Constraint-based Tree Pattern Mining

Sandra de Amo¹, Nyara A. Silva¹, Ronaldo P. Silva¹, Fabiola S.F. Pereira¹

¹Faculdade de Computação – Universidade Federal de Uberlândia (UFU)
Campus Santa Mônica, Bloco B – Uberlândia – MG – Brazil
deamo@ufu.br, {nyarasilva,ronaldopires}@yahoo.com, fabs177@hotmail.com

Abstract. Most work on pattern mining focus on simple data structures like itemsets or sequences of itemsets. However, a lot of recent applications dealing with complex data like chemical compounds, protein structure, XML and Web Log databases, social network, require much more sophisticated data structures (trees or graphs) for their specification. Here, interesting patterns involve not only frequent object values (labels) appearing in the graphs (or trees) but also frequent specific topologies found in these structures. Recently, several techniques for tree and graph mining have been proposed in the literature. In this paper, we focus on constraint-based tree mining. We propose to use tree automata as a mechanism to specify user constraints over tree patterns. We present the algorithm CobMiner which allows user constraints specified by a tree automata to be incorporated in the mining process. An extensive set of experiments executed over synthetic and real data (XML documents) allow us to conclude that incorporating constraints during the mining process is far better effective than filtering the frequent and interesting patterns after the mining process.

Keywords: Frequent Pattern Discovery, Tree Pattern Mining, Tree Automata, Constraint-based Mining, XML Mining, Web Mining.

1. Introduction and Motivation

Recently there has been an increasing interest in developing techniques for the discovery of tree patterns in a dataset of trees (Zaki 2002; Miyahara et al. 2001). Different kind of real data can be modeled using tree structures, such as XML documents, Web usage logs, RNA structures in Bioinformatics, etc. The growing interest in developing methods for tree pattern mining can be justified by its potential applicability in different contexts, as discussed next:

Web Usage Mining: Web Usage Mining consists in using data mining techniques for discovering patterns representing user behavior when browsing an internet site. Such techniques can be useful to a decision maker in order to better understand user’s needs concerning Web applications and Web services (Cooley 2000). For instance, given a dataset of Web access logs related to an e-commerce site, one could be interested in discovering that users frequently first access the link electronic products, then backtrack and access the link cars and inside the car web page, access the link accessories, find the item cd player and buy it. This navigation pattern can be considered as a warning that the site structure should be redesigned, since users frequently find some difficulty in quickly accessing the Web page they want.

XML Mining: The rapid growth of XML repositories in the Web has stimulated the development of systems capable of efficiently storing and handling XML data. Several
problems of knowledge discovery in huge XML repositories have been investigated such as: (1) Frequent Pattern Discovery (Asai et al. 2002): Given a set of XML documents (which naturally conform to labeled trees or graphs), it would be interesting for some users to discover which sub-documents (subtrees) frequently appear within the given XML documents. (2) Frequent Query Pattern Discovery (Yang et al. 2003): Basic features of languages for querying XML documents like XPath (Clark and Derose 1999) or XQuery (Chamberlin 2003) are regular path expressions and tree patterns with selection predicates on multiple elements that specify the tree-structured relationships. Processing such queries can be computationally expensive since it involves navigation through the tree-like (hierarchical) structure of XML documents, which can be deeply nested. Thus, the discovery and storage in cache of frequently requested queries can improve the performance of XML query processing.

Data Mining in Bioinformatics (Wang and Zhao 2003): Huge amounts of RNA molecule structures (represented as trees) are stored in genome databases. In order to extract information about new RNA structures, researchers compare the new structure with well-known RNA structures, looking for topological patterns shared by the new structure and the known ones. The discovered patterns can provide an important insight concerning the functional aspects of the new RNA structure.

In these three different contexts where tree pattern mining techniques have been developed and employed, the amount of discovered patterns is enormous, due to the highly combinatorial nature of the tree pattern search space. Thus, very often, tree pattern mining is a very expensive process. In recent years, a lot of work has been dedicated to the development of methods allowing users to control the patterns produced by the mining systems. This control can take place in a post-processing phase or during the mining process, by incorporating user-specified constraints. This later approach has been largely exploited in the context of association rule mining (Padmanabhan and Tuzhilin 2000) and sequential pattern mining (Garofalakis et al. 1999; de Amo and Furtado 2005).

