{6.1}$$

Since K is a perturbation of J, $\mathsf{val}_J(f) \leq \mathsf{val}_K(f) < \mathsf{val}_J(\mathsf{next}(f))$ and hence as $\mathsf{val}_K(f) = \mathsf{val}_K(e) = \mathsf{val}_J(e)$,

$$\mathsf{val}_J(f) \leq \mathsf{val}_J(e) < \mathsf{val}_J(\mathsf{next}(f)). \tag{6.2}$$

Now we consider two cases based on the element $A_e(f)$. When $A_e(f) = f$, $\mathsf{val}_I(\mathsf{next}(f)) \neq \mathsf{val}_I(e)$ and thus due to Equations 6.2 and Definition 22, $\mathsf{val}_I(f) \leq \mathsf{val}_I(e) < \mathsf{val}_I(\mathsf{next}(f))$; so the claim is true. In case $A_e(f) = \mathsf{next}(f)$, $\mathsf{val}_I(\mathsf{next}(A_e(f))) > \mathsf{val}_I(A_e(f)) = \mathsf{val}_I(\mathsf{next}(f)) = \mathsf{val}_I(e)$ and thus again the claim is true.

Now we show that $i \leq k$, and for every $e \in S$ and any element f of T_e, $A_e(f)$ is promoted with s_i in I. As $\mathsf{diff}(K, M) = a$, for no member f of T_e and value i, $A_e(f)$ may be promoted in M, and so nor in I, with s_j where $j < i$. So Equation 6.1 shows that the perturbation I' of I that changes the value of $A_e(f)$ to $\mathsf{val}_I(e)$, for all $f \in T_e$ is a promotion-preserving perturbation of I causing members $A_e(f)$, for $f \in T_e$ be promoted. Therefore, as I is an aligned input, by Definition 15 $\mathsf{val}_I(A_e(f)) = \mathsf{val}_I(e)$, for all $f \in T_e$, and $\mathsf{val}_I(e)$ is in the result set of I. As we showed, $\mathsf{val}_I(s_{i-1}) < \mathsf{val}_I(e) \leq \mathsf{val}_I(s_i)$. So $\mathsf{val}_I(e) = \mathsf{val}_I(s_i) = \mathsf{val}_I(A_e(f)) < \mathsf{val}_I(\mathsf{next}(A_e(f))$, for all $f \in T_e$. Hence, for all $f \in T_e$, $\mathsf{val}_K(s_i) = \mathsf{val}_J(s_i) < \mathsf{val}_J(\mathsf{next}(A_e(f)) \leq \mathsf{val}_K(\mathsf{next}(A_e(f))$.

We next show that M is maximal at $\mathsf{val}_J(s_i)$. For every element e satisfying $\mathsf{val}_J(e) \leq \mathsf{val}_J(s_i) < \mathsf{val}_J(\mathsf{next}(e))$, we have $\mathsf{val}_I(e) \leq \mathsf{val}_I(s_i) < \mathsf{val}_I(\mathsf{next}(e))$. So, for every perturbation M' of J with $\mathsf{diff}(M', M) = \mathsf{val}_J(s_i)$, we can build a promotion-preserving perturbation I' of I from I in which every leaf containing an element promoted in M' with value $\mathsf{val}_J(s_i)$ contains an element promoted in I' with value $\mathsf{val}_I(s_i)$: for every leaf l containing an element e promoted in M' with value $\mathsf{val}_J(s_i)$, we change the value of e to $\mathsf{val}_I(s_i)$. Then, as I is aligned, the same set of elements are promoted in I and in I'. Therefore, every leaf containing an element promoted in M' with s_i, already contains an element promoted in I and so in M with s_i. Thus, M is maximal at $\mathsf{val}_J(s_i)$.

We next show that $a = \mathsf{val}_J(s_i)$. Assuming to the contrary that $a < \mathsf{val}_J(s_i)$, we prove that M compresses K. As we showed, $\mathsf{val}_K(s_i) < \mathsf{val}_J(\mathsf{next}(A_e(f)) \leq \mathsf{val}_J(\mathsf{next}(\mathsf{next}(f))$ for every $f \in T_e$ and $e \in S$. Also, for every element g promoted in K with a value in the range $[a, \mathsf{val}_J(s_i)]$, by Lemma 40 there is an element e in S promoted with g in K and thus, $A_e(g)$ is promoted with s_i in M. The last condition of Definition 19 is satisfied as well because M is tight (since I is tight). Therefore, K is compressed by M which contradicts the fact that K is the representative perturbation of J. Hence, $\mathsf{diff}(M, K) = a = \mathsf{val}_J(s_i)$ and so as M is maximal at a, $M \preceq K$. On the other hand, since K is the representative perturbation of J, K is also maximal at a, and as a result, $K \preceq M$. Hence $M \equiv K$, contradicting the assumption $a = \mathsf{diff}(M, K)$. □ □

6.2.1 First Lower Bound

Let us first explain the precise form of the first lower bound. For an input I, a maximal sequence of consecutive elements of any leaf that are promoted in I is a *promoted group*. A promoted group is a *floating* one if it does not include the last nor the first element in the leaf. Suppose we use $g(l)$ as the number of floating promoted groups of a leaf l in the representative perturbation of I. Then $A(I)$ is defined as $\sum_{l\text{: leaf node}} \phi(l) + \log \binom{\mathsf{size}(l)}{g(l)}$. In this section, we prove that for any input I and algorithm $\mathcal{A}$ there exists an input J in the same class as I with $A(J) = A(I)$ and such that $\mathcal{A}$ needs $\Omega(A(J))$ comparisons to solve the problem for J. This amount of time is in fact needed to find the elements of each leaf l promoted in the representative perturbation among $\mathsf{size}(l)$ elements of l. For the representative perturbation I' of I (which by definition is aligned), we have $A(I) = A(I')$. Therefore, we only need to verify the claim for aligned inputs. Also, as the next lemma shows, for any aligned input I, there is a tight aligned input J in the same class such that the same set of elements is promoted in I and in J, and hence $A(I) = A(J)$. Therefore, we can focus only on tight aligned inputs.

Lemma 42 *For any aligned input I, there is a tight aligned input J in the same class as I such that the same set of elements is promoted in J and in I.*

Proof Without loss of generality, we assume that the values of elements are real numbers and the difference between the values of any two elements with different values is at least 1. Suppose E is the sequence of elements of I ordered by their I-values. For any input K and value v in the result set of K, the set of nodes in K that have v in their contribution set makes a sub-intersection tree, which we call the *v-tree of K*. The idea here is to obtain J from I such that the v-tree of J, for any v in the result set of J, is non-partitionable. As such, the values of elements in J is defined as follows.

J-values of elements promoted in I: For every value v in the result set of the root in I, we partition the v-tree of I into a number of non-partitionable sub-intersection trees $\mathcal{J}_1, \ldots \mathcal{J}_k$. Then for any i, $1 \leq i \leq k$, each leaf of $\mathcal{J}_i$ has an element promoted in I with value v; we define the J-value of these elements as $v - \frac{i-1}{k}$. Then each element promoted with value v in I will be promoted in J with value $v - \frac{i}{k}$, for some i, $0 \leq i < k$. Also, assuming no new element is promoted in J, for each v in the result set of J, the v-tree in J will be non-partitionable.

J-values of elements not promoted in I: The goal here is to determine J-values of elements not promoted in I such that they are not promoted in J nor in any promotion-preserving perturbation of J. Let $v_{\max}$ denote a value bigger than the I-value of any element in I. For any element e of any leaf l, we define v_e as follows: suppose f is the smallest element of l that is promoted in I and is not smaller than e; if such an element f exists, we define v_e to be J-value of f; otherwise, we define $v_e = v_{\max}$. The value of every non-promoted element e in a leaf l is defined as $v_e - \frac{i}{n^2}$. where i is the number of elements after e in the sequence E. This way J-values of every two elements that are not promoted in I will be different.

One can observe that the same set of elements is promoted in J and in I: every element not promoted in I has a J-value different than the J-value of any other element (because of the $-\frac{i}{n^2}$ term), and every element promoted in I is still promoted in J. Now, it suffices to show that J is an aligned input. Suppose to the contrary that J is not aligned and so by Definition 15 there is a promotion-preserving perturbation J' of J and an element e such that e is promoted in J' but not in J. We can assume J' is minimal. Suppose f is the J-biggest element promoted in J' with e.

First we show that for every element g promoted in J' with f, $v_g = v_f$. If that's not the case, $|v_f - v_g| \geq \frac{1}{n}$. Moreover, no matter whether f is promoted in I or not, $\mathsf{val}_J(f) = v_f - \frac{i}{n^2}$ for $0 \leq i < n$, and similarly $\mathsf{val}_J(g) = v_g - \frac{j}{n^2}$, for $0 \leq j < n$, and also $\mathsf{val}_J(f) \geq \mathsf{val}_J(g)$. Therefore, $\mathsf{val}_J(f) > v_g \geq \mathsf{val}_J(g)$. The equality $v_g = \mathsf{val}_J(g)$ cannot hold because otherwise g is promoted in J and so as J' is a promotion-preserving perturbation, $\mathsf{val}_{J'}(f) = \mathsf{val}_{J'}(g) = \mathsf{val}_J(g) < \mathsf{val}_J(f)$ which contradicts J' being a perturbation. So, $\mathsf{val}_J(f) > v_g > \mathsf{val}_J(g)$ where v_g is the value of an element in the same leaf as g, which means the J'-value of g cannot be more than v_g and again g may not be promoted in J' with f. So the claim is true.

Next we can see that for every element g promoted in J' with f, g is not promoted in J. Otherwise g is promoted in J and so $v_g = \mathsf{val}_J(g)$. Therefore, $v_e = v_f = v_g = \mathsf{val}_J(g) = \mathsf{val}_{J'}(g) = \mathsf{val}_{J'}(e)$. Also, since J' is a permutation of J, $\mathsf{val}_J(e) \leq \mathsf{val}_{J'}(e) < \mathsf{val}_J(\mathsf{next}(e))$ and so $\mathsf{val}_J(e) \leq v_e < \mathsf{val}_J(\mathsf{next}(e))$. Moreover, as $v_e = \mathsf{val}_J(g) \neq v_{\max}$, by definition v_e is the J-value of an element in the same leaf as e. Thus, $v(e) = \mathsf{val}_J(e)$ and hence, e is promoted in J, which contradicts our assumption.

Defining L as the set of leaves containing elements promoted with e in J', for each leaf l in L, we specify an element h_l not promoted in I, such that these new elements can be promoted together in a promotion-preserving perturbation of I, which will be a contradiction proving the lemma. We consider two cases based on whether $v_f = v_{\max}$ or not. If $v_f = v_{\max}$, for each leaf $l \in L$ and the element g of l promoted in J' with e, then neither e nor any element after g in l is promoted in J. In this situation, for each leaf $l \in L$, we define h_l to be the last element of l. Now consider the case $v_f < v_{\max}$. In this case, for every element g of any leaf $l \in L$ that is promoted with e in J', g is not promoted in J, but an element g' after g in l is promoted in J with value $v_g = v_f$, and no element between g and g' is promoted in J. So in this case we define h_l to be the element right before g'. In either of these two cases, we see that elements h_l, for $l \in L$, are not promoted in I and the elements right after them (if any) are promoted in I and have the same J-value (which is v_f) and so also have the same I-value. So we may create a promotion-preserving perturbation I' of I in which every element has the same value as in I except from elements h_l, for $l \in L$, whose value is changed to $\max_{l \in L} \mathsf{val}_I(h_l)$. Clearly elements h_l are promoted in I', which contradicts I being promotion-preserving. □ □

In the next definition, we define a "gap change" variant of an input I as an input J that is similar to I, but the number of elements in "gaps" between floating promoted groups of each leaf in J can be different from that in I. Here the key idea is that an algorithm should distinguish between certain gap changes of an input.

Definition 23 *Consider an input I. A* gap change *of I is an input J with the same signature such that the same number of elements are promoted in I and in J, and there is a function t mapping every element of every leaf in J to an element in the same leaf in I such that*

- *For every element e, e is promoted in J if and only if $t(e)$ is promoted in I.*
- *For elements e_1 and e_2 promoted in J, e_1 is J-smaller than e_2 if and only if $t(e_1)$ is I-smaller than $t(e_2)$.*

- *For elements e_1 and e_2 where exactly one of e_1 and e_2 is promoted in J and $t(e_1)$ and $t(e_2)$ are not of the same I-value, e_1 is J-smaller than e_2 if and only if $t(e_1)$ is I-smaller than $t(e_2)$.*

Note that t may map two elements to one element and need not to be onto, but it can be observed that, for each leaf l, t defines an order-preserving and one-to-one mapping between elements of l that are promoted in J and elements of l that are promoted in I. Intuitively, a gap change of an input changes the size of non-empty gaps between promoted elements and keeps empty gaps empty. Using the lower bound technique of Chapter 3, we prove that the number of gap changes of an aligned input in a class is a combinatorial lower bound on the worst case running time of any algorithm for inputs in the class.

