View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

Adaptive Algorithms for Weighted Queries on Weighted Binary Relations and Labeled Trees

by
Aleh Veraskouski

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

©Aleh Veraskouski 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Keyword queries are extremely easy for a user to write. They have become a standard way to query for information in web search engines and most other information retrieval systems whose users are usually laypersons and might not have knowledge about the database schema or contained data. As keyword queries do not impose any structural constraints on the retrieved information, the quality of the obtained results is far from perfect. However, one can hardly improve it without changing the ways the queries are asked and the methods the information is stored in the database.

The purpose of this thesis is to propose a method to improve the quality of the information retrieving by adding weights to the existing ways of keyword queries asking and information storing in the database. We consider weighted queries on two different data structures: weighted binary relations and weighted multi-labeled trees. We propose adaptive algorithms to solve these queries and prove the measures of the complexity of these algorithms in terms of the high-level operations. We describe how these algorithms can be implemented and derive the upper bounds on their complexity in two specific models of computations: the comparison model and the word-RAM model.

Keywords

Adaptive algorithms, keyword queries, binary relations, labeled trees, threshold set, pertinent set, non-increasing path subset.

Acknowledgements

I'm thankful to everyone who supported me during my master's studies. I would like to thank Professor J. Barbay, my supervisor, who guided me throughout my studies. I'm also thankful to Professor I. Munro and A. Lopez-Ortiz, my reviewers, for their useful comments and suggestions. And, certainly, I'm thankful to my family for the support and encouragement I experienced over the years.

Contents

1 Introduction **1**

2 Survey of Previous Work **2**
2.1 Adaptive Algorithms 2
2.1.1 Sorting 2
2.1.2 Convex Hull 5
2.2 Queries on Binary Relations 6
2.2.1 Intersection 6
2.2.2 Threshold 13
2.3 Queries on Labeled Trees 15
2.4 Data Structures 20
2.4.1 Binary Relations 20
2.4.2 Priority Queues 21

3 Weighted Queries on Weighted Binary Relations **23**
3.1 Basic Definitions 24
3.2 Threshold Set Problem 25
3.2.1 Problem Statement 25
3.2.2 Algorithm 26
3.2.3 Adaptive Analysis 29
3.3 Pertinent Set Problem 32
3.3.1 Problem Statement 32
3.3.2 Algorithm 32
3.3.3 Adaptive Analysis 34
3.4 Implementation 34

4 Weighted Path Subset Queries on Weighted Labeled Trees **36**
4.1 Basic Definitions 37
4.2 Threshold Path Subset Problem 38
4.2.1 Problem Statement 38
4.2.2 Algorithm 40
4.2.3 Adaptive Analysis 41
4.3 Pertinent Path Subset Problem 46

4.3.1 Problem Statement 46
4.3.2 Algorithm 46
4.3.3 Adaptive Analysis 48
4.4 Implementation 48

5 Weighted LCA Queries on Labeled Trees **50**
5.1 Basic Definitions 50
5.2 Threshold LCA Problem 52
5.2.1 Problem Statement 52
5.2.2 Algorithm 53
5.2.3 Adaptive Analysis 56
5.3 Implementation 61

6 Conclusion **63**
Bibliography 64

Appendices **69**

A Improvements on the algorithm for finding Smallest Lowest Common Ancestors (SLCAs) **69**

List of Figures

2.1 One iteration of the In-place Insertion Sort Algorithm. 3
2.2 An example of the Intersection Problem with the number of keywords $k = 3$ and the corresponding answer. 6
2.3 The Galloping Search Technique (searching for 85). 8
2.4 Interleaving perfectly vs. totally disjoin ranges of values in the sets. 11
2.5 An example of the Threshold Set Problem with the parameter $t = 2$ and the number of keywords $k = 3$ together with the corresponding answer. 14
2.6 An extract of the XML file ('books.xml') and the corresponding tree structure. 15
2.7 An example of XQuery for the XML structure shown in Figure 2.6 and the corresponding answer. 16
2.8 The meaningful structural relationship among the nodes. 17
2.9 An example of postings lists representation of a weighted binary relation on two sets. 20

3.1 An example of weighted association between elements of two sets and the corresponding weighted binary relation. 24
3.2 An example of a weighted query. 25
3.3 An example of the problem instance and the corresponding partition certificate for $\mu_R = 2$, $\mu_Q = 5$, and $t = 20$, where all three types of intervals are present. 29
3.4 Label movement during the execution of Algorithm 5 and Algorithm 6. . . . 31

4.1 An example of a simple file system. Each node represents a folder and contains the words associated with it, along with the weight of these associations. . . 37
4.2 The preorder enumeration of the nodes in the tree in Figure 4.1. 38
4.3 An example of weighted query. For the sake of the simplicity of the discussion, all the keywords have the same weight 1, but the algorithm works the same in the general case as well. 39
4.4 The encoding of the example in Figure 4.1 using a weighted binary relation. The null weights are noted by dots for better readability. 43

4.5 An example of the minimal partition-certificate for the weighted tree in Figure 4.4 and the weighted query in Figure 4.3 with $t = 5$. This partition-certificate has all three possible types of intervals. The alternation of the instance is $\delta = 4$. By symbols '$*$' we describe all the nodes that are descendants of the node associated with the current label (this label affects these nodes as well, unlike in the similar case on binary relations). 44

5.1 An example of the postorder enumeration of the nodes in a tree. 52
5.2 An example of the execution of the Algorithm 11 on the labeled tree provided in Figure 5.3. 55
5.3 An example of a multi-labeled tree. The gray nodes are in the 2-CA set and circled nodes are in the 2-LCA set built on the labels {a,b,c}. 57
5.4 The postorder of the nodes of the tree in Figure 5.3 and one of the minimal 2-intervals. 57
5.5 Three types of incoming nodes during filtering. 59
5.6 The global picture of the results of the lemmas. The set of all nodes that correspond to all minimal t-intervals is a subset of t-CA set but contains t-LCA set completely. The algorithm finds the corresponding node for each minimal t-interval and determines whether it is in the t-LCA set or not through filtering. 60

Chapter 1

Introduction

More and more information is produced and stored as digital documents. In order to use this information effectively, we need ways to selectively retrieve the documents without browsing the whole corpus of data. Researchers in the information retrieval field considered this problem and proposed a large number of techniques to do this.

Usually, a user needs to specify the information he is looking for using one of the formal query languages. Then, a search engine searches for those documents that are relevant to the specified query and display them to the user. This process narrows the scope of the information the user deals with and increases his effectiveness.

Simple keyword queries, in which all the keywords present in the query must be found in the retrieved documents, is one of the popular ways to make such requests. First, these queries are extremely easy to use and do not require the user to have any additional knowledge about the database schema or content. Second, they greatly facilitate the writing of schema-independent queries that can be used on different database schemas without being changed, which increase the reusability of the queries in the environments with databases of different schemas and after the database migration from one schema to another.

As keyword queries do not impose any structural constraints on the retrieved information, the quality of the obtained results is far from perfect. However, one hardly can improve it without changing the ways the queries are asked and the methods the information is stored.

The purpose of this thesis is to provide new ways of asking keyword queries and storing information in the database by adding weights to the existing solutions to improve the quality of the information retrieving. We consider weighted queries on two different data structures: weighted binary relations and weighted multi-labeled trees. We propose adaptive algorithms to solve these queries and prove the measures of the complexity of these algorithms in terms of the high-level operations. We describe how these algorithms can be implemented and derive the upper bounds on their complexity in two specific models of computations: the comparison model and the word-RAM model.

The rest of this thesis is organized as follows. In Chapter 2 we give an overview of the related work. In Chapter 3 we discuss weighted queries on weighted binary relations. In Chapter 4 and Chapter 5 we consider two different types of weighted queries on weighted labeled trees. We discuss our results and possible directions for future research in Chapter 6.

Chapter 2

Survey of Previous Work

In this chapter we give an overview of previous work related to the research in this thesis. We start from the notion of an adaptive algorithm and its adaptive analysis in Section 2.1, that we will use heavily for building and analysing our algorithms. Then, we explain the relevant work done on querying in binary relations in Section 2.2 and labeled trees in Section 2.3, which we are extending in the subsequent chapters. Finally, in Section 2.4 we describe how a number of abstract data types that we use later can be implemented in two different models of computations.

2.1 Adaptive Algorithms

2.1.1 Sorting

Worst-case analysis is a standard way to analyse the complexity of algorithms. One defines a function f that shows the dependence between n, the size of the *problem instance* or *input*, and $f(n)$, the complexity of the algorithm in the worst case among all problem instances of size n. Once this function is found, one can estimate the complexity of the algorithm on a particular instance and, more importantly, compare different algorithms for solving the same problem. We term this analysis the *classical worst-case analysis*.

Consider the *Sorting Problem*, where one needs to reorder the objects of a given sequence according to an order defined on them. For the sake of simplicity, we restrict ourselves only to a strict order, i.e. we assume that there is no equal objects in a sequence.

Definition 2.1.1 *Given a sequence of n distinct objects $S^{in} = (x_1, \ldots, x_n)$ from an ordered set, the* Sorting Problem *is to determine the permutation $\pi(i_1, \ldots, i_n)$ such that $\forall i, k, l < k \Rightarrow x_{i_l} < x_{i_k}$ and hence reorder S^{in} to produce the ordered sequence $S^{out} = (x_{i_1}, \ldots, x_{i_n})$.*

Below we show an example of using of the classical worst-case analysis and its drawbacks. We do this for the Sorting Problem because of its popularity, high intuitiveness, and extensive research done [27, 41].

Consider the well-known sorting algorithm, *Insertion Sort*. As show in Figure 2.1.1, at each step the *In-place Insertion Sort Algorithm* moves the next object of the input to the left until it gets the correct place in the already sorted part of the input. The algorithm does this by comparing the object with its left neighbor and exchanging them if they violate the correct ordering. After the first i iterations are done, the intermediate sequence contains the first i entries of the input array that are already sorted. After the algorithm has considered each object in the input, the sequence is completely sorted.

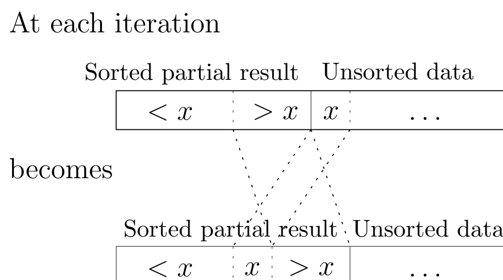


Figure 2.1: One iteration of the In-place Insertion Sort Algorithm.

As the In-place Insertion Sort Algorithm is based on comparing and exchanging objects, we can estimate its running time by counting the number of comparisons performed to solve a given instance. We assume that exchanges are free because here an exchange is always preceded by a comparison.

Consider two instances of the Sorting Problem on integers: $S_1^{\text{in}} = \{7, 6, 5, 4, 3, 2, 1\}$ and $S_2^{\text{in}} = \{7, 1, 2, 3, 4, 5, 6\}$. Both of them are of the same size $n = 7$ and have the same answer $S_0 = \{1, 2, 3, 4, 5, 6, 7\}$.

As S_1^{in} is in inverted order, the In-place Insertion Sort Algorithm starts comparing each object with its left neighbor and exchanges them, until the object appears in the leftmost position of the array. The algorithm performs $0+1+2+3+4+5+6 = 21$ comparisons. If we consider a similar instance of the arbitrary size n, i.e. if the input has the form $S_1^{\text{in}} = \{n, n-1, \ldots, 1\}$, the In-place Insertion Sort Algorithm performs $0+1+\ldots+(n-1) = \Theta(n^2)$ comparisons.

However, when the algorithm is processing S_2^{in}, it performs only two comparisons for each object, except the first one '7', for which it performs only one comparison. In total, the algorithm performs $1+2+2+2+2+2+2 = 13$ comparisons. In the case of a similar problem instance but of an arbitrary size n, $S = \{n, 1, \ldots, n-1\}$, the algorithm performs $2n - 1 = \Theta(n)$ comparisons.

The classical worst-case complexity of the In-place Insertion Sort Algorithm is $f(n) = \mathcal{O}(n^2)$ comparisons. However, while the algorithm reaches this quadratic bound when it deals with the problem instances of the first type S_1^{in}, it performs only a linear number of comparisons to process an instance of the second type S_2^{in}. This shows that classical worst-case analysis, which is based on the size of the instance, does not always correspond to the

number of operations the algorithm really performs to obtain the answer, and one needs a better way to analyse algorithms.

Such an observation was first made by Burge [21] in 1958. He introduced the *instance easiness* or *presortedness* of the sequence – a measure of existing order, a new way to classify problem instances and to build an adaptive analysis. After being considered by many other researchers, the concept of presortedness was formalized by Mannila [38] in 1985.

Here, we consider the *number of inversions* as a measure of presortedness of a problem instance. The *number of inversions* in an object sequence with an order defined on them is the number of pairs of objects that are in the wrong order:

$$Inv(X) = |\{(i,j) \mid 1 \leq i < j \leq n \texttt{ and } x_i > x_j\}|$$

According to the defined inversion measure of presortedness, all the problem instances of type S_1^{in} of size n have $Inv(X) = \Theta(n^2)$. On the other hand, all the problem instances of type S_2^{in} of size n have presortedness $Inv(X) = \Theta(n)$.

For both types of instances described above, the In-place Insertion Sort Algorithm performs $\Theta(Inv(X))$ comparisons, where a problem instance X has presortedness of $Inv(X)$. The same is true for all other instances of the problem, meaning that the complexity of the algorithm is linear in terms of the measure of presortedness.

A sorting algorithm is said to be *adaptive* with respect to a measure of presortedness if it sorts all sequences, but performs particularly well on those that have a high degree of presortedness according to the measure. That is, the more presorted the input is, the faster it should be sorted. Moreover, the algorithm should adapt without knowing the amount of presortedness in advance.

The number of inversions used in our adaptive analysis is a more adequate measure of an instance difficulty than the size of the problem that was used in the case of the classical worst-case analysis. This is also true for most other problems.

Much more research has been done on these issues including defining new measures of presortedness, analysis of existing and new algorithms with the help of these measures, carrying out experiments in real environments, and theoretical analysis of the dependencies between different measures. We refer the reader to the survey by Estivill-Castro and Derick Wood [27] as well as Petersson and Moffat [41] for a good survey of different measures of presortedness and a hierarchy of dependencies between them.

A *model of computation* is the definition of the set of allowable operations for any algorithm and their respective costs. Only after defining a model of computation it is possible to analyze the computational resources required by the algorithm, such as the execution time or memory space, and discuss the limitations of algorithms or computers. It is common to specify a computational model in terms of primitive operations allowed which have unit cost, or simply *unit-cost operations*.

A set of primitive operations defines the exact set of actions any algorithm in this computational model can perform. By assigning costs to the operations, one can determine upper bounds on the complexity of an algorithm in this model, i.e. how costly is the processing of

the problem instances; and prove lower bounds for any algorithm, i.e. what is the minimal possible cost that any algorithm in this computational model that solves this problem might have.

One can prove a lower bound on the complexity of any algorithm based on the problem instance difficulty measure as well. This defines the set of *optimal* algorithms for a given problem: those whose upper bound matches this lower bound. No algorithm that works faster than the lower bound can be developed in a given computational model.

Although there are many models of computation that differ in sets of admissible operations and the cost of their computation, most researchers that work on sorting (as described above) and intersection problems (which we will discuss later) use the *comparison-based* model of computation, which is based on the number of *comparisons* an algorithm performs. This model gives a good approximation of the real environment where large partly-ordered (as in the case of the Sorting Problem) and ordered (as in the case of an intersection problem) sets are operated on.

In the *comparison-based model of computation*, we have an input containing n items, but the only information the algorithm can get about the items is by comparing pairs of them, where each comparison returns YES or NO. Each comparison costs 1 unit, but other operations such as exchanges and moves are free.

2.1.2 Convex Hull

Although, we have discussed the adaptive analysis as it relates to the Sorting Problem, it has successfully been applied to other problems as well. Finding the convex hull of a set of points on the plane, which has received considerable attention in computational geometry, is one of them.

The *convex hull* of a finite set of points S on the plane is the smallest convex polygon containing the set. The vertices of this polygon must be points of S. Thus in order to compute the convex hull of a set S it is necessary to find those points of S that are vertices of the hull.

The first non-trivial algorithm to solve the Convex Hull Problem was proposed by Jarvis [35]. It finds the next edge of the convex hull by comparing slopes of the lines built on the last found vertex and each of the other possible vertices. The complexity of this algorithm is $\mathcal{O}(nh)$, where n is the number of input vertices and h is the number of vertices in the convex hull.

One more algorithm, Graham's scan algorithm [30] is based on sorting of all vertices by the slope they form with a pivot, which is a vertex that is guaranteed to be in the convex hull. The algorithm scans all the vertices in the sorted order and determines whether a particular one belongs to the convex hull by checking the angles it forms with its neighbors. This algorithm has $\mathcal{O}(n \lg n)$ worst-case running time. Note that as the algorithm has to process all the vertices in any case, it does not benefit from a small number of vertices in the output convex hull.

Later, Kirkpatrick and Seidel [36] proposed an adaptive algorithm based on the divide-and-conquer technique with the upper bound of $\mathcal{O}(n \lg h)$ on its complexity. They also proved that this is the worst-case lower bound on the running time of any algorithm that solves this problem if it is measured as a function of both n and h.

Recently, Chan [22] proposed a simple output-sensitive convex hull algorithm in Euclidean space E^2 and its extension to E^3, both running in optimal $O(n \lg h)$ time and both adaptive.

2.2 Queries on Binary Relations

Keyword queries is one of the most popular ways to query search engines. Usually, the answer to a keyword query is the list of all documents that contain all the keywords from the query. In such an interpretation, there is a logical association between the list of documents and the list of keywords: each document is associated with all the words it contains and each keyword is associated with all the documents it can be found in.

A *binary relation* is an association of elements of one set with elements of another set. As a binary relation is a many-many association it is a good choice to express the dependency between documents and keywords.

2.2.1 Intersection

The Intersection Problem often arises when a user is querying a search engine. If a search engine is asked for the web pages that contain all of the entered keywords it has to compute the intersection of the ordered sequences with document references corresponding to these keywords.

A_1	→	2	**5**	7	**20**	23	27	34	38
A_2	→	1	3	4	**5**	**20**	23	29	33
A_3	→	**5**	6	7	14	15	16	**20**	22

→ | **5** | **20** | = I

Figure 2.2: An example of the Intersection Problem with the number of keywords $k = 3$ and the corresponding answer.

Definition 2.2.1 *Given k ordered sequences $A_1, A_2, \ldots, A_k$ of sizes $n_1, n_2, \ldots, n_k$ summing up to n, the Intersection Problem consists of computing their intersection $I = A_1 \cap A_2 \cap \ldots \cap A_k$.*

As the number of pages in the World Wide Web is growing rapidly, the sizes of the sorted sets of references are constantly increasing. The number of queries to search engines is increasing induced by the increase of the number of internet and search engine users [40]. The faster a search engine answers user queries, the more of them it can serve using the

same hardware. The speed of intersection algorithms has strong economical importance in this environment.

A number of intersection algorithms have been proposed in the research literature. We present those of them that are related to our research, their theoretical analysis, and their performance in practice.

Basic Algorithms

The standard algorithm for solving the Intersection Problem in practice is the *SvS* Algorithm (Algorithm 1) [26]. The SvS Algorithm repeatedly intersects the two smallest sets: it performs a binary search in the unexplored portion of the larger set for each element of the smaller set. After two smallest sets were intersected, the algorithm replaces the second set with the next smallest one and intersects them once again. It keeps doing this until the largest set is processed or no elements are left in the first set. Because intersections only make sets smaller, as the algorithm progresses with several sets, the time to do each intersection effectively decreases. In particular, the algorithm benefits greatly if the set sizes vary widely and can perform poorly if the set sizes are all approximately the same.

Algorithm 1 SvS $(A_1, \ldots, A_k)$.

Sort the sets by size and rename them to $A_1, \ldots, A_k$ with A_1 to be the smallest;
for each set A_i, $i = 2$ to k **do**
  Set $l = 1$;
  for all $e \in A_1$ in increasing by value order **do**
    Perform a binary search for e in A_i between l and $|A_i|$;
    if e is not found, remove it from A_1;
    Set l to the location of the smallest element in A_i that is greater than e;
  end for
end for

Demaine *et al.* [25] introduced the *Adaptive* Algorithm (Algorithm 2). It is based on galloping search in the ordered sets: to locate an element the algorithm starts at the beginning of an array and double the index of the queried location until it overshoots the queried element, then it does a binary search between the last two locations considered during the galloping search until it finds the element or proves that there is no such element in the set (Figure 2.3).

