Worst Case Optimal Union-Intersection Expression Evaluation[⋆]

Ehsan Chiniforooshan, Arash Farzan, Mehdi Mirzazadeh[⋆⋆]

David R. Cheriton School of Computer Science, University of Waterloo

Technical Report CS-2008-02

February 8, 2008

[⋆] A preliminary version of this work has been published in the proceedings of International Colloquium on Automata, Languages and Programming(ICALP) 2005 [4].

[⋆⋆] Email addresses: {`echinifo,afarzan,mmirzaza`}`@cs.uwaterloo.ca`

Abstract. We consider the problem of evaluating an expression consisting of unions and intersections of some sorted sets in the comparison model. Given the expression and the sizes of the sets, we are interested in the worst-case complexity of evaluating the expression in terms of the sizes of the sets. We assume the sets in the given expression are independent. We show a lower bound on this problem and present an algorithm that matches the lower bound asymptotically.

1 Introduction

In this paper, we study the problem of evaluating a set expression consisting of a number of union and intersection operators. Sets are known to be sorted and we also assume that sets are independent (that is no set occurs more than once in the expression). Although the worst case complexity in terms of the collective size of the entire input is straightforward, 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.

The problem arises in the context of evaluating search queries in text database systems; most text search engines maintain a set $S(w)$, for each word w, consisting of documents that contain w [1,8,11]. Thus, answering to 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 complicated if the queries are automatically generated [7].

Different variations of the problem have been studied before. The simplest case which is finding intersection or union of two sets is equivalent to the problem of merging two ordered sets of sizes m and n, studied by Hwang and Lin [6]. They present an algorithm that matches the information theoretic lower bound $\lceil \log \binom{m+n}{n} \rceil$. Note that this bound does not allow exhaustive listing of the entire output. They choose sorted arrays as the format of the input and a list of cross pointers between arrays as the output format. Later Brown and Tarjan [2,3] and Pugh [10] showed how data structures such as AVL-tree, B-tree, or skip-list can be used as the format of the input and the output.

Later, Demaine, López-Ortiz, and Munro [5] studied a more general case in which the expression involves more than two sets. The expressions they considered, were limited to union or intersection of a number of sets; that is operations are either all unions or all intersections. 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 $\frac{f(I)}{D(I)}$ among all inputs I of size n, where $f(I)$ is the running time of the algorithm on I. This problem was generalized by Mirzazadeh [9] to general expressions consisting of both union and intersection operators.

Neither of the adaptive algorithms that we mentioned work optimally in the worst case in terms of the sizes of the input sets. In this paper, we consider the worst case complexity as mentioned. We present a lower bound and then an evaluation algorithm that matches the lower bound.

The rest of this paper is organized as follows: In Section 2 we give some definitions and preliminary observations. In Section 3, Theorem 4, we present our lower bound and finally, in Section 4, the optimal algorithm is described.

2 Definitions and Preliminaries

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 formally define an *input* as a pair (T, Γ), where T and Γ are defined as follows. T is an *union-intersection tree* representing the expression: every internal node v is assigned a union or an intersection operator $\pi(v)$ and each leaf v of T corresponds to an input set and is assigned an integer $\mathsf{size}(v)$. We call T the *signature* of the input I. Also, Γ is an *assignment function* that assigns an ordered set of size $\mathsf{size}(v)$ to each leaf v. For an internal node v with k children $u_1, \ldots, u_k$, we denote by $\Gamma(v)$, the union or the intersection of $\Gamma(u_1)$, $\Gamma(u_2)$, $\ldots$, and $\Gamma(u_k)$, depending on the operator assigned to v. By the *result* of an input (T, Γ) we mean the set $\Gamma(\mathrm{Root}(T))$. We denote the set of nodes of a tree T and the set of leaves of T

by V_T and $\mathsf{leaves}(T)$, respectively. Without loss of generality, we assume that every internal node has at least two children, and that the operator assigned to every internal node other than the root differs from the operator assigned to its parent.

In this paper, we focus on the *comparison-based algorithms* which are those that, for any input $I = (T, \Gamma)$, use only comparisons in the input sets to compute the result. In our model, the algorithm has oracle access to Γ, which means that the algorithm reads the signature of the input and can later submit queries of the form (x, y) to the oracle, where x and y are members of the input sets. Then, the oracle informs the algorithm with the comparative values of x and y, that is, the algorithm is told whether x is less than, equal to, or greater than y according to Γ. In such situations we say x and y are *touched* by the algorithm. We show the interaction between the algorithm $\mathcal{A}$ and the oracle O on the input (T, Γ) by $\langle A, O(\Gamma)\rangle(T) = (q_1, r_1, \ldots, q_k, r_k, R)$ where q_i is the ith query of the algorithm, r_i is the response of O to the ith query, and $R = \Gamma(\mathrm{Root}(T))$ is the result. We 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 us 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 1. *Consider an input $I = (T, \Gamma)$ 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 v_i is a leaf of T, $1 \le b_i \le b'_i \le \mathsf{size}(v_i)$, for every $1 \le i \le n$, $\Gamma(v_j)[b'_j] < \Gamma(v_{j+1})[b_{j+1}]$, for every $1 \le j < n$, and $S = \cup_{i=1}^{n} \cup_{j=b_i}^{b'_i} \{\Gamma(v_i)[j]\}$.*

A leaf v of an expression tree T is a *shallow leaf* if v is a child of $\mathrm{Root}(T)$ and the root operation is union: $\pi(\mathrm{Root}(T)) = \cup$.

We define $\binom{s}{s_1, \ldots, s_n}$, when $s \le \sum_{i=1}^{n} s_i$, as the number of ways to select sets $X_1, \ldots, X_n$ of sizes $s_1, \ldots, s_n$, respectively, such that X_i's are subsets of a given set X of size s and $\cup_{i=1}^{n} X_i = X$. Note that this definitions matches definition of the traditional notation $\binom{s}{s_1,\ldots,s_n}$ when $\sum_{i=1}^{n} s_i = s$. Also for a union-intersection tree, we define functions ψ^* and ψ over the set of nodes of T as follows: for a leaf v we define $\psi(v) = \mathsf{size}(v)$. If v is an internal node and $u_1, \ldots, u_k$ is the list of children of v, we define $\psi(v) = \min_{i=1}^{k} \psi(u_i)$ when v is an intersection node, and $\psi(v) = \sum_{i=1}^{k} \psi(u_i)$, when V is a union node. In fact $\psi(v)$ is the maximum potential size of $\Gamma(v)$, considering the subtree rooted at v. Also, for every node v we define $\psi^*(v) = \min \psi(u)$, where the minimum is taken over all ancestors u of v, including v itself. Intuitively, $\psi^*(v)$ is the maximum "contribution" of $\Gamma(v)$ to the result at the root of the tree. We observe that the values of ψ and ψ^* for all nodes of an expression tree T can be evaluated in time $O(|V_T|)$.