Lemma 43 *For any tight aligned input I and algorithm $\mathcal{A}$, there is a reduction J of a gap change of I, where $\mathcal{A}$ needs $\Omega(A(I))$ comparisons when run on J.*

Proof We build the input J by running the algorithm $\mathcal{A}$ against an adversary that assigns actual values to elements and reveals them to the algorithm at the time the algorithm "touches" (inspects) them. In this input, each leaf l will have the same number of promoted groups as in I and each promoted group of I has a corresponding promoted group in J in the same leaf and with the same size. This will be such that for a leaf l with g floating promoted groups in I and a floating promoted group G of l in I, the algorithm does not have a chance to touch any element of the promoted group G' of J that corresponds to G before touching at least $\log \frac{\mathsf{size}(l)-\phi(l)}{g+1}$ elements in the "region" containing G', where $\phi(l)$ is computed for input I.

Here is a more precise definition of "regions" we just mentioned. For a a leaf l with g floating promoted groups and index r where $1 \leq r \leq g$, we use S^l_r to denote the sequence of elements in the rth floating promoted group of l in I. Also, if the first (the last, respectively) element of l is promoted, S^l_0 (S^l_{g+1}, respectively) denotes the first (the last, respectively) promoted group of l; otherwise S^l_0 (S^l_{g+1}, respectively) is empty. Note that for r, $1 \leq r \leq g$, S^l_r is non-empty by definition. We use T^l_r to denote the promoted group of J that corresponds to S^l_r. For a leaf l, the sequence of elements of l is divided as follows: The first $|S^l_0|$ and the last $|S^l_{g+1}|$ elements, called *regions* R^l_0 and R^l_{g+1}, respectively, will contain the elements of T^l_0 and T^l_{g+1}. The rest of the elements are divided into g *regions* $R^l_1, \ldots, R^l_g$, the rth of which being of size $|S^l_r| + \left\lfloor \frac{\mathsf{size}(l)-\phi(l)}{g+1} \right\rfloor$ or $|S^l_r| + \left\lceil \frac{\mathsf{size}(l)-\phi(l)}{g+1} \right\rceil$ where, as mentioned, $\phi(l)$ is the number of elements of l promoted in I.

The adversary uses triplets of integers as values of elements in the instance J it builds. Consider any promoted group S^l_r of a leaf l in I. We define a^l_r as the smallest member of the result set of I that is bigger than the I-value of the element right before S_r^l in l, and b_r^l as the value of the biggest member of the result set of I that is smaller than the I-value of the element right after S_r^l in l. If $v_1, \ldots, v_k$ is the sequence of I-values of elements of S_r^l, the ith element of T_r^l, for each i, will have a value of the form $(v_i, 0, x)$, for some $x \leq 0$. Each element of R_r^l that is before T_r^l will have a value of the form $(a_r^l, x, 0)$, for some $x < 0$, and each element of R_r^l that is after T_r^l will have a value of the form $(b_r^l, x, 0)$, for some $x > 0$.

Algorithm 8 shows how the adversary follows the schema we have just explained to delay touching elements of promoted groups until the aforementioned number of elements are touched. In this algorithm, the variable $\mathcal{C}_r$ stores a subsequence of R_r^l and at each step it is truncated to a smaller sequence. At the beginning, $\mathcal{C}_r$ is initialized to R_r^l and at the end it will be the sequence T_r. Note that this algorithm does not specify how the third coordinates of elements in T_r are set. We discuss that in the rest of the proof.

Algorithm 8: How to determine the first and the second coordinates of members of R_r^l when an element s of R_r^l is touched.

```
if $|\mathcal{C}_r| \leq 2|S_r^l|$ then
    – set the first coordinates of first $|S_r^l|$ members of $\mathcal{C}_r$ equal to $v_1, \ldots, v_{|S_r^l|}$, and
    the second coordinate of them equal to zero;
    – set the first coordinates of the rest of members of $\mathcal{C}_r$ equal to $b_r$, and the
    second coordinate of them equal to $|S_r^l|, \ldots, |\mathcal{C}_r| - 1$;
    – Remove the last $|\mathcal{C}_r| - |S_r^l|$ elements of $\mathcal{C}_r$ from it;
else
    suppose $s$ is the $i$th member of $R_r^l$;
    if $i < |\mathcal{C}_r| - i + 1$ then
        – assign values $-(|\mathcal{C}_r| - 1)$, $-(|\mathcal{C}_r| - 2)$, ..., $-(|\mathcal{C}_r| - i)$ to the second
        coordinates of the first $i$ members of $\mathcal{C}_r$, and $a_r$ to the first coordinate of
        them;
        – Remove the first $i$ members of $\mathcal{C}_r$ from it;
    else
        – assign values $i - 1$, $i$, ..., $|\mathcal{C}_r| - 1$ to the second coordinates of the last
        $|\mathcal{C}_r| - i + 1$ members of $\mathcal{C}_r$, and $b_r$ to the first coordinate of them;
        – Remove the last $|\mathcal{C}_r| - i + 1$ members of $\mathcal{C}_r$ from it;
    end
end
```

Let J be the input generated by the adversary; we claim that if the adversary can force the algorithm to touch elements of all promoted groups, and we obtain the input J' from J by setting the third coordinates of all elements to zero, J' is a gap change of I. To show the correctness of this claim, we define the mapping t required by Definition 23 as follows. Elements of non-floating promoted groups (i.e., R^l_0 and R^l_{g+1}) are mapped to the corresponding elements in T^l_0 and T^l_{g+1}. Now consider any region R^l_r, where $1 \leq r \leq g$. For any member e of R^l_r with second coordinate set to zero, the first coordinate of e equals the I-value of an element e' in the same leaf as e promoted in I; so we define $t(e) = e'$. For every member e of R^l_r with a negative second coordinate, $t(e)$ is defined as the element right before S^l_r, and for every member e of R^l_r with a positive second coordinate, $t(e)$ is defined as the element right after S^l_r. Then it is easy to see that the conditions in Definition 23 hold.

Now we show that the adversary can set the third coordinates of elements in T^l_r, for leaves l and indices r, $0 \leq r \leq g+1$, in such a way that all such elements are touched by the algorithm and J is a reduction of J'. For a value v in the result set of I, we use U_v to denote the sub-intersection tree consisting of all nodes with value v in their contribution set in I. Suppose an element e of R^l_r, for some l and r is touched, and the adversary decides it should make the element a part of T^l_r and, so it assigns to e a value of the form $(v, 0, x)$, for a value v in the result set of I and a value x which we describe shortly. We then mark l as *touched for v*. A node u of U_v is *touched for v* if every leaf of U_v in the subtree rooted at u of U_v, is marked as touched for v. Algorithm 9 shows how we determine the third coordinate of e. It is easy to see that following invariants remain true throughout lifetime

Algorithm 9: How to determine the third coordinates of members of R^l_r.

```
Mark $l$ as "touched for $v$";
Look for the lowest ancestor $u$ of $v$ that is not touched for $v$;
**if** *$u$ does not exist or is an intersection node* **then**
    Set the value of $e$ to $(v, 0, 0)$;
**else**
    Suppose $e$ is the $i$th element of $I$ (according to some fixed arbitrary order);
    Set the value of $e$ to $(v, 0, -i)$;
**end**
```

of the algorithm:

- No value other than $(v, 0, 0)$, for members v of result set of I, becomes a member of the result set of an intersection node.

- For every member v of the result set of I and every node u of U_v that is not touched for v yet:
  - if u is a union node, $(v, 0, 0)$ is not in the result set of any of the children of u that are touched for v.
  - if u is an intersection node, $(v, 0, 0)$ is in the result set of every child of u that is touched for v.

As a result, for every member v of the result set of I and every union node u of U_v, membership of $(v, 0, 0)$ in the result set of u is not determined until u gets touched for v, which means all leaves of U_v have to be touched for v before the algorithm can determine if $(v, 0, 0)$ is in the result set of J. So the total number of elements touched by the algorithm in leaf l with g floating promoted groups will be $\phi(l) + g \log \frac{\mathsf{size}(l)-\phi(l)}{g+1}$. That is because each time an element of a region corresponding to a promoted group is touched, we discard at most half of its "redundant elements" and so it takes at least $\log \frac{\mathsf{size}(l)-\phi(l)}{g+1}$ "touches" to get rid of all of redundant elements of one floating promoted group. So the total number of elements touched in leaf l is in $\Omega(\phi(l) + 2\phi(l) + g \log \frac{\mathsf{size}(l)-\phi(l)}{g+1})$. As $2\phi(l) = \phi(l) + g\frac{\phi(l)}{g} \geq g + g \log \frac{\phi(l)}{g+1}$, the number of elements touched is $\Omega\left(\phi(l) + g(1 + \log \frac{\phi(l)}{g+1} + \log \frac{\mathsf{size}(l)-\phi(l)}{g+1})\right) = \Omega\left(\phi(l) + g(1 + \log \frac{\mathsf{size}(l)}{g+1})\right) = \Omega\left(\phi(l) + g \log \frac{\mathsf{size}(l)}{g}\right)$. Therefore, the total number of elements touched by the algorithm is $\Omega\left(\sum_l \phi(l) + g \log \frac{\mathsf{size}(l)}{g}\right) = \Omega(A(I))$. □ □

Finally the next lemma shows that any gap change of an aligned input I is an aligned input itself and therefore is in the same input class as I. Also the function t described in Definition 23 preserves the number of floating promoted groups in each leaf. As a result, due to Lemma 41 for any reduction J of any gap change of an aligned tight input I, $A(J) = A(I)$. This proves the lower bound for tight aligned inputs and therefore, as discussed previously, holds for any input.

Lemma 44 *Any gap change of an aligned input I is an aligned input.*

Proof If this is not true for a gap change J of I, there is a promotion-preserving perturbation J' of J and an element that is promoted in J', but not in J; suppose a is the J'-value of that element. Also let t be the mapping mentioned in Definition 23 for I and J. We create a promotion-preserving perturbation of I' of I such that an element is promoted in I' but not in I, which contradicts the fact that I is aligned. We define S as the set of elements promoted in J' with value a, and b as the smallest value in the result set of J such that $\mathsf{val}_J(e) \leq b$, for all $e \in S$ (if such a value b does not exist we define

$b = \infty$). When $b < \infty$, we define b_t as the I-value of $t(e)$, for elements e promoted in J with value b (by Definition 23 this value is the same for all such elements e); otherwise we define $b_t = \infty$. For any $f \in S$, if f is promoted in J, then $\mathsf{val}_J(f) = \mathsf{val}_{J'}(f) = a$ is an upper bound on the J-value of all elements in S and thus $a = b = \mathsf{val}_J(f)$. We consider two cases.

The first case is when $b < \infty$ and there is an element $e \in S$ such that the leaf l containing e does not have any element promoted in J with value b. Then, l does not have any elements promoted in I with value b_t. We build a promotion-preserving perturbation I' of I such that any leaf with an element in S, has an element with I'-value b_t.

For this purpose, consider any leaf l with an element $f \in S$ that does not have any element of I-value b_t. If f is promoted in J, then $b = a = \mathsf{val}_J(f)$. The element $t(f)$ already is promoted in I and so by Definition 23 its I-value is b_t, which is a contradiction. Now suppose f is not promoted in J and hence $t(f)$ is not promoted in I. If $t(f)$ has a different I-value than b_t, as $\mathsf{val}_J(f) \leq b$, by Definition 23 $\mathsf{val}_I(t(f)) < b_t$. For any element g of l that is promoted in J and $\mathsf{val}_J(f) < \mathsf{val}_J(g)$, we have $a = \mathsf{val}_{J'}(f) < \mathsf{val}_J(\mathsf{next}(f)) \leq \mathsf{val}_J(g)$, and so, by definition of b, $b \leq \mathsf{val}_J(g)$ (because for all $h \in S$, $\mathsf{val}_J(h) \leq a < \mathsf{val}_J(g)$). Therefore, the leaf l does not have any element g promoted in J where $\mathsf{val}_J(f) < \mathsf{val}_J(g) < b$ and so, l also does not have any element g promoted in I such that $\mathsf{val}_I(t(f)) < \mathsf{val}_I(g) < b_t$. As a result, the I-biggest element of I of I-value less than b_t is not promoted in I and we can define its I'-value to be b_t. This way we build a promotion-preserving perturbation I' of I satisfying our goal.

Next suppose $b = \infty$ or e does not exist. First we prove that $a < b$. The equation $a = b$ may not hold, because then $b \neq \infty$ and so e does not exist, which is a contradiction with the choice of a. Now suppose $b < a$ and consider a leaf l with an element $f \in S$. As e does not exist, l has an element g promoted with value b in J. We have $\mathsf{val}_J(f) \leq b = \mathsf{val}_J(g) < a = \mathsf{val}_{J'}(f) < \mathsf{val}_J(\mathsf{next}(f))$ and hence $\mathsf{val}_J(f) \leq \mathsf{val}_J(g) < \mathsf{val}_J(\mathsf{next}(f))$ which means g and f are the same elements. As a result, since J' is a promotion-preserving perturbation of J, $a = b$, which is a contradiction. Therefore $a < b$.