The Adaptive Algorithm takes the smallest element in the current set whose status in the intersection is unknown and searches for it in all the other sets simultaneously: it does only one step of the doubling search in a set and then proceeds to the next set in the round-robin order. After the algorithm jumps over the element while performing doubling search, it performs complete binary search for this element in the region bounded by the last two probes of the doubling search. After the algorithm has determined whether the current element is in the output or not, it proceeds to the next element in the last set where the search for this element was performed.

Algorithm 2 Adaptive $(A_1, \ldots, A_k)$.

```
Set e = A_1[1]; i = 1; occur = 0; l_i = p_i = 1, ∀i = {1,...,k};
loop
  i = i + 1; if i > k then i = 1;
  l_i = l_i + p_i; p_i = p_i * 2; (one step of doubling search for e in A_i)
  if A_i[l_i] > e then
    Perform a binary search for e in A_i between l_i − p_i and l_i;
    Set l_i to the location of the smallest element in A_i that is greater than e;
    if e was found then
      occur = occur + 1;
      if occur = k then output e;
    end if
    if e was outputted or was not found in A_i then
      occur = 0; p_i = 1, ∀i = {1,...,k};
      Set e to the first element in A_i larger than e; (e = A_i[l_i])
      Exit loop if there is no such an element;
    end if
  end if
end loop
```

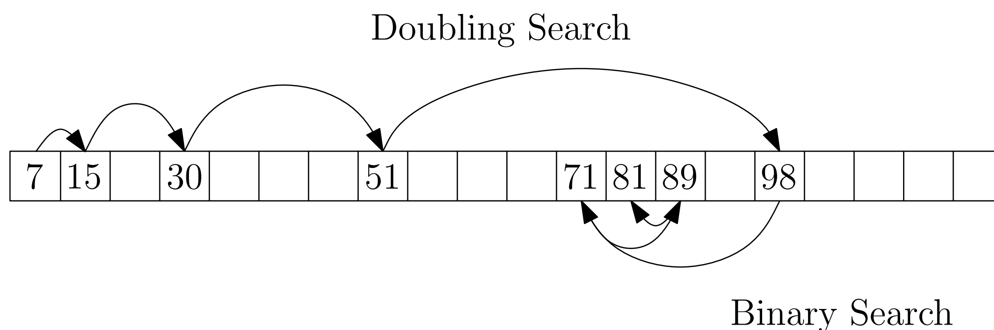


Figure 2.3: The Galloping Search Technique (searching for 85).

The *Sequential* Algorithm (Algorithm 3) for solving the Intersection Problem proposed by Barbay and Kenyon [11, 12] is similar to the Adaptive Algorithm, except that it cycles through the sets performing one entire doubling search at a time in each of them, opposed to a single doubling search step in the Adaptive Algorithm.

Algorithm 3 Sequential $(A_1, \ldots, A_n)$.

Set $e = A_1[1]$; $i = 1$; $\texttt{occur} = 0$; $l_i = 1, \forall i = \{1, \ldots, k\}$;
loop
  $i = i + 1$; if $i > k$ then $i = 1$;
  Perform a doubling search for e in A_i starting from l_i;
  Perform a binary search for e in A_i between two last probes of the doubling search;
  Set l_i to the location of the smallest element in A_i that is greater than e;
  if e was found **then**
    $\texttt{occur} = \texttt{occur} + 1$;
    if $\texttt{occur} = k$ then output e;
  end if
  if e was outputted or was not found **then**
    $\texttt{occur} = 0$;
    Set e to the first element in A_i larger than e; $(e = A_i[l_i])$
    Exit loop if there is no such an element;
  end if
end loop

In their experimental study [26] Demaine *et al.* combined the best properties of the SvS and the Adaptive Algorithm to derive a new one, which they denoted as the *Small Adaptive* (Algorithm 4). For each element in the smallest set, the Small Adaptive Algorithm performs one full galloping search in the second smallest set. If a common element is found, a new search is performed in the remaining sets to determine if the element is indeed in the intersection of all sets, otherwise the algorithm advances to the next element of the smallest set. In addition, the Small Adaptive algorithm repeatedly re-sorts the sets in the increasing according to the sizes of the unexamined intervals order, to minimize the time spent for the search.

Although each of the described above algorithms has a linear worst case time, there is a subset of instances for each of them where it performs better than the others. The Adaptive Algorithm performs well on those instances that have small proofs [25]. The same is true for the Sequential Algorithm. The SvS Algorithm performs especially well when the sizes of the sets are very different. The Small Adaptive Algorithm performs similarly to SvS as long as no element is found to be in the intersection of the two sets, otherwise it does additional checks in the other sets. Note that the Small Adaptive Algorithm and the SvS Algorithm are the only ones taking advantage of the difference of sizes of the sets, and that the Small Adaptive Algorithm is the only one which takes advantage of how this size varies as the algorithm eliminates elements.

Algorithm 4 Small Adaptive $(A_1, \ldots, A_n)$.

Sort the sets by size and rename them to $A_1, \ldots, A_n$ with A_1 to be the smallest;
Set $e = A_1[1]$; $i = 1$; $\texttt{occur} = 0$; $l_i = 1, \forall i = \{1, \ldots, k\}$;
loop
  $i = i + 1$; if $i > k$ then $i = 1$;
  Perform a doubling search for e in A_i starting from l_i;
  Perform a binary search for e in A_i between two last probes of the doubling search;
  Set l_i to the location of the smallest element in A_i that is greater than e;
  if e was found **then**
    $\texttt{occur} = \texttt{occur} + 1$;
    if $\texttt{occur} = k$ then output e;
  end if
  if e was outputted or was not found **then**
    Resort arrays by $|A_j| - l_j, \forall j = \{1, \ldots, k\}$;
    $i = 1$;
    Set e to the first element in A_1 larger than e; $(e = A_i[l_i])$
    Exit loop if there is no such an element;
  end if
end loop

Theoretical Analysis

To express the complexity of an algorithm one needs to define a measure of complexity. A common model in use in the field is the *comparison model* of computation, in which any algorithm can use elements of the sets in an atomic fashion, either through a comparison or data movement.

As search engines work with large data sets, the running time is usually dominated by external memory accesses. The number of comparisons generally shows high correlation with the number of external input and output operations and is considered to be a good approximation of the real world performance.

This cost metric does not always accurately predict running time because of caching effects and data-structuring overhead. However, in most cases data structures are very simple, and memory access patterns have similar regularities. Moreover, comparisons counting is good because of the reproducibility and because they may be directly compared with the theoretical results [26].

For the Intersection Problem on two sets of size n, the classical worst-case analysis shows that on instances where ranges of values interleave perfectly (Figure 2.4a), any algorithm has to do at least $\Omega(n)$ comparisons. On the other hand, if all elements of one set are less than any elements of the another set (Figure 2.4b), the problem can be solved by only one search in a sorted array, which has $\log_2 n + O(1)$ as its upper and lower bounds in terms of comparisons [25].

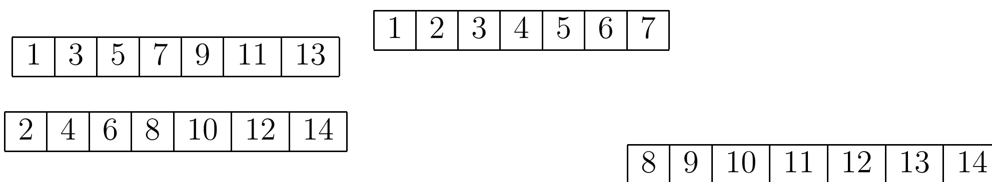


Figure 2.4: Interleaving perfectly vs. totally disjoin ranges of values in the sets.

Clearly, there is a huge gap between the complexities of $\Omega(n)$ and $\log_2 n + O(1)$, and one needs to estimate the algorithm's running time in a better way. One may use an adaptive analysis that ties a specific measure of the complexity of an instance with the number of comparisons done by the algorithm to solve this instance.

Demaine *et al.* [25] addressed this problem and proposed the adaptive algorithms that solves the intersection, union, and difference problems on ordered sets in the comparison model and introduced an adaptive analysis for these algorithms.

They defined a notion of the *proof* for a particular problem as a finite set of inequalities such that all instances that satisfy them have the same solution to the problem. They considered the binary encoding of any proof and the difficulty $D(X)$, which is the smallest encoding length over all proofs for a particular instance. They introduced a lower bound on the complexity of the algorithm, which is the length of the shortest binary encoding of any valid proof.

They considered the worst-case value of the ratio of the running time of an algorithm solving a particular problem instance to the difficulty of this instance or *scaled running time*. This approach allows the running time to be large for difficult instances, but enforces it to be small for easy ones, which is extremely important because it greatly depends on the specifics of the problem instance.

$$\text{worst-case ratio} = \max\{\frac{\texttt{Time}(X)}{D(X)}\}$$

An algorithm that minimizes the worst-case scaled running time is a natural definition of an *optimal adaptive algorithm*. Scaled running time is similar to a "competitive ratio", which measures the quality of the output of an approximation algorithm, instead of its running time, relative to the optimal one.

They proved that for the Intersection Problem the scaled running time of their *Adaptive Algorithm* is $\Theta(kG/D)$, where k is the number of sets to intersect, G is the gap cost, a parameter charactering the proof, and D is the difficulty of the problem.

Later, Barbay and Kenyon [11] introduced the Sequential Algorithm for the Intersection Problem. They used a different measure of difficulty, based on the non-deterministic complexity of the instance, which appears to be easier than the one by Demaine *et al.* .

Any algorithm for the Intersection Problem must certify that the output is correct; i.e. all the elements of the output are really elements of all the sets, and no element of the

intersection has been omitted by exhibiting some inequalities which imply that there can be no other element in the intersection.

The *partition-certificate* of a problem instance is a certain structure that contains a set I, the intersection, as well as all the comparison information required to certify that no other elements are in this intersection. Barbay and Kenyon [11] defined the difficulty of a problem instance as the size of the smallest partition-certificate $\delta = \#I + \min_P \#P$, where $\#I$ is the number of elements in the intersection, and $\#P$ is the number of comparisons needed to verify the non-existence of any other elements in the intersection.

They proved a lower bound of $\Omega(\delta \sum_i \log(n_i/\delta))$ comparisons on average for any randomized algorithm that solves the Intersection Problem, where n_i is the size of the ordered set A_i and show that the complexity of the *Sequential Algorithm* matches this lower bound.

Experiments

The complexity of the algorithm in theory and its actual running time might differ greatly in practice, depending on the way the analysis was made and the algorithm was implemented. To find how good the proposed measure of difficulty is, i.e. whether it catches the complexity of an algorithm in real applications, one should perform a set of experiments. The situations when the superior algorithms according to the theory perform not better or even worse than their competitors are not rare.

Demaine *et al.* [26] carried out the experiments on a 114-megabyte subset of the World Wide Web using a query log from Google (5000 entries). They compared two algorithms: the *SvS Algorithm*, which is used in some of the search engines, and the *Adaptive Algorithm*, which is the theoretically optimal one according to their adaptive measure of difficulty [25].

They concluded that the standard SvS Algorithm outperforms the optimal Adaptive Algorithm on many instances, albeit the minority of them. Moreover, the results show that the Adaptive Algorithm performs frequently better than the SvS only when the number of sets is small.

While the Adaptive Algorithm is wasting time cycling through the sets, SvS gains a significant advantage because the intersection of the smallest two sets is very small and may even be empty. Therefore SvS often terminates after the first pass, having only two sets examined, while adaptive constantly examines all k of them.

While the SvS and the Adaptive Algorithm perform differently and outperform each other on different instances, Demaine *et al.* designed an algorithm to compromise between these two ones in order to obtain the one to be better than both for any number of sets. They decomposed the differences between these algorithms into main techniques and examined their combinations.

The Small Adaptive Algorithm applies a galloping binary search to see how the first element of the smallest set fits into the second-smallest set. If the element is in the second-smallest set, the algorithm checks next whether this element is in the third-smallest set and so on until it determines whether it is in the answer or not (Algorithm 4).

The Small Adaptive Algorithm does not increase the work from the SvS Algorithm because it just moves some of the comparisons ahead in the schedule of SvS. The advantage,

though, is that this action may eliminate a large number of elements from sets and so will change their relative sizes.

An attempt to use Hwang-Lin merging algorithm [33] for intersection of two random sets, which combines the virtues of binary insertion and linear merging, or to alternate the set from which the candidate element comes from, was not successful – it did not make the results better. Demaine *et al.* carried out additional investigation and concluded that the Small Adaptive Algorithm appears to be the best algorithm for computing set intersection on the considered data set.

However, some questions remained open. In the situation when the theoretically optimal algorithm appears not to be best one in practice, one may ask about whether there exists a faster algorithm, and whether there is a better analysis that can strongly link the theoretical and practical results. One more problem was that the corpus of text data used was small which made it not representative enough.

Barbay *et al.* [13] summarized and performed practical testing of all algorithms above on much larger web crawl than the one used before and applied several new techniques to increase the speed of the algorithms on the given data set. They considered the replacement of the binary and doubling search by the interpolation and extrapolation search.

An *Interpolation Search* for an element of value e in an array A on the range a to b estimates the position of the next element of a given range by the values on the borders of this range as given by the formula:

$$I(a,b) = \frac{e - A[a]}{A[b] - A[a]} + a$$

An *Extrapolation Search* [13], in fact, is an extension of interpolation one, where positions of the probes used to estimate the position of the next element as well as the position of the next element can be beyond the interval $[a, b]$. In their studies, the *Extrapolate Ahead Algorithm*, which derives the expected position of the next element based on the value in the current position and the position that is further ahead by l elements, appeared to be the most successful extrapolation technique comparing to several others considered.

Barbay *et al.* [13] concluded that the Small Adaptive Algorithm is superior to all other tested ones even on the large data set. Moreover, they showed how to improve this algorithm by using the Extrapolate Ahead Technique. The resulting *Extrapolate Ahead Small Adaptive Algorithm*, which is a variation of the Small Adaptive Algorithm that at each step computes the position of the next comparison using the values of elements at distance l of each other, outperforms all other tested algorithms and performs the best when $l = \lg n_i$.

2.2.2 Threshold

Although the Intersection Problem is widely used for information retrieving, sometimes, it is too restrictive. If a user types in too many words, the most relevant web pages might not contain all of them and will be excluded from the result of the search. By providing more information to a search engine, a user restricts himself from getting potentially the

best documents. This might be corrected by relaxing requirements the query imposes to the documents in the answer.

Barbay and Kenyon [11] suggested to use the *Threshold Set Problem* which is a relaxation of the Intersection Problem. Given a keyword query from a user, the answer to the Threshold Set Problem with a parameter t is a set of objects such that each of them contains at least t keywords from the query.

A_1	$\rightarrow$	2	**5**	**7**	**20**	**23**	27	34	38
A_2	$\rightarrow$	1	3	4	**5**	**20**	**23**	29	33
A_3	$\rightarrow$	**5**	6	**7**	14	15	16	**20**	22

$\rightarrow$

5	**7**	**20**	**23**

$= I$

Figure 2.5: An example of the Threshold Set Problem with the parameter $t = 2$ and the number of keywords $k = 3$ together with the corresponding answer.

Definition 2.2.2 *Given k ordered sequences $A_1, A_2, \ldots, A_k$ of sizes $n_1, n_2, \ldots, n_k$ summing to n, the t - Threshold Set Problem consists of computing the set of elements that are present in at least t out of the k sets.*

$$T_t = \{x : \sum_i \mathbb{1}(x \in A_i) \geq t\}, \texttt{ where } \mathbb{1}(x \in A_i) = \begin{cases} 1, & \textit{if } x \textit{ belongs to } A_i \\ 0, & \textit{otherwise} \end{cases}$$

The k-threshold Set Problem is the Intersection Problem, i.e. one needs to find a set of all objects that are presented in all k ordered sets defined by the query. The 1-Threshold Set Problem is the Union Problem [25], the answer to which is the set of all the elements that are presented in at least one set in the input. The t-Threshold Set Problem with $t \in \{2, \ldots, k-1\}$) is the compromise between intersection and union problems.

Barbay and Kenyon [11] presented an algorithm for solving the Threshold Set Problem, which is very similar to their Sequential Algorithm for solving the Intersection Problem, and proved an upper bound of $\mathcal{O}(t\delta \log k \log k)$ on its complexity, where δ is the difficulty of a problem instance, i.e. the size of the smallest partition-certificate. In the following work [12] they presented a simpler algorithm based on the same ideas with a better upper bound of $\mathcal{O}(\delta \sum_i \log(n_i/\delta + 1) + \delta k \log(k - t + 1))$ on its complexity. As the Threshold Set Problem is the relaxation of the Intersection Problem, the lower bound of $\Omega(\delta \sum_i \log(n_i/\delta))$ [11] for the latter is true for the former as well.

Barbay and Kenyon [12] proposed an extension of the t-threshold set, an *opt-threshold set*, which is the answer to the t-threshold problem for the largest t for which this answer is not empty. It is always defined, because the 1-threshold is a union and is never empty, unless the input is empty.

The opt-threshold set is a more adequate answer to a simple keyword user query than a t-threshold set: when some elements match all k words of the query, opt-threshold corresponds

to the intersection; otherwise, it corresponds to the set of elements maximizing the number of words matched.

No exact algorithms for solving opt-threshold Problem was provided, except the trivial ones. Further in this thesis the Opt-Threshold Problem is denoted as the *Pertinent Set Problem*.

2.3 Queries on Labeled Trees

The *Extensible Markup Language* (XML) is a W3C-recommended general-purpose markup language [20], the main purpose of which is to facilitate the sharing of data across different applications.

XML provides a text-based means to use a tree-based structure to store information. It combines all information described as text with markup that stores the hierarchy and dependences between pieces of information. Moreover, XML is also designed in a way to be reasonably readable by a person without specialized knowledge.

Although XML is very general, a number of formally defined specialized languages based on XML (such as RSS, XHTML, MusicXML, and others) are defined. While a particular system may rely on the specific characteristics of a particular language, all these sustems are able to understand and work with the data stored in XML.

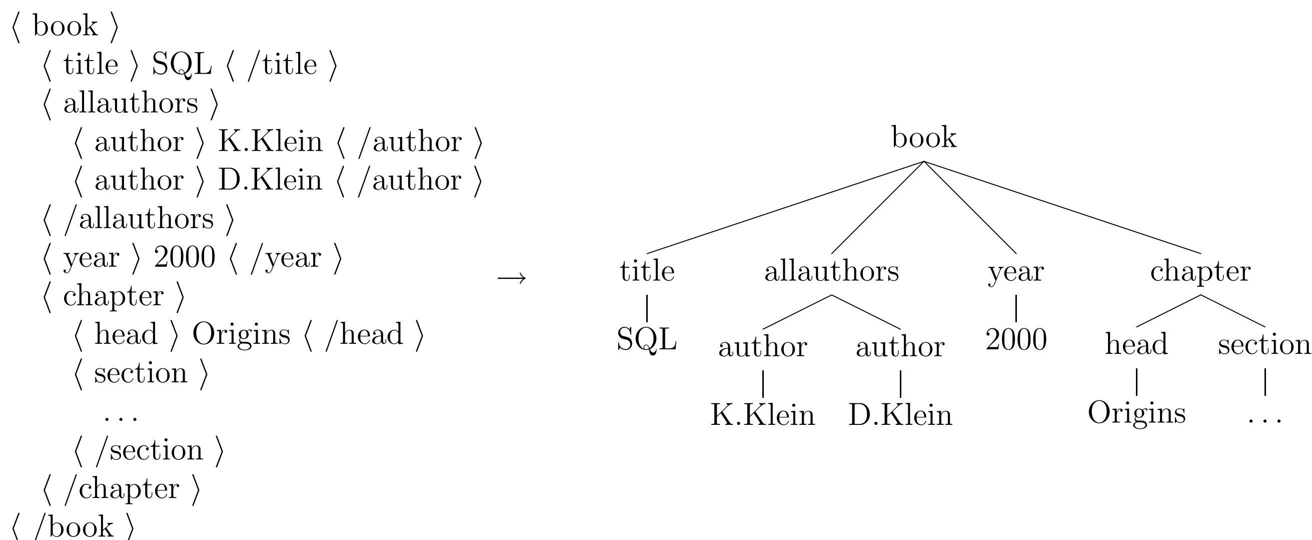


Figure 2.6: An extract of the XML file ('books.xml') and the corresponding tree structure.