Observation 1 *Suppose v is an internal node with k children $u_1, \ldots, u_k$. $\sum_{i=1}^{k} \psi^*(u_i) \geq \psi^*(u)$ if u is a union node and $\sum_{i=1}^{k} \psi^*(u_i) \geq 2\psi^*(u)$, if u is an intersection node.*

We present an algorithm such that for every signature T, the maximum running time of the algorithm, over all possible inputs with the signature T, is asymptotically minized.

3 Lower Bounds

In this section, fixing an arbitrary union-intersection tree T, we present a lower bound on the maximum number of comparisons performed by any algorithm when it is run on inputs with the signature T. For this purpose, we design an adversary $\mathcal{B}$ that for any given algorithm $\mathcal{A}$ and any signature T, as the algorithm $\mathcal{A}$ proceeds and compares members of the input, $\mathcal{B}$ fixes comparative values of members and responds to $\mathcal{A}$. In this manner, an assignment function Γ is constructed gradually and we make sure that there is always an assignment

Γ such that the responses of $\mathcal{B}$ to $\mathcal{A}$ are consistent with it. For two members x and y, if some certain conditions (which we define later in this section) hold, we say x and y are *similar*. We empower the oracle by assuming that when a query (x, y) is submitted, in addition to comparative values of x and y, $\mathcal{A}$ is informed of whether x and y are similar or not. It is clear that any lower bound for algorithms working in this new model is a lower bound for algorithms working in the comparison model as well.

Fixing a set O_T of size $\psi^*(\mathrm{Root}(T))$, $\mathcal{B}$ responds queries so that O_T becomes the final result. We spread the elements of O_T over the nodes of T such that every vertex v is labeled by a subset of O_T of size $\psi^*(v)$ in a manner that for every union (intersection) vertex v, the union (the intersection) of labels of its children is the label of v.

Let us define the previously mentioned labeling more formally. For convenience, rather than using real numbers, we will use triples of integers for representing members of our sets. Triples are compared to each other according to their lexicographic order. We define the set O_T as $\{(1,0,0),(2,0,0),\ldots,(m,0,0)\}$ where $m = \psi^*(\mathrm{Root}(T))$. Given a triple $x = (i, j, k)$, we call i, j, and k the first, the second, and the third coordinates of x, respectively.

Definition 2. *Given a signature T, $\Lambda : V_T \mapsto 2^{O_T}$ is a* **proof labeling** *for T if it has the following properties: First, $\Lambda(Root(T)) = O_T$. Second, for every vertex $v \in V_T$, $|\Lambda(v)| = \psi^*(v)$. Third, for a non-leaf node v is with children $u_1, \ldots, u_k$, if v is a union node: $\cup_{i=1}^{k} \Lambda(u_i) = \Lambda(v)$, and if v is an intersection node: $\Lambda(u_i) = \Lambda(v)$, for $1 \le i \le k$.*

The adversary $\mathcal{B}$ chooses a proof labeling Λ arbitrarily from all possible labelings. Then, $\mathcal{B}$ divides the sequence of members of every leaf v of T into $\psi^*(v)$ *regions*, each of sizes $\left\lfloor \frac{\mathsf{size}(v)}{\psi^*(v)} \right\rfloor$ or $\left\lceil \frac{\mathsf{size}(v)}{\psi^*(v)} \right\rceil$. For a leaf v and integers i and a, if the ith member of $\Lambda(v)$ is $(a, 0, 0)$, then the ith region of v is called an *a-region*. For any a and any a-region $\mathcal{R}$, $\mathcal{B}$ sets the first coordinates of all members of $\mathcal{R}$ to a at the beginning. Thus, given a member x of an a-region and a member y of a b-region such that $a \neq b$, whenever a query (x, y) is submitted, $\mathcal{B}$ can answer the query without knowing anything about the second and the third coordinates.

For any region $\mathcal{R}$, the second coordinate of exactly one element of $\mathcal{R}$, which is called the *key member of* $\mathcal{R}$, will be zero. The strategy is to determine the second coordinates of triples of a region $\mathcal{R}$ in such a way that $\mathcal{A}$ does not touch the key member of $\mathcal{R}$ before touching at least $\log |\mathcal{R}|$ members of $\mathcal{R}$ where $|\mathcal{R}|$ denotes the length of $\mathcal{R}$. The second coordinates of members of a region are all distinct and the third coordinates of non-key members are zero. Moreover, the third coordinates of the key members are determined in such a way that $\mathcal{A}$ needs to touch all key members (actually we will prove a stronger fact).

We now explain the strategy of determining second coordinates of the triples, which is essentially an adversary to binary searching. For every region $\mathcal{R}$ we consider a variable $\mathcal{S}$ storing a subsequence in $\mathcal{R}$, initially $\mathcal{R}$. At any point, the following condition will hold: The second coordinate of every member in $\mathcal{R} \setminus \mathcal{S}$ is already fixed, the second coordinate of every member of $\mathcal{R}$ placed before members of $\mathcal{S}$ is at most $-|\mathcal{S}|$, and the second coordinate of every member of $\mathcal{R}$ 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 $\mathcal{S}$, $\mathcal{B}$ sets its second coordinate to zero. 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 Figure 1, and deletes them from $\mathcal{S}$. Then, by touching a member of a region $\mathcal{R}$ of size n, the length of $\mathcal{S}$ is reduced to at most $\lfloor \frac{n}{2} \rfloor$. Since the value of 0 is not assigned to the second coordinate of any member unless $|\mathcal{S}| = 1$, $\log |\mathcal{R}|$ members of $\mathcal{R}$ have already been touched at the time the key member of $\mathcal{R}$ is being touched. Whenever a member is touched in which the second coordinate is not determined before, before attempting to answer the query, $\mathcal{B}$ determines the second coordinate according to the method we described here. Therefore, we have the following theorem.