The main goal of this paper is the introduction of a method for constraint-based tree pattern mining allowing users to specify the format of the tree patterns they are most interested in. These user-specified constraints are pushed inside the mining process in such a way that only patterns satisfying them are produced. Tedious and expensive post-processing work is largely reduced. Several algorithms for tree pattern mining have been proposed recently, with applications in Web Mining and XML Mining (Zaki 2002; Miyahara et al. 2001). The algorithm TreeMiner introduced in (Zaki 2002) is one of the most efficient. Different from algorithms for tree and graph mining based on the Apriori technique (Miyahara et al. 2001), it uses a depth-first search strategy in the generation phase and an embedding store approach for support counting. According to this approach, at the first two passes over the dataset, essential and compact information for support counting is obtained and stored, preventing further scans of the dataset. In this paper, we present the algorithm CoBMiner which uses TreeMiner as a core mining technique, and incorporates user constraints in the generation phase.

The mechanism we use for specifying user constraints is Tree Automata. A short survey on the use of tree automata in XML research can be found in (Neven 2002). For a more comprehensive survey on tree automata and tree grammars see (Murata et al. 2005). Briefly, a non deterministic tree automata (NTA) is a structure $B = (Q, q_0, L, \delta)$, where
\( Q \) is a finite set of states, \( q_0 \) is the initial state, \( L \) is an alphabet of labels and \( \delta \) is a mapping \( \delta : Q \times L \rightarrow 2^Q \) associating to each pair \((q, a) \in Q \times L\) a regular language over the set \( Q \) (that is, a set of strings of states satisfying a regular expression). While (simple) finite automata are designed to accept words (sequences), tree automata are designed to accept trees labeled over the alphabet \( L \). Then, tree automata can naturally be thought as a mechanism to characterize a specific set of trees, that is, to specify a constraint over trees. The following example illustrates our approach:

![Diagram](image)

**Figure 1.** (a) A site tree structure, (b) A navigation pattern (c) A run of the tree automaton.

**Example 1** Let us suppose an e-commerce site where users navigate through items and eventually make a purchase. Let us suppose that items are classified into two categories: Computers (C) and Printers (Pr). Computers are classified into Notebooks (Nb) and Desktops (Dt). Let us suppose we are interested in discovering specific navigation patterns conforming the following format: the client accesses the site, browses different notebooks brands, buys at least one notebook, besides browsing the printer page, without necessarily making a purchase. The tree patterns fitting to this format are exactly the ones accepted by following tree automaton:

\[
\begin{align*}
\delta(q_0, \text{home}) &= q_1q_2 \\
\delta(q_3, N_i) &= \epsilon \quad \text{for } i \in \{1, 2, 3, 4\} \\
\delta(q_6, B) &= \epsilon \\
\delta(q_5, P_i) &= \epsilon \quad \text{for } i \in \{1, 2\} \\
\delta(q_2, Pr) &= q_5^+ \\
\delta(q_4, N_i) &= q_6 \quad \text{for } i \in \{1, 2, 3, 4\} \\
\delta(q_3, Nb) &= q_3q_4 q_3^+ q_3^+ \\
\delta(q_6, B) &= \epsilon \\
\delta(q_5, P_i) &= \epsilon \quad \text{for } i \in \{1, 2\} \\
\end{align*}
\]

Figure 1(a) illustrates the (very simplified) tree structure of the site. In Figure 1(b), we present a pattern verifying the given constraint and in 1(c) how the tree automaton navigates through it, starting at its root and reaching its leaves. The navigation proceeds as follows: \((q_0, \text{root}) \rightarrow q_1q_2; (q_1, Nb) \rightarrow q_3q_4; (q_3, N_1) \rightarrow \epsilon; (q_4, N_2) \rightarrow q_6; (q_6, B) \rightarrow \epsilon; (q_2, Pr) \rightarrow q_5q_5; (q_5, P_1) \rightarrow \epsilon; (q_5, P_2) \rightarrow \epsilon. \) Notice that for a pattern to be accepted, the automaton rules must associate the regular expression \( \epsilon \) to each reached leaf, meaning that this node has no children.