There are a number of commonly used XML extensions, such as XPath, XQuery, XML Namespaces, XPointer, and others, that facilitate data searching and retrieving. XQuery [17] is a query language for querying collections of XML data. It provides flexible query possibilities to extract data from documents in the World Wide Web and access collections of XML files in the way which is similar to accessing databases.

```
XQuery:

for $book in doc('books.xml')
where $book/year = '2000'
and $book/allauthors/author = 'K.Klein'
return $book/title

the answer:
SQL
```

Figure 2.7: An example of XQuery for the XML structure shown in Figure 2.6 and the corresponding answer.

While the initial versions of XQuery support only highly-structured queries, many researchers noticed the necessity to make them unstructured. There are two main reasons to do this: first, most of the users are laypersons and do not know neither query language semantics nor database schema and, second, most of the highly-structured queries become useless if the schema of the database is changed, as in the case when one attempts to apply the same query to different databases with different schemas.

Query languages such as XML-QL, Lorel, XQL, Quilt, and others were introduced in the scientific literature to permit more flexible queries. All these languages support some sort of schema wildcards that allow the user to specify only partial paths to the data. We refer the reader to Bonifati and Ceri's comparative analysis of these languages [19].

Although such an approach gives much more flexibility and requires less knowledge from the user composing a query, sometimes, the solution they produce is not precise enough. Another approach to relax querying is to use structure-free queries, such as keyword queries, i.e. the queries that do not depend on the schema of the data storage at all.

Schmidt *et al.* [45] exploited the tree structure of XML documents to give a powerful way to query databases with whose content the users are familiar, but without requiring knowledge of tags and hierarchies, by means of a *meet* operator.

A *meet* operator determines the *Lowest Common Ancestor (LCA)* or *Nearest Common Ancestor (NCA)* of nodes in the XML syntax tree [2, 3, 4, 32]. The LCA of two nodes is the deepest ancestor of both of them: $lca(x_1, x_2)$ is the node x that is an ancestor of both of them, such that there is not other node x' that has the same property and for which x is an ancestor. The result of meet operation is equal to the LCA in the case of two nodes.

Schmidt *et al.* extended the notion of the meet operator to the case of a number of subsets of nodes of the XML tree, each of which has only nodes of the same type, i.e. the nodes that have the same rooted path built on structure nodes of the tree. They augmented queries with this operator in order to narrow the scope the information is retrieved.

They demonstrated that the operator can be implemented efficiently: they implemented it on top of the Monet XML module [44] within the Monet database server [18] and tested it on two XML sources: a description of multimedia data items and DBLP bibliography. They found out that the execution time of the meet operator is insignificant comparing to

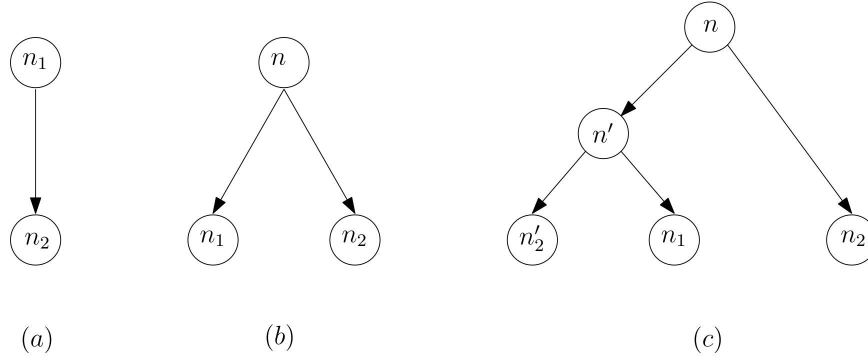


Figure 2.8: The meaningful structural relationship among the nodes.

the execution time of the full-text search involved in the query and that meet scales well with respect to distance of the objects.

In addition, Schmidt *et al.* proposed a way to improve the quality of the results to the queries by incorporating two additional restrictions: *a maximum distance* that says how many edges may lie between two input objects, and *the type of results*, i.e. for each nodes that is a candidate for the output set, its type should be compared to the types allowed to be output.

The first approach does not take any additional computation time and might be quite useful. The second one requires some additional evaluation over the requirements imposed on the possible types of the output nodes. Nevertheless, both of the approaches require extensive knowledge about XML structure that might not be in place.

Li *et al.* [37] extended XQuery to Schema-Free XQuery by adding an additional function *mlcas* that helps to narrow the tree context from which nodes are outputted to the *meaningful one*. Their Meaningful Lowest Common Ancestor Structure (MLCAS) is based on the idea for a node to *'meaningfully relate'* to other nodes in the XML tree.

Figure 2.8 gives an example of which nodes 'meaningfully' relate to others and which are not. A node and its descendant (a) meaningfully relate to each other as they often relate to the same entity: usually a descendant represents a subpart of entity or a feature of it. Two nodes that are descendants of the same ancestor (b) meaningfully relate to each other as they represent the same entity either. However, if a node has an alternative 'step-brother' (c), it does not meaningfully relate to it, because the first brother is of the same type and represents the same type of information, which is more relevant.

A new *mlcas* function in XQuery language allowed Li *et al.* to impose the mentioned structural relationship on the results of the query without making it more complex for a user. They proposed a way to incorporate it into the existing XQuery without burdening a user too much: he needs only to add the word *mlcas* to the query to make it be processed in the mlcas-aware way, but a user might not add it and will have a query processed in the original way.

Li *et al.* proposed two algorithms for computing MLCAS [37]. The first one is slower but can be easily implemented using the standard operators that most of the XML search

engines support. The second algorithm is more efficient but, on the other hand, requires a non-standard operator: stack-based *structural join* [1] to be supported by a search engine.

Consider an XML tree and m keywords involved into the Schema-Free XQuery. For each keyword e_i there is a list of nodes of length n_i associated with it. The basic idea of the first algorithm is the total enumeration of all possible vectors of size m each having a node of the mth set put to the mth position, building rooted paths for each of these nodes in the same vector, and merging these paths into a tree such that its leafs are nodes from this vector. Then, for any pair of trees, the algorithm eliminates the one whose root is an ancestor (in the original database tree) of the root of the other, as it conflicts with the definition of MLCAS. The roots of the remaining trees are returned as the MLCAS.

They showed that the time complexity of the straightforward implementation of this algorithm is $\mathcal{O}(h^m \Pi_{i=1}^m n_i)$, where h is the maximum height of the XML data tree.

The second algorithm developed by them is more efficient but requires an additional operator that is not commonly supported. The algorithm is inspired by the stack-based family of algorithms for structural join. They claimed that the time complexity of the stack-based MLCAS algorithm is $\mathcal{O}(h\Sigma_{i=1}^m n_i + \Pi_{i=1}^m n_i)$, where h denotes the height of the XML data tree.

Li *et al.* implemented Schema-Free XQuery using the Timber [34] native XML database and evaluated the system on two aspects: search quality in terms of search accuracy and completeness on standard and heterogeneous XML benchmarks and search performance, where they measured the overhead caused by evaluating schema-free queries versus schema-aware queries.

To evaluate the relative strength of Schema-Free XQuery, they compared it with two previously known techniques that allow search over XML documents without knowledge of XML schema: Meet [45] and XSEarch [24]. While Meet returns the subtree rooted by LCA for the set of keywords given in the query as the answer to the query, XSEarch is better for a pure keyword-based search as it distinguishes tag names from textual content and determines meaningful relationships among nodes in a better way.

While in the experiment Schema-Free XQuery achieves perfect precision and recall for all the queries (all the results returned were correct and all the possible correct results were returned), Meet and XSearch perform poorly on many of the queries. They often produce results that are correct but too inclusive: unrelated elements are returned along with the meaningful ones.

Experiments with performance showed that the stack-based MLCAS algorithm is up to 16 times faster than the basic MLCAS computation using standard operators. Moreover, Schema-Free XQuery queries evaluated with the stack-based algorithm incur an overhead no more than 3 times the execution time of an equivalent schema-aware query.

Xu and Papakonstantinou [46] considered the problem of keyword search in XML trees. They use the notion of *Smallest Lowest Common Ancestor (SLCA)*, which is essentially the same as the notion of Meaningful Lowest Common Ancestor (MLCAS) introduced by Li *et al.* [37], to define the answer to such queries and propose new algorithms to perform

pure keyword search. They measured the complexity of the proposed algorithms in terms of number of disk accesses and performed experiments with their implementation.

They used the conventional label-ordered tree model to represent XML trees, where all the nodes are enumerated in preorder and each of them is labeled with a tag, and present two new algorithms: the *Indexed Lookup Eager Algorithm* and the *Scan Eager Algorithm*. Both of which are based on several properties of the preorder enumeration of nodes and definitions of the lowest common ancestor and the smallest lowest common ancestor in a labeled tree.

The Indexed Lookup Eager Algorithm is based on a number of structural properties that holds for the SLCA. These properties allow scanning through the sets of nodes in increasing in preorder order and finding all the SLCA in the already scanned parts of the tree. By the time the algorithm completes the scanning of all the sets of nodes associated with keywords from the query, it has all the SLCAs in this tree for this query found.

The only difference between the Scan Eager algorithm and the Indexed Lookup Eager algorithm is that the former uses a simple scanning in the lists of nodes ordered in preorder instead of binary search.

Both of these algorithms are non-blocking, i.e. they produce part of the answers quickly so that users do not have to wait long to see the first few answers. Moreover, if a user is interested in any correct answer or is satisfied with any of a few first answers, one can reduce the total execution time dramatically by stopping the algorithm after it produced the first result.

The complexity of the Indexed Lookup Eager Algorithm is $\mathcal{O}(|S_1|kd\log|S|)$ where $|S_1|$ is the minimum and $|S|$ is the maximum size of keyword lists S_1 through S_k. The complexity of the Scan Eager Algorithm is $\mathcal{O}(k|S_1| + d\sum_2^k|S_i|)$ or $\mathcal{O}(kd|S|)$. The complexity of Stack is $\mathcal{O}(d\sum_{i=1}^k|S_i|)$ since both the number of lca operations and the number of Dewey number comparisons are $\sum_{i=1}^k|S_i|$. Some improvements on these algorithms are proposed in Appendix A.

To perform practical evaluation of the newly proposed algorithms, Xu and Papakonstantinou implemented them in a XKSearch system. Also they implemented the stack-based sort-merge algorithm (DIL) in XRANK described by Guo *et al.* [31] and denoted here as the *Stack Algorithm*, which also uses Dewey numbers.

Xu and Papakonstantinou evaluated Scan Eager, Indexed Lookup Eager, and Stack algorithms on the DBLP data. The analytical results match with the theoretical ones and show that the Indexed Lookup Eager algorithm outperforms other algorithms when the keywords have different frequencies, while the Scan Eager Algorithm appears to be better in the case where the keywords have similar frequencies.

The idea of using the full-text search in querying XML-based data sources is very useful and is becoming so popular that the support of it was incorporated in the latest version of XML and XQuery specifications [5].

The work described in Chapter 4 extends the idea of full-text search on multi-labeled trees. It introduces a new type of queries and provides new algorithms to answer them.

2.4 Data Structures

Although representation and implementation issues of the abstract data types used by us are out of the scope of this thesis, we provide a brief overview of the results obtained in this direction because they explicitly affect the possibility of an effective implementation of the algorithms we propose.

2.4.1 Binary Relations

Postings Lists

A binary relation between two sets $R : [\sigma] \times [n] \to \{0, 1\}$ can be encoded as the sequence of pairs from this relation, ordered in some convenient manner. If we have a set of labels, each of which is associated with a subset of objects, we can encode this by the sorted lists of references to objects for each label.

A weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, i.e. the one with weights associated with each pair, can be encoded as a list of postings lists that for each label α contains all the objects that are associated with this label (have $R(\alpha, x) > 0$) in increasing order together with these weights (see Figure 2.9).

R	1	2	3	4	5	6	...	n
1	0	0	2	0	0	5	...	...
2	4	0	0	1	0	0	...	...
...	...	...	...	...	...	...	...	...
σ	0	2	0	0	0	3	...	...

$\to$

1	$(3, 2)$	$(6, 5)$	...	...
2	$(1, 4)$	$(4, 1)$	...	...
α	$\cdots$	$\cdots$	$(x, R[\alpha, x])$	...
σ	$(2, 2)$	$(6, 3)$	...	...

Figure 2.9: An example of postings lists representation of a weighted binary relation on two sets.

Consider a pair (α, x) of a binary relation R where both sets are ordered. Further, we will need the possibility to find the pair with a given label α and the smallest object x' larger than x, s.t. $R(\alpha, x') > 0$. We denote this operation by α-successor of the object x if we consider x as the possible answer and strict α-successor if we do not.

The same operation but in backward direction, i.e. the finding the pair with a given label α and the largest object x' smaller than x, s.t. $R(\alpha, x') > 0$, we call α-predecessor of the object x if we consider x as the possible answer and strict α-successor if we do not.

Lemma 2.4.1 *Consider a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$ implemented as a list of postings lists, a label $\alpha \in [\sigma]$, and an object $x \in [n]$. Let n_α be the number of objects associated with each label α, then one can support the search for the α-successor of x in time $O(\lg n_\alpha)$ and the search for the α-successors of all objects in an ordered subset of size d in time $O(d \lg(n_\alpha/d))$ in the comparison model. One can support the searches for the strict α-successor, the α-predecessor, and the strict α-predecessor of x in the same time as well.*

Proof 2.4.1 A simple binary search finds the α-successor of x in $O(\lg n_\alpha)$ comparisons. A sequence of doubling searches finds a sequence of α-successors of d consecutive objects in $O(d \lg(n_\alpha/d))$ comparisons [42]. The other operators are supported using the same technique. □

Succinct Encoding

A weighted binary relation can also be encoded as a set of compressed bit-vectors [39, 43]. Such encodings take advantage of a small index and a word parallelism of the word-RAM model.

The succinct encoding proposed by Barbay *et al.* [10] supports the search for the insertion rank of a particular element in $O(\lg \lg \sigma)$ time.

Lemma 2.4.2 *Consider a weighted binary relation $R : [\sigma] \times [n] \to \{0, \dots, \mu_R\}$ implemented as a succinct encoding where $\sigma < n$, a label $\alpha \in [\sigma]$ and an object $x \in [n]$. One can search for the α-successor of x in time $O(\lg \lg \sigma)$ in the word-RAM model, with word size $\Theta(\lg \max\{\sigma, n\})$. One can support the searches for the strict α-successor of x, α-predecessor of x, and strict α-predecessor of x in time $O(\lg \lg \sigma)$ as well.*

Proof 2.4.2 The succinct encoding [10] supports the *rank* and *select* operators on the rows of the matrix in time $O(\lg \lg \sigma)$, on the positive entries in the rows of the matrix representing the weighted binary relation. One can search for the α-successor, α-predecessor, strict α-successor or strict α-predecessor of an object in time $O(\lg \lg \sigma)$ using these operators. The other operators are supported using the same technique. □

2.4.2 Priority Queues

There are a number of different ways to implement a priority queue abstract data type. We consider only two of them: a complete binary tree implemented by an array and the advanced tree structure described by Andersson and Thorup [6].

The classical heap implementation of the priority queue based on an array representing a full binary tree requires logarithmic time to insert an element in and to find and retrieve the minimal element from the priority queue.

The advanced structure described by Andersson and Thorup [6] is based on the exponential search trees that use the benefits of the word-RAM model. This structure supports general insertion and deletion of an element in $\mathcal{O}((\lg \lg k)^2)$ time amortized and search for the minimal element in constant time in the word-RAM model.

Considering the variety of data-structures that one can use to implement binary relations, labeled trees, and priority queues, each of them with a different trade-off between the space used and the time required to search in it, we express the complexity of our algorithms in the number of high-level operations such as α-successor and α-predecessor searches, and priority queue operations. Then we infer the final complexity of each algorithm for each

data-structure we consider. One can do the same for any other data structure that supports the mentioned operations required by our algorithms.

Chapter 3

Weighted Queries on Weighted Binary Relations

In this chapter we deal with *weighted queries* on *weighted binary relations* to solve the problem of retrieving relevant documents from a large data storage using keywords. While weights in binary relations allow us to express the degree of dependence between keywords and documents in the data storage more precisely, weights in queries differentiate the importance of each keyword for the result.

We consider the *Threshold Set Problem* to find all those documents that are relevant enough according to the given weighted binary relation, weighted query, and a threshold value. We provide an adaptive algorithm to solve this problem and find such set of documents. We analyse presented algorithm through introducing an adaptive measure of difficulty and proving the upper bound on the number of high-level operations this algorithm performs.

Then, we explain the shortcomings of the provided approach coming from the inability to set the threshold value manually precise enough in advance and introduce the *Pertinent Set Problem*, which is free of them. We change our algorithm for the Weighted Threshold Set Problem to deal with the pertinent case and prove the upper bound on the number of high-level operations this algorithm performs in the worst case as well.

Finally, we describe two ways the presented algorithms can be implemented and derive the upper bounds on their complexity in comparison-based and word-RAM models of computations.

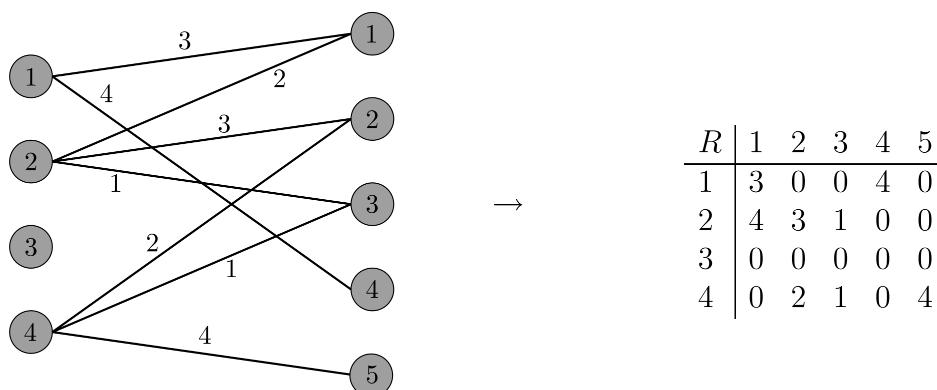
The chapter is organized as follows. In Section 3.1 we provide the basic definitions that the reader will need in the subsequent sections. In Section 3.2 we introduce the Threshold Set Problem and provide an algorithm to solve it followed by its adaptive analysis. In Section 3.3 we introduce the Pertinent Set Problem and an algorithm to solve it as well as its analysis. In Section 3.4 we show how one can implement these algorithms in practice and derive the upper bounds on the complexity of these implementations.

3.1 Basic Definitions

In practice, one often needs to express a dependence between elements of two sets of objects: some elements of the first set are associated with some elements of the second set. In the general case, each object can be associated with another independently from the associations between itself and other objects or between other objects in these sets, a non-negative integer value.

A binary relation is an easy way to define such an association. A binary relation on two sets does not have any restrictions on what items of the first set can be associated with the items of the second one. Here we define weighted binary relations, i.e. binary relations where each association of two elements is augmented with a weight.

Definition 3.1.1 *Consider a set of labels $[\sigma]$, a set of documents $[n]$, and a positive integer μ_R. A* weighted binary relation *R is a function $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$ that associates an integer value $R(\alpha, x)$ with each pair formed by a label $\alpha \in [\sigma]$ and a document $x \in [n]$.*



R	1	2	3	4	5
1	3	0	0	4	0
2	4	3	1	0	0
3	0	0	0	0	0
4	0	2	1	0	4

Figure 3.1: An example of weighted association between elements of two sets and the corresponding weighted binary relation.

If $\mu_R = 1$, a weighted binary relation reduces to an unweighted (or boolean) binary relation [11].

In practice, a binary relation can be used to represent an index in a search engine. A search engine usually stores references to all the documents that contain a particular word from the set of all words the user can type in. In this situation, weights may express the importance of a given word in the document: for example, whether the word is in the title, in the text, or in the footnote.

Keyword queries are a popular way for a user to ask for information. These queries consist of keywords, by which a user expresses what he is looking for. Usually, the answer to this query is related to these keywords in some way. For example, one can require the retrieved documents to contain all these words.