Fig. 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 $-(|\mathcal{S}|-1)$, $-(|\mathcal{S}|-2)$, $\ldots$, $-(|\mathcal{S}|-i)$ 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 $i-1$, $i$, $\ldots$, $|\mathcal{S}|-1$ 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;
```

Theorem 1. *If all key members of $L \subseteq \mathsf{leaves}(T)$ are touched by $\mathcal{A}$ in $\langle \mathcal{A}, \mathcal{B} \rangle(T)$, then $\mathcal{A}$ has submitted at least $\sum_{v \in L} \psi^*(v) \cdot \lg\lceil \frac{\mathsf{size}(v)}{\psi^*(v)} + 1 \rceil$ queries.*

From now on, when we talk about the strategy of $\mathcal{B}$ for responding a given query (x, y), we assume second coordinates of x and y have already been determined.

Given a query (x, y) if x and y are from two a-regions, for some a, and the second coordinate of one of x or y is non-zero (that is at most one of x and y is a key member), $\mathcal{B}$ has enough information to answer the query. We define two members x and y to be *similar* if x and y are key members of two a-regions, for some a. As noted before, $\mathcal{A}$ will be informed by $\mathcal{B}$ if x and y are similar upon comparing them. In Subsection 3.1 we show that $\mathcal{B}$ can respond to queries on similar members in such a manner that in the end for each member x, $\mathcal{A}$ knows which members are similar to x.

Theorem 2. *For any signature T and any deterministic comparison-based algorithm $\mathcal{A}$, if after an interaction $\langle A, B\rangle(T) = (q_1, r_1, \ldots, q_k, r_k, R)$, $\mathcal{A}$ knows all sets of similar members, then $k \geq \frac{1}{2}(l_1 + \log_6 l_2)$ where l_1 and l_2 are defined below, L is the set of non-shallow leaves of T, and $u_1, \ldots, u_k$ are children of v in the expressions.*

$$l_1 = \textstyle\sum_{v \in L} \psi^*(v) \cdot \lg\left(\frac{\mathsf{size}(v)}{\psi^*(v)} + 1\right)$$
$$l_2 = \textstyle\prod_{v:\pi(v)=\cup} \binom{\psi^*(v)}{\psi^*(u_1), \ldots, \psi^*(u_k)}$$

Proof. For any proof labeling Λ that $\mathcal{B}$ fixes, since every non-shallow leaf v has an intersection ancestor, any member of $\Lambda(v)$ appears in $\Lambda(u)$ for at least another leaf u of the tree. Thus, any key member of a non-shallow leaf is similar to at least another member. Therefore, $\mathcal{A}$ has touched all key members of non-shallow leaves of T in $\langle \mathcal{A}, \mathcal{B}\rangle(T)$. Thus, using Theorem 1, we have $k \geq l_1$.

Moreover, for any member of the result R, $\mathcal{A}$ is aware of all similar members to that member (as an assumption in the description of the theorem; to be proved in Section 3.1). This means that $\mathcal{A}$ has enough information to figure out what proof labeling $\mathcal{B}$ has fixed and thus Λ can be expressed as a function of the sequence of responses of $\mathcal{B}$, $(r_1, \ldots, r_k)$. Hence, $(r_1, \ldots, r_k)$ is different for different Λ's. Using the fact that queries can have only six different values together with the fact that the number of possible proof labelings is l_2, we have $k \geq \log_6 l_2$. Putting these lower bounds for k together, we conclude that $k \geq \frac{1}{2}(l_1 + \log_6 l_2)$. □

3.1 The Game

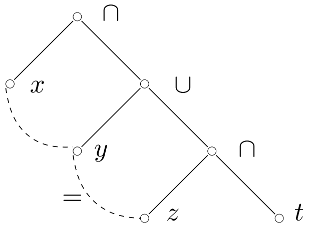
In this part we fix an a and focus on answering queries on key members of a-regions (which are similar). Let's focus on the subtree of T consisting of leaves of T that contain a-regions and their ancestors, and ignore the rest of the tree. We invent a two player game between $\mathcal{A}$ and $\mathcal{B}$, in which $\mathcal{A}$ submits queries between key members of a-regions until it knows whether a key member of an a-region appears in the result; $\mathcal{B}$ aims to avoid premature ending of the game before $\mathcal{A}$ has enough information for determining the set of members similar to a key member of an a-region. The configuration of the game at some specific point is a tuple $(\mathcal{T}, \mathcal{G})$ where $\mathcal{T}$ is a union-intersection tree in which internal nodes can have one or more children; the additional rule is that if the root is a union node, it should only have one child. Also, $\mathcal{G}$ is a graph defined on leaves of $\mathcal{T}$ storing the history of queries submitted. Throughout the rest of the paper, we assume implicitly that the history of queries are augmented to contain its transitive closure, so it represents not only the history of queries but whatever knowledge we can infer from submitted queries as well.

Each leaf is representing only one key member (the key member of its a-region, in our application) and thus we can view a query as comparing a pair of leaves of $\mathcal{T}$. Moreover, each edge of $\mathcal{G}$ is labeled with one of $<$, $=$, or $>$, demonstrating the response to that query. We assume $\mathcal{A}$ does not submit the same query more than once during the game. $v \in \mathsf{leaves}(\mathcal{T})$ is *$\mathcal{G}$-identical to* $u \in \mathsf{leaves}(\mathcal{T})$ if there is an edge labeled with $=$ between v and u in $\mathcal{G}$.

The game finishes when all possible queries are exhausted or as soon as the root in $(\mathcal{T}, \mathcal{G})$ has a *witness* which proves the key member belongs to the result. A witness of a node v is a subset of nodes where every two leaves belonging to S are $\mathcal{G}$-identical. Furthermore, it is defined recursively as follows. The only witness of a leaf v is $\{v\}$. A set S is a witness of a union node v if $v \in S$ and $S \setminus \{v\}$ is a witness of a child of v. For an intersection node v, S is a witness of v if $v \in S$ and every child u of v has a witness W_u such that $S = \{v\} \cup \bigcup_u W_u$. When the game finishes, $\mathcal{B}$ wins if $\mathcal{G}$ is connected which means the set of similar members to the key member is determined. Otherwise if the graph is disconnected $\mathcal{A}$ is the winner. We prove that $\mathcal{B}$ has a winning strategy.

Figure 2 illustrates a game in which $\mathcal{A}$ has won. However, in the illustrated

Fig. 2 An example of a game in which $\mathcal{A}$ has won. $\mathcal{A}$ has compared y and z and the answer has been $=$. Then, $\mathcal{A}$ compares x and y and regardless of the reply he can compute the value of the root without submitting any query about t.



