Weighted Queries on Binary Relations and Multi-Labeled Trees

Jérémy Barbay, Aleh Veraskouski [1]

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

Technical Report CS-2006-23

Abstract. The common way to search large indexed databases is through conjunctive queries on binary relations, such as those associating keywords to web page references. We extend this model by adding positive weights to the terms of the query. In this context we give and analyze two algorithms to answer such queries in a pertinent way. We extend this approach to solve weighted queries on tree-structured objects such as file-systems or XML documents.

Keywords. Adaptive algorithms, intersection, conjunctive queries, labeled tree, unordered subset-path queries, pattern matching queries.

1 Introduction

Consider the task of a search engine answering conjunctive queries: given a set of keywords, it should return a list of references to the objects relevant to all those keywords. These objects can be web pages as in the case of a web search engine like Google, or documents as in a search engine on a file system, or any kind of data searched by keywords. Rather than roam the set of all objects (which is usually very large – think about the set of web pages indexed by Google), a good search engine uses a *precomputed index*, which represents the binary relation between the set of objects $\{1, \ldots, n\} = [n]$ and the set of admissible keywords $\{1, \ldots, \sigma\} = [\sigma]$, so that it can be easily searched.

Each keyword $\alpha \in [\sigma]$ of a query corresponds to a set $S \subseteq [n]$ of objects, and the answer to a conjunctive query is the intersection $S_1 \cap \ldots \cap S_k$ of the sets $S_1, \ldots, S_k$ corresponding to the labels $\alpha_1, \ldots, \alpha_k$ composing the query. The intersection problem has been largely studied when the sets are represented by sorted arrays, in which case variants of binary search or interpolation search allows to computer intersection efficiently, both in theory [2, 3, 6, 14] and in practice [8, 15]. For certain applications it is known that there are alternative representations with better performance, e.g. Barbay *et al.* [2, 3, 6, 14] showed that encoding directly the binary relation improves performance. Most intersection algorithms can be adapted to this setting, and even the search in labeled trees such as file-systems and XML documents benefits from their results.

Conjunctive queries are attractive by their simplicity, but their efficiency is reduced as the amount of data indexed by search engines is rapidly growing: more and more keywords are required to reach an answer of size manageable by the user, with an increasing risk of over-constraining the query, in which case the answer to the query is empty. A solution to this problem, made famous by the Google search engine, is to *rank* the objects (web pages, in this context): users type only a few keywords, the size of the answer is massive, but hopefully the best results are among the first offered to the user. We consider an alternate solution, where the user types as many keywords as desired, some being eventually added automatically from his profile, and where the search engine returns the *most pertinent* answer, which is never empty but whose size can always be made reasonable by adding more keywords.

The concept, based on the threshold set problem, as studied by Barbay and Kenyon [6], is a generalization of the intersection problem, where one requires the set of objects that matches t labels out of the k composing

[1] The Corresponding Author. E-mail: averasko@uwaterloo.ca

the query. For $t = 1$ the answer is the union of the corresponding sets, for $t = k$ their intersection, and for t between 1 and k a relaxation of the intersection that is especially interesting when the query is over-constrained and the intersection is empty. Our results are twofold:

- First, we generalize the adaptive algorithm for the threshold set problem from Barbay and Kenyon [6] to weighted queries on binary relations. We introduce a variant of threshold set queries, the *pertinent set* queries, whose answer corresponds to the smallest non-empty threshold set and we give a partial adaptive analysis of its computational complexity.
- Second, we transpose these two problems, their algorithms and their analysis to the context of the search in labeled trees: we consider unordered subset path weighted queries.

To make our complexity results simpler, we express them in terms of the number of "object searches" – searches for a particular object in an indexed set. Object searches in binary relations can be supported either using the encoding for binary relations defined by Barbay *et al.* [5] or through a binary search, doubling search, interpolation or extrapolation search in a sorted array [8]. Node searches in a labeled tree for nodes matching an ancestor, descendant, or follower relation can be supported either using the encoding for labeled trees from Barbay *et al.* [9], or through doubling searches using a complex index structure [4]. When using indexed encodings [5, 9] in Random Access Memory (RAM) of word size $\theta(\max\{\sigma, n\})$, we deduce the complexity in term of number of accesses by multiplying the number of object searches by their constant cost. In the case of sorted arrays, we deduce the complexity in the comparison model from the expression of the number of object searches using the concavity of the logarithm, as done previously in [3, 6].

We organize our presentation as follows. Section 2 deals with weighted queries on binary relations: Section 2.1 gives some preliminary knowledge and results, Section 2.2 considers threshold set queries and Section 2.3 transposes the results to pertinent set queries. Section 3 deals with weighted queries on multi-labeled trees in the same manner: Section 3.1 considers threshold set of the unordered subset path weighted queries, while Section 3.2 transpose the results to the pertinent set of the unordered subset path weighted queries. Finally, we conclude with a discussion of the impact of our results and some perspectives of future research in Section 4.