For any element $f \in S$, f is not promoted in J because otherwise as we proved, b should equal a. So, for any $f \in S$, since $\mathsf{val}_J(f) \leq a < b$, $t(f)$ has an I-value less than b_t and is not promoted in I. Furthermore, for any element g in the leaf l containing f that is promoted in J and is J-bigger than f, we have $\mathsf{val}_{J'}(f) < \mathsf{val}_J(\mathsf{next}(f)) \leq \mathsf{val}_J(g)$ and thus by choice of b, $\mathsf{val}_J(g) \geq b$. So, l does not have an element g promoted in I where $\mathsf{val}_I(t(f)) < \mathsf{val}_I(g) < b_t$. Hence, if we consider the I-biggest element e of the leaf l I-less than b_t, e is not promoted in I. So we take the maximum I-value v of such elements e and we set the I'-values of all such elements e to v. Then we have a promotion-preserving perturbation of I in which all such elements are promoted while none of them are promoted in I. This contradicts the alignedness of I. □ □

6.2.2 Second Lower Bound

The key idea behind the second lower bound is that, roughly speaking, any algorithm should be able to distinguish any canonical proof labeling P of a signature S from other canonical proof labelings of S. More specifically, we show that there is an input J, where P corresponds to the representative perturbation I of J, and $\mathcal{A}$ has enough information to compute P after running on J. In order to prove this fact, we first need to show there is an aligned input for each proof labeling.

Lemma 45 *Every non-empty proof labeling P of a signature S corresponds to an aligned input I with signature S.*

Proof We consider a sub-union tree U in S and create the input I with signature S as follows. For any leaf ℓ of U, the last $|P(\ell)|$ elements of ℓ, called *middle elements*, will have values equal to members of $P(\ell)$ and all other elements of ℓ, called *prior elements* will have values less than the value of the minimum value v in $P(\ell)$ but bigger than any other value less than v that appears in $P(\text{root})$. For leaves ℓ that are not a part of U, the first $|P(\ell)|$ elements, called *middle elements*, of ℓ will have values equal to members of $P(\ell)$ and all other elements, called *post elements*, will have values greater than the value of the maximum value in $P(\text{root})$. As U is a sub-union tree, it is easy to see that in any input J with the same signature as I and for any value v in the result set of the root in J, there is an element of a leaf in U, and an element of a leaf outside U that is promoted with value v in J.

Now we show that for any promotion-preserving perturbation J of I, middle elements are promoted with their I-values, and no other element is promoted. Let's investigate correctness of this claim for each of the three groups of elements separately. The correctness of the claim for middle elements comes from the fact that they are already promoted in I (as their values come from a proof labeling) and J is a promotion-preserving perturbation.

Now consider any prior element e of a leaf l and consider the smallest middle element f of l. The J-value of e will be in the range $[\mathsf{val}_I(e), \mathsf{val}_I(f))$ and, due to the way we assigned I-values of prior elements, there is no middle element with an I-value in this range. Also all post elements have I-values, and thus J-values, bigger than $\mathsf{val}_I(f)$. Thus, the J-value of no non-prior element equals the J-value of a prior element, which means no prior element can be promoted in C (because prior elements only appear in leaves of U). Moreover, this means no non-post element will have a J-value equal to that of a post-element, which means no element of leaves of U has a J-value equal to that of a post-element. As a result, post elements may not be promoted in J either. Therefore the same set of elements is promoted in J and in I, which proves I is aligned. $\square$ $\square$

In Chapter 3, for any algorithm $\mathcal{A}$ and signature S, we showed that the responses the algorithm gets from the adversary for comparisons it makes is actually an encoding scheme for the proof labeling corresponding to the input. More precisely, we designed an encoding scheme for canonical tight proof labelings P of S with the following property: there is a set $\mathcal{S}_P$ of inputs with signature S such that if $\mathcal{A}$ needs at most k comparisons to solve each of the instances in $\mathcal{S}_P$, the schema can encode P with $O(k)$ bits. It can be observed that all inputs in $\mathcal{S}_P$ are reductions of aligned inputs I where P is the proof labeling of I. Also, as we showed, such an input I and its reductions are in the same class. So the minimum number of bits required to encode canonical tight proof labelings corresponding to aligned inputs in a class is a lower bound on the worst-case running time of any algorithm on inputs in that class.

Given any function f, one can find $\prod \binom{\mathsf{share}_f(v)}{\mathsf{share}_f(u)} = 2^{B_f}$ canonical tight proof labelings P such that $|P(l)| = \mathsf{share}_f(l)$ for all leaves, where the product is over all union nodes v and children u of v. So for at least one of these proof labelings, the aforementioned encoding schema needs at least B_f bits to encode it. This means there is an input in the class requiring at least B_f comparisons to be solved by $\mathcal{A}$. This proves the second lower bound.

6.3 The Algorithm

Our algorithm works by scanning all sets in a synchronized fashion and at each iteration a new member of the result set of the tree is discovered. Hence, the members of the result set are discovered one at a time in increasing order. At each iteration of the algorithm, for each node v, we store the best lower bound we have for the next member of the result set of the subtree rooted at node v, in an array denoted by $\mathsf{min}[v]$. Values in the min array guide the algorithm to decide what parts of the tree to explore at each step in search of the next member of the result set of the root.

Algorithm 10: update(v, k): boolean

```
if $v$ is a leaf then
    /* using gallop search:                                                */
    $e$ := the first element of $v$ with value more than $k$;
    $\mathsf{min}[v] = \mathsf{val}(e)$;
    return true iff $\mathsf{val}(\mathsf{prev}(e))$ equals $k$;
end
if operation(v)= $\cap$ then
    forall the $u$ child of $v$ do update$(u, k)$;
    $\mathsf{min}[v]$ = maximum of $\mathsf{min}[u]$ over all children $u$ of $v$;
    return true iff all calls to update(u,k) returned true;
end
if operation(v)= $\cup$ then
    /* use the FindSmalls operation of the "min-update" structure to
       implement the next line                                             */
    forall the children $u$ of $v$ s.t. $\mathsf{min}[u] \leq k$ do update$(u, k)$;
    /* use the UpdateSmall operation of the "min-update" structure to
       implement the next line                                             */
    update the values in the "min-update" structure;
    /* use the FindMin operation of the "min-update" structure to
       implement the next line                                             */
    $\mathsf{min}[v]$ = minimum of $\mathsf{min}[u]$ of all children $u$ of $v$;
    return true iff at least one call to update$(u, k)$ returned true;
end
```

6.3.1 The Sketch of the Algorithm

Each iteration of the algorithm is an invocation of a recursive function update(v, k) which is called on a node v with a suggested lower bound k on the next element to be promoted. After the execution of the function on v, $\mathsf{min}[v]$ will be updated to a lower bound on the smallest member of the result set of v that is larger than k. The function also returns a boolean value to indicate if k is a member of the result set of v.

The update function is given in Algorithm 10 and works based on the type of node v. When v is a leaf, a variation of binary search called *gallop search* [11] is used to look for the first element e of value greater than k in $\mathsf{elements}(v)$, and $\mathsf{min}[v]$ is set to e.[6]

[6] For the leaves v, we need to remember the element of value $\mathsf{min}[v]$ as well and so, in addition to the

Here is a short explanation of gallop search, for our use-case. Suppose f is the element of v pointed to by $\mathsf{min}[v]$ just before running an update on v, and as mentioned we look for the element e, and assume e is the m'th element after f in the leaf. We first find an element g with a value not smaller than k after f, and this way we limit the candidates to g and the elements between f and g. We do this using at most $\lceil \log m \rceil + 1$ comparisons in a way that there are at most $2m$ elements between f and g. Here is what we do to find g: for $i = 0, 1, 2, \ldots$, we compare the value of the 2^ith element after f with k, until we reach one not smaller than k (which is the element g we mentioned). Then if the value of g is bigger than k, we know the element e we look for is between f and g, so using a binary search in $O(\log m)$ we can find e.

When searching for element e, we run two gallop searches in parallel: one starting from f going forward, and one starting from the end of the sequence $\mathsf{elements}(v)$ going backward. In this way, if there is a large gap between f and e, and e is closer to the end of $\mathsf{elements}(v)$, we find e faster.

When calling update recursively, at an intersection node, update is called recursively for all children and then the min value is set to the maximum of those of children. In contrast, at a union node, only those children u with a value $\mathsf{min}[u]$ not exceeding k are recursively updated. To find and update this proper set of children at a union node, we design an efficient data structure "min-update" described in Section 6.3.2.

Algorithm 11: The main function that computes the answer.

```
update(root, $-\infty$);
while $\mathsf{min}[root] \neq \infty$ do
    if update(root, $\mathsf{min}[root]$) then
        output $\mathsf{min}$[root];
    end
end
```

The main function of the algorithm is given in Algorithm 11. It consists simply of iterative invocations of function UPDATE() on the root of the expression tree. Each of these calls is a *round*. In case the function evaluates to true, the current min[root] value belongs to the result set, and therefore it is reported in the output. Also as explained, the min[root] is updated to a larger value on which the function is called in the next iteration.

value of the element, we also store a reference to the actual element. As such in the rest of the chapter, when v is a leaf, $\mathsf{min}[v]$ may refer to the value or to the element, depending on the context.

We define the *target value of a round* as the value k for which update(root, k) is invoked in that round.

6.3.2 The "min-update" Structure

In the update function of Algorithm 10, on a union node v, we find children u of v with $\mathsf{min}[u] \leq k$ and invoke update on these children recursively. We cannot afford to perform a linear scan of all children of v in every update iteration on a union node v. Furthermore, over the course of many invocations of update on v, some children of v are updated more frequently than others. The overall time spent to update node v must be adaptive to the update frequencies on different children of v.

We introduce a data structure which we name *Huffman heap* to perform operations on the min array efficiently. It stores a set of entries (the values of min array for children of v), and performs the following operations efficiently:

FINDMIN: retrieves the smallest value.

FINDLESSTHAN(key): report all the entries with value at most key.

UPDATEHEAP: updates the entries according to the new values in min array.

The operation UPDATEHEAP is used after operation FINDLESSTHAN and before any other operations is performed on the array. We run this operation once we finish running update recursively on selected children of the union node.

To be more precise, for a union node v with children $u_1, \ldots, u_r$, we define a *v-round* as a round in which update is called on v and we assume there are m v-rounds. For $0 \leq t \leq m$ and $1 \leq i \leq r$, we denote the number of times update is invoked on child u_i in the first t v-rounds by $b_i^{(t)}$, and define $b^{(t)} = \sum_i b_i^{(t)}$. The data structure is a binary tree over r leaves, where each leaf of the binary tree corresponds to a child u of v. We store $\mathsf{min}[u]$ in the leaf node corresponding to the child u of v; also, in every internal node x, we store the minimum value stored in the subtree rooted at x. Consequently, the cost of FINDLESSTHAN(k) is asymptotically the number of leaves with value at most k together with their ancestors. For a set S of leaves in this tree, we use Ancestors(S) to denote the set containing members of S and their ancestors. The running time of UPDATEHEAP is also asymptotically $|\text{Ancestors}(S)|$, for S the set of leaves found in the previous FINDLESSTHAN. The operation FINDMIN is clearly done in constant time.

The tree structure is updated in such a way that just before running the t-th v-round, the height of each child u of v is at most $-\log p_i^{(t)} + 2$ where $p_i^{(t)} = \frac{b_i^{(t)}}{b^{(t)}}$. In order to achieve this goal we rebuild the tree after v-round numbers τ_1, τ_2, ... (values to be set shortly). Rebuilding the tree after v-round $t = \tau_j$, for some j, is done using the Shannon-Fano algorithm [26]:

1. Let $u_{x_1}, \ldots, u_{x_k}$ be children of v such that $(p_{x_i}^{(t)})$ is a non-decreasing sequence. Find a value d minimizing the difference between $\sum_{i=1}^{d} p_{x_i}^{(t)}$ and $\sum_{i=d+1}^{k} p_{x_i}^{(t)}$.
2. Recursively, create the tree T_1 with nodes $u_{x_1}, \ldots, u_{x_d}$ and the tree T_2 with the rest of the nodes. T_1 and T_2 are the left and the right subtrees of the root, respectively.

The depth of each node u_i will be $-\log p_i^{(\tau_j)} + 1$ [26]. The values of τ_j are chosen as follows: Defining $\tau_0 = 0$, τ_j is the smallest value bigger than τ_{j-1} where for some child u_i of v, $b_i^{(\tau_j)} > 2b_i^{(\tau_{j-1})}$. Then in any round t, the height of each node u_i will be at most $-\log p_i^{(t)} + 2$.