game, $\mathcal{B}$ could answer in a smarter way and win the game. In this section, our aim is to prove that if $\mathcal{G}$ is initially an empty graph, then $\mathcal{B}$ always can win the game. We show a game with starting tree $\mathcal{T}$ and starting graph $\mathcal{G}$ by $(\mathcal{T}, \mathcal{G})$.

We define $(\mathcal{T}, \mathcal{G})$ *is reducible to* $(\mathcal{T}', \mathcal{G}')$ if $\mathcal{B}$ has a wining strategy in $(\mathcal{T}', \mathcal{G}')$, then $\mathcal{B}$ has a winning strategy in $(\mathcal{T}, \mathcal{G})$. In addition, we say that a vertex of $\mathcal{T}$ is *unmatched* if either it is an internal node or if it is the only vertex of its connected component in $\mathcal{G}$. Graph $\mathcal{G}$ is an *equivalency graph* when all the edges in $\mathcal{G}$ are labeled with $=$. A graph is called *v-small* when each outgoing edge (v, i) from node v is labeled with $v < i$, and every edge with no endpoint in v is labeled with $=$. If $\mathcal{G}$ is v-small, for some v, then $(\mathcal{T}, \mathcal{G})$ is a *v-small game.*

We define four operations which can be used to reduce a game instance to a smaller one. The operations have particular preconditions which should hold before the operation is applied.

Contract:

Preconditions:

1. $\mathcal{G}$ is an equivalency graph.

2. An internal vertex $u \in V_T - (\mathsf{leaves}(T) \cup \{\mathrm{Root}(T)\})$ has exactly one child v

Operation: If v is a leaf, u is deleted and v becomes a child of the parent of u. Otherwise, u and v are both deleted and children of v become the children of the parent of u.

Intersection-pruning:

Preconditions:

1. $\mathcal{G}$ is an equivalency graph.
2. v_1, v_2 children of an intersection node u are connected in $\mathcal{G}$ with an edge label $=$.

Operation: v_1 is deleted.

Union-pruning:

Preconditions:

1. $\mathcal{G}$ is v-small. v is a child of a union node u, and u has at least two children.

Operation: v is removed.

Intersection-deletion:

Preconditions:

1. $\mathcal{G}$ is v_1-small when $v_1, \ldots, v_k$ are children of an intersection node u other than root.
2. Each of $v_2, \ldots, v_k$ has a $\mathcal{G}$-identical node outside the set $v_2, \ldots, v_k$ (when $k \geq 2$).
3. The parent of u has at least two children.

Operation: $u, v_1, \ldots, v_k$ are removed.

Lemma 1. *if $(\mathcal{T}', \mathcal{G}')$ is obtained by application of one of the above four operations on $(\mathcal{T}, \mathcal{G})$, then $(\mathcal{T}, \mathcal{G})$ can be reduced to $(\mathcal{T}', \mathcal{G}')$.*

Proof. $(\mathcal{T}', \mathcal{G}')$ are obtained from $(\mathcal{T}, \mathcal{G})$ by removing some vertices from $\mathcal{T}$, and $\mathcal{G}'$ is essentially the relevant part of $\mathcal{G}$ on $\mathcal{T}'$; more specifically, $\mathcal{G}'$ is the projection of the query history on the existing vertices in $\mathcal{T}'$. We show a strategy for $\mathcal{B}$ to reduce $(\mathcal{T}, \mathcal{G})$ to $(\mathcal{T}', \mathcal{G}')$. Each query asked by $\mathcal{A}$ in $(\mathcal{T}, \mathcal{G})$ involves comparing two leaves x, y. The idea is that $\mathcal{B}$ simulates on $(\mathcal{T}, \mathcal{G})$, the query-answering strategy in $(\mathcal{T}', \mathcal{G}')$, and as new queries in $(\mathcal{T}', \mathcal{G}')$ arrive, we keep both $\mathcal{G}$ and $\mathcal{G}'$ updated.

Let us assume a new query (x, y) in $(\mathcal{T}', \mathcal{G}')$ arrives. If both x, y are present in $\mathcal{T}'$ then $\mathcal{B}$ imitates the answer to query in $(\mathcal{T}', \mathcal{G}')$ and also updates $\mathcal{G}'$ accordingly. The case where x or y has been deleted and not present in $\mathcal{T}'$ is more interesting; we first determine if x, y have $\mathcal{G}$-identical nodes in $\mathcal{G}'$; if both x, y have $\mathcal{G}$-identical nodes x', y' in $\mathcal{G}'$, the query (x, y) is answered as if query (x', y') is asked in $(\mathcal{T}', \mathcal{G}')$, and edges $(x, y) \in G$ and $(x', y') \in G'$ are updated accordingly. If only one node, say y, has a $\mathcal{G}$-identical node, $\mathcal{B}$ answers the query declaring that $x < y$. In this case the edge x, y in $\mathcal{G}$ is labeled accordingly, however no update in $\mathcal{G}'$ is necessary. The last case where none of x, y has $\mathcal{G}$-identical nodes in $\mathcal{G}'$ never happens, since in each operation at most one node that has no $\mathcal{G}$-identical node is deleted.

It is easy to check that $\mathcal{B}$ can use the previously mentioned strategy and answer queries with consistency. We now reason on the correctness of the reduction. We have to show that $\mathcal{B}$ losing in $(\mathcal{T}, \mathcal{G})$ implies that $\mathcal{B}$ also loses in $(\mathcal{T}', \mathcal{G}')$; Or equivalently, if $\mathcal{A}$ has a winning strategy in $(\mathcal{T}, \mathcal{G})$, then it also wins in $(\mathcal{T}', \mathcal{G}')$.

We argue by showing that there is a witness in $(\mathcal{T}, \mathcal{G})$ if and only if there is a witness in $(\mathcal{T}', \mathcal{G}')$. The manner by which we keep G and G' updated, guarantees us that two nodes are $\mathcal{G}$-identical if and only if they are $\mathcal{G}'$-identical (provided they exist in $\mathcal{G}'$). Hence, a witness in $(\mathcal{T}', \mathcal{G}')$ can be clearly extended to a witness in $(\mathcal{T}, \mathcal{G})$. The reverse is less trivial and one has to observe that the node v of a v-small game that is deleted in $(\mathcal{T}', \mathcal{G}')$ can never be part of a witness in $(\mathcal{T}, \mathcal{G})$. Thus, the projection of the witness of $(\mathcal{T}, \mathcal{G})$ is a witness of $(\mathcal{T}', \mathcal{G}')$.

If $\mathcal{A}$ has a winning strategy then two cases are possible. First, after submitting a number of queries $\mathcal{B}$ finds a witness W for $\mathcal{T}$ without making $\mathcal{G}$ connected. In this case, we know there also exists a witness for $(\mathcal{T}', \mathcal{G}')$. We argue that $\mathcal{G}'$ is also not connected and thus, $\mathcal{A}$ wins in $(\mathcal{T}', \mathcal{G}')$. It is easy to check that none of the above four operations– contract, intersection-pruning, union-pruning, and intersection-deletion – deletes an entire connected component in $\mathcal{G}$, so each connected component in $\mathcal{G}$ is also present in $\mathcal{G}'$ (although with possibly fewer vertices or broken apart into more than one component).