2 Weighted Queries on Binary Relations

2.1 Binary Relations

Definition 1. *Consider an ordered set of objects* $\{1, \ldots, n\} = [n]$ *and an ordered set of labels* $\{1, \ldots, \sigma\} = [\sigma]$. *A* binary relation R *is a function* $R : [\sigma] \times [n] \to \{0, 1\}$, *which associates a boolean value* $R(\alpha, x)$ *to each pair formed by a label* $\alpha \in [\sigma]$ *and an object* $x \in [n]$.

Note that a binary relation can be encoded as a list of posting lists (which lists all the objects associated to a label in increasing order).

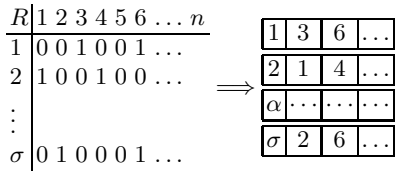

Fig. 1. An example of posting lists representation of a binary relation.

Also note that a list of posting lists is not necessarily the best way to encode a binary relation: Barbay *et al.* [5] suggested another encoding for binary relations, which permits faster searches. For any two objects

$x, y \in [n]$ we say that x *precedes* (*succeeds*) y if $x < y$ ($x > y$) in the order defined on $[n]$. We express our results in term of the number of searches performed on the binary relation. We define these searches as follows:

Definition 2. *Consider a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, a label $\alpha \in [\sigma]$ and an object $x \in [n]$.*

- *The* label α successor of x, *denoted by* $\texttt{label} - \texttt{successor}(\alpha, x)$, *is the first object y succeeding or equal to x, such that $R(\alpha, x) = 1$.*
- *The* label α strict successor of x, *denoted by* $\texttt{label}-\texttt{ssuccessor}(\alpha, x)$, *is the first object y succeeding x, such than $R(\alpha, x) = 1$.*
- *The* label α predecessor of x, *denoted by* $\texttt{label} - \texttt{predecessor}(\alpha, x)$, *is the last object y preceding or equal to x, such that $R(\alpha, x) = 1$.*
- *The* label α strict predecessor of x, *denoted by* $\texttt{label}-\texttt{spredecessor}(\alpha, x)$, *is the last object y preceding x, such that $R(\alpha, x) = 1$.*

Those searches are supported in different times depending on a particular encoding of the binary relation. An encoding based on posting lists supports them in time that depends on the number of objects associated to each label:

Lemma 1. *Consider a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$ implemented as a list of posting lists, a label $\alpha \in [\sigma]$ and an object $x \in [n]$. And let n_α be the number of objects associated to each label α.*

- *The search for the label α successor of x is supported in time $O(\lg n_\alpha)$ in the comparison model.*
- *The search for the label α successors of d objects in increasing order is supported in time $O(d \lg(n_\alpha/d))$ in the comparison model.*

The searches for the label α strict successor, the label α predecessor, and the label α strict predecessor of x are supported in the same time.

Proof. A simple binary search finds the label α successor of x in $O(\lg n_\alpha)$ comparisons. A sequence of doubling searches finds a sequence of label α successors of d consecutive objects in $O(d \lg(n_\alpha/d))$ comparisons, by concavity of the logarithm. The other searches are supported using the same techniques. □

An encoding that takes advantage of the word parallelism of the RAM model supports these operators in time which depends only on the smallest dimension of the binary relation, usually σ:

Lemma 2. *Consider a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, implemented as a multi-labeled string, a label $\alpha \in [\sigma]$ and an object $x \in [n]$. Without loss of generality, suppose $\sigma \leq n$. The search for the label α successor of x is supported in time $O(\lg \lg \sigma)$ in the RAM model, with word size $\Theta(\lg \max\{\sigma, n\})$. The searches for the label α strict successor of x, label α predecessor of x, and label α strict predecessor of x are supported in $O(\lg \lg \sigma)$ as well.*

Proof. The encoding described by Barbay *et al.* [5] supports the *rank* and *select* operators on the positive entries in the rows of the matrix representing binary relation in time $O(\lg \lg \sigma)$. One can do search for label α successor in the same time by using these operators. The other searches are supported using the same techniques. □

2.2 Weighted Queries and Threshold Set

Although in common settings conjunctive queries don't make any difference between the keywords composing the query, its variant that associates a weight to each keyword was suggested by Barbay and Kenyon [7, Perspectives]. We define it here more formally:

Definition 3. *Consider a positive integer μ and an ordered set of labels $\{1, \ldots, \sigma\} = [\sigma]$. A* weighted query *is a function $Q : [\sigma] \to \{0, \ldots, \mu\}$, which associates a non-negative weight $Q(\alpha)$ to each label α.*

These weights allow, for instance, to differentiate between some important keywords typed by the user and some less important ones added automatically by the system to personalize the search. The definition of a conjunctive query has to be extended to take into account the additional information from the weightiness of the query. We propose the first extension that generalize the threshold set to weighted queries.

Definition 4. *Consider a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu\}$, and a non-negative integer t. The* score *of an object x on R and Q is defined as* $\texttt{score}(R, Q, x) = \sum_{\alpha \in [\sigma]} Q(\alpha) R(\alpha, x)$.