However, usually not all the keywords in a query are equally important. While some of them are very specific and point to the revel ant documents, others are more general and appear in the documents not relevant to the query as well.

Here we consider *weighted* queries. While simple queries treat each keyword as equally important, weighted queries associate a weight with each keyword to say which of them are more important and which are less. One can use this to facilitate the calculation of the relevance of documents: a document associated with a few very important keywords might be more relevant than the one associated with a huge amount of less ones.

Definition 3.1.2 *Consider a set of labels $[\sigma]$ and a positive integer μ_Q. A* weighted query *is a function $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$, mapping $\alpha \in [\sigma]$ to a weight $Q(\alpha) \in \{0, \ldots, \mu_Q\}$.*

3	2	5	1	5
how	to	build	a	shuttle

Figure 3.2: An example of a weighted query.

Keyword queries are a convenient way to build a request: it does not require any special knowledge about the data storage from the user. This is especially important when most of the users are ad hocs as in the case of the most publicly used search engines.

Associating weights with keywords in queries brings much more flexibility to the system. They, if used explicitly by a user, can help him define the degree of relevance of each word to the query. On the other hand, a system can use weights automatically to adjust the search results to a specific user, i.e. to make the search personalized, like Google does nowadays [29].

To impose more structure on the sets of documents under consideration, we require them to be ordered. This ordering does not restrict usability of the sets in any way, but is critical for efficient implementation of search in these sets. All the implementation issues are discussed in Section 3.4.

3.2 Threshold Set Problem

3.2.1 Problem Statement

For any query entered by a user, there are more relevant and less relevant documents in the data storage. In order to differentiate them, we define a notion of relativeness of an document to the specified query and show how to search and retrieve the most relevant ones.

In line with the common approach, the more keywords from the query are associated with a document, the more relevant this document is to the query. In the case of a weighted query, the documents associated with the keywords of greater weights are more relevant than those associated with low-weighted keywords. We use the notion of *score* of a document, which is a weighted sum of the weights of all keywords from the query associated with the document, to expresses the measure of its relativeness to a given query.

Definition 3.2.1 *Consider a weighted binary relation* $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$ *and a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$*. The* score *of a document* x *for* Q *on* R *is the sum of the keyword's* α *weight* $Q(\alpha)$ *multiplied by the weight of its association* $R(\alpha, x)$ *to the document* x *over all labels:*

$$\texttt{score}(R, Q, x) = \sum_{\alpha \in [\sigma]} Q(\alpha) R(\alpha, x).$$

Note that if μ_Q is quite large, even documents associated only with the very low-weighted keywords in the query may contribute a large weight to the total sum, if the weight of the corresponding association is large enough.

As all the documents in the storage have a particular relevance to a given query expressed in terms of the integer value *score*, one can define a set of 'relevant enough' documents he wants to retrieve by setting a threshold for the value of score. In this case, the problem of solving weighted queries on weighted binary relations will be to find all the documents that have the score of at least a given threshold value.

Definition 3.2.2 *Consider a weighted binary relation* $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$*, a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$*, and a non-negative integer* t*. The* threshold set answering Q on R with threshold t *is the set of all documents of score of at least* t *for* Q *on* R*:*

$$\texttt{Thres}(R, Q, t) = \{x \in [n], \texttt{score}(R, Q, x) \geq t\}$$

The problem of finding such threshold set is defined as the Threshold Set Problem*.*

In the case where $\mu_Q = \mu_R = 1$, the definition matches the one for unweighted (simple) queries on unweighted (boolean) binary relations. If $\mu_R = 1$ and t is equal to the sum of the label weights in Q, i.e. $t = \sum_{\forall \alpha} Q(i)$, the threshold set corresponds to the result of a traditional keyword query – the intersection of the sets of documents associated with the labels from the query.

3.2.2 Algorithm

To proceed further, we introduce some operations on binary relations that our algorithm will use. These operations are very common in the literature in the field [10] and there are a lot of different ways to support them that depend on the way the weighted binary relation is encoded. We will discuss some of these ways in Section 3.4.

Definition 3.2.3 *Consider the ordered sets* $[\sigma]$ *and* $[n]$*, a weighted binary relation* $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$*, an arbitrary label* $\alpha \in [\sigma]$*, and an arbitrary document* $x \in [n]$*.*

- *The* α-successor of x*, denoted by* $\texttt{label_successor}(\alpha, x)$*, is the smallest document* y *larger or equal to* x*, such that* $R(\alpha, y) > 0$*.*
- *The* strict α-successor of x*, denoted by* $\texttt{label_ssuccessor}(\alpha, x)$*, is the smallest document* y *larger than* x*, such that* $R(\alpha, y) > 0$*.*

Our algorithm, formally described as Algorithm 5 and Algorithm 6, is inspired by the algorithm for unweighted threshold set queries on unweighted binary relations proposed by Barbay and Kenyon [12].

Algorithm 5 scans through the set of all documents in increasing order executing an iteration for those of them that might be in the threshold set. At each iteration it checks whether the current document x is in the threshold set or not by calculating the score value of x. Once the decision on the current document is made, the algorithm advances to the next possible candidate document, which is determined by Algorithm 6.

Algorithm 5 operates with k labels of positive weights from the query Q distributed among three sets: a `YES`-set, a `MAYBE`-set, and a `NO`-set. For any current document x, `YES`-set contains labels that are associated with x, `NO`-set contains labels that are not associated with it, and `MAYBE`-set contains labels that were not checked yet. Each label appears in exactly one of these sets at any given moment.

The algorithm makes a decision regarding the current document x based on the value that the labels that are currently in `YES`-set add to the score value and the labels in the `MAYBE`-set may potentially add to the score value. If the decision cannot be made at the moment, additional computation follows and more labels from `MAYBE`-set are moved to either `YES`-set or `NO`-set. The algorithm continue doing this until the decision is made. It is guaranteed to make a final decision before there will be no more labels in `MAYBE`-set to retrieve.

We use the notions of *weight* and *optimistic weight* of a set of labels in the formal descriptions of Algorithm 5 and Algorithm 6. We use *weight* to denote the cumulative weight a subset of labels gives to the score by being associated with the current document. On the other hand, we use the notion of the *optimistic weight* to estimate the upper bound of the value of score that a given subset of labels can potentially contribute to a given document.

$$\mathtt{WEIGHT}(S, x) = \sum_{\alpha \in S} Q(\alpha) R(\alpha, x)$$

$$\mathtt{WEIGHT}^{\mathtt{OPT}}(S) = \mu_R \sum_{\alpha \in S} Q(\alpha)$$

We use the notion $X \leftarrow \alpha \leftarrow Y$ in the description of the algorithms to represent the operation of moving a label α from set Y to set X. Omitting α, as in the case $X \leftarrow Y$, means that we move all the labels from set Y to set X.

Algorithm 6 advances x to the next value in the following way. First, it discards all the labels from `YES`-set to `MAYBE`-set. Then the algorithm starts moving labels from `MAYBE`-set to `NO`-set one by one until the optimistic weight of `MAYBE`-set is about to drop below the threshold value t. Then it sets x to the lowest document greater than the current x, associated with at least one label in `NO`-set.

Note that Algorithm 5 is non-blocking. It outputs a document from the threshold set immediately after it finds it. If a user needs only the first few documents from the threshold set, as in the case of most search engines, the algorithm does not have to compute the entire threshold set – it can stop processing after retrieving enough documents to display to a user, a number that is often very small.

Algorithm 5 Answering Threshold Set Queries (R, Q, t)

$x = -\infty$, $\texttt{NO} = \texttt{YES} = \emptyset$, $\texttt{MAYBE} = \{\forall\alpha | Q(\alpha) > 0\}$;
$\texttt{Advance}(x, \texttt{YES}, \texttt{NO}, \texttt{MAYBE})$ (Algorithm 6);
while $x < +\infty$ **do**
  $\alpha \leftarrow$ the next label from MAYBE in round-robin order;
  if $\texttt{label_successor}(\alpha, x) = x$ **then**
    $\texttt{YES} \leftarrow \alpha \leftarrow \texttt{MAYBE}$;
    if $\texttt{WEIGHT}(\texttt{YES}, x) \geq t$ **then** Output x;
  else
    $\texttt{NO} \leftarrow \alpha \leftarrow \texttt{MAYBE}$; (using the result from $\texttt{label_ssuccessor}(\alpha, x)$ and pushing to the priority queue)
  end if
  if $t \leq \texttt{WEIGHT}(\texttt{YES}, x)$ or $\texttt{WEIGHT}(\texttt{YES}, x) + \texttt{WEIGHT}^{\texttt{OPT}}(\texttt{MAYBE}) < t$ **then**
    $\texttt{Advance}(x, \texttt{YES}, \texttt{NO}, \texttt{MAYBE})$ (Algorithm 6);
  end if
end while

Algorithm 6 $\texttt{Advance}(x, \texttt{YES}, \texttt{NO}, \texttt{MAYBE})$

$\texttt{MAYBE} \leftarrow \texttt{YES}$; $(\texttt{YES} = \emptyset;)$
while $\texttt{WEIGHT}^{\texttt{OPT}}(\texttt{MAYBE}) \geq t$ **do**
  $\alpha \leftarrow \texttt{MAYBE}$ in round-robin order;
  $\texttt{NO} \leftarrow \alpha \leftarrow \texttt{MAYBE}$ ($\texttt{label_ssuccessor}(\alpha, x)$-operator);
end while
Find $S \subset \texttt{NO}$ of labels, such that $x_{\texttt{next}} = \texttt{label_ssuccessor}(\alpha, x)$ is the same and minimal;
$\texttt{YES} \leftarrow S \leftarrow \texttt{NO}$; (at most σ priority queue pops)
$x = x_{\texttt{next}}$;

3.2.3 Adaptive Analysis

Any algorithm that solves the Threshold Set Problem has to *certify* the correctness of its output, i.e. to present a *certificate* that guarantees that the answer is correct. A certificate can be defined in different ways. Here we define it as a partition of the problem instance with specific properties.

Definition 3.2.4 *Consider a weighted binary relation $R : [\sigma]\times[n] \to \{0,\dots,\mu_R\}$, a weighted query $Q : [\sigma] \to \{0,\dots,\mu_Q\}$, and a non-negative integer t. A* partition-certificate *is a partition $(I_i)_{i\in[\delta]}$ of the set $[n]$ of objects, such that for any $i \in [\delta]$, one of the three options is true:*

- *I_i is a single document $\{x\}$ with $\texttt{score}(R, Q, x) \geq t$ (a positive singleton);*
- *I_i is an interval that has a set $S \subseteq [\sigma]$ of labels, such that $\mu_R \sum_{\alpha\notin S} Q(\alpha) < t$ and $\sum_{x\in I_i} R(x, \alpha) = 0$, $\forall\alpha \in S$ (a negative interval)*
- *I_i is a single document $\{x\}$ with $\texttt{score}(R, Q, x) < t$, but does not have a set $S \subseteq [\sigma]$ of labels, such that $\sum_{x\in I_i} R(x, \alpha) = 0$, $\forall\alpha \in S$ and $\mu_R \sum_{\alpha\notin S} Q(\alpha) < t$ (a negative singleton).*

Keywords	Weights		1	2	3	4	5	6	7	8	9
how	3	→	1	0	1	2	2	1	0	0	1
build	5	→	2	0	0	0	1	2	2	1	1
shuttle	5	→	2	2	0	2	2	0	0	2	1

Figure 3.3: An example of the problem instance and the corresponding partition certificate for $\mu_R = 2$, $\mu_Q = 5$, and $t = 20$, where all three types of intervals are present.

In the partition-certificate presented in Figure 3.3 there are two positive singletons: $\{1\}$ and $\{5\}$ (with $\texttt{score}(\{1\}) = 23$, $\texttt{score}(\{5\}) = 21$, both are greater than threshold $t = 20$). $\{2, 3, 4\}$ and $\{6, 7\}$ are negative intervals: the first does not have a word 'build' which prevents the score of any document inside the interval to be more than 16, the second one has the same problem with a word 'shuttle'. $\{8\}$ and $\{9\}$ are negative singletons: their potential maximum score is 20 and 26 respectively, but real score is only 15 and 18.

There are a lot of ways to define the difficulty of an intersection instance, such as the minimal encoding size of a certificate [25], the minimal number of comparisons [11], or the size of signatures [9]. Here to measure the difficulty of a problem instance we define its *alternation*, the size of its smallest partition-certificate.

Definition 3.2.5 *Consider a weighted binary relation $R : [\sigma]\times[n] \to \{0,\dots,\mu_R\}$, a weighted query $Q : [\sigma] \to \{0,\dots,\mu_Q\}$, and a non-negative integer t. The* alternation *of Q on R with threshold t is the size δ of the smallest partition-certificate of the instance.*

We use alternation as our measure of difficulty for the adaptive analysis of the Algorithm 5 that solves the Threshold Set Problem. We derive the complexity of Algorithm 5 processing a specific instance as a function of the alternation of this instance, i.e. its 'hardness'.

We consider `YES`-set and `MAYBE`-set represented by a list abstract data type (ADT), because we only need to push and pop labels in and out in FIFO order. And we consider `NO`-set represented by a priority queue ADT, because we need to push elements and pop those that have the smallest value of `label_ssuccessor`(α, x) with regard to the current document x.

We assume that the functions `WEIGHT` and $\texttt{WEIGHT}^{\texttt{OPT}}$ do not take more than a constant time to compute. In fact, this can be easily done. One should keep the last values of `WEIGHT` and $\texttt{WEIGHT}^{\texttt{OPT}}$ functions for all sets. Each time a labels is pushed in or pop out of the set, these function values are updated. This does not take more than a constant time for each push and pop and does not affect the total complexity of the algorithm.

We express our results in the number of searches performed (`label_ssuccessor`(α, x) operations) on the weighted binary relation and the number of priority queue operations (inserts as well as finding and removing the minimal element). The ways these operations can be supported and the final complexity of the algorithm in that cases will be discussed in the subsequent section.

In the formulation of all following theorems, we refer to α-successor and strict α-successor operations as *search operations.*

Theorem 3.2.1 *Consider a weighted binary relation* $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$*, a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$*, and a non-negative integer* t*. There is an algorithm that computes the threshold set for* Q *on* R *with threshold-value of* t *in* $\mathcal{O}(\delta k)$ *search and priority queue operations, where* δ *is the alternation of the instance and* k *is the number of labels of positive weight in* Q*.*

Proof 3.2.1 Algorithm 5 starts from the first document and scans through the remaining ones in increasing order. It jumps over some of them to speed up the process. For each document under consideration, the algorithm explicitly determines whether this document is in the threshold set or not. Thus, to show the correctness of the algorithm we only need to show that no documents skipped by Algorithm 6 is in the threshold set.

Consider the moment when Algorithm 5 is jumping from the current value of x to the next value of x, which we denote as x'. It sets x' to the smallest document greater then x which is associated with a label in `NO`-set, skipping all the documents between x and x' which, nevertheless, might have labels in `MAYBE`-set associated with them. However, even if all the labels from `MAYBE`-set are associated with an element between x and x', their cumulative weight of $\sum_{\alpha \in \texttt{MAYBE}} Q(\alpha)$ combined with the highest possible value of an association μ_R will not be enough to quality this document for the threshold set, because $\texttt{WEIGHT}^{\texttt{OPT}}(\texttt{MAYBE}) < t$, which follows from the condition Algorithm 6 exists its main loop.

During processing of a positive or negative singleton, Algorithm 5 performs at most k iterations of the main loop to move enough labels from `MAYBE`-set to `YES`-set in order to decide whether x is in the threshold set or not. Each iteration takes at most $\mathcal{O}(1)$ search

and priority queue operations, resulting in $\mathcal{O}(k)$ for any singleton in total. Once the decision is made, Algorithm 6 jumps to the next candidate document x. Algorithm 6 performs at most $\mathcal{O}(k)$ search and priority queue operations as well. Hence, the total complexity of processing any singleton is $\mathcal{O}(k)$.

During the processing of a negative interval, Algorithm 5 might have to jump to the next candidate document x more than once. We analyse it by putting an upper bound on the number of operations the algorithm performs in this situation.

Algorithm 5 moves labels only from `MAYBE`-set to either `YES`-set or `NO`-set. Algorithm 6, on the other hand, move labels from `YES`-set to `MAYBE`-set, from `MAYBE`-set to `NO`-set, and from `NO`-set to `YES`-set in the mentioned order.

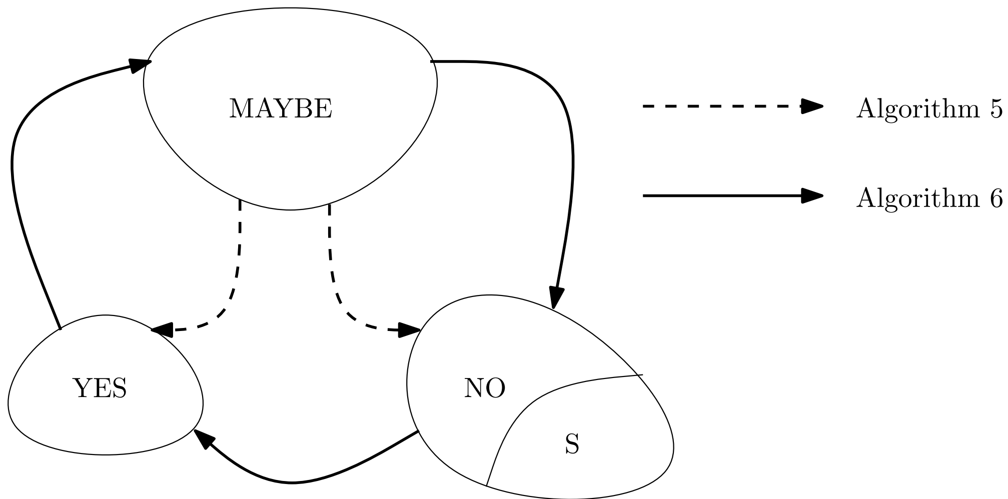


Figure 3.4: Label movement during the execution of Algorithm 5 and Algorithm 6.

However, during the processing of the same interval of a partition certificate, the algorithm does not move any labels belonging to S (Definition 3.2.4), which is the subset of `NO`-set, to `YES`-set because the α-successor of this label is out of the current interval and will be processed only during processing of the the next one. While the algorithm retrieves labels from `MAYBE`-set in round-robin order, it cannot retrieve any label from `MAYBE`-set for the second time until all the labels from subset S appear in set `NO`, which effectively means that the next element x will be outside of the current interval and the next iteration will be done on the next interval. While it takes a constant number of operations for the algorithm to move each label from `MAYBE`-set back to `MAYBE`-set, it needs $\mathcal{O}(k)$ search and priority queue operations to complete this interval of the partition-certificate.

As the algorithm spends $\mathcal{O}(k)$ operations to process each interval, and the instance has δ intervals, the total complexity of the algorithm is $\mathcal{O}(\delta k)$ search and priority queue operations. □

3.3 Pertinent Set Problem

3.3.1 Problem Statement

The threshold set for unweighted queries on unweighted binary relations is a compromise between the intersection ($t = k$) and the union ($t = 1$) of k sets: it requires the parameter t to specify what compromise is desired. The threshold set for weighted queries on weighted binary relations, discussed in the previous section, gives a possibility to show the importance of each word in the query and its relevance in the storage.

However, in most applications, defining a proper threshold value explicitly is very impractical: any predetermined threshold value can result in either very many partly relevant documents or no documents at all, depending on what labels are associated with what documents and with what weights.

The *Pertinent Set* is a solution to this problem: we define it as a set of the documents that give the highest score value, i.e. the most relevant documents to a given query. We formalize this notion and define the problem of finding it, the smallest non-empty threshold set.

The *pertinent score* for a given binary relation is the maximum score obtained by at least one document.

Definition 3.3.1 *Consider a weighted binary relation* $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$ *and a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$*. The* pertinent score for Q on R *is the maximal score for* Q *on* R *over all documents in* $[n]$*:*

$$\texttt{pertinent_score}(R, Q) = \max_{x \in [n]} \{\texttt{score}(R, Q, x)\}$$

Having a pertinent score defined, we may define a set of all documents that obtain this score. Note that it is never empty.