Second, the other way $\mathcal{A}$ can win is by exhausting all possible queries without giving a witness. In this case, since there is no witness in $(\mathcal{T}, \mathcal{G})$, none exists in $(\mathcal{T}', \mathcal{G}')$ either. Moreover, since all queries are exhausted in $(\mathcal{T}, \mathcal{G})$, queries in $(\mathcal{T}', \mathcal{G}')$ are exhausted as well. Hence, $\mathcal{A}$ also wins in $(\mathcal{T}', \mathcal{G}')$. □

Theorem 3. *Suppose that (T, G) is a game with the following properties:*

1. *G is an equivalency graph.*
2. *All children of every union node are unmatched.*
3. *Each intersection node has at least one unmatched child.*
4. *Every intersection node with an internal node child has at least two children.*
5. *If the root is a union node, it has only one child.*

Then, $\mathcal{B}$ has a winning strategy.

Proof. We use induction on the number of components of G plus the number n of leaves of $\mathcal{T}$ to prove the theorem. The base case $n = 1$ is trivial as G is complete when $n = 1$. We assume that no contract nor intersection-pruning operation is possible; otherwise, due to Lemma 1 we can repeatedly apply these operations, each time obtaining a smaller game, until no more operation is possible. It is easy to check that after these changes, still the five properties mentioned in the description of the theorem holds. After all these changes if $n = 1$ the problem is trivial as explained. So, assume $n > 1$ and thus by property 5 $\mathcal{T}$ has no shallow leaf.

Let us first prove that in $(\mathcal{T}, \mathcal{G})$ the game has not finished yet. For this purpose, we first prove that $(\mathcal{T}, \mathcal{G})$ does not have a witness. Assume to the contrary that W is a witness. A witness of size 1 is not possible because $\mathcal{T}$ does not have a shallow leaf, every internal node other than the root has at least two children, and $n > 1$. Therefore, there is no unmatched leaf in W because all leaves in W are $\mathcal{G}$-identical. This contradict properties 2 and 3 mentioned in the theorem which imply every witness has an unmatched leaf. Thus, $\mathcal{A}$ has not won yet and so it must submit a query. Also, still more queries can be submitted because due to properties 2 and 3 there is still an unmatched leaf and we assumed $\mathcal{T}$ has at least two leaves. So, by definition the game is not finished yet and hence $\mathcal{A}$ must submit a new query.

If the algorithm submits a query (u, v) in which u and v are $\mathcal{G}$-identical, $\mathcal{B}$ trivially answers $u = v$. Considering the first query (u, v) submitted by $\mathcal{A}$ such that u and v are not $\mathcal{G}$-identical, the adversary $\mathcal{B}$ responds to this query according to the following strategy. In the following, a *matched leaf* is a leaf that is not unmatched. Also, an unmatched leaf, depending on it having an unmatched sibling or not, is called a *normal-unmatched* or a *last-unmatched* leaf, respectively.

If the parent of u or the parent of v, say that of u, is a union node, the u is normal-unmatched, by property 2 and the fact that no contraction is possible. Then, $\mathcal{B}$ would answer u is less than v and then it obtains a u-small game in which preconditions of union-pruning(u) hold. So, by applying this operation we reduce the game to a smaller one in which the five properties mentioned in the theorem hold and thus $\mathcal{B}$ wins by induction.

Otherwise, if u or v, say u, is a last-unmatched leaf, we can observe that the second and the third preconditions for the operation intersection-deletion(p), for p the parent of u, hold (note that all siblings of u are in different components otherwise an intersection-pruning is possible). Now if $\mathcal{B}$ says u is smaller than v, $\mathcal{G}$ will get u-small and so we can apply intersection-deletion(p) to reduce the game to a smaller one.

Otherwise, $\mathcal{B}$ will say u and v are equal and then the number of components is reduced. Since neither u nor v is last-unmatched, after this response still property 3 holds and so by induction in the new game $\mathcal{B}$ can win. $\square$

Having proved Theorem 3, we know the adversary can respond to queries on key members of a-regions such that the algorithm knows all members similar to the key member of each a-region, for every $1 \leq a \leq m$, and also that $(a, 0, 0)$ is in the result of the root. Thus, Theorem 2 yields the following theorem.

Theorem 4. *For any signature T and any deterministic comparison-based algorithm $\mathcal{A}$, there is an input with the signature G such that $\mathcal{A}$ submits $\Omega(l_1 + \lg l_2)$ queries where l_1 and l_2 are defined in the same way as in Theorem 2.* $\square$

4 The Worst-Case Optimal Algorithm

In this section, we present our algorithm which matches the lower bound in Section 3. First we study two special cases separately; these special problems come in handy in solving the general problem.

The first special case involves evaluating union of a series of sets: $X_1 \cup X_2 \cup \ldots \cup X_k$. Hwang and Lin [6] studied this problem for $k = 2$. They showed to compute $A \cup B$, tight lower and upper bounds of $\Theta\left(\lg\binom{|A|+|B|}{|A|}\right)$ exist. We extend their result to values of $k > 2$. We Define $s_i = |X_i|$, for $1 \leq i \leq k$ and $s = \sum_{i=1}^{k} s_i$. We first show a non-optimal result in Lemma 2 and immediately improve it to the optimal result in Lemma **??**.

Lemma 2. *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 (non-optimal) time $O\left(\sum_{i=1}^{k} \lg\binom{s+s_{\max}}{s_i}\right)$.*

Proof. We use the follwoing algorithm to compute $\cup_{i=1}^{n} X_i$.

1. If there is only one set in $\mathcal{X}$, we are done; otherwise, we select the two smallest sets in $\mathcal{X}$, say X_a and X_b.
2. We compute $X_a \cup X_b$, using the trivial merge algorithm, with at most $|X_a| + |X_b|$ comparisons.
3. We replace X_a and X_b with $X_a \cup X_b$ in $\mathcal{X}$, and repeat the procedure from step 1.

Clearly, the worst case running time happens when all sets in $\mathcal{X}$ are disjoint. In this case, the above-mentioned algorithm works similar to Huffman coding; consider each set X_i as a symbol with frequency $|X_i| / \sum_{i=1}^{n} |X_i|$. Then, in each step of the algorithm, we select two elements with smallest frequencies and create a new element based on them. 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. So, the overall number of comparisons will be $\sum_{i=1}^{n} |X_i| h(X_i)$, where $h(X_i)$ is the depth of X_i in the corresponding Huffman tree. Also, Huffman trees have the property that the depth of a node with frequency p is at most $\lg(p^{-1}) + 1$. This means the number of comparisons in our algorithm is at most $\sum_{j=1}^{n} |X_j| \left(\lg(\sum_{i=1}^{n} |X_i| / |X_j|) + 1\right) = \sum_{i=1}^{n} s_i \left(\lg(s/s_i) + 1\right)$.