Definition 5. *The* threshold set on R and Q with the threshold t *is the set* $\texttt{Thres}(R, Q, t) = \{x \in [n], \texttt{score}(R, Q, x) \geq t\}$ *of objects having score of at least t.*

In the case where μ is equal to 1, the definition matches the one given by Barbay and Kenyon [6] on unweighted queries. Note that in this context, if t is equal to the number of labels of positive weight in Q, the threshold set corresponds to the result of a traditional conjunctive query, the intersection of the sets of objects associated with the labels from the query.

The analysis in the weighted case is similar to the original one in the unweighted case: any algorithm has to *certify* the correctness of its output, hence we measure the complexity of our algorithm as a function of the size of a certificate, which is a measure of the hardness of the instance. A certificate can be defined in several ways, we define it as a partition of the universe, from which the result is easy to check.

Definition 6. *Consider a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu\}$, and a non-negative integer t. A* partition-certificate *of the instance is a partition $(I_i)_{i \in [\delta]}$ of the set $[n]$ of objects such that for any $i \in [\delta]$, either I_i is a singleton $\{x\}$ and $\texttt{score}(R, Q, x) \geq t$, or I_i is an interval and there is a set $S \subset [\sigma]$ of labels such that $\sum_{\alpha \notin S} Q(\alpha) < t$ and $\sum_{x \in I_i} R(x, \alpha) = 0$, $\forall \alpha \in S$.*

There are several ways to define the difficulty of an intersection instance, such as the minimal encoding size of a certificate or the minimal number of comparisons in a certificate [3, 6, 14]. Similarly, there are several ways to define the difficulty of a threshold set instance. We define one derived from the alternation measure proposed by Barbay and Kenyon [6] for the intersection problem.

Definition 7. *Consider a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu\}$, and a non-negative integer t. The* alternation *of the instance is the size δ of the smallest partition-certificate of the instance.*

The alternation corresponds exactly to the non-deterministic complexity, the complexity required from a non-deterministic algorithm to solve the instance, or equivalently to the number of comparisons required to check the correction of the output. We use it as our measure of difficulty for the adaptive analysis of the threshold set computation.

Theorem 1. *Given a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu\}$, and a non-negative integer t, there is an algorithm which performs $O(\delta k)$ searches and heap operations to compute the corresponding threshold set, where δ is the alternation of the instance, and k is the number of labels of positive weights in Q.*

Proof (sketch). We extend the algorithm given by Barbay *et al.* [7] on binary relations and unweighted queries in the next way. The algorithm determines iteratively the smallest possible object which can eventually be in the threshold set, checks it, and proceeds to the next potential object. We decompose the execution of the algorithm into phases, each corresponding to an interval of the partition-certificate in which all objects have been considered.

As the algorithm considers the labels in round-robin order, at most k searches are performed in each phase, after which the algorithm has at least all the objects of the next interval of the partition certificate eliminated. The algorithm features a heap to determine the smallest possible object that can eventually be in the threshold set. This heap is updated at most k times per phase, hence the cost of $O(k)$ heap operations per phase. As there is a partition certificate of length δ, the algorithm terminates after δ phases, thus the total complexity is $O(\delta k)$ searches and heap operations. □

Corollary 1. *Under the same conditions as Theorem 1, there is an algorithm which performs* $O(\delta \sum_{\alpha \in S} \lg(n_\alpha/\delta) + \delta k \lg k)$ *comparisons to compute the threshold set, when* R *is implemented as a list of posting lists.*

Proof. This is a direct combination of the results of Theorem 1 and Lemma 1: at most δ searches are performed in each posting lists, for objects in increasing order, and each heap operations costs at most $O(\lg k)$ comparisons. □

Corollary 2. *Under the same conditions as Theorem 1, there is an algorithm which computes the threshold set in time* $O(\delta k \lg\lg\sigma + \delta k(\lg\lg k)^2)$ *in the RAM model with word size* $\Theta(\lg\max\{\sigma, n\})$*, when the heap is implemented as in [1], and* R *is implemented as described by Barbay* et al. [5].

Proof. This is a direct combination of the results of Theorem 1, Lemma 2, and priority queues implementation and analysis in [1]. □

Similarly, the results of Theorem 1 can be applied to other implementations of binary relations and posting lists, and in other computational models, such as the cache-oblivious model [16]. Several data-structures have been proposed to search efficiently in sorted sets while taking advantage of the cache [10, 11].

2.3 Pertinent Set

The beauty of conjunctive queries comes in part from their simplicity. The threshold set on binary relations and weighted queries is a compromise between the intersection ($t = k$) and the union ($t = 1$) of k sets: it requires the parameter t to specify which compromise is desired. In most applications aimed to the large public what is desired is only a non-empty answer of maximal precision: we formalize this notion as a new type of query, corresponding to the smallest non-empty threshold set.