Finite automata (for string processing) have been used as a mechanism to specify constraints in sequential pattern mining (Garofalakis et al. 1999; de Amo and Furtado 2005). Tree automata are usually used in a different context, as a mechanism for specifying XML schemas or DTD (Papakonstantinou and Vianu 2000; Murata et al. 2005). A classical problem in this context is, given a set of XML documents, find a common schema (DTD) to which the XML documents conform (Wang and Liu 1997). This problem reduces to discover a tree automaton which accepts the majority of the XML documents in the given set. Differently from this setting, our approach consists in using tree automata as a mechanism for controlling the generated
patterns in the mining process. The user’s constraints are translated into a tree automaton. Thus, in our approach, a tree automaton is given and our task consists in discovering tree patterns conforming to this automaton.

**Our Contribution.** Our main contributions can be summarized as follows: (1) the introduction of a mechanism for specifying constraints over patterns in a tree pattern mining context. (2) the design and implementation of an algorithm (CoBMiner) for tree pattern mining with constraints (3) an extensive set of experiments over synthetic data (4) a preliminary study of using CoBMiner in XML Mining.

**Paper Organization.** This paper is organized as follows. In Section 2, we formalize the main concepts related to the problem of mining tree patterns with tree automata constraints. In Section 3, we present our algorithm CoBMiner (Constraint-Based Miner) In Section 4 we present and analyse some experimental results obtained by executing CoBMiner over synthetic and real data. Finally in Section 5, we present our concluding remarks and discuss our ongoing and further research.

2. Problem Formalization

In this section, we present the theoretical concepts underlying the description of our mining method and we conclude the section by describing the main task the algorithm CoBMiner is designed to accomplish.

2.1. Tree Pattern Mining

In order to present the tree pattern mining problem, we adopt the notation introduced in (Zaki 2002).

**Trees and String Encodings.** Let $L = \{l_1, ..., l_m\}$ be a set of labels. We can assume that labels are mapped into natural numbers, so $L \subseteq \mathbb{N}$. A labeled tree over $L$ is a directed, acyclic and connected graph $T = (N, E)$, where $N = \{n_0, n_1, ..., n_p\}$ and $E$ denote the set of nodes and edges respectively and where the following conditions are verified: (1) node $n_0$ is a special and unique node in $N$ such that there is no $u \in N$ with $(u, n_0) \in E$. This node is called the root of $T$. (2) there is a mapping $l : N \rightarrow L$ which associates labels to each node of $T$. A tree is said to be ordered if for each node $n \in N$, the set $\text{Child}(n) = \{u \in V \mid (n, u) \in E\}$, whose elements are called children of $n$, is total ordered. From now on, we refer to ordered labeled trees simply as trees. Let $x$ be a node of a tree $T$. A node $y$ on the unique path linking $x$ to the root is called an ancestor of $x$. $x$ is called a descendant of $y$. A direct ancestor is called a parent and obviously, a direct descendent is called a child. Nodes $y$ and $z$ which have the same parent are called siblings. For each node $x$ is a tree $T$ we associate a pair of numbers $[i, k]$, called the scope of $x$, where $i$ corresponds to the position of $x$ in the tree $T$ and $k$ corresponds to the position of the rightmost descendent of $x$ in the tree $T$. In Figure 2, we illustrate the scopes of each node of tree $T_2$.

There are several ways of representing ordered trees. As in (Zaki 2002), we have chosen the string encoding. This encoding is built as follows. We assume that the nodes of $T$ are numbered according to their positions in the depth-first traversal of the tree $T$. So, the root is the node $n_0$, its first child is node $n_1$, the first child of $n_1$ is node $n_2$ and so on. The string encoding of $T$, denoted by $s_T$ is obtaining by performing a depth-first traversal of $T$, starting at the root and adding the labels of the traversed nodes into $s_T$. Whenever we backtrack from a child to its parent we add a unique symbol -1 to the string encoding $s_T$. This format for tree representation is one of many existing formats. We
have chosen it since it allows more efficient subtree counting and manipulation, besides being very performant in terms of storage space usage (see [Zaki 2002] for details). In Figure 2, the string encodings for trees $T_1$ and $S$ are illustrated.