The following inequalities complete the proof:

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

□

Lemma 3. *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 (optimal) time $O\left(\sum_{i=1}^{k} \lg \binom{s}{s_i}\right)$.*

Proof. We separate out the largest set (say X_1), and apply Lemma 2 to compute $X_{2,k} = X_2 \cup \ldots \cup X_k$ in $O\left(\sum_{i=2}^{k} \lg \binom{s'+s'_{\max}}{s_i}\right)$, where $s', s'_{\max}$ are the sum and the maximum of set sizes taken over $2, \ldots, k$. Since $s' + s'_{\max} \leq s$, the time is $O\left(\sum_{i=2}^{k} \lg \binom{s}{s_i}\right)$.

We use the algorithm of Hwang and Lin [6] to compute the union of the largest set and the remaining sets ($X_1 \cup X_{2,k}$) in $O(\lg \binom{s}{s_1})$. Therefore, the overall time is $O(\sum_{i=1}^{n} \lg \binom{s}{s_i})$.

The next step is to show the bound we achieved in Lemma 3 is indeed optimal. That is we must show that $\sum_{i=1}^{n} \lg \binom{s}{s_i}$ is in $O(\lg \binom{s}{s_1, s_2, \ldots, s_n})$:

Lemma 4. *If $s \leq \sum_{i=1}^{n} s_i$, then $\sum_{i=1}^{n} \lg \binom{s}{s_i} = O(\lg \binom{s}{s_1, s_2, \ldots, s_n})$.*

Proof. We define $t_i = \min\left\{s, \sum_{j=1}^{i} s_j\right\}$ and $t'_i = \min\left\{s, \sum_{j=i}^{n} s_j\right\}$), for $1 \leq i \leq n$. We define $t_0 = t'_{n+1} = 0$.

We prove the lemma in a series of steps:

Step 1. $\binom{s}{s_1, \ldots, s_n} \geq \prod_{i=1}^{n} \binom{t'_i}{s_i}$**.**

Define a set $X = \{x_1, \ldots, x_s\}$. By induction on i we prove the number of ways to select subsets $X_1, \ldots, X_i$ of sizes $s_1, \ldots, s_i$ of X such that $|X - \cup_{j=1}^{i} X_j| \leq t'_{i+1}$ is great than or equal to $\prod_{i=1}^{j} \binom{t'_i}{s_i}$. The base case where $i = 0$ is trivial. We show that if sets $X_1, \ldots, X_{i-1}$ have been selected such that $|X_j| = s_j$ for every j, $1 \leq j \leq i-1$, and $|X - \cup_{j=1}^{i-1} X_j| \leq t'_i$, there are at least $\binom{t'_i}{s_i}$ ways to choose the set X_i of size s_i such that $|X - \cup_{j=1}^{i} X_j| \leq t'_{i+1}$. Set $Y = X - \cup_{j=1}^{i-1} X_j$. 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. Thus, the induction hypothesis for $i = n$ proves the claim.

Step 2. $\binom{t_i}{s_i}\binom{t'_i}{s_i} \geq \binom{s}{s_i}$**.**

We consider the set $X = \{x_1, \ldots, x_s\}$, and define $Y = \{x_1, \ldots, x_{t_i}\}$ and $Y' = \{x_1, \ldots, x_{s_i}, x_{t_i+1}, \ldots, x_s\}$. 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 which is $\binom{s}{s_i}$. We define the *result* of a subset pair (A, B) as the set $(A - Y') \cup (B - Y) \cup (Y \cap Y' \cap A \cap B)$. 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 X.

Step 3. $\frac{1}{2} \sum_{i=1}^{n} \lg \binom{s}{s_i} \leq \lg \binom{s}{s_1, \ldots, s_n}$**.**

Step 1 implies that $\lg \binom{s}{s_1,\ldots,s_n} \geq \sum_{i=1}^{n} \lg \binom{t'_i}{s_i}$. One can prove similarly that $\lg \binom{s}{s_1,\ldots,s_n} \geq \sum_{i=1}^{n} \lg \binom{t_i}{s_i}$. Therefore, $2 \lg \binom{s}{s_1,\ldots,s_n} \geq \sum_{i=1}^{n} \left(\lg \binom{t_i}{s_i} + \lg \binom{t'_i}{s_i} \right)$. Step 2 implies that $\lg \binom{t_i}{s_i} + \lg \binom{t'_i}{s_i} \geq \lg \binom{s}{s_i}$. These two facts together show that $\frac{1}{2} \sum_{i=1}^{n} \lg \binom{s}{s_i} \leq \lg \binom{s}{s_1,\ldots,s_n}$.

Step 3 directly implies the lemma. □

Lemma 3 and Lemma 4 together imply the following corollary:

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

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

The second special case has the form $Y \cap (X_1 \cup X_2 \cup \ldots \cup X_k)$, given that $|Y| \geq |X_i|$ for all i. This problem for the case where $k = 1$ (i.e. computing $Y \cap X$) has been studied and tight lower and upper bounds of $\Theta(|X| \lg \frac{|X|+|Y|}{|X|})$ already exist [6]. 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 $B[y]$. We consider a specific representation for members of Y such that the representation for each member y of Y also includes a point to $B[y]$; then we can access $B[y]$ in constant time. We initialize all the elements in it to false. We compute the intersection of each X_i with Y separately ($Y_i = Y \cap X_i$) in $O\left(\sum_{i=1}^{k} |X_i| \lg \frac{|X_i|+|Y|}{|X_i|}\right)$ time using Hwang and Lin's algorithm [6]. When Y_i's are all computed, we consider them one by one and for each Y_i, for all $y \in Y_i$, we set $B[y] = true$. Then we scan array $\mathcal{B}$ and return as output each element b such that $B[b]$ 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 all Y_i's. Also creating B in the beginning and scanning it in the end takes time $O(|Y|)$; therefore:

Theorem 5. *The result set of $Y \cap (X_1 \cup X_2 \cup \ldots \cup X_k)$ can be computed in $O\left(|Y| + \sum_{i=1}^{k} |X_i| \lg \frac{|X_i|+|Y|}{|X_i|}\right)$ time.* □

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, we are interested in computing $\Gamma(v) \cap U$, for a given "universe set" U. In the second type we are asked to compute $\Gamma(v)$. The procedures Compute (v, U) and Compute(v) (Figure 3) 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 somehow have processed the subtree rooted at u_i for some i, and have obtained $\Gamma(u_i)$. It makes perfect sense to pass $\Gamma(u_i)$ as a universe set to 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. As for Compute(v) it turns out that, for some nodes v, the size of the possible result of a node is less than any universe set we can possibly provide with in advance. In these cases we do not pass any universe set as it will not save any computation time.

Next, we investigate the correctness and the running time of the algorithm.

Theorem 6. *Every time that procedure* Compute$(v,\ U)$ *in Algorithm 1 is called, the precondition $|U| \leq \psi^*(v)$ holds (Line 1) and the procedure computes $\Gamma(v) \cap U$. Also, every time that procedure* Compute(v) *in Algorithm* 2 *is called, the precondition $\psi(v) = \psi^*(v)$ holds (Line 1) and the procedure computes $\Gamma(v)$.*

Proof. The fact that the procedures produce the right output is trivial by using an induction on the height of the tree.

Fig. 3 The general algorithm.