Definition 8. *Consider a binary relation* $R : [\sigma] \times [n] \to \{0, 1\}$ *and a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu\}$*. The* pertinent score *on* R *and* Q *is the maximal score* $\texttt{pertinent\_score}(R, Q) = \max_{x \in [n]}\{\texttt{score}(R, Q, x)\}$ *on* R *and* Q *over all objects in* $[n]$*. The* pertinent set *on* R *and* Q *is the set* $\texttt{Pert}(R, Q) = \{x \in [\sigma], \texttt{score}(R, Q, x) = \texttt{pertinent\_score}(R, Q)\}$ *of objects that have the maximum score on* R *and* Q*.*

Theorem 2. *Given a binary relation* $R : [\sigma] \times [n] \to \{0, 1\}$ *and a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu\}$*, there is an algorithm that computes the pertinent set on* R *and* Q*. Once it has found the first object of the pertinent set, this algorithm performs* $O(\delta k)$ *additional object searches and heap operations, where* δ *is a alternation of the given instance with* $t = \texttt{pertinent\_score}(R, Q)$*.*

Proof. Consider an instance I of the pertinent set problem defined by R and Q with $\texttt{pertinent\_score}(R, Q) = t^\star$. In the contrast to the algorithm for the threshold set problem, the algorithm for the pertinent set problem is not given a definite threshold value t in the very beginning, thus it initializes it to $t = 1$ and starts scanning objects in increasing order.

After finding an object x with the score of at least t it does not just add it to the output set, but firstly checks if the score of x is greater than t. And if it is, the algorithm stores this value as a new threshold $t = \texttt{score}(x)$ and resets the current output set, because all the objects currently in it have score less than t and thus are not in the pertinent set on R and Q. Finally, it finds the first object of the pertinent set.

After the first object of the pertinent set is found the algorithm works in the same way as the threshold set algorithm for the threshold $t = t^\star$, except that for each positive interval it continues to verify that the score of the object x that goes to the output is not greater than the threshold $t^\star$. This verification consists of searching for x in all the sets in #MAYBE. Each of these searches gives negative answer, and while all of these sets have to be checked by the threshold set algorithm in the worst case as well, this verification does not take the pertinent set algorithm more operations than what it takes to the threshold set algorithm in the worst case. Thus the pertinent set algorithm does not increase the upper bound for the threshold set algorithm, which in couple with the Theorem 1 gives an upper bound of $O(\delta k)$ object searches and heap operations for the pertinent set algorithm after it finds the first object of the pertinent set. □

3 Weighted Queries on Multi-Labeled Trees

We generalize conjunctive queries on labeled trees to the weighted case in the similar way as we did it for search queries on binary relations. One can use them for search in file systems or XML documents by setting large weights to more general labels or by adding labels with low weights from a user-specific profile.

Weighted queries in file systems allow to highlight the most relevant keywords. Weights can be set up by the user manually or by the system automatically. For example, a file system can set weights to adjust relevance of the answer for the given query having internal knowledge about its structure and stored data, or it eventually can put additional keywords with low weights from the user profile to make the search personalized.

We give an algorithm that finds a threshold set for such kind of weighted queries and analyse it in Section 3.1. We extend the idea to the pertinent variant of this problem in a similar way as it was done for the pertinent set in Section 3.2.

3.1 Threshold Set

XML documents as well as structure of file systems can be represented by rooted trees with labels associated to their nodes [5].

Definition 9. *Consider a rooted tree M with a set of nodes $[n]$ enumerated in pre-order and an ordered set of labels $[\sigma]$. A* multi-labeled tree *is a rooted tree M with associated to it binary relation $R : [\sigma] \times [n] \to \{0, 1\}$ between the set of labels $[\sigma]$ and the set of nodes $[n]$.*

We use the notion of *weighted query* fro the previous section. And we denote the set of all the ancestors of a given node x of a multi-labeled tree by $\texttt{ancestors}(x)$.

Definition 10. *Consider a multi-labeled tree M with associated binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu\}$, and a non-negative integer t. An* unordered subset path weighted query *Q is a weighted query in the context of multi-labeled trees, where rooted paths are searched for nodes matching the labels of Q. The* path score *of a node x on M, R, and Q is the sum of the weights of all labels associated with any of the ancestors of the node x or with the node x itself, i.e. $\texttt{path\_score}(M, R, Q, x) = \sum_{\alpha \in [\sigma]} Q(\alpha) \max_{y \in \texttt{ancestors}(x) \cup \{x\}} R(\alpha, y)$.*

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

Definition 11. *Consider a multi-labeled tree M with a set of nodes $[n]$ and an associated binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, an unordered subset path weighted query $Q : [\sigma] \to \{0, \ldots, \mu\}$, and a non-negative integer t. The* threshold set *of an unordered subset path weighted query Q with threshold t is the set of nodes that have path score at least t and do not have any ancestors with this property. More formally, $\texttt{ThresM}(M, R, Q, t) = \{x \in [n] : \texttt{path\_score}(M, R, Q, x) \geq t$ and $\nexists y \in \texttt{ancestors}(x)$, such that $\texttt{path\_score}(M, R, Q, y) \geq t\}$.*

Now we present the algorithm that finds the threshold set of an unordered subset path weighted query Q on M and R with a threshold t. It might use the following operations on the multi-labeled tree M in addition to the standard ones constant-time ones. For any label $\alpha \in [\sigma]$ and any node $x \in [n]$ we use the same notion of the operations *label α successor of x*, *label α strict successor of x*, *label α predecessor of x*, and *label α strict predecessor of x* as it was done for the case of binary relations but with respect to the pre-order defined on the set of nodes $[n]$.