Lemma 46 *The total time spent on min-update operations at a union node v with r children is $O(r^2 \log^2 r + \sum_{i=1}^{r} b_i \log \frac{m+b_i}{b_i})$, where the b_i's and m are defined as above.*

Proof The running time of heap operations can be divided into two parts: performing individual FindMin, FindLessThan, and UpdateHeap operators, and rebuilding the Huffman heap in rounds τ_1, τ_2, ...

Part 1. The running time of this part is $m + |\sum_{t=1}^{m} \text{Ancestors}(S_t)|$, where S_t is the set of leaves updated in v-round t. Using induction on the height of the tree, it is easy to show that in a binary tree, for any set S of leaves, $\sum_{l \in S} |\text{Ancestors}(l)| - |\text{Ancestors}(S)| \geq |S| \log |S| - 2|S|$. Thus, defining $s_t = |S_t|$, $b = b^{(m)}$, and, for each i,

$b_i = b_i^{(m)}$ (and so, $b = \sum_{t=1}^{m} s_t$),

$$\begin{aligned}
&\sum_{t=1}^{m} |\text{Ancestors}(S_t)| \\
\leq\ & \sum_{t=1}^{m} \left(\sum_{l \in S_t} |\text{Ancestors}(l)| - s_t \log s_t + 2s_t \right) \\
\leq\ & \left(2b + \sum_{t=1}^{m} \sum_{u_i \in S_t} \left(-\log p_i^{(t)} + 2 \right) \right) - \sum_{t=1}^{m} (s_t \log s_t - 2s_t) \\
=\ & \left(4b + \sum_{t=1}^{m} \sum_{u_i \in S_t} \log \frac{b^{(t)}}{b_i^{(t)}} \right) - \sum_{t=1}^{m} (s_t \log s_t) + 2b
\end{aligned}$$

which due to convexity of $x \log x$ is at most

$$\begin{aligned}
&6b + \left(\sum_{i=1}^{r} \sum_{S_t, u_i \in S_t} \log \frac{b}{b_i^{(t)}} \right) - \sum_{t=1}^{m} \left(\frac{b}{m} \log \frac{b}{m} \right) \\
&= 6b + b \log m - \sum_{i=1}^{r} \sum_{j=1}^{b_i} \log j.
\end{aligned}$$

By Stirling's bounds, this is asymptotically equal to $\left(\sum_{i=1}^{r} b_i \log \frac{m+b_i}{b_i} \right)$.

Part 2. The algorithm spends $O\,(r \log r)$ time each time it rebuilds the Huffman heap. So, in total, the algorithm spends at most $O\,(r \log r \sum_{i=1}^{r} \log b_i)$ time to keep the Huffman heap approximately updated. Let us consider each term of the sum individually: if $b_i \leq r \log r \log b_i$, then $\log b_i \in O(\log r)$ and so $r \log r \log b_i \in O\left(r \log^2 r\right)$. Therefore, in any case $r \log r \log b_i \in O(r \log^2 r + b_i)$ and hence the total rebuilding time is in $O\left(r^2 \log^2 r + \sum_{i=1}^{r} b_i\right)$.

□

6.3.3 Generating Representative Perturbation

In Section 6.1, we defined $\phi(v)$ as the size of the contribution set of v in the representative perturbation. Here, we bound the number of update calls on each node of the tree by this value. For this, we show that the algorithm "generates" an aligned input, which is indeed the representative perturbation of the input.

After each round in the execution of the algorithm on an input I, we define the perturbation J "generated" by the algorithm as follows. For a round r and leaf l updated in round r, the *stop element of l in round r* is defined as the biggest element of l with a value no more than the target-value of r. In other words, this is the element $\mathsf{prev}(\mathsf{min}[\ell])$ at the end of round r. For every round r so far and leaf ℓ updated in round r, we define the J-value of the stop element of ℓ in r to be k, where k is the target value of round r; we define the J-value of every other element to be the I-value of that element. By the "perturbation generated by the algorithm" we mean the perturbation generated after all rounds.

Compactness

A perturbation J of an instance I is said to be *compressible* at a value u if there is an aligned perturbation K of I with $\mathsf{diff}(J, K) = u$ which compresses J. Clearly a compact perturbation is one that is not compressible at any point. In order to preserve the compactness constraint of Definition 21, after detecting a value b in the result set of the root of the original input, the algorithm analyzes the previous round r_1 and the current round r_2 (the one with target value b) to decide if the algorithm has generated a perturbation that is compressible at u, for u the target value of r_1. If the perturbation generated is compressible at u, then rounds r_1 and r_2 should be "replaced" with a single round of target value b. So we undo them and we call $\text{update}(\text{root}, b)$ to have a single round r_3 of target value b instead, and we say the rounds r_1 and r_2 are *rolled-back by* r_3 and remove the rolled-back rounds from consideration. Next we will show that there cannot be more than one roll-back for the same value in the result set of I thus bounding the roll-back time to within a constant factor of the non-rolled-back rounds.

We also alter the definition of the perturbation J generated by the algorithm after a round r as follows: we only consider r and its previous rounds that are not rolled-back until the end of round r, and for every such round r' and leaf ℓ updated in round r', we set the J-value of $\mathsf{prev}(\mathsf{min}_{r'}[\ell])$ to k, where k is the target value r'. We set the J-value of every other element to its I-value.

An important issue here is how to detect the compactness of the perturbation. To be more precise consider the situation after rounds r_1 and r_2 described above. The problem is to decide if the perturbation P_1 generated after round r_2 is compressed by the perturbation P_2 generated after rolling back r_1 and r_2 by the round r_3, and if such a perturbation P_2

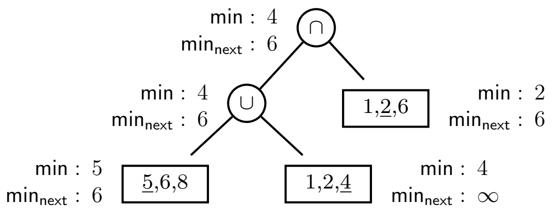


Figure 6.6: An example of min and $\mathsf{min}_{\mathsf{next}}$ values. Next element to be processed in each leaf is underlined.

is aligned. The conditions in Definition 19 for P_2 compressing P_1 can be easily checked by examining the set of leaves visited in the previous round. The challenging part is how to determine if P_2 is aligned. We discuss that in the rest of this section, but assuming we can do that properly, we can make the following claim.

Observation 47 *The final perturbation generated by the algorithm compresses the perturbation generated by the algorithm immediately after any rolled-back round r.*

In order for the algorithm to predict the alignedness of the perturbation that will be generated after rolling back (note that we need to determine it before doing the actual roll-back), we create an array called $\mathsf{min}_{\mathsf{next}}$ on the nodes of the tree, similar to the min array, but instead of being computed on the first non-processed element of each leaf, it is computed on the second non-processed element of each leaf. Figure 6.6 shows an example. At the end of each call Update(v, k), we update $\mathsf{min}_{\mathsf{next}}$ as follows:

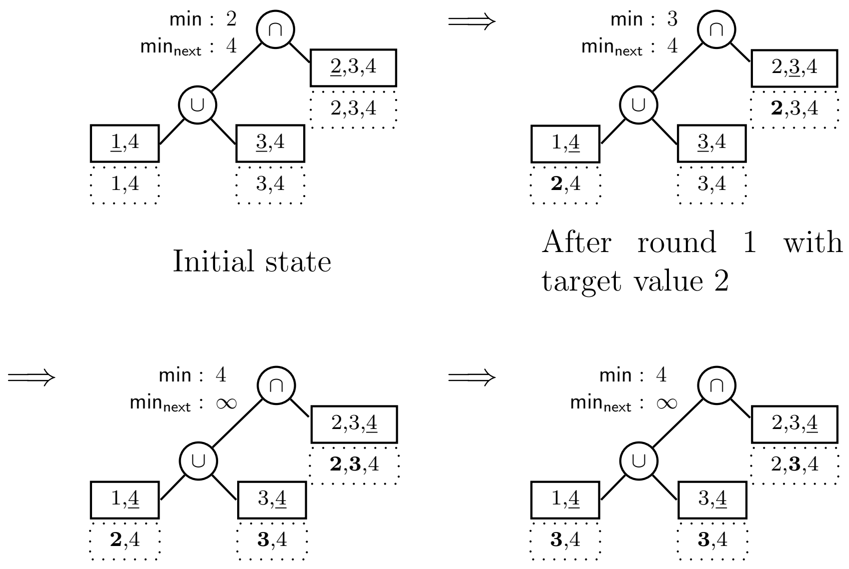
- if v is a leaf, $\mathsf{min}_{\mathsf{next}}[l]$ will be $\mathsf{next}(e)$ for e the smallest element of l with value bigger than k, or an element of value ∞ if all elements of l have values at most k.
- if v is an intersection (union, respectively) node, $\mathsf{min}_{\mathsf{next}}[v]$ is set to the maximum (minimum, respectively) of $\mathsf{min}_{\mathsf{next}}[u]$, for the children u of v.

Lemma 48 *The perturbation generated after rolling back rounds r_1 and r_2 with a round r_3 of target value b is aligned only if $\mathsf{min}_{\mathsf{next}}[root] > b$ right before round r_1.*[7]

Proof For any node v, $\mathsf{min}_{\mathsf{next}}[v]$ is the minimum number a for which v has a sub-intersection tree U such that the second non-processed element of each leaf has a value of at most a. Now suppose despite the fact that $\mathsf{min}_{\mathsf{next}}[v] \leq b$, we rollback the round. For all nodes v of U, we have $\mathsf{min}[v] \leq \mathsf{min}_{\mathsf{next}}[v] \leq b$ and so Update(v, b) is run on nodes of U

[7] Although in this lemma we prove the “only if” part, the “if” part also holds and is proved in Lemma 49.

recursively. As the second non-processed element before round r_1 has a value of at most b, in the perturbation J generated after the roll-back, for every leaf l of U, an element e_l of l is promoted with value b in J and $\mathsf{prev}(e_l)$ has a value less than b and is not promoted in J. Then, defining $a = \max_{l \in \{\text{leaves of } U\}} \mathsf{val}(\mathsf{prev}(e_l))$, one can create a perturbations K of J by changing values of elements $\mathsf{prev}(e_l)$, for leaves l of U, to a, obtaining a perturbation in which elements $\mathsf{prev}(e_l)$ are promoted. So J is not aligned. $\square$ $\square$



Initial state

After round 1 with target value 2

After round 2 with target value 3

After rolling back rounds 1 and 2 with a round of target value 3

Figure 6.7: An example of a situation where we roll-back rounds. The next element to be processed in each leaf is underlined. Also, the sequences in “dotted” boxes show the perturbation generated by the algorithm at each stage.

So we use the condition $\mathsf{min}_{\mathsf{next}}[\text{root}] > b$ to decide if we should do the actual roll-back. Figure 6.7 shows an example.

Now the question is how we keep the $\mathsf{min}_{\mathsf{next}}$ values up-to-date in an affordable time. Anytime the function update is called on an intersection node, it is also called on all its children, so the cost of updating $\mathsf{min}_{\mathsf{next}}$ is within a constant factor of the calls. For union nodes, we use a similar technique as for min to prevent it from incurring an additional cost:

in the Huffman heap constructed for the node, at each vertex of the heap, in addition to the minimum value among the min-values in the subtree at that vertex, we also store the minimum $\mathsf{min}_{\mathsf{next}}$-value of those nodes. This way, $\mathsf{min}_{\mathsf{next}}$ is updated as fast as min and the overhead is just a change in the constant factor.

Lemma 49 *The algorithm generates the representative perturbation of the input.*

Proof Considering the perturbation J generated by the algorithm for the input I, we show that J satisfies the conditions mentioned in Definition 21 one by one.

Claim 2 *J is maximal at any point b that is in the result set of the input.*

Proof This is because for any perturbation J' of I with $\mathsf{diff}(J, J') = b$ and for any leaf l containing an element e promoted with value b in J', $\mathsf{min}[l]$ just before the non-rolled-back round r of target value b is at most b because otherwise, e is promoted with a value less than b in J, which contradicts $\mathsf{diff}(J, J') = b$. As such leaves make a sub-intersection tree, just before round r, $\mathsf{min}[l] \leq b$, for all ancestors u of such leaves l. Therefore, in round r, update(l, b) is called, for all such leaves l, and hence an element of value b is promoted in J from each of those leaves. Thus, $J \preceq J'$, which means J is maximal at b. □

Claim 3 *For any promotion-preserving perturbation K of I where $J \npreceq K$, there is a perturbation K' with $K' \preceq K$ such that J compresses K' but K does not.*

Proof For $a = \mathsf{diff}(J, K)$, by definition J is not maximal at a and thus by Claim 2 a is not in the result set of I. We define b_1 as the biggest value in the result of I smaller than a ($-\infty$ if no such value exists) and b_2 as the smallest value in the result of I bigger than a (∞ if no such value exists). Consider the last round r_1 with target value c_1 less than a (if such a round exists), and the last round r_2 with target value c_2 less than or equal to a (if such a round exists). First assume any of r_1 or r_2, say r_i for $i \in \{1, 2\}$, is a rolled-back round, and define K' as the perturbation generated after r_i. Then it can be seen that c_i is greater than b_1 and less than or equal to a. As a result, a is not in the result set of J and hence it is in the result set of K. Also, by Observation 47, J compresses K' and so $K' \preceq J$. Therefore, if $c_i < a$, by Observation 36, $K' \preceq K$; otherwise K' is maximal at a (due to a very similar argument to Claim 2) and so again $K' \preceq K$. Also, K does not compress K' because it has a in its result set. So the claim is true.