Definition 3.3.2 *Consider a weighted binary relation* $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$ *and a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$*. The* pertinent set *answering* Q *on* R *is the set of documents that have the maximum score for* Q *on* R*:*

$$\texttt{Pert}(R, Q) = \{x \in [\sigma], \texttt{score}(R, Q, x) = \texttt{pertinent_score}(R, Q)\}$$

The problem of finding such pertinent set is defined as the Pertinent Set Problem*.*

3.3.2 Algorithm

Algorithm 7 for finding the Pertinent Set does not differ much from Algorithm 5 for finding the Threshold Set. It scans through the set of all documents in the same way and checks whether a particular document is in the pertinent set or not as it was done by Algorithm 5. Advancing to the next element is performed in the same way and by the same auxiliary Algorithm 6.

However, while Algorithm 7 is looking for a pertinent set, a set of documents with maximal value of score (which is not known in advance), it needs to keep a value $t^\star$, the maximum value of the score obtained by any of the previous scanned documents, and a set R of all previously scanned documents that have score value of $t^\star$.

For each document x under consideration, the algorithm checks its score value $\texttt{score}(x)$, and adds x to the current pertinent set R if $\texttt{score}(x)$ is equal to $t^\star$, the score value of all other documents in R. If the $\texttt{score}(x)$ is greater than $t^\star$, the algorithm updates $t^\star$ to this new value and resets R to the $\{x\}$, because x is the only document with this score value so far.

Algorithm 7 Answering Pertinent Set Queries

```
x = −∞, NO = YES = ∅, MAYBE = {∀α|Q(α) > 0};
t* = 0, R = ∅;
Advance(x, YES, NO, MAYBE) (Algorithm 6);
while x < +∞ do
  α ← the next label from MAYBE in round-robin order;
  if label_successor(α, x) = x then
    YES ← α ← MAYBE;
    if WEIGHT^OPT(MAYBE) = 0 then
      if WEIGHT(YES, x) = t* then
        R ← x;
      end if
      if WEIGHT(YES, x) > t* then
        t* = WEIGHT(YES, x);
        R = {x};
      end if
    end if
  else
    NO ← α ← MAYBE; (using the result from label_ssuccessor(α, x) and pushing to the
    priority queue)
  end if
  if t ≤ WEIGHT(YES, x) or WEIGHT(YES, x) + WEIGHT^OPT(MAYBE) < t then
    Advance(x, YES, NO, MAYBE) (Algorithm 6);
  end if
end while
Output t* and R;
```

Note that unlike Algorithm 5, Algorithm 7 is blocking. It has to scan all the documents before it gives a user any documents that are in the pertinent set. The reason for this is that the current pertinent set can be reseted by the algorithm at any moment if it finds a document with the score greater than the maximum one obtained before, making preliminary output of documents impossible.

3.3.3 Adaptive Analysis

Theorem 3.3.1 *Consider a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$, and a non-negative integer t. There is an algorithm that computes the pertinent set for Q on R. Once it has found the first document of the pertinent set, this algorithm performs $O(\delta k)$ document searches and priority queue operations, where δ is the alternation of the instance and k is the number of labels of positive weight in Q.*

Proof 3.3.1 Once Algorithm 7 has found the first document of the pertinent set, its execution differs from the execution of Algorithm 5 only in that it continues to check whether the document under consideration has larger score value than $t^\star$ (the algorithm does not know that the $t^\star$ found is the maximal one).

To perform this check, for each document x under consideration, Algorithm 7 continues to move labels from `MAYBE`-set to `NO`-set or `YES`-set until `MAYBE`-set is empty. It retrieves at most k labels in this way for each document x, the amount that Algorithm 5 is still allowed to do by Theorem 3.2.1 without increasing of upper bound.

While during processing of each interval Algorithm 7 does not perform more operations than Algorithm 5 is allowed to do, the complexity of Algorithm 7, after it has found the first document in the pertinent set, does not differ from the complexity of Algorithm 5. □

Note that before finding the first element of the pertinent set, Algorithm 7 might perform an amount of operations that does not depend on the alternation of the instance. In the worst case, the index of the first document in the pertinent set is very large and the algorithm needs to increase $t^\star$ up to `pertinent_score`(R, Q) times and put up to n documents to the temporary pertinent sets before finding the first document from the real pertinent set. The algorithm might compute a temporary result of size n even though the final result is small or null.

In practice, it might not be an issue, if the algorithm finds the first element of the pertinent set and the optimal threshold-value t fast in most cases.

3.4 Implementation

In the previous sections we derived an upper bound on the complexity of the Threshold Set and the Pertinent Set Algorithms expressed in the number of searches in a binary relation and priority queue operations (insertions and deletions). There are a number of ways one can support these primitive operations.

In this section we consider some of the ways a binary relation and a priority queue can be implemented that were discussed in Section 2.4 and derive exact upper bounds on the complexity of the Threshold Set and the Pertinent Set Algorithms. We consider two different ways to implement a binary relation: using postings lists and using a succinct encoding [10], as well as two ways to implement a priority queue: using a classical heap structure and using the advanced structure proposed by Andersson and Thorup [6].

Corollary 3.4.1 *If a binary relation is implemented by postings lists and a priority queue is implemented by a classical heap structure, the complexity of Algorithm 5 / 6 and Algorithm 7 / 6 is* $O(\delta \sum_{\alpha\in[\sigma]} \lg(n_\alpha/\delta) + \delta k \lg k)$*, where* t *is the sum of the sizes of all postings lists and* k *is the number of labels with a positive weight.*

Proof 3.4.1 This follows directly from Theorem 3.2.1, Theorem 3.3.1, and Lemma 2.4.1: at most δ searches are performed in each of postings lists, for documents in increasing order, and each priority queue operation costs at most $O(\lg k)$ comparisons. □

Corollary 3.4.2 *If a binary relation is implemented by a succinct encoding and a priority queue is implemented in the way proposed by Andersson and Thorup, the complexity of Algorithm 5 / 6 and Algorithm 7 / 6 is* $O(\delta k \lg\lg\sigma + \delta k(\lg\lg k)^2)$ *in the RAM model with word size* $\Theta(\lg\max\{\sigma, n\})$*, where* k *is the number of labels with a positive weight.*

Proof 3.4.2 This follows directly from Theorem 3.2.1, Theorem 3.3.1, Lemma 2.4.2, and the priority queue implementation described by Andersson and Thorup [6]. □

After one find other ways to implement binary relations, he can apply Theorem 3.2.1 and Theorem 3.3.1 to derive new upper bounds on the complexity of Algorithm 5 and Algorithm 7.

Chapter 4

Weighted Path Subset Queries on Weighted Labeled Trees

In this chapter we deal with *weighted path subset queries* on *weighted labeled trees* to solve the problem of retrieving relevant documents from a large tree-like data storage using keywords. While weights in labeled trees allow to express the degree of dependence between keywords and their nodes more precisely, weights in queries differentiate the importance of each keyword for the result.

We consider the *Threshold Path Subset Problem* to find all those documents that are relevant enough according to the given weighted binary relation, weighted query, and a threshold value. We provide an adaptive algorithm to solve this problem and find such set of documents. We analyse the given algorithm by introducing an adaptive measure of difficulty and proving an upper bound on the number of high-level operations this algorithm performs.

Then, we explain the shortcomings of the provided approach coming from the inability to set the threshold value manually precise enough in advance and introduce the *Pertinent Path Subset Problem*, which is free of them. We change our algorithm for the Threshold Path Subset Problem to deal with the pertinent case and prove the upper bound on the number of high-level operations this algorithm performs in the worst case as well.

Finally, we describe two ways the presented algorithms can be implemented and derive the upper bounds on their complexity in comparison-based and word-RAM models of computations.

The chapter is organized as follows. In Section 4.1 we provide the basic definitions that the reader will need in the subsequent sections. In Section 4.2 we introduce the Threshold Path Subset Problem and provide an algorithm to solve it followed by its adaptive analysis. In Section 4.3 we introduce the Pertinent Path Subset Problem and an algorithm to solve it as well as its analysis. In Section 4.4 we discuss how one can implement these algorithms in practice and derive the upper bounds on the complexity of these implementations.

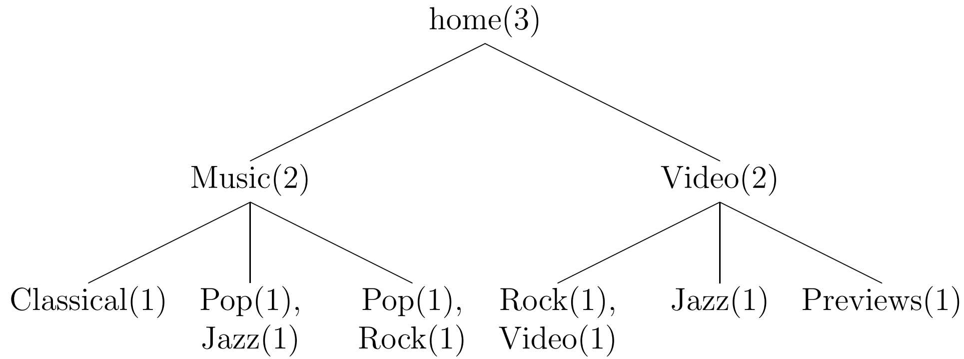


Figure 4.1: An example of a simple file system. Each node represents a folder and contains the words associated with it, along with the weight of these associations.

4.1 Basic Definitions

Tree structures are very important in information retrieval. The information stored in a tree-like structure often has much more useful context than the information stored without it. This additional structural dependence between pieces of information allows to increase the quality of the information retrieval.

One of the ways to store information in a tree is by using the multi-labeled tree structure: a rooted tree that has labels associated with its nodes. While each node represents a piece of information, linking edges between nodes show the structural dependence between these pieces.

Following the ideas presented in the previous chapter, we assign weights to the associations between nodes and labels. This helps to express even more complex context than unweighted (simple) multi-labeled trees allow.

Definition 4.1.1 *Consider a rooted tree T with a set of nodes $[n]$ and a set of labels $[\sigma]$. A* weighted multi-labeled tree *is a rooted tree T with defined weighted binary relation $R : [\sigma] \times [n] \rightarrow \{0, \ldots, \mu_R\}$ between the set of labels $[\sigma]$ and the set of nodes $[n]$.*

Note that if $\mu_R = 1$, a weighted multi-labeled tree reduces to a simple (unweighted) multi-labeled tree described by Barbay *et al.* [10].

To work with individual nodes in a tree, one needs to assign unique numbers to them, i.e. to index them. There are many different ways to enumerate nodes in a tree: the common preorder, postorder, inorder or a quite sophisticated but still very useful DFUDS order [14].

We use preorder ordering to enumerate nodes of a tree in this chapter. As one can convert an index of the node from one ordering to another, i.e. find indexes of the same node in different orderings, in constant time, the choice of the ordering does not restrict us.

Here we define the weighted path subset query in the same way the weighted query was defined in the previous chapter.

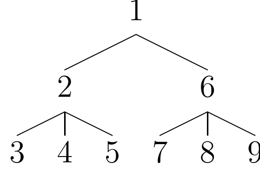


Figure 4.2: The preorder enumeration of the nodes in the tree in Figure 4.1.

Definition 4.1.2 *Consider a set of labels* $[\sigma]$ *and a positive integer* μ_Q*. The* weighted path subset query *is a function* $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$*, mapping* $\alpha \in [\sigma]$ *to a weight* $Q(\alpha) \in \{0, \ldots, \mu_Q\}$.

4.2 Threshold Path Subset Problem

4.2.1 Problem Statement

As in the case of weighted binary relations, for any query entered by a user, there are more relevant and less relevant nodes in the tree-like data storage. In order to differentiate them, we define a notion of relevance of a node relative to the query and show how to search and retrieve the most relevant ones.

However, here we need to take the tree structure into account. Commonly, a label that is associated with a node characterizes not only the node itself but all the nodes of its subtree as well. To express this we need to propagate the influence of each label to all the nodes in the subtree.

Given a weighted path subset query Q on a tree T labeled through the relation R, the path-score of a node x is defined as the sum of maximum values of $Q(\alpha)R(\alpha, y)$ for each node y which is x or one of its ancestor over all labels $\alpha \in [\sigma]$. Each label is counted only once, i.e. a label α contributes only $\max_y R(\alpha, y)$ to node x, where y is x or one of its ancestors.

Definition 4.2.1 *Consider a weighted multi-labeled tree* T*, a weighted binary relation* $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$*, a weighted path subset query* $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$*. We define the* path-score *of a node* x *for* Q *on* T *and* R *as*

$$\texttt{path_score}(T, R, Q, x) = \sum_{\alpha \in [\sigma]} Q(\alpha) \max_{y \in \{x\} \cup \texttt{ancestors}(x)} R(\alpha, y)$$

where $\texttt{ancestors}(x)$ *denotes the set of all ancestors of a given node* x *in* T.

The further a node is from the root, the higher path-score it gets: any descendant of the node x gets path-score of at least the same value as x. We are interested in the closest to the root nodes – the first ones that get the path-score of at least t, as they are sufficient to describe the set of all such nodes.

Definition 4.2.2 *Consider a weighted multi-labeled tree T, a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, a weighted path subset query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$, and a non-negative integer t. The* threshold path subset *of a weighted path subset query Q with the threshold t is the set of nodes that have path-score of at least t and do not have any ancestors with this property, i.e.*

$$\texttt{ThresTree}(T, R, Q, t) =$$
$$\{x \in [n] : \texttt{path_score}(T, R, Q, x) \geq t \wedge \forall y \in \texttt{ancestors}(x) \Rightarrow$$
$$\texttt{path_score}(T, R, Q, y) < t\}.$$

The problem of finding such threshold path subset is defined as the Threshold Path Subset Problem*.*

1	1	1	1
Home	Music	Pop	Previews

Figure 4.3: An example of weighted query. For the sake of the simplicity of the discussion, all the keywords have the same weight 1, but the algorithm works the same in the general case as well.

In the case where $\mu_Q = \mu_R = 1$, the definition matches the one for unweighted (simple) queries on unweighted multi-labeled trees. If $\mu_R = 1$ and t is equal to the sum of the label weights in Q, i.e. $t = \sum_{\forall\alpha} Q(i)$, the threshold path subset only those nodes that have all the labels from the query associated with them or one of their ancestors.

We restrict the problem to the case where the labels are associated with the nodes on the same root-to-leaf path with non-increasing weights, i.e. there is no node x that has a label α associated with it with some weight $R(x, \alpha)$ and that has a descendant x' associated with the same label with larger weight $R(x', \alpha) > R(x, \alpha)$.

Definition 4.2.3 *A weighted multi-labeled tree T is called* a non-increasing weighted multi-labeled tree *if $\forall x_1, x_2 \in [n], x_1 \in \texttt{ancestors}(x_2) \implies \nexists \alpha, \alpha \in [\sigma], s.t.\ R(\alpha, x_1) > 0 \wedge R(\alpha, x_2) > 0)$.*

This non-increasing property does not restrict the instances of the problem when $\mu_R = 1$, i.e. if a labeled tree is non-weighted, because in this case a tree is non-increasing by definition: if a node has a label associated with it, it cannot have an ancestor with the same label but with more weight – the maximum weight is 1.

The multi-labeled tree in Figure 4.1 is non-increasing. Although a label 'Video' is associated with two different nodes one of which belongs to the rooted path of the another, an association with the descendant has weight 1, which is smaller than the association with the ancestor that has weight 2. If we exchange these weights, the tree will cease to be non-increasing.

The presence of the non-increasing property makes the contribution of a label α to the path-score of a node x depend only on the weight of the closest to the root ancestor of

the node x associated with the label α, instead of depending on the arbitrary one with the largest weight of its association with the label α. To solve weighted threshold path subset queries in the general case, an algorithm would have to compute $\max_{y \in \texttt{ancestors}(x) \cup \{x\}} R(\alpha, y)$ regularly, which makes it more complex.

Further in this thesis, we assume that weighted multi-labeled trees are non-increasing ones, unless the other is specified explicitly.

4.2.2 Algorithm

For any label $\alpha \in [\sigma]$ and any node $x \in [n]$ we use the same notion of the operations *α-successor of x* and *strict α-successor of x* as for weighted binary relations but with respect to the preorder defined on the set of nodes $[n]$.

However, our algorithm needs a number of additional operations specific to multi-labeled trees because of the new structural dependence a tree introduces.

Definition 4.2.4 *Consider a weighted multi-labeled tree T and a weighted binary relation $R : [\sigma] \times [n] \rightarrow \{0, \ldots, \mu_R\}$. For a node $x \in [n]$ and a label $\alpha \in [\sigma]$ we define:*

- *the* α-ancestor of x, *denoted by* `label_ancestor`(α, x), *is the closest to the root ancestor y of the node x or node x itself, such that $R(\alpha, y) > 0$, or $\emptyset$ if there is no such node.*
- *the* follower *of x, denoted by* `follower`(x), *is the node with the smallest preorder rank greater than x that is not a descendant of x, or $\emptyset$ if there is no such node.*

The notion of `follower`(x) correlates with the notion of `following`(x) in XPath queries [15, 23]. Basically, `follower`(x) is the first node in the set `following`(x) of all nodes greater than node x in document order, excluding any descendants, attribute, and namespace nodes.

Our algorithm for answering weighted queries on non-increasing weighted multi-labeled trees, formally described as Algorithm 8 and Algorithm 9, is inspired by Algorithm 5 and Algorithm 6 for weighted queries on weighted binary relations described in Section 3.2. The main difference is that once the next node of the output set is found, Algorithm 8 skips all the nodes in its subtree, because they are guaranteed not to be in the threshold path subset.

Algorithm 8 scans through the set of nodes in increasing in preorder order. At each iteration it checks whether the current node x is in the threshold set or not by calculating the path-score value of x. Once the decision on the current node is made, the algorithm advances to the next possible candidate node, which is determined by Algorithm 9.

Algorithm 8 operates with k labels of positive weights from the query Q distributed among `YES`-set, `MAYBE`-set, and `NO`-set in the similar way as Algorithm 5 does. For any current node x, `YES`-set contains labels that are associated with at least one node on the rooted path of x or with a node x itself, `NO`-set contains labels that are not associated with any node on the rooted path of x or with itself, and `MAYBE`-set contains labels that were not checked yet. Each label appears in exactly one of these sets at any given moment.

The algorithm makes a decision regarding the current node x based on the value that labels that are currently in YES-set bring to the path-score value and labels that are in MAYBE-set might potentially bring. If the decision cannot be made immediately, additional computation follows and more labels from MAYBE-set are moved to either YES-set or NO-set. The algorithm keeps doing this until the decision is made. It is guaranteed to make a final decision before there will be no more labels in MAYBE-set to retrieve.

Algorithm 9 differs from Algorithm 6 in the following way: once it has found a next node x from the threshold path subset, it jumps directly to the follower x' of x skipping all the nodes from its subtree, because they are guaranteed not to be in the threshold path subset. Then the algorithm discards all the labels α from YES-set that have $\texttt{label_successor}(\alpha, x) < x'$, because the nodes these labels were associated with were just skipped and nothing can be assumed about them without making a fresh search (which is done when moving a label from MAYBE-set to YES-set or NO-set).

We use the notions of *weight* denoted by WEIGHT and *optimistic weight* denoted by $\texttt{WEIGHT}^{\texttt{OPT}}$ of a set of labels in the formal descriptions of Algorithm 8 and Algorithm 9. We use *weight* to denote the cumulative weight a subset of labels gives to the path-score by being associated with the current node or one of its ancestors. On the other hand, we use the notion of the *optimistic weight* to estimates the upper bound of the value of path-score that a given subset of labels can potentially contribute to a given node. For readability, we assume that $\texttt{label_ancestor}(\alpha, \emptyset) = 0$.

$$\texttt{WEIGHT}(S, x) = \sum_{\alpha \in S} Q(\alpha) R(\alpha, \texttt{label_ancestor}(\alpha, x))$$

$$\texttt{WEIGHT}^{\texttt{OPT}}(S) = \mu_R \sum_{\alpha \in S} Q(\alpha)$$

Algorithm 9 is the same as Algorithm 6 described in Section 3.2. It advances a node x to the next value in the following way. First, it discards all the labels from YES-set to MAYBE-set. Then the algorithm starts moving labels from MAYBE-set to NO-set one by one until the optimistic weight of MAYBE-set is about to drop below the threshold value t. Then it sets x to the lowest node greater than the current x, associated with at least one label in NO-set.

Note that Algorithm 9 is non-blocking (as Algorithm 6 in the previous section). It outputs each node from the threshold path subset immediately after it finds it.