Definition 12. *Consider a multi-labeled tree M with a set of nodes $[n]$ enumerated in pre-order, a set of labels $[\sigma]$, and a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$. For a node $x \in [n]$ and a label $\alpha \in [\sigma]$ define:*

- *the* label α ancestor of x, *denoted by $\texttt{label}-\texttt{ancestor}(\alpha, x)$, is the closest to the root ancestor y of the node x, such that $R(\alpha, y) = 1$.*

– *the* label α descendant of x, *denoted by* $\texttt{label}-\texttt{descendant}(\alpha, x)$, *is the first in pre-order descendant* y *of the node* x, *such that* $R(\alpha, y) = 1$.
– *the* follower *of* x, *denoted by* $\texttt{follower}(x)$, *is the node with the smallest in pre-order number greater than* x *that is not the descendant of* x.
– *the set* $\texttt{labels}(S)$ *is the set of labels associated to at least one node from* S. *Formally,* $\texttt{labels}(S) = \{\alpha \in [\sigma] : \exists x \in S, R(\alpha, x) = 1\}$.

The notion of $\texttt{follower}(x)$ correlates with the notion of $\texttt{following}(x)$ in XPath queries [12, 13]. Basically, $\texttt{follower}(x)$ is the first node in the set $\texttt{following}(x)$ of all nodes greater than node x in document order, excluding any descendants, attribute and namespace nodes.

The partition-certificate we use here is similar to the partition-certificate of queries on binary relations. The difference is that its positive intervals might be non-singletons.

Definition 13. *Consider a multi-labeled tree* M, *binary relation* $R : [\sigma] \times [n] \to \{0, 1\}$, *a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu\}$, *and a non-negative integer* t. *A* partition-certificate *of the instance is a partition* $I = (I_i)_{i \in [\delta]}$ *of size* δ *on the set of nodes* $[n]$, *such that for any* $i \in [\delta]$, *either the node in* I_i *with the smallest number* $x = \min\{\forall x \in I_i\}$ *has path score of at least* t *and the node* $y = \max\{\forall y \in I_i\} + 1$ *is the follower of the node* x, *or there is a set* $S \subseteq [\sigma]$ *of labels such that* $\sum_{\alpha \notin S} Q(\alpha) < t$ *and* $\sum_{x \in I_i} \sum_{y \in \texttt{ancestors}(x)} R(\alpha, y) = 0, \forall \alpha \in S$.

In other words, the partition-certificate consists either of the nodes from the whole sub-tree with the root x from the subset path or of the subset of consecutive in pre-order nodes that have a subset $S \in [\sigma]$ of the labels they all miss of the weight enough to guarantee these nodes not to be in the threshold set.

Each instance potentially admits different partition-certificates. We study the performance of the algorithm in function of the size of the smallest one.

Definition 14. *Consider a multi-labeled tree* M, *a binary relation* $R : [\sigma] \times [n] \to \{0, 1\}$, *a weighted query* $Q : [\sigma] \to \{0, \ldots, \mu\}$, *and a non-negative integer* t. *The* alternation *of the instance is the smallest size* δ *of all possible partition-certificates of the given instance.*

The alternation allows to distinguish between hard and easy instances. We show then that there is an algorithm that performs well on instances with small alternation.

To make the algorithm simpler, we slightly preprocess its input. We add two auxiliary nodes: a node $x = -\infty$ that is the parent of the initial tree root and does not have any associated labels, i.e. $R(\alpha, -\infty) = 0, \forall \alpha \in [\sigma]$, and a node $x = +\infty$ that is a descendant of the first auxiliary node $x = -\infty$ and has all the labels from $[\sigma]$ associated to it, i.e. $R(\alpha, +\infty) = 1, \forall \alpha \in [\sigma]$. The algorithm starts its work from by setting its current node to the new root $x = -\infty$.

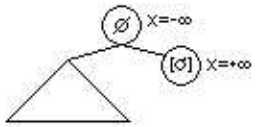

Fig. 2. Modified input

At any time the algorithm has a current node x and two structures: the `rootSet` - a set that contains all the labels associated with the ancestors of the x and the node x itself; and the `heap` - a heap that contains one pair (α', x') for all other labels from $[\sigma]$, where $x' = \texttt{label}-\texttt{ssuccessor}(\alpha', x)$, which is the key in this heap. The `skipList` is just a temporary list of the labels.

We define *path score* of the `rootSet` and `skipList` as the sum of the weights of all the labels in them, i.e. $\texttt{path\_score}(\texttt{rootSet}) = \sum_{\forall\alpha\in\texttt{rootSet}} Q(\alpha)$ and $\texttt{path\_score}(\texttt{skipList}) = \sum_{\forall\alpha\in\texttt{skipList}} Q(\alpha)$.

On each step the algorithm determines if the current node x belongs to the threshold set and advances x to the next node that might be in skipping all the nodes that are guaranteed not to be there (because labels associated with their ancestors do not have enough weight).

---

lgorithm 3.1. Search for **the Threshold Set of the Unordered Subset Path Weighted Query** Q **on** M, R **with threshold** t.