Next, suppose neither r_1 nor r_2 is rolled-back. Consider the time t right after round r_1 if r_1 exists, or just at the beginning of the algorithm otherwise. Then, for every leaf l,

$\mathsf{min}[l]$ is the first element, or $\mathsf{prev}(\mathsf{min}[l])$ is an element promoted in J and so in K (because $\mathsf{diff}(J, K) = a > c_1$) with a value at most c_1. So, for every leaf l containing an element promoted with a value of a in K, $\mathsf{min}[l]$ is not bigger than a and hence $\mathsf{min}[\text{root}] \leq a$ which means the first round r' after time t has a target value at most a. Also, due to the choice of r_1, the target value of r' is no less than a. So, the target value of r' is a, that is, r' and r_2 are the same rounds. As for all leaves l containing an element promoted with a value of a in K, $\mathsf{min}[l]$ has an I-value of at most a, for all ancestors v of such leaves l, $\mathsf{min}[v] \leq a$ and thus, all such leaves are visited in round r_2. Therefore, because r_2 is not rolled-back, $J \preceq K$, which contradicts the assumption of the claim. □

Claim 4 *The perturbation J is compact.*

Proof Suppose J is not compact and so it is compressed by an aligned perturbation K. Define $a = \mathsf{diff}(J, K)$ and b as the smallest value in the result set of I bigger than a. Then by Definition 19, K is tight at b. Consider the last round r of target-value less than b and define a' as the target value of r. Then, $a \leq a' < b$ and hence, the perturbation J' generated after r is compressed by K and so r is rolled-back. On the other hand, because a is in the result set of J, the algorithm has not rolled-back the round of target value a. Thus, $a < a' < b$.

Because K is tight at b, there is an element e of K-value b with the property that if we changed its K-value, b would not be in the result set of K anymore. Note that by Definition 19, the set of leaves containing an element promoted with value a, a', or b in J' is a subset of leaves containing elements promoted with value b in K. Hence, the the leaf l containing e has elements f and f' promoted with values a and a' in J', respectively, while e is promoted with value b in J' (because a, a', and b all are in the result set of J'). The element f is promoted with value a in J as well. So, by Definition 19 $\mathsf{next}(\mathsf{next}(f)) > b$. This is a contradiction because two elements after f (not necessarily right after f) are promoted with values at most a' and b in J'. □

Claim 5 *The perturbation J is aligned.*

Proof Suppose J is not aligned and so there is a minimal promotion-preserving perturbation K of J for which there is an element promoted in K but not in J. We define $a = \mathsf{diff}(J, K)$ and S as the set of elements promoted with value a in K.

Consider the time t right after the last non-rolled-back round with a target value a' less than a (or the beginning of the algorithm if no such round exists in which case we define

$a' = -\infty$), and define b as the smallest value greater than or equal to a in the result set of I (if such a value exists). Observe that at time t, $\mathsf{min}[l]$ is e or an element smaller than e, for every leaf l with a member e in S, because neither e nor any element after e is promoted with a values less than a in J. Therefore, at time t, $\mathsf{min}[\text{root}] \leq a$. So due to the choice of t, $\mathsf{min}[\text{root}]$ at time t is a, which means a round r of target value a is executed at time t and visits all leaves with an element in S. So round r should be rolled-back with a round of target value b which visits all leaves visited in round r. Thus, every leaf with an element in S has an element promoted with value b in J, which is different from the element of l promoted with value a in K (because K is a promotion-preserving perturbation of J). So, because we just showed that at time t $\mathsf{min}[l]$ is e or an element smaller than e, for every leaf l with a member e in S, we have $\mathsf{min}_{\mathsf{next}}[l] \leq b$ at time t and so the round r could not be rolled-back. □

We proved J satisfies the conditions mentioned in Definition 21 and so the lemma is true.
□ □

Now we can see that for every leaf l, every time we call function Update on l in a non-rolled-back round, the gallop search stops right after an element that is promoted in the representative perturbation. Also, rolled-back rounds are replaced by a non-rolled-back round in which the same set of leaves (and possibly some additional leaves) are updated (condition 4 in Definition 19). Moreover, as is clear from the algorithm, at most two rounds are rolled-back by a single round. This results in the following corollary.

Corollary 50 *The number of update calls on each node v is no more than $3\phi(v)$.*

6.3.4 Analysis

We analyze the running time as the sum of running times on individual nodes.

Theorem 51 *For an input I, if t is the number of nodes in the expression tree, the running time of Algorithm 11 is $O\left(t^2 \log^2 t + A(I) + B_\phi\right)$.*

Proof We analyze the running time of all invocations of Algorithm 10 as the sum of running times on individual nodes of the tree. The processing in Algorithm 10 on a node v varies depending on the type of the node.

First suppose v is a leaf and consider non-rolled back rounds, which are $\phi(v)$ calls, each of which is a progressive gallop search, stopping right after an element e of v, which is promoted in the representative perturbation generated by the algorithm. In cases where $\mathsf{prev}(e)$ is also promoted in the representative perturbation (and so is in the same promoted group as e), the gallop search takes constant time. Also when e is not part of a floating promoted group, the gallop search will have a constant amortized running time. That's because if there are k elements in the promoted group at the end of the list of elements of v, for the first of these k elements, the gallop search takes $O(\log k)$ (because we run a gallop search from the end in parallel to the forward gallop search), and for $k-1$ other ones it takes $O(1)$. The remaining $g(v)$ gallop searches (where $g(v)$ was defined as the number of floating promoted groups of v) take $O\left(g(v) + \log \binom{\mathsf{size}(v)-g(v)}{g(v)}\right)$ time. Also for rolled-back rounds r, if r stops at a leaf l at an element e, then e or $\mathsf{next}(e)$ is promoted in the representative perturbation, so the gallop searches in those rounds do not change the running time by more than a constant factor.

When v is an intersection node, the computation is only $O(r)$ where r is the number of children of v (and the cost of recursion which we account for in the descending nodes). Finally, if v is a union node, the computation uses the "min-update" structure of section 6.3.2. Due to Lemma 46, the running time is $O\left(r^2 \log^2 r + \sum_{i=1}^{r} b_i \log \frac{m+b_i}{b_i}\right)$. As Corollary 50 states, the b_i's are within a constant factor of the share_ϕ values for the children of v and m is at most $3\mathsf{share}_\phi(v)$. Hence, the running time on a union node v is $O\left(r^2 \log^2 r + \sum_{i=1}^{r} \mathsf{share}_\phi(u_i) + \log \binom{\mathsf{share}_\phi(v)}{\mathsf{share}_\phi(u_i)}\right)$. So, as $\sum \mathsf{share}_\phi(u_i)$ for the children u_i of union nodes is in $O\left(\sum_{\text{leaves } l} \phi(l)\right)$, the theorem is correct. □

Chapter 7

The t-Threshold Problem

The algorithm we presented in Chapter 5 covered trees with any possible unary or binary operator[1] on sets. But how about operators defined on a higher number of operands? In this section we consider one possible such operator called “t-threshold”. Given k sets and an integer t, the *t-threshold* operator selects all members appearing in at least t of the k sets. By a *threshold* operator, we mean the t-threshold operator, for some value t.

The t-threshold problem was introduced by Barbay and Kenyon [6]. They investigated the problem of evaluating an expression tree of height one consisting of a single t-threshold operator, and presented a lower bound and optimal algorithms for cases where t is at least half of the number of operands.

In this chapter we consider trees of arbitrary heights, containing only threshold operators, which we call *threshold trees*. Note that union and intersection are also threshold operators. We look for an algorithm that is worst-case optimal for inputs with each possible signature.

7.1 Preliminaries

A generalization of the concepts we defined in Chapter 3 for inputs with threshold operators will be useful throughout this chapter. We extend the definitions for the functions cap and share, and the concept of proof labelings for inputs with threshold operator.

[1]Here we only consider operators in which membership of a value a in the result depends only on membership of a (and not any other value) in the operand sets.

The first question we try to answer is the maximum result set of a threshold input with a given signature. We first address this question for a tree of height one, and then we consider trees with arbitrary heights.

Lemma 52 *Consider a tree of height one with k leaves $l_1, \ldots, l_k$, where $\textsf{size}(l_1) \le \textsf{size}(l_2) \le \cdots \le \textsf{size}(l_k)$, and a t-threshold operator, for some t. The maximum size of the result set of the tree is*

$$\min_{1 \le p \le t} \left\lfloor \frac{1}{p} \sum_{i=1}^{k-t+p} \textsf{size}(l_i) \right\rfloor .$$

We will prove this lemma in Section 7.2.2, but to intuitively see why this is true, consider any p between 1 and t, define $m = k - t + p$, and consider the m leaves with the smallest set sizes. Each member of the result set should appear in at least t sets, and so it should appear in at least $t - (k - m) = p$ of these m sets. Thus, the size of the result set cannot exceed $\frac{1}{p}$ of total sizes of these m sets.

Inspired by Lemma 52, we define functions $\textsf{cap}$ and $\textsf{share}$ for nodes of threshold trees as follows. Similar to before, as we will show, $\textsf{cap}$ captures the maximum sizes of result sets of nodes and $\textsf{share}$ will define the maximum sizes of contribution sets of nodes.

Definition 24 *We define the function $\textsf{cap}$ over nodes of a threshold tree recursively as follows.*

- *Given a leaf l, we define $\textsf{cap}(l) = \textsf{size}(l)$.*
- *For an internal t-threshold node v with k children $u_1 \ldots, u_k$, where $\textsf{size}(u_1) \le \textsf{size}(u_2) \le \cdots \le \textsf{size}(u_k)$, we define $\textsf{cap}(v) = \min_{1 \le p \le t} \left\lfloor \frac{1}{p} \sum_{i=1}^{k-t+p} \textsf{cap}(u_i) \right\rfloor$.*

Moreover, for a node v, we define $\textsf{share}(v) = \min_u \textsf{cap}(u)$, where the minimum is taken over all ancestors u of v, including v.

Given a t-threshold node v, for some value t, we define t as the *threshold of* v. Moreover, if $u_1, \ldots, u_k$ are the children of v, we define the *maximized-threshold of* v as $\left\lfloor \frac{\sum_i \textsf{share}(u_i)}{\textsf{share}(v)} \right\rfloor$. The intuition behind maximized-threshold is that for a t-threshold node v with maximized-threshold value of t', we can determine members of sets such that each member of the contribution set of v appears in contribution sets of at least t' of its children. In other words, we increase the threshold-value of each node up to its maximized-threshold value such that the share and cap values for nodes do not change.

We next extend the definition of a proof labeling (Definition 9) to threshold trees.

Definition 25 *A function Λ is a proof labeling for a threshold signature if for any threshold node v with children $u_1, \ldots, u_k$, $\bigcup_{i=1}^{k} \Lambda(u_i) = \Lambda(v)$ and each element in $\Lambda(v)$ appears in at least t of $\Lambda(u_1), \ldots, \Lambda(u_k)$, where t is the threshold of v.*

A proof labeling Λ is *maximal* if for any node v, $|\Lambda(v)| = \mathsf{share}(v)$. Also, Λ is *canonical* if $\Lambda(\text{root}) = \{1, 2, \ldots, |\Lambda(\text{root})|\}$. For a proof labeling Λ and a node v, we may use the term Λ-*set of* v to refer to $\Lambda(v)$.

In this chapter, without loss of generality, we assume no union node (i.e. a node with threshold 1) is a parent of any other union node; otherwise we can just merge those two nodes. Also we assume every internal node has at least two children.

7.2 Lower Bound

7.2.1 Overview

The lower bound has a form similar to the one we proved for the union-intersection problem. We will prove the following two lower bounds:

- The first lower bound is the logarithm of the number of ways to select $\mathsf{share}(l)$ elements for each non-shallow leaf l out of its $\mathsf{size}(l)$ elements as the set of elements promoted from l. This means at least $\Omega\left(\sum_l \log \binom{\mathsf{size}(l)}{\mathsf{share}(l)} + \mathsf{share}(l)\right)$ comparisons. Note that the term $\Omega\left(\sum_l \mathsf{share}(l)\right)$ reflects that at least one comparison is needed on each of the $\mathsf{share}(l)$ elements promoted from l.
- The second lower bound is the logarithm of the number of proof labelings, that is the number of ways to set values of elements promoted from each leaf so that the result set of the root is $\{1, \ldots, \mathsf{share}(\text{root})\}$.