4.2.3 Adaptive Analysis

We perform an adaptive analysis of the complexities of Algorithm 8 and Algorithm 9 by using an extension of the alternation as our measure of difficulty, the smallest size of the partition-certificate that we introduce below. The definition of the partition-certificate is inspired by the similar structure for the adaptive analysis of the algorithm solving the Threshold Set Problem, described in Section 3.2. However, its intervals are different.

Algorithm 8 Answering Threshold Path SubSet Queries (T, R, Q, t)

$x = -\infty$, $\texttt{NO} = \texttt{YES} = \emptyset$, $\texttt{MAYBE} = \{\forall\alpha | Q(\alpha) > 0\}$;
$\texttt{Advance}(x, \texttt{YES}, \texttt{NO}, \texttt{MAYBE})$ (Algorithm 9);
while $x < +\infty$ **do**
  $\alpha \leftarrow$ the next label from $\texttt{MAYBE}$ in round-robin order;
  if $\texttt{label_ancestor}(\alpha, x) \neq \emptyset$ **then**
    $\texttt{YES} \leftarrow \alpha \leftarrow \texttt{MAYBE}$;
    if $\texttt{WEIGHT}(\texttt{YES}, x) \geq t$ **then**
      Output x;
      $x' = \texttt{follower}(x)$
      $\texttt{MAYBE} \leftarrow \forall\alpha$, s.t. $\texttt{label_successor}(\alpha, x) < x' \leftarrow \texttt{NO}$;
      $x = x'$;
    end if
  else
    $\texttt{NO} \leftarrow \alpha \leftarrow \texttt{MAYBE}$; (using the result from $\texttt{label_ssuccessor}(\alpha, x)$ and pushing to the priority queue)
  end if
  if $t \leq \texttt{WEIGHT}(\texttt{YES}, x)$ or $\texttt{WEIGHT}(\texttt{YES}, x) + \texttt{WEIGHT}^{\texttt{OPT}}(\texttt{MAYBE}) < t$ **then**
    $\texttt{Advance}(x, \texttt{YES}, \texttt{NO}, \texttt{MAYBE})$ (Algorithm 9);
  end if
end while

Algorithm 9 $\texttt{Advance}(x, \texttt{YES}, \texttt{NO}, \texttt{MAYBE})$

$\texttt{MAYBE} \leftarrow \texttt{YES}$; ($\texttt{YES} = \emptyset$;)
while $\texttt{WEIGHT}^{\texttt{OPT}}(\texttt{MAYBE}) \geq t$ **do**
  $\alpha \leftarrow \texttt{MAYBE}$ in round-robin order;
  $\texttt{NO} \leftarrow \alpha \leftarrow \texttt{MAYBE}$ ($\texttt{label_ssuccessor}(\alpha, x)$-operator);
end while
Find $S \subset \texttt{NO}$ of labels, such that $x_{\texttt{next}} = \texttt{label_ssuccessor}(\alpha, x)$ is the same and minimal;
$\texttt{YES} \leftarrow S \leftarrow \texttt{NO}$; (at most σ priority queue pops)
$x = x_{\texttt{next}}$;

As before, any algorithm answering a weighted threshold query has to check the correctness of its result. It corresponds to produce a certificate that each node in the answer set has a path-score of at least the threshold, and that each node that is not in the answer set either has an ancestor that is in this set or has a path-score smaller than the threshold.

Definition 4.2.5 *Consider a weighted multi-labeled tree T, a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$, and a non-negative integer t. A* partition-certificate *is a partition $I = (I_i)_{i \in [\delta]}$ of size δ on the set of nodes $[n]$, such that for any $i \in [\delta]$ one of the next three options is true:*

- *the node in I_i with the smallest number $x = \min I_i$ has path-score of at least t, and the node $y = \max I_i + 1$ is the follower of the node x (*a positive interval*);*
- *I_i is an interval that has a set $S \subseteq [\sigma]$ of labels, such that $\mu_R \sum_{\alpha \notin S} Q(\alpha) < t$ and $\sum_{x \in I_i} \sum_{y \in \mathtt{ancestors}(x) \cup \{x\}} R(\alpha, y) = 0, \forall \alpha \in S$ (*a negative interval*);*
- *I_i is a single node x with $\mathtt{path_score}(x) < t$ but does not have a set of labels $S \subseteq [\sigma]$, such that $\sum_{x \in I_i} \sum_{y \in \mathtt{ancestors}(x) \cup \{x\}} R(\alpha, y) = 0, \forall \alpha \in S$ and $\mu_R \sum_{\alpha \notin S} Q(\alpha) < t$ (*a negative singleton*).*

The partition-certificate consists of intervals of nodes. Each of them contains either all nodes from the sub-tree rooted in x that is in the threshold path subset, or a set of the consecutive nodes that have a set $S \in [\sigma]$ of labels they all lack with enough weight to guarantee these nodes not to be in the threshold set, or just a single node that is not in the threshold set but does not get into the second category.

		1	2	3	4	5	6	7	8	9
Classical	→	.	.	1	.	.	.	.	.	.
Home	→	3	.	.	.	.	.	.	.	.
Jazz	→	.	.	.	1	.	.	.	1	.
Music	→	.	2	.	.	.	.	.	.	.
Pop	→	.	.	.	1	1	.	.	.	.
Previews	→	.	.	.	.	.	.	.	.	1
Rock	→	.	.	.	.	1	.	1	.	.
Video	→	.	.	.	.	.	2	1	.	.

Figure 4.4: The encoding of the example in Figure 4.1 using a weighted binary relation. The null weights are noted by dots for better readability.

An instance might admit more than one partition-certificate. As we said, we study the performance of the algorithm as the function of the size of the smallest one, denoted by *alternation.*

		1	2	3	4	5	6	7	8	9
Home	→	3	.	.	.	.	.	.	.	.
Music	→	.	2	.	.	.	.	.	.	.
Pop	→	.	.	.	1	1	.	.	.	.
Previews	→	.	.	.	.	.	.	.	.	1

	1	2	3	4	5	6	7	8	9
→	3	*	*	*	*	*	*	*	*
→	.	2	*	*	*	.	.	.	.
→	.	.	.	1	1	.	.	.	.
→	.	.	.	.	.	.	.	.	1

Figure 4.5: An example of the minimal partition-certificate for the weighted tree in Figure 4.4 and the weighted query in Figure 4.3 with $t = 5$. This partition-certificate has all three possible types of intervals. The alternation of the instance is $\delta = 4$. By symbols '$*$' we describe all the nodes that are descendants of the node associated with the current label (this label affects these nodes as well, unlike in the similar case on binary relations).

Definition 4.2.6 *Consider a weighted multi-labeled tree T, a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$, and a non-negative integer t. The* alternation *of the instance is the smallest size δ of all possible partition-certificates for the given instance.*

As in the case of the previous chapter, we assume that `YES`-set and `MAYBE`-set are represented by a list abstract data type (ADT), because we only need to push and pop labels in and out in FIFO order. We assume that `NO`-set is represented by a priority queue ADT, because we need to push elements and pop those that have the smallest value of `label_ssuccessor`(α, x) with regard to the current node x.

We assume that the functions `WEIGHT` and $\texttt{WEIGHT}^{\texttt{OPT}}$ do not take more than a constant time to compute. In fact, this can be easily done. One should keep the last values of `WEIGHT` and $\texttt{WEIGHT}^{\texttt{OPT}}$ functions for all sets. Each time a labels is pushed in or popped out of a set, these function values are updated. This does not take more than a constant time for each push and pop and does not affect the total complexity of the algorithm.

We express our results in terms of the number of searches (performed by the `label_ssuccessor`(α, x) and `label_ancestor`(α, x) operators), as well as the number of priority queue operations (pushes as well as pops). The ways these operations can be supported and the final complexity of the algorithm in that cases will be discussed in the subsequent section.

In the formulations of all following theorems, we refer to `label_ancestor`()-ancestor and `follower`() operations as *advanced tree operations.*

Theorem 4.2.1 *Consider a weighted multi-labeled tree T, a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$, and a non-negative integer t. There is an algorithm that computes the threshold set for Q on non-increasing T, R with threshold-value of t in $\mathcal{O}(\delta k)$ search, priority queue, and advanced tree operations, where δ is the alternation of the instance and k is the number of labels of positive weight in Q.*

Proof 4.2.1 Algorithm 8 works in a similar to Algorithm 5 way. It starts from the first node in the interval under consideration and scans through all the nodes in increasing in preorder order. It jumps over some nodes to speed up the process. For each node under consideration, the algorithm explicitly determines whether this node is in the threshold path subset or not. Thus, to show the correctness of the algorithm we only need to show that all the nodes skipped by Algorithm 9 are not in the threshold set.

Consider the algorithm is jumping from the current node x to the next node x'. It sets x' to the smallest node greater then x which is associated with a label in `NO`-set, skipping all the intermediate nodes that might have labels in `MAYBE`-set associated with them. However, even if all the labels from `MAYBE`-set are associated with a node between x and x', their cumulative weight of $\sum_{\alpha \in \texttt{MAYBE}} Q(\alpha)$ combined with the highest possible value of an association μ_R will not be enough to quality this node for the threshold path subset, because $\texttt{WEIGHT}^{\texttt{OPT}}(\texttt{MAYBE}) < t$, which follows from the condition Algorithm 9 exists its main loop.

However, all the nodes skipped by this jump form a subtree of the node x and cannot be in the threshold path subset by definition of the partition-certificate (Definition 4.2.5). Because of this, Algorithm 8, unlike Algorithm 5, performs one additional jump from current node x to its follower $\texttt{follower}(x)$ just after it has qualified node x for threshold path subset.

Algorithm 8 processes a negative singleton in the similar way as Algorithm 5 does. It performs at most k iterations of the main loop to move enough labels from `MAYBE`-set to `YES`-set in order to decide whether x is in the threshold set or not. Each iteration takes at most $\mathcal{O}(1)$ search, priority queue, and advanced tree operations, resulting in $\mathcal{O}(k)$ for a negative singleton in total. Once the decision is made, Algorithm 9 jumps to the next candidate node x. Algorithm 9 performs at most $\mathcal{O}(k)$ search and priority queue operations as well. Hence, the total complexity of processing any singleton is $\mathcal{O}(k)$.

A positive interval starts with the node that is in the threshold path subset followed by all the nodes from its subtree. Algorithm 8 processes the first node of the interval spending the same time as Algorithm 5 does, except additional $\mathcal{O}(k)$ advanced tree operations. Before calling Algorithm 9 the algorithm has to skip to $\texttt{follower}(x)$. This takes one `follower` operation and at most $\mathcal{O}(k)$ priority queue operations. Hence, the total complexity of processing a positive interval is $\mathcal{O}(k)$.

When processing a negative interval, unlike Algorithm 5, Algorithm 8 might have to jump to the next candidate node x more than once, we analyse it by consider the way the algorithm moves labels from one set to another. The algorithm needs $\mathcal{O}(k)$ search, priority queue, and advanced tree operations to complete this interval of the partition-certificate (we refer a reader to the proof of Algorithm 5 for more details).

As the algorithm spends $\mathcal{O}(k)$ to process any interval, and any instance has δ intervals, the total complexity of the algorithm is $\mathcal{O}(\delta k)$ search and priority queue operations. □

4.3 Pertinent Path Subset Problem

4.3.1 Problem Statement

Threshold path subset for weighted queries on weighted multi-labeled trees, discussed in the previous section, is a proper relaxation of the path subset queries previously introduced [12]. But, as in the case of the threshold set for unweighted queries on unweighted binary relations, it is not enough in some situations.

In most applications, defining a proper threshold value will bother the user and, moreover, any predetermined threshold value can result in either very many partly relevant nodes or no nodes at all, depending on how labels are associated with the nodes and with what weights.

In this section, we consider a solution to this problem inspired by the results obtained in the previous chapter. The *Pertinent Path Subset* is the set of the most relevant nodes to a given query, those that give the highest path-score value. We formalize this notion and define a new type of query which corresponds to finding the *pertinent path subset*, the smallest non-empty threshold path subset.

The pertinent path score for a given tree is the maximum path score obtained by at least one node in this tree.

Definition 4.3.1 *Consider a weighted multi-labeled tree T, a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, and a weighted query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$. The* pertinent path-score *on T, R, and Q is the maximum path-score of any node x in $[n]$ on T, R, and Q:*

$$\texttt{pertinent_path_score}(T, R, Q) = \max_{x \in [n]} \{\texttt{path_score}(T, R, Q, x)\}$$

Having a pertinent path score defined, we may define a path subset of all nodes that obtain this path score, which is never empty.

Definition 4.3.2 *Consider a weighted multi-labeled tree T, a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, and a weighted query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$. The* pertinent set of the weighted unordered subset path query Q on T, R *is the set of nodes that have the maximum path-score on T, R, and Q:*

$$\texttt{PertT}(T, R, Q) =$$

$$\{x \in [\sigma], \texttt{path_score}(T, R, Q, x) = \texttt{pertinent_path_score}(T, R, Q)\}$$

The problem of finding such pertinent path subset is defined as the Pertinent Path Subset Problem.

4.3.2 Algorithm

Algorithm 10 for finding a pertinent path subset is similar to Algorithm 7 that finds a threshold path subset. It scans through the set of all nodes in the same way and checks

whether a particular node is in the pertinent path subset or not as it was done by Algorithm 7. Advancing to the next node is performed in the same way and by the same auxiliary Algorithm 9.

However, while Algorithm 10 is looking for a pertinent path subset, a set of nodes with maximal value of path-score which is not known in advance, it needs to maintain a value $t^\star$, the maximum value of the path-score obtained by any of the previous scanned nodes, and a set R of all previously scanned nodes that have score value of $t^\star$.

For each node under consideration x, the algorithm checks the score value $\texttt{path_score}(x)$, and put x to the current pertinent path subset R if $\texttt{path_score}(x)$ is equal to $t^\star$, the path-score value of all other nodes in R. If the $\texttt{path_score}(x)$ is greater than $t^\star$, the algorithm updates $t^\star$ to this new value and resets R to the one node x, because x is the only node with this score value so far.

Algorithm 10 Answering Pertinent Path SubSet Queries

```
x = −∞, NO = YES = ∅, MAYBE = {∀α|Q(α) > 0};
t* = 0, R = ∅;
Advance(x, YES, NO, MAYBE) (Algorithm 9);
while x < +∞ do
  α ← the next label from MAYBE in round-robin order;
  if label_ancestor(α, x) ≠ ∅ then
    YES ← α ← MAYBE;
    if WEIGHT^OPT(MAYBE) = 0 then
      if WEIGHT(YES, x) > t* then
        t* = WEIGHT(YES, x);
        R = ∅;
      end if
      if WEIGHT(YES, x) ≥ t* then
        R ← x;
        x′ = follower(x)
        MAYBE ← {∀α, s.t. label_successor(α, x) < x′} ← NO;
        x = x′;
      end if
    end if
  else
    NO ← α ← MAYBE; (using the result from label_ssuccessor(α, x) and pushing to the
    priority queue)
  end if
  if t ≤ WEIGHT(YES, x) or WEIGHT(YES, x) + WEIGHT^OPT(MAYBE) < t then
    Advance(x, YES, NO, MAYBE) (Algorithm 9);
  end if
end while
```

Note that as in the case of Algorithm 7, Algorithm 10 is blocking. It has to scan all the nodes and computer full pertinent path subset in order to give a user any nodes from the answer pertinent set. The reason for this is that the current pertinent path subset can be reseted by the algorithm at any moment if it finds a node with the path score greater than the maximum that each of the nodes found before has, making preliminary output of the nodes impossible.

4.3.3 Adaptive Analysis

Theorem 4.3.1 *Consider a weighted multi-labeled tree T, a weighted binary relation $R : [\sigma] \times [n] \to \{0, \ldots, \mu_R\}$, and a weighted query $Q : [\sigma] \to \{0, \ldots, \mu_Q\}$. There is an algorithm that computes the pertinent path subset of the weighted unordered subset path query Q on T, R. Once it has found the first node of the pertinent set, this algorithm performs $O(\delta k)$ additional search, priority queue, and advanced tree operations, where δ is an alternation on this instance with $t =$ `pertinent_path_score`(T, R, Q) and k is the number of labels of positive weight in Q.*

Proof 4.3.1 Once Algorithm 10 has found the first node of the pertinent path subset, its execution does not differ from the execution of Algorithm 8, except that it continues to check whether an node under consideration has larger path-score value than $t^\star$ (the algorithm does not know that the $t^\star$ found is the maximal one).

To make this check, for each node x under consideration, Algorithm 10 continues to move labels from `MAYBE`-set to `NO`-set or `YES`-set until `MAYBE`-set is empty. It retrieves at most k labels in this way for each node x, the amount that Algorithm 8 is still allowed to do by Theorem 4.2.1 without increasing of upper bound.

As during the processing of each interval Algorithm 10 does not perform more operations than Algorithm 8 in the worst case, the complexity of Algorithm 10 does not differ from the complexity of Algorithm 8 after it has found the first node in the pertinent set. □

Note that, as in the case of the algorithm for finding pertinent set for a weighted query on weighted binary relations, before finding the first node of the pertinent set, Algorithm 10 might perform an amount of operations that does not depend on the alternation of the instance. In the worst case, the first node from the pertinent path subset is very large and the algorithm needs to increase $t^\star$ up to t times and put up to n nodes to the temporary pertinent sets before finding the first node from the real pertinent path subset set. The algorithm might compute a temporary result as large as n, even though the final result is small or null.

4.4 Implementation

In the previous sections we derive an upper bound on the complexity of the Threshold Path Subset and Pertinent Path Subset Algorithms expressed in the number of searches in a

binary relation, a priority queue, and advanced tree operations. There are a number of data structures to support these operations.

In this section we consider the same ways a binary relation and a priority queue can be implemented that were discussed in Section 3.4 and derive exact upper bounds on the complexity of the Threshold Path Subset and the Pertinent Path Subset Algorithms.

Corollary 4.4.1 *If a labeled tree is implemented by postings lists and a priority queue is implemented by a classical heap structure, the complexity of Algorithm 8 / 9 and Algorithm 10 / 9 is* $O(\delta \sum_{\alpha \in [\sigma]} \lg(n_\alpha/\delta) + \delta k \lg k)$*, where* t *is the sum of the sizes of all postings lists and* k *is the maximum size of the priority queue.*

Proof 4.4.1 Similar to Corollary 3.4.1, except the support for the `follower`(x) operation. However, if one keeps a tree structure in the memory, which one is needed to do to preserve the tree structure information anyway, one can always find the parent and look for the next child in the tree structure in a constant time, i.e. without influencing the overall complexity of the algorithm. □

Corollary 4.4.2 *If the labeled tree is implemented by a succinct encoding introduced by Barbay* et al. *[10] and the priority queue is implemented in the way proposed by Andersson and Thorup [6], the complexity of Algorithm 8 / 9 and Algorithm 10 / 9 is* $O(\delta k \lg\lg \sigma + \delta k (\lg\lg k)^2)$ *in the RAM model with word size* $\Theta(\lg \max\{\sigma, n\})$*, where* k *is the maximum size of the priority queue.*

Proof 4.4.2 Similar to Corollary 3.4.2, except the support for the `follower`(x) operation. However, it can be implemented by adding a number of a node descendants to the current node number (standard for labeled tree `tree_nbdesc`(x) operation that is supported in $\mathcal{O}(\lg\lg \sigma)$ time [10] by the same encoding). □

After one find other ways to implement binary relations, he can apply Theorem 4.2.1 and Theorem 4.3.1 to derive new upper bounds on the complexity of Algorithm 8 and Algorithm 10.

Chapter 5

Weighted LCA Queries on Labeled Trees

In this chapter we deal with *weighted lowest common ancestor (lca) queries* on *weighted labeled trees* to solve the problem of retrieving relevant documents from a large tree-like data storage using keywords. While weights in labeled trees allow to express the degree of dependence between keywords and their nodes more precisely, weights in queries differentiate the importance of each keyword for the result.

We consider the *Threshold LCA Problem* to find all those documents that are relevant enough according to the given weighted binary relation, weighted query, and a threshold value. We provide an adaptive algorithm to solve this problem and find such set of documents. We analyse presented algorithm through introducing an adaptive measure of difficulty and proving the upper bound on the number of high-level operations this algorithm performs. Finally, we describe two ways the presented algorithms can be implemented and derive the upper bounds on their complexity in comparison-based and word-RAM models of computations.

The chapter is organized as follows. In Section 5.1 we provide the basic definitions that the reader will need further. In Section 5.2 we introduce the Threshold LCA Problem and provide an algorithm to solve it followed by the analysis of its complexity. In Section 5.3 we discuss how this algorithm can be implemented in practice and derive upper bounds on its complexity for different implementations.