```
Procedure COMPUTE($v$, $U$);
1 // precondition: $|U| \leq \psi^*(v)$.
begin
    switch type of node $v$ do
2       case Leaf: return $\Gamma(v) \cap U$ ;
        case Union:
            foreach $u_i$ child of $v$ do
                if $\psi(u_i) < |U|$ then
3                   $X_i \longleftarrow Compute(u_i)$
                else
4                   $X_i \longleftarrow Compute(u_i, U)$
5           return $U \cap (X_1 \cup X_2 \cup \ldots \cup X_k)$
        case Intersection:
            $X \longleftarrow U$ ;
            foreach $u_i$ child of $v$ do
6               $X \longleftarrow Compute(u_i, X)$
            return $X$
end
```

Algorithm 1 Computing the intersection of U with the result set of the subtree rooted at v (i.e. $\Gamma(v) \cap U$).

```
Procedure COMPUTE($v$);
1 // precondition: $\psi(v) = \psi^*(v)$.
begin
    switch type of node $v$ do
2       case Leaf: return $\Gamma(v)$ ;
        case Union:
            foreach $u_i$ child of $v$ do
3               $X_i \longleftarrow Compute(u_i)$
4           return $X_1 \cup X_2 \cup \ldots \cup X_k$ ;
        case Intersection:
            $j \longleftarrow minindex(\psi(u_i))$ ;
5           $X \longleftarrow Compute(u_j)$ ;
            foreach $u_i$ child of $v$ do
                if $i \neq j$ then
6                   $X \longleftarrow Compute(u_i, X)$
            return X
end
```

Algorithm 2 Computing the result set of the subtree rooted at v (i.e. $\Gamma(v)$).

Thus we only show that preconditions mentioned in line 1 of Algorithm 1 and in line 1 of Algorithm 2 always hold whenever they are called.

In Algorithm 1, we have three recursive calls. The first one is in line 3: since $\psi(u_i) < |U|$ and, by precondition, $|U| \leq \psi^*(v)$, therefore $\psi(u_i) < \psi^*(v)$. By definition of $\psi^*(u_i)$, the latter inequality implies that $\psi^*(u_i) = \psi^(u_i)$ which means the precondition will hold. The second recursive call occurs in line 4; we know that $\psi(u_i) \geq |U|$ and since $\psi^*(v) \geq |U|$, we can deduce $\psi^*(u_i) \geq |U|$. Thus the precondition will hold. The last recursive call occurs in line 6; since v is an intersection node, $\psi(u_i) \geq \psi(v)$ for each i. By definition $\psi(v) \geq \psi^*(v)$; hence $\psi(u_i) \geq \psi^*(v)$. Since $\psi^*(u_i) = \min\{\psi(u_i), \psi^*(v)\}$, so $\psi^*(u_i) = \psi^*(v)$. By precondition $\psi^*(v) \geq |U|$, and therefore $\psi^*(u_i) \geq |U|$. As $|U| \geq |X|$, the precondition will hold.

Similarly in Algorithm 2, there are three recursive calls. The first one is in line 3; since v is of type union $\psi(u_i) \leq \psi(v)$ and by precondition $\psi(v) = \psi^*(v)$;

so $\psi(u_i) = \psi^*(u_i)$ which implies the precondition will hold. Second recursive call occurs in line 5; because v is of type intersection, $\psi(v) = \psi(u_j)$ for $j = \mathit{minindex}(\psi(u_i))$ where u_i's are children of v. By precondition $\psi(v) = \psi^*(v)$; so $\psi^*(u_j) = \min\{\psi(u_j), \psi^*(v)\} = \min\{\psi(u_j), \psi(v)\} = \psi(u_j)$, and hence the precondition will hold. The third and last recursive call occurs in line 6. we know that $\psi^*(u_i) = \min\{\psi(u_i), \psi^*(v)\} = \min\{\psi(u_i), \psi(v)\} = \psi(u_j)$, and $|X| \le \psi(u_j)$, so $|X| \le \psi^*(u_i)$. □

Next, we analyze the running times of the procedures by measuring 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 will be, of course, the sum of such processing times in nodes of the tree. It is easy to see that no computation is involved in intersection nodes. Here, we analyze two other types of nodes (i.e. leaf and union) separately:

Processing Time in Union Nodes: Line 5 is the only one in Algorithm 1 on which we spend some computing time. Also, in Algorithm 2, only line 4 involved computation. These two are exactly the special cases we studied in the beginning of this section. We can prove the following lemma.

Lemma 5. *Processing a union node v takes time of $O\left(\sum_{i=1}^{k} \psi^*(u_i) + \lg \binom{\psi^*(v)}{\psi^*(u_1),\ldots,\psi^*(u_k)}\right)$ where $u_1, \ldots, u_k$ are children of v.*

Proof. In Algorithm 1, the only line that we spend some time on computing is line 5. Since $|X_i| \le |U|$, the computation can be done in $O\left(|U| + \sum_{i=1}^{k} |X_i| \lg \frac{|X_i|+|U|}{|X_i|}\right)$, by Theorem 5. As $|U| < \psi^*(u_i)$, by the precondition, and $|X_i| \le |U|$, $|X_i| < \psi^*(u_i)$. Given that $|U| < \psi^*(v)$ and $|X_i| \le \psi^*(u_i)$, $|U| + \sum_{i=1}^{k} |X_i| \lg \frac{|X_i|+|U|}{|X_i|}$ is of $O\left(\psi^*(v) + \sum_{i=1}^{k} \psi^*(u_i) \lg \frac{\psi^*(u_i)+\psi^*(v)}{\psi^*(u_i)}\right)$. Finally, since by Observation 1, $\psi^*(v) \le \sum_{i=1}^{k} \psi^*(u_i)$ and the term $\lg \frac{\psi^*(u_i)+\psi^*(v)}{\psi^*(u_i)}$ is not less than one, we can eliminate the term $\psi^*(v)$. Thus, the processing time of line 5 in Algorithm 1 is $O\left(\sum_{i=1}^{k}\psi^*(u_i)\lg\frac{\psi^*(u_i)+\psi^*(v)}{\psi^*(u_i)}\right) = O\left(\sum_{i=1}^{k}\psi^*(u_i) + \sum_{i=1}^{k}\lg\binom{\psi^*(v)}{\psi^*(u_i)}\right)$ which is, by Lemma 4, of $O\left(\sum_{i=1}^{k}\psi^*(u_i) + \lg\binom{\psi^*(v)}{\psi^*(u_1),\ldots,\psi^*(u_k)}\right)$.

In Algorithm 2, only line 4 is important. Due to Corollary 2, the result can be computed in $O(s + \lg\binom{s}{|X_1|+\ldots+|X_k|})$ where $s = \sum_{i=1}^{k}|X_i|$. By precondition of this procedure $\psi^*(v) = \psi(v)$, for each child u_i of v, $\psi^*(u_i) = \min\{\psi(u_i), \psi^*(v)\} = \min\{psi(u_i), \psi(v)\} = \psi(ui)$. Therefore, $\psi^*(v) = \psi(v) = \sum_{i=1}^{k}\psi(u_i) = \sum_{i=1}^{k}\psi^*(u_i)$. Also, $|X_i| \leq \psi(u_i) = \psi^*(u_i)$ for every i. Thus, $\binom{\sum_{i=1}^{k}|X_i|}{|X_1|,\ldots,|X_k|} \leq \binom{\sum_{i=1}^{k}\psi^*(u_i)}{\psi^*(u_1),\ldots,\psi^*(u_k)} = \binom{\psi^*(v)}{\psi^*(u_1),\ldots,\psi^*(u_k)}$. Hence, since $s \leq \sum_{i=1}^{k}\psi^*(u_i)$ the running time is in $O\left(\sum_{i=1}^{k}\psi^*(u_i) + \lg\binom{\psi^*(v)}{\psi^*(u_1),\ldots,\psi^*(u_k)}\right)$. $\square$