(01) $\texttt{output} \leftarrow \emptyset$
(02) $\texttt{rootSet} \leftarrow \emptyset$
(03) for $\forall\alpha \in [\sigma]$ do $\texttt{heap} \leftarrow (\alpha, \texttt{label}-\texttt{ssuccessor}(\alpha, -\infty))$
(04) $x = -\infty$
(05) while $x \neq +\infty$ do
(06) if $\texttt{path\_score}(\texttt{rootSet}) \geq t$ then
(07) $\texttt{output} \leftarrow x$
(08) $x \leftarrow \texttt{follower}(x)$
(09) $\texttt{updateRootSet}(x)$
(10) $\texttt{updateHeap}(x)$
(11) else
(12) $\texttt{skipList} = \emptyset$
(13) $x_s \leftarrow$ the smallest node x in the `heap`, s.t. $\texttt{path\_score}(\texttt{rootSet}) + \texttt{path\_score}(\texttt{skipList}) \geq t$
(14) do
(15) $(\alpha', x') \leftarrow \texttt{heap}.pop()$
(16) $\texttt{skipList} \leftarrow \alpha'$
(17) until $x' \leq x_s$
(18) $x \leftarrow x_s$
(19) $\texttt{updateRootSet}(x)$
(20) $\texttt{updateSkipList}(x)$
(21) end if
(22) end while

Function $\texttt{updateHeap}(x)$ for each pair (α', x') in the heap updates its value x' to $\texttt{label}-\texttt{ssuccessor}(\alpha, x)$, where x is the current node. Basically, one can do this simply by moving all the pairs from the heap to a temporary list, updating them and placing back to the heap.

Function $\texttt{updateRootSet}(x)$ is more complicated: for each label α' from the `rootSet` it checks if α' is associated with any ancestor of the x or with the current node x itself. If yes, the label α' stays in the `rootSet`, otherwise it is augmented with the value $\texttt{label}-\texttt{ssuccessor}(\alpha', x)$ and is put to the heap.

Function $\texttt{updateSkipList}(x)$ does the same check as $\texttt{updateRootSet}(x)$, but with the labels in the `skipList`. If a label α is on the rooted path it's inserted to the `rootSet`, otherwise – to the `heap` as a part of the newly formed pair $(\alpha, \texttt{label}-\texttt{ssuccessor}(\alpha, x))$.

Lemma 3. *The Algorithm 3.1 is correct, i.e. it produces output equal to the* $\texttt{ThresM}(M, R, Q, t)$. *Moreover, it supports the next invariants between iterations:*

- $\{\forall\alpha \in Q, Q(\alpha) > 0\} = \texttt{rootSet} \cup \texttt{labels}(\texttt{heap})$ *and* $\texttt{rootSet} \cap \texttt{labels}(\texttt{heap}) = \emptyset$.
- `rootSet` *contains all and only the labels with* $Q(\alpha) > 0$ *associated with any of the ancestors of the current node* x *or with the node* x *itself.*

Proof. First, we prove the invariants that are kept by the Algorithm 3.1 in the order we claim them and then proceed to the correctness of the algorithm.

– Initially, all the labels are in the `heap`. Functions `updateRootSet`(x) and `updateHeap`(x) do not remove any labels from the `heap` without placing it to the `rootSet` and vice-versa. If the algorithm on steps (15)-(16) move any labels from the `heap` to the `skipList`, the function `updateSkipList` will move them back to the `heap` or to the `rootSet` by the end of the iteration. The invariant is preserved.
– Initially, the second invariant is true. At each iteration the current node x is changed that might cause the violation of the invariant. But in the end of each iteration either a call to the `updateRootSet`(x) and `updateHeap`(x) or call to the `updateRootSet`(x) and `updateSkipList`(x) is made. The function `updateRootSet`(x) remove all the extra nodes from the `rootSet` and one of the functions `updateHeap`(x) or `updateSkipList`(x) adds missing nodes. The invariant is preserved.

To prove the correctness we need to prove that all the nodes that the algorithm outputs are in the threshold set and that no nodes from the threshold set are missing in the output.

The Algorithm 3.1 can put only current node x to the output set. While the algorithm maintains the second invariant, it can find the real path score for the current node x. And while it does not process any of the nodes y that have an ancestor x with `path_score`$(x) \geq t$, because it skips such nodes while processing their ancestor x, we get that all the nodes that the algorithm puts to the output are nodes from the threshold set.

To prove that no nodes are missing in the output we show that all the skipped nodes are not in the threshold set. The algorithm skips nodes in two cases: if it just put a node to the output or if the current node does not go to the output and the algorithm search for the next node in the heap that might go. In the first case, the algorithm skips only descendants of the node from the threshold set that are not in the threshold set by definition. In the second case, the path score of the nodes skipped while popping pairs from the `heap` up to the s^{th} pair (α_s, x_s) is less than the threshold t and thus they cannot belong to the threshold set as well. □

Theorem 3. *Given a multi-labeled tree M with a set of nodes $[n]$, a binary relation $R : [\sigma] \times [n] \to \{0, 1\}$, a weighted query $Q : [\sigma] \to \{0, \ldots, \mu\}$, and a non-negative integer t. There is an algorithm which performs $O(\delta k)$ searches for label ancestor, searches for label strict successor, and heap operations to compute the corresponding threshold set* `ThresM`(M, R, Q, t) *of the unordered subset path weighted query Q, where δ is the alternation of the instance and k is the number of labels of positive weights in Q.*

Proof. The existence of the algorithm follows from the Lemma 3. Now we proof its upper bound.