5.1 Basic Definitions

In this chapter, we deal with simple (unweighted) multi-labeled trees. These trees are similar to the ones used in Chapter 4, except they have boolean weights associated with pairs of nodes and labels.

Definition 5.1.1 *Consider a rooted tree T with a set of nodes $[n]$ and a set of labels $[\sigma]$. A* multi-labeled tree (T, R) *is composed of a rooted tree T and a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$ between the set of labels $[\sigma]$ and the set of nodes $[n]$.*

Given two nodes in a multi-labeled tree, one often needs to find a dependency between them. The notion of the *Nearest Common Ancestor (NCA)* or *Lowest Common Ancestor (LCA)* [2, 3, 16, 32] is widely-used for this purpose.

Definition 5.1.2 *Given a rooted tree T, the Lowest Common Ancestor (LCA) of two nodes x_1 and x_2 is the node y with the property that it is an ancestor of x_1 and x_2 and no descendant of y satisfies this property.*

One can define the Lowest Common Ancestor for more nodes in a similar way.

Definition 5.1.3 *Given a rooted tree T, the Lowest Common Ancestor of k nodes $\{x_1, \ldots, x_k\}$ is the ancestor y of all k nodes, s.t. there is no descendant of y with the same property.*

The concept of the lowest common ancestors is relevant in the case when we have a family of sets of nodes and want to identify a structural dependency between these sets. To do this, we can find such nodes that have at least one node from each set in their subtrees. This idea was used, for example, by Xu and Papakonstantinou [46]. We define it here more formally:

Definition 5.1.4 *Consider a multi-labeled tree (T, R). Given a subset of labels $S \subseteq [\sigma]$, which define the sets of nodes $S_1, \ldots, S_{|S|}$, where each node $x \in S_i$ is associated with at least i-th label from the set S,* the Common Ancestor set (CA-set) *of sets $S_1, \ldots, S_{|S|}$ is the set of all nodes, each of which has at least one node from each set $S_i \in \{1, \ldots, |S|\}$ in its subtree.*

As a node is closer to the root, the number of its descendants is larger. We are interested only in those nodes in the CA set, that are farthest from the root, i.e. in those ones that have no descendants that are also in the CA set. Xu and Papakonstantinou [46] defined the set of all such nodes as the set of the *Smallest Lower Common Ancestor (SLCA)*, however we call it the *Lower Common Ancestors (LCA) set* due to the linguistic meaning.

Definition 5.1.5 *Consider a multi-labeled tree (T, R). Given a subset of labels $S \subseteq [\sigma]$, which define the sets of nodes $S_1, \ldots, S_{|S|}$, where each node from set S_i is associated with the i-th label from set S,* the Lowest Common Ancestor set (LCA set) *of sets $S_1, \ldots, S_{|S|}$ is the set of all nodes, each of which has at least one node from each set S_i in its subtree and does not have a descendant with the same property.*

When dealing with the Threshold and the Pertinent Path Subset Problems in Chapter 4, the path-score of a node depends on the nodes on its path to the root, which are higher in the tree. However, the nodes of interest in this chapter are located deeper, so that the postorder ordering of the nodes of the tree is more convenient for threshold LCA problem considered below.

Weighted queries used in the previous chapters are convenient for expressing user's needs. In this chapter, we define them in a similar way as it was done in Chapter 4.

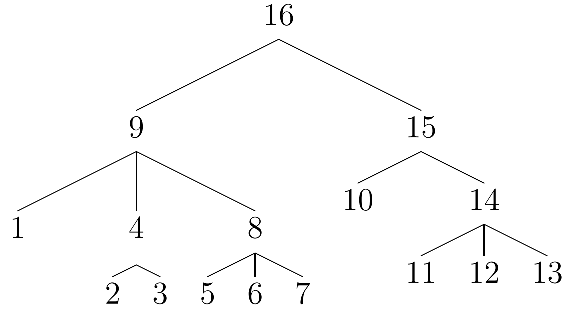


Figure 5.1: An example of the postorder enumeration of the nodes in a tree.

Definition 5.1.6 *Consider a set of labels* $[\sigma]$. *A* weighted LCA query *is a function* $Q : [\sigma] \to \Re^+$, *where each label* $\alpha \in [\sigma]$ *is associated with a non-negative weight* $Q(\alpha) \geq 0$.

5.2 Threshold LCA Problem

5.2.1 Problem Statement

As in the case of path subset problems, for any query entered by a user, there are more relevant and less relevant nodes in a multi-labeled tree. In order to differentiate them, we define a notion of relevance of an object to the query and show how to search and retrieve the most relevant objects.

While in the path subset queries the pertinence of a node depends on the labels associated with this node and the nodes on its rooted path, in the case of LCA queries, the pertinence depends on the labels associated with the nodes in the subtree of a given node.

For a given weighted LCA query Q on a multi-labeled tree (T, R), we define *the lca-score of a node* x as the sum of weights of all labels that are associated with at least one node y from the subtree rooted by x.

Definition 5.2.1 *Consider a multi-labeled tree* (T, R) *and a weighted LCA query* $Q : [\sigma] \to \Re^+$. *We define the* lca-score *of a node* x *for* Q *on* T *and* R *as*

$$\texttt{lca_score}(T, R, Q, x) =$$

$$\sum_{\alpha \in [\sigma]} Q(\alpha)\ \mathbb{1}(\exists y \in \texttt{descendants}(x), s.t.\ R(\alpha, y) = 1)$$

where $\texttt{descendants}(x)$ *denotes the set of all descendants of a given node* x *together with node* x *and* $\mathbb{1}(X) = 1$ *is an indicator whether* X *is true.*

The further a node is from the bottom of the tree, the higher its lca-score: any ancestor of the node x gets a lca-score of at least the same value as the lca-score of x. We are interested in the closest to the bottom (or the farthest from the root) nodes – the first ones that get high enough lca-score, as they are sufficient to describe the set of all such nodes.

In previous chapters, we discussed the usefulness of relaxation of search problems in order to improve search results. By using weighted LCA query Q and setting a threshold t on the required value of lca-score here, we relax CA and LCA sets to their threshold versions.

Definition 5.2.2 *Consider a multi-labeled tree* (T, R)*, a weighted LCA query* $Q : [\sigma] \rightarrow \Re^+$*, and a non-negative integer* t*. The* t-CA set *of a weighted LCA query* Q *of threshold* t *is the set of nodes that have lca-score of at least* t*:*

$$\texttt{ThresCA}(T, R, Q, t) = \{x \in [n] : \texttt{lca_score}(T, R, Q, x) \geq t\}$$

Definition 5.2.3 *Consider a multi-labeled tree* (T, R)*, a weighted LCA query* $Q : [\sigma] \rightarrow \Re^+$*, and a non-negative integer* t*. The* t-LCA set *of a weighted LCA query* Q *of threshold* t *is the subset of* t*-CA set of nodes that do not have any descendants that are in* t*-CA set as well, i.e. they are deepest ones with such property:*

$$\texttt{ThresLCA}(T, R, Q, t) =$$

$$\{x \in \texttt{ThresCA}(T, R, Q, t) : \forall y \in \texttt{ThresCA}(T, R, Q, t) \Rightarrow y \notin \texttt{descendants}(x)\}.$$

Solving the Threshold LCA Problem *consists in finding the* t*-LCA set of a weighted LCA query.*

5.2.2 Algorithm

For any label $\alpha \in [\sigma]$ and any node $x \in [n]$ we use the same notion of the operation α-*successor of* x as in Chapter 4 but with respect to the postorder defined on the set of nodes $[n]$. In addition, our algorithms require some operations that were not defined previously.

Definition 5.2.4 *Consider a multi-labeled tree* (T, R) *with nodes enumerated in postorder. For a node* $x \in [n]$ *and a label* $\alpha \in [\sigma]$ *we define:*

- *the* α-predecessor of x*, denoted by* $\texttt{label_predecessor}(\alpha, x)$*, outputs the largest in preorder node* y *that is less than node* x*, such that* $R(\alpha, y) = 1$*, or* $\emptyset$ *if there is no such node.*

- *the* lowest common ancestor (LCA) of x_1 and x_2*, denoted by* $\texttt{lca}(x_1, x_2)$*, outputs the node that is an ancestor of both of them and that does not have any descendant node with the same property (see Definition 5.1.2).*

Our algorithm for solving the Threshold t-LCA Problem starts from the first node in postorder and scans through the list of all the nodes enumerated in postorder. For each node x under consideration the algorithm finds the smallest node x_{right} larger than x, such that all the labels associated with the nodes between x and x_{right} add weight to the lca-score of their Lowest Common Ancestor of at least t, i.e. $\sum_{\alpha\in[\sigma]} Q(\alpha) \max_{y\in\{x,\ldots,x_{right}\}} R(\alpha, y) \geq t$. Then it finds the largest node x_{left} with the same property but starting from x_{right} and

progressing backward: x_{left} is the largest node smaller than x_{right}, such that all the labels associated with the nodes between x_{left} and x_{right} add weight to the lca-score of their Lowest Common Ancestor of at least t. The algorithm outputs the lowest common ancestor of these two nodes $\texttt{lca}(x_{left}, x_{right})$ and proceeds further, by setting the current node x to the smallest successor of node x_{left} that matches at least one label from the query. The algorithm finishes when it cannot find the next node x_{right} (see Algorithm 11).

The function $\texttt{WEIGHT}(x_{left}, x_{right})$ denotes the weight of all labels associated with the nodes placed between nodes x_{left} and x_{right} in postorder including themselves. These nodes will contribute this weight to the lca-score of their lowest common ancestor (see Definition 5.1.3), which is the same as the LCA of the leftmost and rightmost nodes (Definition 5.1.2).

$$\texttt{WEIGHT}(x_{left}, x_{right}) = \sum_{\alpha \in [\sigma]} Q(\alpha) \max_{x \in \{x_{left}, \ldots, x_{right}\}} R(\alpha, x)$$

Algorithm 11 Solving the Threshold t-LCA Problem (T, R, Q, t)

$x = x_1$;
loop
  $x_{right} \leftarrow$ the first successor of x, s.t. $\texttt{WEIGHT}(x, x_{right}) \geq t$;
  if $x_{right} = +\infty$ then EXIT;
  $x_{left} \leftarrow$ the first predecessor of x_{right}, s.t. $\texttt{WEIGHT}(x_{left}, x_{right}) \geq t$;
  $\texttt{output} \leftarrow \texttt{lca}(x_{left}, x_{right})$
  $x \leftarrow$ the next node larger than x_{left} associated with any label α, s.t. $Q(\alpha) > 0$;
end loop

To find node x_{right} the algorithm computes nodes $x_\alpha = \texttt{label_successor}(\alpha, x)$ for each label $\alpha, Q(\alpha) > 0$ and builds a min-priority queue from them. Then it retrieves the smallest set of the nodes such that the sum $Q(\alpha)$ of all corresponding labels is at least t. The algorithm sets x_{right} to the last retrieved node.

The algorithm finds x_{left} in a similar way: it computes nodes $x_\alpha = \texttt{label_predecessor}(\alpha, x_{right})$ for each label $\alpha, Q(\alpha) > 0$ and builds a max-priority queue from them. Then it retrieves the smallest set of the nodes such that the sum $Q(\alpha)$ of all corresponding labels is at least t. The algorithm sets x_{left} to the last retrieved node.

The result of the algorithm solving the Threshold t-LCA Problem is a sequence of nodes. Any outputted node x is in the t-CA set whose lca-score is of at least t by construction of the algorithm. However, not all of returned nodes belong to the t-LCA set, because the algorithm does not check any ancestor-descendant dependencies between outputted nodes. To remove nodes that are not in the t-LCA set from the output, we need an additional *filtering* step.

Our on-line filtering algorithm is inspired by the one provided implicitly by Xu and Papakonstantinou [46]: it receives nodes from the first algorithm one by one and checks whether each of them is in the t-LCA set or not. The algorithm stores the most recent received node as $\mathtt{x_r}$ and check the relationship between $\mathtt{x_r}$ and the new node x that comes

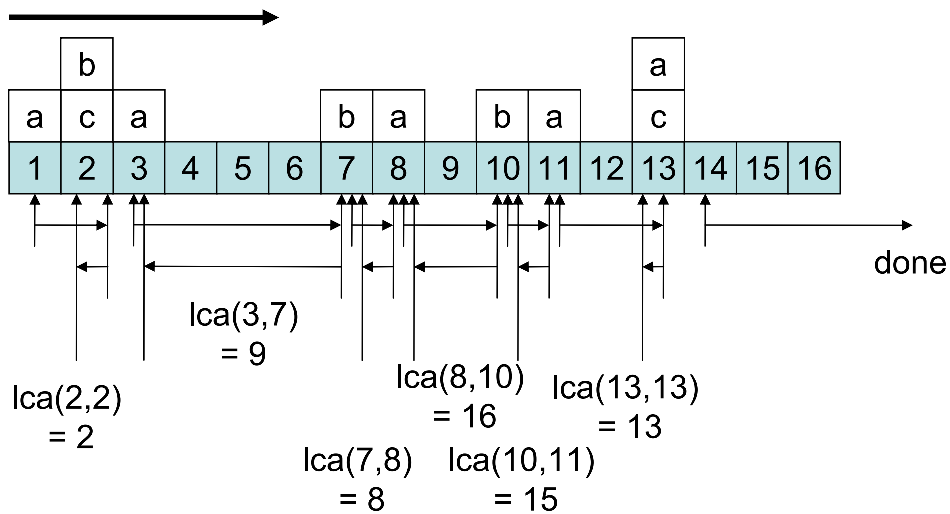


Figure 5.2: An example of the execution of the Algorithm 11 on the labeled tree provided in Figure 5.3.

to the input. If x is an ancestor of $\mathtt{x_r}$, x is discarded, because it cannot be in t-LCA set by definition. If x is a descendant of $\mathtt{x_r}$, $\mathtt{x_r}$ is discarded and replaced by x, because $\mathtt{x_r}$ cannot be in t-LCA set. Otherwise, our filtering algorithm sends $\mathtt{x_r}$ to the output (i.e. confirms that it is in t-LCA set), and set $\mathtt{x_r}$ to the newcomer node x. When there is no more nodes coming to the input, the algorithm outputs $\mathtt{x_r}$ to the t-LCA set and finishes.

Algorithm 12 Filtering Algorithm (T, t-CA)

```
x_r ← the first node coming to the input;
for each x coming to the input do
  x_lca = lca(x_r, x);
  if x_lca = x_r then
    do nothing (discard x)
  else if x_lca = x then
    x_r = x
  else
    output x_r;
    x_r = x
  end if
end for
output x_r;
```

5.2.3 Adaptive Analysis

In order to analyse the algorithm, we define a notion of the complexity of the problem instance that shows which problem instances are easy and which are difficult to solve. To do this, we highlight the most important parts of the problem instance.

Consider we take x_{left} and x_{right} and compute the possible additional weight they and all the nodes between them can contribute to the lca-score value of their common ancestor. If we increase this interval, the weight will increase or, at least, remain the same. If we decrease this interval, the weight will decrease or remain the same. We define a notion of the smallest interval that still contributes enough weight to the lca-score of a node to make sure it is in the t-CA set.

Definition 5.2.5 *Consider a multi-labeled tree* (T, R)*, a weighted LCA query* $Q : [\sigma] \rightarrow \Re^+$*, and a non-negative integer* t*.* The Minimal t-interval *of the problem instance is an interval* $(x_{left}, \ldots, x_{right})$ *of nodes in postorder, s.t.* $\sum_{\alpha\in[\sigma]} Q(\alpha) \max_{x\in\{x_{left},\ldots,x_{right}\}} R(\alpha, x) \geq t$ *and there is no subinterval with the same property.*

Each node in the t-LCA set corresponds to a minimal interval that has lca-score of at least t; note that the inverse is not always true. As the number of the intervals of minimal size increases the number of nodes in the t-LCA set increases as well. We take the number of minimal t-intervals as our *measure of difficulty* of the corresponding problem instance.

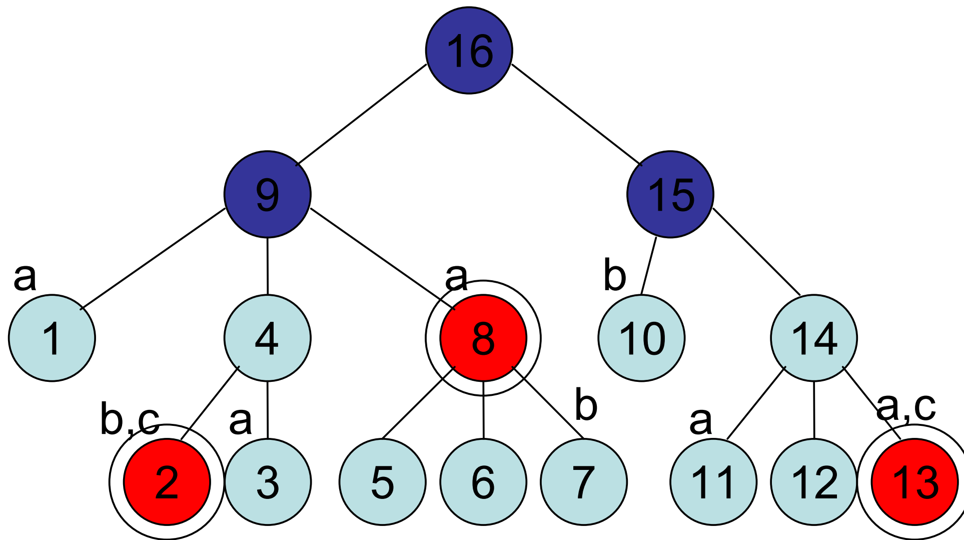


Figure 5.3: An example of a multi-labeled tree. The gray nodes are in the 2-CA set and circled nodes are in the 2-LCA set built on the labels {a,b,c}.

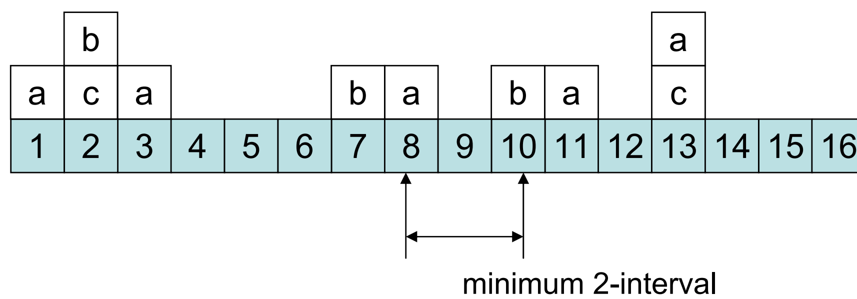


Figure 5.4: The postorder of the nodes of the tree in Figure 5.3 and one of the minimal 2-intervals.

Definition 5.2.6 *Consider a multi-labeled tree (T, R), a weighted LCA query $Q : [\sigma] \rightarrow \Re^+$, and a non-negative integer t.* The difficulty δ *of the problem instance is the number of distinct minimal t-intervals of the given problem instance.*

We prove dependency between nodes in the t-CA set, t-LCA set, minimal t-intervals, and iterations of Algorithm 11. We represent these dependences graphically in Figure 5.6.

Lemma 5.2.1 *For a given problem instance defined by a multi-labeled tree (T, R), a weighted LCA query $Q : [\sigma] \rightarrow \Re^+$, and a non-negative integer t, there is a bijection $f_1()$ between the set of minimal t-intervals and each iteration of Algorithm 11.*

Proof 5.2.1 We prove this lemma by considering a function $f_1()$ from the set of all minimal t-intervals to the set of all iterations of the algorithm and proving that it is injective and surjective, which leads to being bijective.

To prove injectivity of $f_1()$ we only need to show that for each two different minimal t-intervals I_1 and I_2 the corresponding iterations $f_1(I_1)$ and $f_1(I_2)$ of Algorithm 11 are different. As the intervals has to start from different nodes (otherwise they cannot be different, because they are minimal), and by the fact that each iteration of the algorithm starts from one of the successors of the node the previous iteration started, $f_1()$ is injective.

To prove surjectivity of $f_1()$ we need to show that for each iteration of Algorithm 11, there exists a minimal t-interval. This is true, because an interval the algorithm comes up at each iteration has the property that if one removes either the first or the last node from it, it will have weight less than t by definition of the algorithm, i.e. it is a minimal t-interval that will certainly have a correspondence in the set of *all* t-intervals.