7.2.2 The Number of Proof Labelings

The number of proof labelings is the number of ways we can start from root, and at each node v we determine Λ-sets of the children of v so that the result of applying the operator associated with v on Λ-sets of children of v is $\Lambda(v)$. Of course, in order for this measure to be finite, we need to confine $\Lambda(\text{root})$ to a fixed set.

Let us focus on a single node v and count the number of choices we have at v. Suppose v is a t-threshold node, for some t, with children $u_1, \ldots, u_k$, and maximized-threshold $t_{\max}$. Without loss of generality we assume $\Lambda(v) = \{1, \ldots, \mathsf{share}(v)\}$ and $\mathsf{share}(u_1) \leq \mathsf{share}(u_2) \leq \cdots \leq \mathsf{share}(u_k)$. We propose a schema for determining the Λ-sets of the children of v and then we count the number of ways that this schema can work. The schema determines Λ-sets of the children of v such that:

- They are all subsets of $\{1, \ldots, \mathsf{share}(v)\}$.
- For each $1 \leq j \leq \mathsf{share}(v)$, j appears in the Λ-sets of at least $t_{\max}$ children of v.
- For each child u_i, $\Lambda(u_i)$ has $\mathsf{share}(u_i)$ members.

Then clearly Λ is a proof labeling.

To setup the schema, we create $t_{\max} + 1$ *baskets* out of the $\sum_i \mathsf{share}(u_i)$ members of Λ-sets (whose values are yet to be determined).[2] Each of the baskets will be of size $\mathsf{share}(v)$, except the last one whose size is at most $\mathsf{share}(v)$ (and can be zero). The members in each basket will have different values, so, for the case of the first $t_{\max}$ baskets, the members of each basket will cover the whole set $\{1, \ldots, \mathsf{share}(v)\}$. Here is the intuition behind baskets: We have $\mathsf{share}(v)$ values in $\Lambda(v)$, each to be appear in at least $t_{\max}$ sets. So we can roughly say that we have $t_{\max}$ copies of $\Lambda(v)$ to be distributed among Λ-sets of children of v, and one possible extra partial copy containing elements appearing in more than $t_{\max}$ sets. Each basket will represent one of these "copies".

Algorithm 12 specifies how the members of the Λ-sets are divided between different baskets. Note that in this algorithm we specify how many members of each Λ-set are going to be in each basket, but the algorithm does not specify the actual values of these members; that's the non-deterministic part of the schema. The baskets partition each

[2]Note that members of different Λ-sets hare are considered distinct "objects" even if they have the same value. So we can talk about $\sum_i \mathsf{share}(u_i)$ members of Λ-sets even though some of them may have same values.

Algorithm 12: Determining sizes of Λ-subsets.

```
for $1 \leq b \leq t_{\max}$ do
    the entire Λ-set of the $b$th biggest child (the $b$th biggest according to sets sizes,
    which is $u_{k-b+1}$) is allocated to the $b$th basket.
end
Initialize $b = 1$ and $i = 1$;
while $b \leq t_{\max} + 1$ and $i \leq k$ do
    if the $b$th basket already has $\mathsf{share}(v)$ members then
        increment $b$
    else
        Assuming there are $p$ unallocated members remaining in $\Lambda(u_i)$ and the $b$th
        basket already has $q$ members, allocate $\min\{p, \mathsf{share}(v) - q\}$ members of
        $\Lambda(u_i)$ to the $b$th basket;
        if all members of $\Lambda(u_i)$ are allocated then
            Increment $i$
        end
    end
end
```

Λ-set into a number of subsets which we call Λ-subsets (in other words, all members of a Λ-set that belong to the same basket create one Λ-subset).

Next we show how the schema determines actual values in Λ-sets non-deterministically. A Λ-subset is *partial* if it is not equal to its Λ-set, that is, if its Λ-set is partitioned into multiple Λ-subsets. We define the following order on Λ-subsets: the Λ-subsets are ordered by basket number, and within the same basket, partial Λ-subsets come before non-partial ones, and partial Λ-subsets of a basket are ordered by their child index. A Λ-subset S is *restricted by a* Λ*-subset* T (intuitively meaning that the choices we make for members of T restrict our choices for members of S) if T comes before S in the order we mentioned, and S and T are in the same basket, or belong to the same child of v. We start from the first Λ-subset in this order, and for each Λ-subset S, we choose the actual values in S so that they are distinct from values assigned previously to Λ-subsets of the same basket or the same child as S (i.e. the Λ-subsets that restrict S).

Next we present a lower bound on the number of choices we have for determining elements of Λ-subsets. We define the *cost of a* Λ*-subset* S as $\log \binom{\mathsf{share}(v)}{|S|}$ (please note that $|S|$ is fixed even when the actual members of S are yet to be determined). We show that the logarithm of the total number of choices is within a constant factor of the total cost of Λ-subsets. Note that due to the order in which we constructed the Λ-subsets, partial Λ-subsets are only restricted by prior partial Λ-subsets, not by any non-partial Λ-subset. So we first estimate the number of choices we have when we determine members of these sets.

Lemma 53 *The logarithm of the number of ways we may choose members of partial Λ-subsets is at least $\frac{1}{4}$ of their total costs.*

Proof Consider the sequence $u_{l_1}, \ldots, u_{l_p}$ of children of v whose Λ-subsets are partial, in order, and define P_i as the sequence of Λ-subsets of u_{l_i} ordered by basket number. We also define P_0 as the empty sequence. It suffices to show that for any i, $1 \leq i \leq p$, the logarithm of the number of choices we have for determining Λ-subsets in P_i and P_{i-1} is at least half of their total costs.

To prove this claim for P_i, for some i, suppose $P_i = S_1, \ldots, S_r$, and, when $i > 1$, $P_{i-1} = T_1, \ldots, T_q$. Note that S_j, for $j > 1$ is only restricted by $S_1, \ldots, S_{j-1}$ Also S_1 is only restricted by T_q if $i > 1$ and T_q and S_1 are in the same basket; otherwise S_1 is not restricted at all. Thus, when $i = 1$ or T_q and S_1 are not in the same basket, the number of choices for $S_1, \ldots S_p$ is the number of ways to select disjoint subsets of sizes $|S_1|, \ldots, |S_p|$ from $\mathsf{share}(v)$ elements, whose logarithm is at least $\frac{1}{2}\sum_{S \in P_i} \log \binom{\mathsf{share}(v)}{|S|}$ by Lemma 9.

Now we consider the case where $i > 1$ and S_1 and T_q are in the same basket B. B is the first basket containing a Λ-subsets of u_{l_i} and so, as Λ-subsets of u_{l_i} are partial, B is not the basket number $t_{\max} + 1$. Therefore, the first Λ-subset allocated to this basket in Algorithm 12 is the whole Λ-sets of one of the $t_{\max}$ children w with biggest Λ-sets. Other than the Λ-set of w, there are only two other Λ-subsets in B: T_q and S_1. Hence, T_q is the first Λ-subset of B whose members are determined and so, the number of choices when selecting members of T_q was

$$\binom{\mathsf{share}(v) - \mathsf{share}(u_{l_{i-1}}) + |T_q|}{|T_q|}.$$

Since $u_{l_{i-1}}$ has partial subsets and so is not among the $t_{\max}$ children with biggest Λ-sets, $\mathsf{share}(u_{l_{i-1}}) \leq \mathsf{share}(w) = \mathsf{share}(v) - (|T_q| + |S_1|)$. Thus, the number of choices when selecting T_q was at least $\binom{|T_q|+|S_1|}{|T_q|} = \binom{|T_q|+|S_1|}{|S_1|}$. Also, the number of choices when selecting S_1 was $\binom{\mathsf{share}(v)-|T_q|}{|S_1|}$. So, the number of choice when selecting S_1 and T_q in total was at least $\binom{|T_q|+|S_1|}{|S_1|} \times \binom{\mathsf{share}(v)-|T_q|}{|S_1|} \geq \binom{\mathsf{share}(v)}{|S_1|}$. This means that the logarithm of the number of choices we had for selecting T_q plus the logarithm of the number of choices we had for selecting $S_1, \ldots S_p$ is at least the logarithm of the number of ways to select disjoint subsets of sizes $|S_1|, \ldots, |S_p|$ out of $\mathsf{share}(v)$ members, which is again at least $\frac{1}{2}\sum_{S\in P_i}\log\binom{\mathsf{share}(v)}{|S|}$, by Lemma 9. □

Next we obtain a lower bound on the number of choices if we also consider non-partial Λ-subsets.

Lemma 54 *For any basket B, the logarithm of the number of choices for members of Λ-subsets of B is at least half of the total cost of Λ-subsets of B minus the total cost of partial Λ-subsets of B.*

Proof Suppose $S_1, \ldots, S_p$ are Λ-subsets of B in the order we defined. It is easy to see that the only Λ-subset of B that can be restricted by Λ-subsets of other baskets is S_1 and that this may happen only when S_1 is partial. Thus, if S_1 is not partial, then the only restriction on the choices for the members of the S_i's is that the S_i's should be disjoint and of pre-defined sizes. Thus, by Lemma 9, the logarithm of the number of choices is at least half of their total costs.

Now consider the case where S_1 is partial. The logarithm of the number of choices when determining members of S_i, for $2 \le i \le p$, is

$$\log\binom{\mathsf{share}(v)-\sum_{j=1}^{i-1}|S_j|}{|S_i|}.$$

Therefore, the logarithm of the number of these choices plus the cost of S_1 is

$$\sum_{i=1}^{p}\log\binom{\mathsf{share}(v)-\sum_{j=1}^{i-1}|S_j|}{|S_i|},$$

which is the number of ways to choose disjoint sets of sizes $|S_1|, \ldots, |S_p|$ from a set of size $\mathsf{share}(v)$. Therefore, again by Lemma 9, the logarithm of the number of choices plus the cost of partial sets is at least half of total costs of Λ-subsets of B. □

We hence conclude from Lemmas 53 and 54 that the logarithm of the number of choices for Λ-subsets is at least a constant factor of the total cost of Λ-subsets. Moreover, each setting yields a distinct Λ function, because, given a function Λ constructed by the schema we described, we can compute values of Λ-subsets in the order Algorithm 12 introduces them. So, the logarithm of the number of canonical maximal proof labelings is at least a constant factor of the total cost of Λ-subsets. Also, the cost a Λ-set is not less than the total cost of its Λ-subsets. This proves the next theorem.

Theorem 55 *The logarithm of the number of canonical maximal proof labelings for a threshold signature is at least $\sum_v \log\binom{\mathit{share}(\mathit{parent}(v))}{\mathit{share}(v)}$.*

7.2.3 Proving the Lower Bounds

The First Lower Bound

To prove the correctness of the first lower bound, we can follow the exact same logic explained in Section 3.3. Basically, we choose and fix an arbitrary maximal canonical proof labeling Λ (which due to Theorem 55 exists). For each value $a \in \Lambda(\text{root})$, we create a subtree of the original tree, denoted by $T^{(a)}$, that consists of nodes v for which $\Lambda(v)$ contain a. Also, similar to Section 3.3, we divide the sequence of elements of each leaf l into $\mathsf{share}(l)$ regions of equal sizes (up to one element) each hiding one "crucial element" having a value $(a, 0, x)$, for some $a \in \Lambda(l)$, and some x. Then, using the technique described in Algorithm 1, we prevent the algorithm from touching any crucial member before touching at least $\frac{1}{\mathsf{share}(l)} \log \binom{\mathsf{size}(l)}{\mathsf{share}(l)}$ other elements of the region containing it.

The only place in Section 3.3 where we used the fact that the operators are simple set operators (we allowed union, intersection, delta and minus), is when we provided the strategy for the adversary to force the algorithm to touch all crucial members. In the following section titled "The Second Lower Bound", we prove that the game defined in Section 3.4 has a winning strategy for the adversary in the case of threshold trees. We note that when the adversary wins in the game, the algorithm had touched all crucial members. Thus, the first lower bound holds.

The Second Lower Bound

We use the same technique as in Section 3.4 to prove that the logarithm of the number of canonical tight proof labelings is a lower bound. We only need to show how to extend the strategy designed for the adversary in the game explained in Section 3.4 to work for threshold trees.

Given a t-threshold node v with k children, where $t = k$, the node v is in fact an intersection node and it is treated the same way it was treated in Section 3.4. In the case when $t < k$, we first treat it as a union node. Then the adversary starts "eliminating" its children. Once $k - t$ children are eliminated (so there are only t children remaining), then the adversary should treat v as an intersection node. As a result, in the input constructed, each element in the set assigned to v by the proof labeling will be promoted from exactly t children of v.

7.3 The Algorithm

The algorithm skeleton is similar to the algorithm presented in Section 6.3.1: we process all sets in parallel, each set from the smallest to the biggest member, and we keep an array min which shows a pointer to the next element to be processed in each leaf. So, elements before min[l], for a leaf l, are *processed* elements, and elements from min[l] onwards are *unprocessed*. The algorithm is run in a number "rounds", where we assume at the beginning of each round that no element among the processed elements can be added to the result, so all processed elements can be ignored. For each leaf l, we call the element pointed by min[l] at the beginning of each round the *active element of l* in that round.