Consider an arbitrary instance of the threshold set problem and the corresponding smallest partition-certificate I with δ intervals. At first, we prove that the Algorithm 3.1 spends at most one iteration for each interval and then we count the number of searches for label ancestor, label strict successor, and heap operations performed during each iteration.

Consider an arbitrary interval $I_i \in I$. It is either positive or negative. If it is positive, the algorithm has to process the first node of the interval as the current node. While processing it just outputs the current node $x \in I_i$ and skips all the other nodes from this interval by setting $x \leftarrow$ `follower`(x), i.e. processes the whole interval in one step.

Imaging the algorithm is processing any node $x \in I_i$ as the current node in the negative interval. While by definition of the partition-certificate there is a subset $S \subseteq [\sigma]$ of labels whose nodes are not associated with any of the nodes in the interval I_i, they are all in the `heap` with key nodes having larger numbers than any node in the I_i. But while the sum of weights of all the other labels less than the threshold ($\sum_{\alpha \notin S} Q(\alpha) < t$), the last node (α_s, x_s) popped out from the `heap` will have the number greater than $\max\{\forall x \in I_i\}$.

The number of pairs in the `heap` and labels in `rootSet` and `skipList` is always bounded by k. The function `updateHeap`(x) performs one heap deletion, one heap insertion and one `label−ssuccessor` search for at most any pair in the `heap`. This gives the function `updateHeap`(x) the complexity of $O(k)$ in terms of these operations.

The function `updateRootSet`(x) and the function `updateSkipList`(x) performs one heap insertion, one `label−ancestor` search, and one `label−ssuccessor` search for at most any label in the `rootSet` or in the `skipList` respectively. It's complexity is $O(k)$.

If the interval is positive, the algorithm performs a call to `updateRootSet`(x) and a call to `updateRootSet`(x). If the interval is negative, the algorithm makes one call to `updateRootSet`(x) and one call to `updateSkipList`(x) together with performing up to k heap deletions. This implies the complexity of $O(k)$ for any iteration.

The upper bound of $O(\delta)$ on the number of iterations and the upper bound of $O(k)$ in terms of described operations on the complexity of each iteration imply the upper bound of $O(\delta k)$ for the Algorithm 3.1. □

Lemma 4. *Consider a multi-labeled tree M with a set of nodes $[n]$, a set of labels $[\sigma]$, and a binary relation $R : [\sigma]\times[n] \to \{0,1\}$. Without loss of generality, suppose $\sigma \leq n$. Given a label $\alpha \in [\sigma]$ and a node $x \in [n]$, there is an encoding of this data that supports operations* `label−ancestor`(α, x) *and* `label−ssuccessor`(α, x) *in time $O(\lg\lg\sigma)$ and* `follower`(x) *in time $O(1)$ in the RAM model, with word size of $\Theta(\lg\max\{\sigma, n\})$.*

Proof. The encoding for multi-labeled trees described by Barbay *et al.* [5] supports the `label−ancestor`(α, x) and `label−ssuccessor`(α, x) operators in the time $O(\lg\lg\sigma)$ and allows to implement `follower`(x) in constant time. □

Corollary 3. *Under the same condition as Theorem 3, there is an algorithm which computes the threshold set of the unordered subset path weighted query in time $O(\delta k \lg\lg\sigma + \delta k(\lg\lg k)^2)$ in the RAM model with word size $\Theta(\lg\max\sigma, n)$, when the heap is implemented as in [1], and R is implemented as described by Barbay* et al. [5].

Proof. This is a direct combination of the results of the Theorem 3, Lemma 4, and priority queue implementation and analysis in [1]. □

3.2 Pertinent Set

In the similar way to what we have done for weighted queries on binary relations, we extend threshold set queries to pertinent set queries on multi-labeled trees.

Definition 15. *Consider a multi-labeled tree M, a binary relation $R : [\sigma] \times [n] \to \{0,1\}$, and a weighted query $Q : [\sigma] \to \{0,\ldots,\mu\}$. The* pertinent path score *on M, R, and Q is the maximum path score* `pertinent_path_score`$(M, R, Q) = \max_{x\in[n]}\{$`path_score`$(M, R, Q, x)\}$ *of any node in $[n]$ on M, R, and Q. The* pertinent set *of the unordered subset path weighted query Q on M, R is the set* `PertM`$(M, R, Q) = \{x \in [\sigma],$ `path_score`$(M, R, Q, x) =$ `pertinent_path_score`$(M, R, Q)\}$ *of nodes that have the maximum path score on M, R, and Q.*

Theorem 4. *Given a multi-labeled tree M with a set of nodes $[n]$, a binary relation $R : [\sigma] \times [n] \to \{0,1\}$, and a weighted query $Q : [\sigma] \to \{0,\ldots,\mu\}$. There is an algorithm that computes the pertinent set of the unordered subset path weighted query Q on M, R. Once it has found the first object of the pertinent set, this algorithm performs $O(\delta k)$ additional searches for label ancestor, searches for label strict successor, and heap operations, where δ is a alternation on this instance with $t =$* `pertinent_path_score`(M, R, Q).