From the injectivity and surjectivity of $f_1()$, its bijectivity follows. □

Lemma 5.2.2 *For a given problem instance defined by a multi-labeled tree (T, R), a weighted LCA query $Q : [\sigma] \rightarrow \Re^+$, and a non-negative integer t, there is an injection $f_2()$ from t-LCA set to a set of all minimal t-intervals.*

Proof 5.2.2 We prove this by showing that for any pair of different nodes x_1, x_2 in t-LCA set, their images $f_2(x_1)$, $f_2(x_2)$ in the set of all minimal t-intervals are different.

As nodes x_1 and x_2 cannot have ancestor-descendant dependency, because an ancestor is not in t-LCA set in this case, they have totally disjoint subtrees (the trees that do not have any common node). The minimal t-interval that gives the required weight to the node has to be inside the interval corresponding to the entire subtree of a node. As the subtrees of these two nodes do not intersect, the minimal t-intervals do not intersect either, meaning they cannot be the same. □

Lemma 5.2.3 *For a given problem instance defined by a multi-labeled tree (T, R), a weighted LCA query $Q : [\sigma] \rightarrow \Re^+$, and a non-negative integer t, and assuming that Algorithm 11 outputs all nodes that are in t-LCA set and some or no nodes that are not in t-LCA set but are in t-CA set, Filter Algorithm 12 outputs all and only nodes from t-LCA set.*

Proof 5.2.3 We prove the lemma's statement by proving that all the nodes discarded by the filtering algorithm are not in t-LCA set and all the nodes it outputs are in t-LCA set.

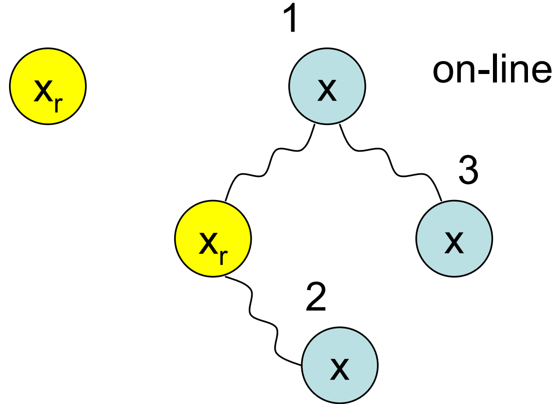


Figure 5.5: Three types of incoming nodes during filtering.

The filtering algorithm discards nodes only in two cases. First, it discards $\mathtt{x_r}$ when it finds that the next coming node is its child, which means that node $\mathtt{x_r}$ cannot be in the t-LCA set by definition. Second, it discards the coming node x when it appears to be an ancestor of node $\mathtt{x_r}$, which means it cannot be in the t-LCA set as well.

All the time the filtering algorithm is working, it keeps node $\mathtt{x_r}$ to be not an ancestor of any of the nodes in the input (here we use the order the Algorithm 11 outputs the nodes in t-CA set to the filtering algorithm). The filtering algorithm outputs $\mathtt{x_r}$ if the coming node belongs to the next branch of the tree and cannot ban $\mathtt{x_r}$ from being in t-LCA set as well as all the subsequent nodes coming. □

Theorem 5.2.1 *For a given problem instance defined by a multi-labeled tree* (T, R)*, a weighted LCA query* $Q : [\sigma] \rightarrow \Re^+$*, and a non-negative integer* t*, there is an algorithm that solves the Threshold* t*-LCA Problem performing* $\mathcal{O}(k\delta)$ *priority queue, label-successor, label-predecessor operations and* $\mathcal{O}(\delta)$ *lca operations.*

Proof 5.2.1 The correctness of the algorithm follows from Lemma 5.2.1, Lemma 5.2.2, and Lemma 5.2.3: the minimal t-interval of any node in the t-LCA set will be presented in the set of all minimal t-intervals (Lemma 5.2.2), will be found by Algorithm 11 and passed to the input of the Filtering Algorithm 12 (Lemma 5.2.1) which will output it successfully to the result set (Lemma 5.2.3).

Each iteration of Algorithm 11 costs $\mathcal{O}(k)$ priority queue and label-successor operations for finding x_{right}, $\mathcal{O}(k)$ label-predecessor and priority queue operations for finding x_{left}, and one lca operation to determine a node to output to Filtering Algorithm 12, and one lca operation performed by Filtering Algorithm for every node from the input. As the number of iterations of Algorithm 11 is δ, the total worst-case complexity is bounded by $\mathcal{O}(k\delta)$ of priority queue, label-successor, label-predecessor operations and $\mathcal{O}(\delta)$ lca operations.

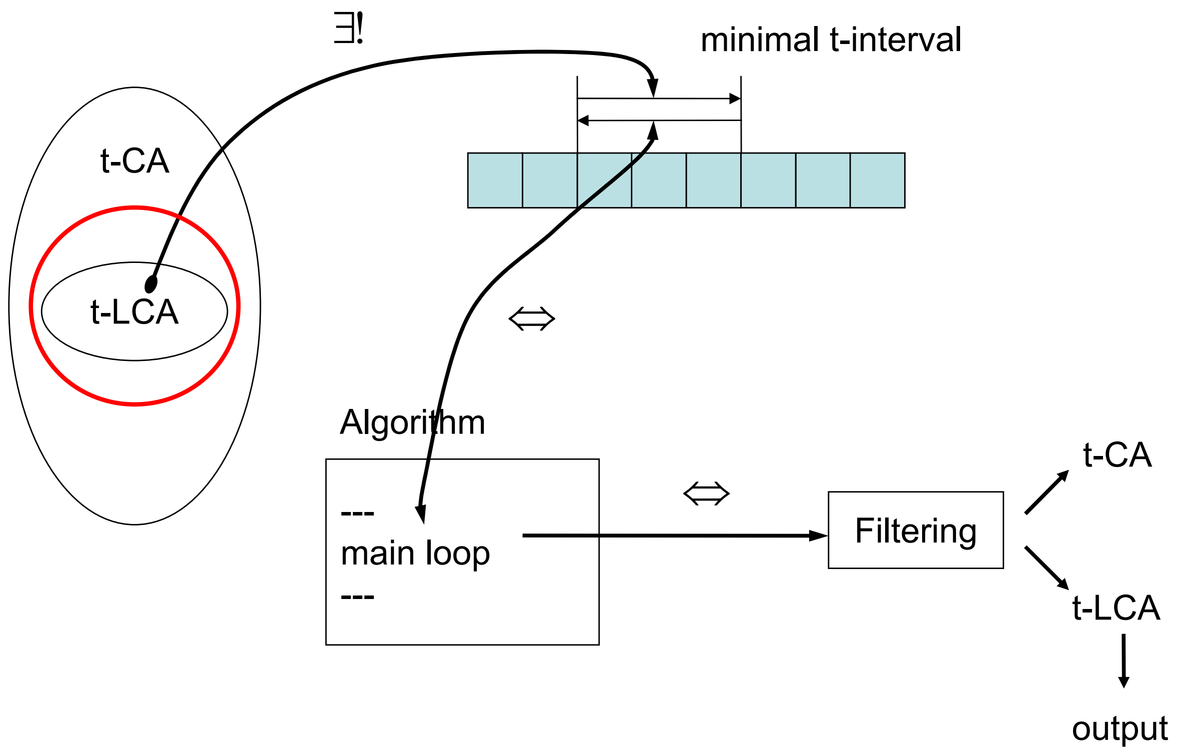


Figure 5.6: The global picture of the results of the lemmas. The set of all nodes that correspond to all minimal t-intervals is a subset of t-CA set but contains t-LCA set completely. The algorithm finds the corresponding node for each minimal t-interval and determines whether it is in the t-LCA set or not through filtering.

Lemma 5.2.1 and the definition of the instance difficulty through the number of distinct minimal t-intervals (Definition 5.2.6) together with the work flow of Algorithm 11 gives the mentioned worst-case complexity. □

It is natural to add weights to the multi-labeled tree T through making a binary relation R weighted, as it was done in the case of path subset queries in Chapter 4. However, this strategy changes the problem greatly and our algorithm cannot be easily adapted to it.

In addition, we cannot put a restriction similar to the one of non-increasing trees used for solving path subset problem, because the position of the nodes that contribute to the lca-score of a given node now is too unpredictable: they are not grouped on the same rooted path as in the case of path subset queries, but are spread across the whole subtree of the node randomly, thus Algorithm 11 cannot search for the next interval that can contribute to the t-CA set effectively.

Unlike two previous chapters, the idea of pertinent t-LCA set is not too meaningful here. The node that has the maximum lca-score is always known – it is the root of the whole tree, even though it is not always unique. And looking for the nodes with the same lca-score as the root is trivial: one can easily find the maximal threshold value $t^\star$, which brings us to solving of the original Threshold t-LCA Problem.

5.3 Implementation

In the previous sections we derived an upper bound of the complexity of the Threshold t-LCA Problem solving expressed in the number of priority queue, label-successor, label-predecessor operations and lca operations. There are a number of data structures to support these operations.

In this section we consider the same ways a binary relation and a priority queue can be implemented as the ones discussed in Section 4.4 and derive upper bounds on the worst-case complexity of the algorithm for solving the Threshold t-LCA Problem.

Corollary 5.3.1 *If a multi-labeled tree is implemented by postings lists and a priority queue is implemented by a classical heap, the complexity of the combination of Algorithm 11 and Algorithm 12 is* $O(\delta \sum_{\alpha\in[\sigma]} \lg(n_\alpha/\delta) + \delta k \lg k + \delta h)$*, where* k *is the number of labels* α *with* $Q(\alpha) > 0$ *and* h *is the height of the tree.*

Proof 5.3.1 One can implement the algorithms in the same way as in Corollary 3.4.1, except the support for the $\texttt{lca}(x_1, x_2)$ operation, which can be done by scanning of a rooted path in time linear in the size of the tree. □

Corollary 5.3.2 *If the labeled tree is implemented using the succinct encoding introduced by Barbay* et al. *[10] and the priority queue is implemented as proposed by Andersson and Thorup [6], the complexity of the combination of Algorithm 11 and Algorithm 12 is*

$O(\delta k \lg\lg \sigma + \delta k(\lg\lg k)^2)$ in the RAM model with word size $\Theta(\lg \max\{\sigma, n\})$, where k is the number of labels α with $Q(\alpha) > 0$.

Proof 5.3.2 One can implement the algorithms in the same way as in Corollary 3.4.2, except for the support of `lca`(x_1, x_2) operation, which can be done in the way proposed by Geary *et al.* [28] in $\mathcal{O}(1)$ time. □

Chapter 6

Conclusion

In this thesis we provided new ways of asking keyword queries and storing information in the database by adding weights to the existing solutions to improve the quality of the information retrieving. We considered weighted queries on two different data structures: weighted binary relations and weighted multi-labeled trees. We proposed adaptive algorithms to solve these queries and proved the measures of the complexity of these algorithms in terms of the high-level operations. We described how these algorithms can be implemented and derived the upper bounds on their complexity in two specific models of computations: the comparison model and the word-RAM model.

First, we gave a broad overview of the work done in the related areas such as the adaptive analysis of algorithms and presented some results about queries on binary relations and multi-labeled trees. Then, taking into account the benefits of the pure keyword search, we extended it by adding weights to the keywords in the query. These weights allow to specify which keywords are more or less important in order to increase the relevance of the information retrieved.

Next, we identified the intuition behind previous work on threshold set queries on binary relations [11] and applied it in much more general contexts: where weights are associated with the relation between objects and labels. We applied the threshold set concept to path-subset [10] and LCA queries and derived algorithms for solving them. In the contexts of weighted queries and weighted path-subset queries we defined pertinent queries that find the optimal threshold and are more informative than the queries previously considered, while being not substantially more expensive to answer.

Finally, we performed an adaptive analysis of all presented algorithms by defining measures of difficulty of the problems instances. We elaborated on the implementation issues and derived upper bounds on the complexity of all given algorithms in the specific models.

The concept of weighted threshold and pertinent set queries can be applied to some other type of schema-free queries on multi-labeled trees, among which we describe three in particular:

- *additive path subset queries*, which are similar to path-subset queries but with a different score function, where for each node x the contribution of its ancestors labeled α adds up to form its score;

- *path subsequence queries*, which are similar to path-subset queries but with a required order on the labels of the query (obviously, the path-score can then be defined in an additive or non-additive way);

Additive path subset queries is a natural extension of the path subset queries considered in this thesis. They allow multiply instances of the same label on the same rooted path add weight to the score of the considered node. They would be very useful in practice because, as in the case of the file system, if a keyword appears more than once in the path to a file, this file is more relevant to a query containing this keyword. However, considering such queries will make our algorithm much more complex: it will not have a bound on the score that a given label may contribute and will have to consider all occurrences of each label without skipping them, as it currently does. This will make the algorithm's complexity to be linear in the total number of labels, unless one find another way to perform the scanning.

Path subsequence queries is one more possible extension of the path subset queries. By imposing the order constraints on the label appearance in the rooted path, one can filter out unnecessary branches of the tree. However, the algorithm should be changed greatly in this case. Moreover, the adaptive analysis of the algorithm in this situation is an open question as well.

In addition, as threshold set queries generalize simple keyword queries, for which many algorithms have been studied [7, 8, 25, 26], many other algorithms should be considered, some of which could take advantage of properties of the instances other than those described by the *alternation* measure of difficulty.

Although the adaptive analysis is finer than the classical worst-case analysis, its value for a particular application depends of the appropriateness of the corresponding difficulty measure: some experimentations will be necessary. It is reasonably easy to generate a weighted index, for instance by assigning different weights to the labels associated with the links to a web page, in the title or in a simple paragraph. It will be harder to generate realistic user queries for the threshold set: the users are most often used to using keyword queries and adapting their queries to the type of results returned: they give a small number of keywords to avoid receiving a null answer. A possible solution is to consider queries *extended* with some labels of small weight, corresponding to the profile of the user: such queries would help to adjust the answer to the initial keyword query based according to the user preferences.

Bibliography

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural joins: A primitive for efficient XML query pattern matching.

[2] S. Alstrup. Optimal algorithms for finding nearest common ancestors in dynamic trees. Technical Report 95-30, Universitetsparken 1, DK-2100 Denmark, 1996.

[3] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[4] S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Automata, Languages and Programming*, pages 73–84, 2000.

[5] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Holstege, D. McBeath, M. Rys, and J. Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text. Technical report, W3C Working Draft, May 2006. http://www.w3.org/TR/xquery-full-text/.

[6] A. A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 335–342, New York, NY, USA, 2000. ACM Press.

[7] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3109 of *Lecture Notes in Computer Science (LNCS)*, pages 400–408. Springer, 2004.

[8] R. A. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proceedings of 12th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 13–24, 2005.

[9] J. Barbay. Optimality of randomized algorithms for the intersection problem. In *Proceedings of the Symposium on Stochastic Algorithms, Foundations and Applications (SAGA)*, volume 2827 of *Lecture Notes in Computer Science (LNCS)*, pages 26–38. Springer, 2003.

[10] J. Barbay, A. Golynski, J. I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proceedings of the 17th*

Annual Symposium on Combinatorial Pattern Matching (CPM), volume 4009 of *Lecture Notes in Computer Science (LNCS)*, pages 24–35. Springer-Verlag, 2006.

[11] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–399. Society for Industrial and Applied Mathematics (SIAM), 2002.

[12] J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In *Proceedings of the 14th International Symposium Algorithms and Computation (ISAAC)*, volume 2906 of *Lecture Notes in Computer Science (LNCS)*, pages 575–584. Springer, 2003.

[13] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA)*, volume 4007 of *Lecture Notes in Computer Science (LNCS)*, pages 146–157. Springer Berlin / Heidelberg, 2006.

[14] D. Benoit, E. D. Demaine, J. I. Munro, and V. Raman. Representing trees of higher degree. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures (WADS)*, volume 1663 of *Lecture Notes in Computer Science (LNCS)*, pages 169–180. Springer-Verlag, 1999.

[15] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML Path language (XPath) 2.0. Technical report, W3C Working Draft, November 2003. http://www.w3.org/TR/xpath20/.

[16] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.

[17] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. Technical report, W3C Recommendation, January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[18] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal: Very Large Data Bases*, 8(2):101–119, 1999.

[19] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, 2000.

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup language (XML) 1.1 (Second Edition). Technical report, W3C Recommendation, September 2006.

[21] W. H. Burge. Sorting, trees, and measures of order. *Information and Control*, 1(3):181–197, 1958.

[22] Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *GEOMETRY: Discrete and Computational Geometry*, 16, 1996.

[23] J. Clark and S. DeRose. XML Path language (XPath). Technical report, W3C Recommendation, November 1999. http://www.w3.org/TR/xpath/.

[24] S. Cohen. XSearch: A semantic search engine for XML, 2003.

[25] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[26] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments*, volume 2153 of *Lecture Notes in Computer Science (LNCS)*, pages 5–6, Washington DC, January 2001.

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

[28] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.

[29] Google. www.google.com/history.

[30] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[31] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over XML documents, 2003.

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

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

[34] H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber: A native XML database, 2002.

[35] R. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2:18–21, 1973.

[36] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm. *SIAM J. Comput.*, 15(1):287–299, 1986.

[37] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, 2004.

[38] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Computers*, 34(4):318–325, 1985.

[39] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 14th International Workshop on Algorithms and Data Structures (WADS)*, volume 1671 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2007.

[40] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[41] O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995.

[42] J.-C. Raoult and J. Vuillemin. Optimal unbounded search strategies. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 512–530. Springer-Verlag, 1980.

[43] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm*, pages 1230–1239, 2006.

[44] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. *Lecture Notes in Computer Science*, 1997:137+, 2001.

[45] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying XML documents made easy: Nearest concept queries. In *ICDE*, pages 321–329, 2001.

[46] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2005. ACM Press.

Appendix A

Improvements on the algorithm for finding Smallest Lowest Common Ancestors (SLCAs)

As mentioned in Section 2.3, Xu and Papakonstantinou [46] considered the problem of pure keyword search in XML trees and presented two algorithms: *Indexed Lookup Eager Algorithm* and *Scan Eager Algorithm* for finding Smallest Lowest Common Ancestor (SLCA). Both of these algorithms are based on several properties of the preorder enumeration of nodes and definitions of LCA and SLCA in a labeled tree. They are different only in the way they perform *left match* and *right match* operations.

If we consider all the nodes that have a given label are contained in an ordered set S, then the left (right) match of v in an ordered set of nodes S is the node of S that has the largest (smallest) ID that is smaller (greater) than or equal to ID of v.

We propose two modifications to these two most frequent operations of both algorithms. First, we replace binary search in left match and right match operations by doubling search described in Section 2.2.1. Second, while the initial algorithms look for the elements in each set S_i in increasing order, we can make them not consider already scanned portion of these elements by introduction of the cursor that splits arrays on already scanned and not scanned parts.

Using of doubling search in the implementations of left match and right match operations remove the difference between Indexed Lookup Eager and Scan Eager Algorithms. A doubling search performs not worse and in most cases better than the simple scanning of the ordered set of elements and, in the same time, performs not more than two times worse than a binary search.

Introduction of the cursor between already scanned and not scanned parts of an ordered set of elements allows to optimize searching by removing already considered part from the scope of the future searches. The upper bound on the complexity of such searches was obtained by Raoult and Vuillemin [42] and is formulated by us in this thesis in Lemma 2.4.1.

Xu and Papakonstantinou proved an upper bound $\mathcal{O}(|S_1|kd\log|S|)$ on the complexity of the Indexed Lookup Eager Algorithm, where $|S_1|$ is the minimum and $|S|$ is the maximum size of keyword lists S_1 through S_k. They obtained an upper bound $\mathcal{O}(k|S_1| + d\sum_2^k |S_i|)$ or $\mathcal{O}(kd|S|)$ on the complexity of the Scan Eager Algorithm. Two modification proposed by us and Lemma 2.4.1 allow to obtain a new upper bound

$$\mathcal{O}(|S_1|kd\log\frac{|S|}{|S_1|})$$

on the complexity of the improved algorithm.

Both initial algorithms are non-blocking, i.e. they produce part of the answers quickly so that users do not have to wait long to see the first few answers. Moreover, if a user is interested in any correct answer or is satisfied with any of a few first answers, one can reduce the total execution time dramatically by stopping the algorithm after it produced the first result. Our improvements of these algorithms do not void their non-blocking property.