We first present the algorithm for the simpler case of a threshold tree of height one. This is equivalent to the t-threshold problem investigated before by Barbay and Kenyon [6, 7]. We then generalize the algorithm to general threshold trees.

7.3.1 Trees of Height One

The algorithm is based on the following simple idea, which was also used by Barbay and Kenyon [6]: if we consider active elements of the leaves, nothing smaller than the tth smallest of them should be added to the result in the future because it is not possible to find t elements with the same value of any of them among unprocessed elements (as a reminder, we do not consider elements before active elements of leaves). As such, at any point, by *the active cut-off*, we mean the value of the tth smallest active element of leaves.

We need a data structure that stores the k leaves, and at each stage can provide the list of all leaves with active elements not bigger than the active cut-off. This is the list of leaves to inspect to decide if the cut-off value is in the result set, and to determine the new cut-off value. Please note that such a list may contain more than t leaves if there are several leaves with active elements of the same value as the active cut-off. Algorithm 13 provides a high level description of the algorithm, given such a data structure.

Now the question is how to design the aforementioned data structure in a way we can find the cut-off value and leaves with active elements not bigger than the cut-off value efficiently. Suppose we store the values of the active elements of the leaves in a balanced binary search tree. Then we have immediate access to the t smallest ones in $O(t)$ time, and for updating the active element of each leaf, we need to spend $O(\log k)$ comparisons. But the point is that, as Algorithm 13 shows, we need to extract the t smallest active elements, but we do not care if the data structure reports these t in sorted order or not, while in this data structure we are extracting them in sorted order. In other words, we are potentially

Algorithm 13: The algorithm for threshold trees of height 1.

```
while not finished do
    Select the sets with active element not bigger than the cut-off value;
    In each leaf, use gallop search, to decide if the cut-off value is in the set, and
    jump to the first element bigger than the cut-off value;
    If the number of the sets containing the cut-off value is at least $t$, add the cut-off
    value to the result set;
end
```

wasting $O(t \log t)$ comparisons, which is $O(\log t)$ per set. The question is that how we can address this inefficiency?

The idea is to replace the lower part with height at most $O(\log t)$ of this binary search tree with unsorted arrays of sizes $O(t)$. To be more precise, we select a number of internal nodes as *array nodes* satisfying these conditions:

- The number of leaves in the subtree rooted at each array node is between t and $2t$.
- No array node is a descendant of another array node.
- Each leaf is a descendant of an array node.

Then, we replace the subtree rooted at the array nodes with arrays storing leaf values (not necessary in sorted order). Note that there are possibly multiple ways to choose array nodes.

Now let us explain how we can use this data structure. If we look at the leftmost leaf of the tree, we can find an array containing between t and $2t$ smallest members. Using a linear selection algorithm [14] we can find the member rank t in this array. Then we look into this array and next leaves to find all other occurrences of the member rank t in this data structure. This is all done in $O(t + m)$ time where m is the number of occurrences of the member rank t.

Next we explain how we update the tree. We first remove elements that just got updated from the tree. By doing this one or more leftmost leaves may completely be removed from the tree, and also one additional leaf may end up with less than t elements, in which case we merge that leaf with its next leaf. As a result of this process, the tree may need re-balancing (please refer to the standard insert/delete algorithm for balanced trees), in which case we do so. Then, for each leaf being updated, we find the position where it should be inserted and we insert it. If the destination array exceeds the $2t$ limit, we split it into two, and a new node is inserted and the tree is rebalanced if needed. To ensure that rebalancing of the tree when doing insertions or deletions does not change the running time asymptotically, we use B$^+$-trees of order 4 or more as the form of the balanced binary tree; then we can spread the balancing cost at each internal node among nodes added or removed under that node, and this way each leaf is "charged" a constant amount.

7.3.2 General Trees

The algorithm for general trees at the high level is very similar to the algorithm we presented in Section 6.3.1. We generalize the min array to all nodes in the tree, where for each node v the min array stores the next potential member of the contribution set of v. Then, each round is implemented as a recursive top-down function (called "update") on nodes of the tree which tries to skip elements with values less than min[root] and find the next potential member of the result set. At each round the recursive function *visits* (i.e. is called on) only a subset of nodes. For a node v, by a v-round we mean a round in which v is visited.

Algorithm 14: update(v, k): boolean

```
if $v$ is a leaf then
    /* using gallop search:                                    */
    $e$ := the first element of $v$ with value more than $k$;
    $\mathsf{min}[v] = \mathsf{val}(e)$;
    return true iff $\mathsf{val}(\mathsf{prev}(e))$ equals $k$;
else  /* Suppose $v$ is a $t$-threshold node                   */
    Select the children $u$ with min$[u] \leq k$;
    forall the each selected child $u$ do
        Update($u, k$);
        Update the value of min$[u]$ in the data structure;
    end
    Set min$[v]$ equal to the $t$th smallest min$[u]$, for children $u$ of $v$;
    return true if at least $t$ of calls to Update for children of $v$ returned true;
end
```

The implementation of the "update" function is presented in Algorithm 14. Here, when "update" is called on an internal threshold node v, the behavior of the algorithm is similar to the case of a tree of height 1, when v was the root. For the case of trees of height 1, we built and maintained the data structure that stores children u of v partially ordered by values of their active elements, and this data structures allowed us to select the ones with t smallest active element values. The generalization for any internal node of general threshold trees, will be the same data structure that stores children u of v partially ordered by $\min[u]$ instead. Then, as explained in Algorithm 14, we use this data structure to select children that need update, and at the end, we select the tth smallest min-values.

7.4 Analysis

The idea is to bound the number of times we visit each node by the maximum size of the contribution set of the node in inputs with the same signature. To do this, we define the perturbation generated by the algorithm, the same way it was defined in Section 6.3.3. Please note that here we never roll-back the rounds, and so we are referring to the simple definition provided at the beginning of Section 6.3.3. The perturbation will have the same signature as the original input, and the number of times a leaf is selected is the number of elements promoted from it in the perturbation.

Considering a leaf l, the number of times l is selected is the number of elements promoted from l in the perturbation which is at most $\mathsf{share}(l)$, thus the time we spend in gallop searches in l is at most $\mathsf{share}(l) \log \frac{\mathsf{size}(l)}{\mathsf{share}(l)} = O\left(\log \binom{\mathsf{size}(l)}{\mathsf{share}(l)}\right)$. Therefore, the total time we spend in gallop searches is bounded by the first lower bound.

Next we show that the time we spend in updating the tree is bounded by the sum of the first and the second lower bounds.

Lemma 56 *For a t-threshold node v with children $u_1, \ldots, u_n$ and a value k, the time spent at v when update(v, k) is called is within a constant factor of*

$$O\left(\sum_i \mathsf{share}(u_i) + \sum_i \log \binom{\mathsf{share}(v)}{\mathsf{share}(u_i)}\right).$$

Proof The time we spent at node v is in three parts:

1. Finding the nodes with min-values less than or equal to k.

2. Updating the data structure after recursive calls on children of v.

3. Finding the tth smallest min-value at the end.

We discuss each of these three separately. We define Λ as the function that assigns each node v the contribution set of v in the perturbation generated by the algorithm.

We first investigate the first item in the list. It can be observed that $\mathrm{Update}(v, k)$ is called only if $\min[v] \leq k$, which means there are at least t children of v with min-value at most k. To select these children, we consider array-nodes of the tree from left to right, and for each one in this order, we go over all children u of v in the array-node and select ones with $\min[u] \leq k$. If there is any child in the array-node for which $\min[u] > k$, we do not proceed to the next array nodes. Since size of each array-node is at most $2t$ and at least t nodes are selected, the time consumed in this part is proportional to the number of children selected.

To estimate the updating cost (second item in the list), we consider the cost of inserting each child in the data structure separately. We first prove the average position of insertion of a child u_i in the tree is $t_{\max}\frac{|\Lambda(v)|}{|\Lambda(u_i)|}$ and then we come up with the update cost.

To estimate the average position of insertion of a child u_i, we suppose u_i is inserted at position x of the tree at round r, and suppose r' is the first round after r that visits u_i. We define the *fan-out* of a v-round as the number of children of v visited during that round. The sum of fan-outs of rounds between r and r' is at least x. On the other hand, we know u_i is visited in total $|\Lambda(u_i)|$ times out of $|\Lambda(v)|$ times that v is selected, and the sum of fan-outs of all rounds is $\sum_i |\Lambda(u_i)| \leq \sum_i \mathsf{share}(u_j) = t_{\max}|\mathsf{share}(v)|$. Thus, the average of the positions of insertion of u_i is at most $t_{\max}\frac{|\mathsf{share}(v)|}{|\Lambda(u_i)|}$.

Now let's find the cost of an insert at a position x. The leaf where u_i should be inserted will be $\Theta(\frac{x}{t_{\max}})$ leaves away from the left most leaf of the tree. Therefore, we can find that leaf in time proportion to $\log\frac{x}{t_{\max}}$. So the total cost is at most within a constant factor of $|\Lambda(u_i)|\left(1 + \log\frac{|\mathsf{share}(v)|}{|\Lambda(u_i)|}\right)$ which is in $O\left(\mathsf{share}(u_i) + \log\binom{\mathsf{share}(v)}{\mathsf{share}(u_i)}\right)$ as $|\Lambda(u_i)| \leq \mathsf{share}(u_i)$.

Finally, as explained before, the time we spend for finding the t smallest min-value m at the end is proportional to the number of children of v with min-values less than m, which is at least the number of children of v that are visited the next time v is visited. Therefore, the running time in this part is proportional to $\sum_i |\mathsf{share}(u_i)|$. □

Lemma 56 proves a bound on the time we spend in internal nodes. To handle special case of shallow leaves, we need to make an exception when the root is a union node. In such cases, we just solve the problem for each child of the root independently, and then we compute the union of result sets using the technique described in Section 4.1.1. Then, by

Corollary 21, the running time in the root is $O(\sum_u \log \binom{\mathsf{share}(\mathrm{root})}{\mathsf{share}(u)})$, where the sum is over children of the root.

To wrap up the running time analysis, we need to sum up the running time mentioned in Lemma 56 for all internal nodes, except the root when it is a union node, plus the running time we mentioned for the root when it is a union node. The summation is a constant factor of $O\left(\sum_{u \in V} \log \binom{\mathsf{share}(\mathsf{parent}(u))}{\mathsf{share}(u)} + \sum_{u \in U} \mathsf{share}(u)\right)$, where V is the set of all nodes except root, and U is the set of all nodes except root and shallow leaves. Note that a union node may not be a parent of another union node, and for non-union internal nodes v, $\mathsf{share}(v) \le \frac{1}{2}\sum_u \mathsf{share}(u)$, where sum is over children u of v. Therefore, $\sum_{u \in U} \mathsf{share}(u) \le 2\sum_{l \in L} \mathsf{share}(l)$. Thus, the total running time is $O\left(\sum_{u \in V} \log \binom{\mathsf{share}(\mathsf{parent}(u))}{\mathsf{share}(u)} + \sum_{l \in L} \mathsf{share}(l)\right)$, which is not more than sum of the first and the second lower bounds.

Theorem 57 *The algorithm is worst-case optimal for inputs with each possible signature.*

Chapter 8

Further Thoughts and Conclusion

8.1 Possible Extensions

8.1.1 Repeated Sets in Expressions

The reason we do not allow sets being repeated in the expression is that if we do so, the problem gets readily intractable. The next theorem illustrates this problem. We defining *the cost of a signature* S as the smallest value m such that there is an algorithm that can solve the problem with at most m comparisons on any input with signature S.

Theorem 58 *Given any operator* $\pi \in \{-, !, \Delta\}$*, unless* $\mathcal{P} = \mathcal{NP}$*, there are no constants* c_1 *and* c_2 *and polynomial time algorithm* $\mathcal{A}$ *such that for any input* I *containing only operators* $\{\cap, \cup, \pi\}$*, the number of comparisons* $\mathcal{A}$ *performs on* I *is at most* $c_1 s + c_2$*, for* s *the cost of the signature of* I*.*

Proof We present a reduction of the SAT problem to this problem. Consider a SAT expression on n variables $x_1 \ldots, x_n$.We define a set expression E in which all sets are of size one. There is a main set a, and for each variable x_i, there is a set a_i. The idea is that x_i is true if and only if $a_i = a$. We create a set expression S from E by replacing operators $\wedge$ and $\vee$ with $\cap$ and $\cup$ respectively, and replacing any term of the form x_i, for some i, with a_i, and replacing any term of the form $\bar{x}_i$, for some i, with one of $!a_i$, $a - a_i$, or $a \Delta a_i$, depending on whether $\pi =!$, $\pi = -$, or $\pi = \Delta$, respectively. The whole expression will be $\mathcal{E} = a \cap E$. It is easy to see that given this expression as input to any algorithm, if the original SAT expression cannot be satisfied, the result of $\mathcal{E}$ is empty, regardless of the values of set a and sets a_i. So in such cases, the cost of the signature is just zero.