We make a slight change in the algorithm to save time: in the case when the root of the whole tree is of type union, we take union using the algorithm in Corollary 1 instead of that in Corollary 2 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 5.

Lemma 6. *If the root is a union node, processing time in the root takes time of* $O\left(\lg\binom{\psi^*(root)}{\psi^*(u_1),\ldots,\psi^*(u_k)}\right)$ *where* $u_1, \ldots, u_k$ *are children of the root.* $\square$

Here we claim that the term $\sum_{i=1}^{k}\psi^*(u_i)$ in Lemma 5 is negligible when it is summed over all union nodes. In the sum, ψ^* 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 $\sum_{v\in S}\psi^*(v) \leq \sum_{v\in L}\psi^*(v)$ where L is the set of non-shallow leaves. This can be proved by writing the inequalities of Observation 1 for all nodes of the tree, and summing them all together.

Theorem 7. *Processing in union nodes and leaves takes time of*
$O\left(t + \sum_{v\in L}\psi^*(v) + \sum_{union\ node\ v}\lg\binom{\psi^*(v)}{\psi^*(u_1),\psi^*(u_2),\ldots,\psi^*(u_k)}\right)$ *where* L *is the set of non-shallow leaves and* t *is the time we spend in non-shallow leaves.* $\square$

Processing Time in Leaf Nodes: If v is a leaf, in line 2 in Algorithm 1, we compute the intersection of $\Gamma(v)$ and U. As a precondition, we know that $|U| < \psi^*(v)$ and also by definition that $\psi^*(v) \leq \psi(v) = \mathsf{size}(v)$, so $|U| < \mathsf{size}(v)$. In the second special case, Theorem 5, it is shown how to compute the intersection in time $O(|U| \lg \frac{|U|+\mathsf{size}(v)}{|U|})$. Since $|U| < \psi^*(v) \leq \mathsf{size}(v)$, the processing time is in $O\left(\psi^*(v) \lg \frac{\psi^*(v)+\mathsf{size}(v)}{\psi^*(v)}\right)$.

In line 2 of Algorithm 2, we simply return $\Gamma(v)$ which, by precondition, has size $\psi^*(v)$. In case v is a shallow leaf by the argument mentioned in Theorem 7, 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 Theorem 7.) Thus the following theorem holds:

Theorem 8. *In Algorithms 1 and 2, the time spent in each non-shallow leaf is* $O\left(\psi^*(v) \lg(\frac{\mathsf{size}(v)}{\psi^*(v)} + 1)\right)$ *and we spend no time in shallow leaves.* $\square$

We conclude from Theorems 4, 8, and 7 that our algorithm is optimal.

5 Conclusion

We studied the problem of evaluating an expression of sorted sets with union and intersection operands. Complexity of algorithms were measured in terms of the sizes of the input sets. We proved lower bounds on the worst case complexity of algorithms that can solve this problem, and later presented an algorithm that asymptotically matches the lower bound.

An immediate extension to this work is changing its format of input/output to a more appropriate format. Our assumption for the format of the input is lists of elements of sets, and the format of the output is a list of cross-references which specify the ranges of the elements. However, with a little effort, the format of both the input and the output can be changed to *balanced search trees*. More specifically, we choose B-trees. Adapting the lower bounds is straightforward; Theorem 4 will still hold without any modification. As for the upper bound, it is sufficient to show that we can handle the two special cases in Section 4 with the same time complexity, since the general algorithm only uses these two for computation. These two special cases can be dealt with in the same way Demaine et al. [5] handled B-tree representations of their input sets. It is easy to see that the extra work for assembling and disassembling the B-trees in their scheme does not affect our bounds in Corollary 2 and Theorem 5.

As a future work, one can consider expressions that can have operands of type complement besides those of type union and intersection.

6 Acknowledgments

The authors would like to thank Peyman Afshani, Alex Lópex-Ortiz, Reza Dorri, Narges Simjour, and anonymous referees for their useful comments.

References

1. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
2. Mark R. Brown and Robert E. Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.
3. Mark R. Brown and Robert E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.
4. Ehsan Chiniforooshan, Arash Farzan, and Mehdi Mirzazadeh. Worst case optimal union-intersection expression evaluation. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 179–190. Springer, 2005.
5. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *SODA '00: Proceedings of the eleventh annual*

ACM-SIAM symposium on Discrete algorithms, pages 743–752, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

6. 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.
7. G. Lee, M. Park, and H. Won. Using syntactic information in handling natural language quries for extended boolean retrieval model, 1999.
8. M.I. Mauldin. Lycos: design choices in an internet search service. *IEEE Expert*, 12(1):8–11, 1997.
9. Mahdi Mirzazadeh. Adaptive comparison-based algorithms for evaluating set queries, 2004.
10. William Pugh. A skip list cookbook. Technical report, University of Maryland at College Park, College Park, MD, USA, 1990.
11. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.