Proof. Consider an instance I of the pertinent set problem defined by M, R and Q with `pertinent_path_score`$(M, R, Q) = t^\star$.

In the contrast to the algorithm for the threshold set problem, the algorithm for the pertinent set problem is not given a definite threshold value t in the very beginning, thus it initializes it to $t = 1$ and starts scanning nodes.

After finding the node x with the path score `path_score`(x) greater than the current threshold t the algorithm stores the new value of $t =$ `path_score`(x) and reset current output set as it was done by the algorithm for the pertinent set on binary relation.

Moreover, after finding the node x that goes to the output set the algorithm proceeds not to the follower of the x as it was done by the algorithm for the threshold set, but to the smallest node in the `heap`, because if it resides in the subtree of x it will have greater threshold.

Formally, we make the next changes to the Algorithm 3.1:

```
(00)  t = 1
(06)    if path_score(x) ≥ t then
(07)      if path_score(x) > t then
            t = path_score(x)
            output = ∅
          end if
          output ← x
(08)      (α′, x′) ← heap.pop()
          rootSet ← α′
          x ← x′
```

After the first node of the pertinent set is found, each time the algorithm proceeds to the smallest node in the heap after it puts one more node x to the output set, it get new current node that is at least follower of x in pre-order, because otherwise the new current node would reside in the subtree of x and would have path score greater than $t^\star$, which contradicts the assumption that $t^\star$ is the maximum threshold. Considering this, we do the same analysis as we did in the Theorem 3 that implies the upper bound of the pertinent set algorithm.

□

4 Conclusion

In this paper we generalize threshold set queries on binary relations, the corresponding algorithms and adaptive analysis to weighted queries. We introduce pertinent set queries, whose answers correspond to the smallest non-empty threshold sets and represent the most relevant subsets of the objects to the given query. We transpose threshold and pertinent set queries, their algorithms and their analysis to the context of search in multi-labeled trees.

This work opens many new directions for future research. Here we mention the most prominent ones.

Although we give algorithms to solve pertinent set queries in the contexts of binary relations and multi-labeled trees, we do only its partial adaptive analysis. We cannot do a complete analysis because the current definitions of the partition-certificates for these two problems do not fully represents the algorithms' complexity in the pertinent case: for a specific partition-certificate there might be a lot of different instances of the problem on which the algorithm proposed would have drastically different running time, depending on how fast it reveals the maximum score or path score value for the instance. To solve this problem one may define another partition-certificate that will take into account this issue.

While we extended the threshold set problem by adding weights to the keywords of the query, one more very natural extension is to give weights to the objects of the binary relations, i.e. to introduce weighted binary relations. This is useful for gradation between objects of binary relation and describing of more sophisticated dependences between them, but it is complicate to implement efficiently as well. The structure of the problem instance becomes too unpredictable and our algorithms cannot get benefit in object intervals skipping without checking them inside for possible answers.

While in this paper we study the nodes of a tree closest to the root that have enough labels from the given query on their rooted paths, in some applications one may care about the deepest nodes having enough labels associated with their descendants. Although this problem looks similar to the subset path problem, the algorithm for the latter must be changed greatly to deal with the former.

Bibliography

[1] 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.

[2] 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 *LNCS*, pages 400–408. Springer, 2004.

[3] J. Barbay. Optimality of randomized algorithms for the intersection problem. In A. Albrecht, editor, *Proceedings of the Symposium on Stochastic Algorithms, Foundations and Applications (SAGA)*, volume 2827 / 2003, pages 26–38. Springer-Verlag Heidelberg, 2003.

[4] J. Barbay. Index-trees for descendant tree queries in the comparison model. Technical Report TR-2004-11, University of British Columbia, July 2004.

[5] 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*, pages 24–35. Springer-Verlag LNCS 4009, 2006.

[6] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the thirteenth ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 390–399, 2002.

[7] J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In H. O. Toshihide Ibaraki, Noki Katoh, editor, *Lecture Notes in Computer Science*, pages 575–584. Springer-Verlag, 2003. Proceedings of the 14th Annual International Symposium on Algorithms And Computation (ISAAC).

[8] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In M. S. Carme lvarez, editor, *Experimental Algorithms: 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006. Proceedings*, volume 4007 of *Lecture Notes in Computer Science (LNCS)*, pages 146–157. Springer Berlin / Heidelberg, 2006.

[9] J. Barbay and S. Rao. Succinct encoding for XPath location steps. Technical Report CS-2006-10, University of Waterloo, Ontario, Canada, 2006.

[10] M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 195–207. Springer-Verlag, 2002.

[11] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *IEEE Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[12] 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/.

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

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

[15] 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, Lecture Notes in Computer Science*, pages 5–6, Washington DC, January 2001.

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285. IEEE Computer Society, 1999.