Now assume there are constants c_1 and c_2 and an algorithm $\mathcal{A}$ such that $\mathcal{A}$ solves any given input with at most $c_1 s + c_2$, where s is the cost of the input. Then we obtain an algorithm for solving SAT for a given expression as follows. We build the expression E as explained, and run the algorithm on E. For any query the algorithm makes on variables, we try all three possible answers to the query, so exploring a tree of possibilities on relative values of members of sets in E. If on any of the paths we explore in this tree algorithm asks more than c_2 questions, we know the cost of the input is more than zero, thus the input SAT expression is satisfiable and we can output "Yes". Otherwise, we let the algorithm finish running on all possibilities we generated (which are not more than 3^{c_2}) and if for any of them the algorithm generated a non-empty solution we output "Yes"; otherwise we know every input with same signature as E has an empty solution and thus the SAT expression is not satisfiable; hence we can output "No". This way we have solved SAT in polynomial time, which is impossible unless $\mathcal{P} = \mathcal{NP}$. $\square$

Note that the proof of Theorem 58 does not prove anything if we restrict the problem to the inputs with only union and intersection operators. It can be proven that for inputs with only union and intersection operators, one can obtain a polynomial time algorithm solving the problem with at most $\log n$ times more comparisons than the cost of the input.

It would be also interesting to think of parameterized complexity or approximation algorithms for the problem when repeated sets are allowed. For example, what can be achieved if we restrict the number of repetitions to a parameter k? Let's consider the union-intersection problem, and see how the problem can be solved by an optimal number of comparisons using the techniques we developed in the thesis.

We first discuss the lower bound. Consider the game we presented in Section 3.4. We defined a graph on leaves of the expression tree, showing pairs of leaves that algorithm knows how they compare. The algorithm starts from an initial state where the graph is empty, and then by a sequence of comparisons, the algorithm moves to states with some edges in the graph. However, if we allow some sets to repeat in the expression, it means the game is not started from a clean state; from the very beginning there are some edges in the graph showing which leaves are equal. So it might be that some of invariants mentioned in that section to be violated from very beginning. If that's not the case and all invariants hold, as we proved, the adversary has a winning strategy, thus, the second lower bound still holds. We here focus on this type of inputs. The first lower bound still needs some treatment. In the bound mentioned in Theorem 12, there is a term for each leaf. When a set is repeated, different terms that correspond to the same set need to be combined. As an example if the same set of size s is repeated in leaves with contribution limits of sizes $t_1, \ldots, t_r$, the contribution of this set to the formula mentioned for lower bound is $\sum_i (t_i + \log \binom{s}{t_i})$. We should replace this with a single term of $t + \log \binom{s}{t}$, where t is the total number of values appearing in contribution sets of these r leaves in the specific proof labeling we are considering.

Now lets discuss the algorithm part. We use the algorithm presented in Chapter 6, which is as explained in the chapter is worst-case optimal. The only change we make to the algorithm is that when the same set appears more than once in the tree, once we do gallop search in one leaf, we update the min-values in other leaves corresponding to the same set as well. As a result, we do not repeat the gallop-search on the same range for multiple leaves corresponding to the same set. The running time consists of two terms, one used to match the first lower bound, one used to match the second lower bound. The term that was supposed the first lower bound still matches it (due to the change we mentioned) and we are doing optimal there. But for the term matching the second lower bound, we could be off by a factor of k.

8.1.2 Unsorted Sets

Throughout the thesis we required the input sets to be pre-sorted and represented by a data structure that shows this sorted order. In this section we argue that with a small modification to the algorithm in Chapter 4, we may process inputs in which some sets are sorted and some are not in a worst-case optimal way. Here we define the fact that each set is pre-sorted or not as a part of the signature of the input and, again we look for an algorithm that works optimally in the worst case among inputs of the same signature.

The idea is that given an input I with some unsorted input sets, we replace each unsorted set of size s with a union node with s leaves of size 1, containing the s members of the set, and this way we obtain an "equivalent" input J. Clearly I and J have the same solution and so the algorithm works correctly with the same running time for I and J.

The only remaining issue is to show that the same lower bound we proved for the signature of I holds for the signature of J. To see why it is true, the reader may confirm that the only difference between the two signatures is that for the signature of the original input, no member is allowed to be repeated in an unsorted set while in its union version there is no such restriction enforced by the signature. The signature of the original input is giving this extra information to the algorithms before they make any comparisons. Now if we review the way we proved the lower bound, when we construct inputs in which the worst case happens, the leaf sets that are directly unioned with each other are disjoint. In other words, all the lower bounds still hold even if we restrict the problem to the inputs where the sets that are directly unioned with each other are disjoint. So the algorithm runs in optimal time for these inputs.

8.1.3 Order-Independent Adaptive Algorithm

In Chapter 6 we developed an algorithm that could take advantage of nice properties of the order of elements in the sets to solve the problem faster for instances where elements are not completely shuffled in sets. In "easy instances", similar elements were next to each other in input sets and could be "eliminated" together.

One may think of the problem in the case where the ordering of elements in sets does not have such nice properties and so those kind of adaptive algorithms do not function much better than worst-case optimal algorithms presented in earlier chapters. An example is when the key used for sorting elements in sets is a hash function.

Let us formulate the problem in such situations. Given two inputs I and J with the same signature, I is a *permutation* of J if there is a one-to-one mapping between the set of values appearing in I and the set of values in J such that a value is in a set in I if and only if the corresponding value is in the same set in J. In fact two inputs are permutations of each other if the only difference between them is the global order used to sort elements of sets in each input. Then, we look for an algorithm that, given any input I, works optimally in the average case (or in the worst case) on all permutations of I.

Here we limit the problem to inputs that are the intersection of a number of sets. The reader may verify that a class of permutations of an input with k sets can be identified by 2^k numbers representing the sizes of intersections of any number of input sets. Given an input I chosen randomly from the class of permutations of an input with k sets, one can look at a small fraction of elements of each set, and from these elements, estimate the size of the intersection of any subset of the k sets.

The idea then is to come up with an optimal ordering $p_1, \ldots, p_k$ of input sets and compute the intersection in that order, that is first to compute $S_{p_1} \cap S_{p_2}$, then take the intersection of the result with S_{p_3}, and so on. Suppose $R_i = \bigcap_{1 \leq j \leq i} S_{p_j}$ and we want to compute $R_i \cap S_{p_{i+1}}$. Evaluating this intersection in the average case can be done in a way better than the naive intersection algorithms we have seen before: for every member e of R_i, we already know the position of e in S_{p_j} for $j < i$ and thus we have some idea about the approximate position of e in S_{p_i}. Define b such that S_{p_b} has the biggest size among $S_{p_1}, \ldots S_{p_i}$, and let l be the index of e in S_{p_b}. Then we can start looking for e in $S_{p_{i+1}}$ from position $\left\lfloor l \frac{|S_{p_{i+1}}|}{|S_{p_b}|} \right\rfloor$. Since the total order used to sort members of input sets is chosen uniformly random, we expect the algorithm to find the position of e in $S_{p_{i+1}}$ in time $\log(2 + \frac{|S_{p_b}|}{|S_{p_{i+1}}|})$.

The challenge here is how to find the optimum ordering $p_1, \ldots, p_n$. This does not look to be solvable in polynomial time, but we might be able to design a good approximation algorithm. This remains an open problem, but here we mention a result in a similar problem. As mentioned, the expected time spent for verifying membership of each element of $\bigcap_{1\leq j\leq i} S_{p_i}$ in $S_{p_{i+1}}$ is a logarithmic term based on the ratio of the sizes of the sets. If instead, the cost of this verification process was $O(1)$, the problem would be equivalent to the problem of pipelined set cover [37], which has a polynomial-time 4-approximation algorithm.

8.2 Conclusion

In this thesis we considered the problem of evaluating set expressions when the input sets are preprocessed and are pre-sorted. We first considered the complexity of evaluating the expression when it contains complement, union, intersection, difference, and symmetric difference. These expressions are the most general types of expressions with binary and unary operations. We gave an algorithm to evaluate such expressions in the comparison model.

We argued that the algorithm performs the optimal number of comparisons by giving a matching lower bound in the cases where the expression does not contain complement and the only difference operation is at the root of the corresponding expression tree. We conjecture that the algorithm is optimal over all set expressions. We showed that the first lower bound in section 3.3 applies to all types of set expressions. However, the second lower bound applies only to special types of expressions as stated. Proving the second lower bound for all types of expressions remains open.

Then, we proposed an adaptive algorithm to evaluate a given set expression consisting of unions and intersections over sorted sets. We partitioned the set of instances into finely-sized difficulty classes according to the level of interleaving of values of the sets. The algorithm proposed has asymptotically optimal running time in each individual difficulty class.

The study of this problem in the (word) RAM model remains the subject of future work. We potentially might be able to improve the running time by a factor of $w^{1-o(1)}$, as it was possible for the simple case of intersection problem [13]. Extension of the operators to difference and complement also remains open.

References

[1] S.G. Akl and H. Meijer. Parallel binary search. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):247–250, April 1990.

[2] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 400–408. Springer Berlin Heidelberg, 2004.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[4] R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval*, SPIRE'05, pages 13–24, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] J. Barbay, A. Golynski, I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284 – 297, 2007. The Burrows-Wheeler Transform.

[6] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 390–399, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[7] J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono, editors, *Algorithms and Computation*, volume 2906 of *Lecture Notes in Computer Science*, pages 575–584. Springer Berlin Heidelberg, 2003.

[8] J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Algorithms*, 4(1):4:1–4:18, March 2008.

[9] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14:7:3.7–7:3.24, January 2010.

[10] J. Barbay, R. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Conference on Experimental Algorithms*, WEA'06, pages 146–157, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] J.L. Bentley and A.C.C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[12] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson. A time-optimal multiple search algorithm on enhanced meshes, with applications. *Journal of Parallel and Distributed Computing*, 22(1):113–120, 1994.

[13] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In Takeshi Tokuyama, editor, *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 739–750. Springer, 2007.

[14] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448 – 461, 1973.

[15] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[16] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.

[17] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[18] WH Burge. Sorting, trees, and measures of order. *Information and control*, 1(3):181–197, 1958.

[19] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In *Proceedings of the International Symposium on Algorithms*, SIGAL '90, pages 251–260, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[20] E. Chiniforooshan, A. Farzan, A. López-Ortiz, and M. Mirzazadeh. Evaluation of general set expressions: An adaptive approach. *submitted*.

[21] E. Chiniforooshan, A. Farzan, and M. Mirzazadeh. Worst case optimal union-intersection expression evaluation. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming*, ICALP'05, pages 179–190, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] E. Chiniforooshan, A. Farzan, and M. Mirzazadeh. Evaluation of general set expressions. In *Proceedings of the 19th International Symposium on Algorithms and Computation*, ISAAC '08, pages 366–377, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA '00: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[24] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Revised Papers from the 3rd International Workshop on Algorithm Engineering and Experimentation*, ALENEX '01, pages 91–104, London, UK, UK, 2001. Springer-Verlag.

[25] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.

[26] R. M. Fano. The transmission of information. Technical Report 65, Research Laboratory of Electronics, M.I.T, Cambridge, Mass., 1949.

[27] W. Fernandez de la Vega, A. M. Frieze, and M. Santha. Average-case analysis of the merging algorithm of hwang and lin. *Algorithmica*, 22(4):483–489, 1998.

[28] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[29] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[30] F. K Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, 1(2):145–158, 1971.

[31] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[32] G. Lee, M. Park, and H. Won. Using syntactic information in handling natural language quries for extended boolean retrieval model, 1999.

[33] J. Li, B.T. Loo, J.M. Hellerstein, M.F. Kaashoek, D.R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 207–215. Springer Berlin Heidelberg, 2003.

[34] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. *World Wide Web*, 9(4):369–395, 2006.

[35] M. I. Mauldin. Lycos: design choices in an internet search service. *IEEE Expert*, 12(1):8–11, 1997.

[36] M. Mirzazadeh. Adaptive comparison-based algorithms for evaluating set queries. Master's thesis, School of Computer Science, University of Waterloo, 2004.

[37] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. *Database Theory-ICDT 2005*, pages 83–98, 2005.

[38] W. Pugh. A skip list cookbook. Technical report, University of Maryland at College Park, College Park, MD, USA, 1990.

[39] V. Raman, L. Qiao, W. Han, I. Narang, Y.L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 773–784, New York, NY, USA, 2007. ACM.

[40] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Databases*, pages 553–564. VLDB Endowment, 2005.

[41] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *PVLDB*, 2(1):838–849, 2009.

[42] Z. Wen. Parallel multiple search. In *Parallel Processing Symposium, 1991. Proceedings., 5th International*, pages 114–119, Apr-2 May 1991.

[43] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.