**Data trees and tree patterns.** Let $T = (N_t, E_t)$ and $S = (N_s, E_s)$ be trees. We say that $S$ is an embedded subtree of $T$ (denoted as $S \preceq T$) if: (1) $N_s \subseteq N_t$ and (2) $(x, y) \in E_s$ if and only if $x$ is an ancestor of $y$ in $T$. Thus, an edge $(x, y)$ of $S$ corresponds to a path in $T$ lying in a path going from the root to a leaf. If $S$ is an embedded subtree of $T$, we say that $T$ contains $S$, or $S$ is contained in $T$. Conceptually, a tree pattern is simply a tree. We use the name tree when referring to a data tree (for instance, a tree representing a XML document), and tree pattern when referring to patterns “hidden” (i.e., contained) in a data tree. The size of a tree (or a tree pattern) $T = (N_t, E_t)$ is the number of nodes in $N_t$. It is denoted by $|T|$. For instance, tree $T_1$ depicted in Figure 2 has size 7. The tree pattern $S$ is contained in $T_1$ and $T_3$ but not in $T_2$.

Let $S \preceq T$. Each occurrence of $S$ in $T$ can be identified by its match label, which is defined as the sequence of positions in $T$ where $S$ occurs. More precisely:

**Definition 1 (Match label - Support)** Let $S = (N_s, E_s)$ and $T = (N_t, E_t)$, where $S \preceq T$. Let $N_s = \{s_1, \ldots, s_k\}$ and $N_t = \{t_1, \ldots, t_m\}$. Since $S \preceq T$ then there exists a set of nodes $\{t_{i_1}, \ldots, t_{i_k}\}$ in $N_t$ such that: (1) $l(s_j) = l(t_{i_j})$ for all $j = 1, \ldots, k$, (2) $(s_p, s_q) \in E_s$ if and only if $t_{i_p}$ is an ancestor of $t_{i_q}$ in $T$. The sequence of indices (positions) $(t_{i_1}, \ldots, t_{i_k})$ is called a match label for $S$ in $T$. Clearly, there may be several match labels for a same tree pattern $S$ in a data tree $T$, each one corresponding to an occurrence of $S$ in $T$. For instance, let us consider the data trees $T_1, T_2, T_3$ and tree pattern $S$ depicted in Figure 2. The match labels for $S$ in $T_1$ are $\{(025), (015)\}$. There is only one match label for $S$ in $T_3$, $(134)$.

![Figure 2. A tree database and tree pattern $S$](image)

Let $T$ be a set of (data) trees with $N$ trees (i.e. a tree database) and $S$ a tree pattern. For each $T \in T$, let $d_S(T) = 1$ if $S$ has an occurrence in $T$ and $d_S(T) = 0$ otherwise. The support of $S$ with respect to $T$ (denoted by $sup(S, T)$ or simply $sup(S)$ for the sake of simplifying the notation) is defined as $sup(S) = \frac{\sum_{T \in T} d_S(T)}{N}$. We say that a tree pattern $S$ is frequent with respect to the tree database $T$ and a minimum support threshold $\alpha$ ($0 \leq \alpha \leq 1$) if $sup(S) \geq \alpha$.

For instance, considering the trees depicted in Figure 2: $S$ has two occurrences in $T_1$, no occurrence in $T_2$ and one occurrence in $T_3$. So the support of $S$ with respect to the tree database $T = \{T_1, T_2, T_3\}$ is $\frac{2}{3}$.

**2.2. Tree Pattern Mining with Constraints**

In this section, we introduce the theoretical concepts and definitions closely related to the main contribution of this paper. First of all, we introduce a special class of tree automata
The following tree automaton

**Definition 2 (Tree Automata and Local Tree Automata)** Let \( L \) be a finite alphabet. A tree automaton over \( L \) is a structure \( A = (Q, q_0, L, \delta) \), where \( Q \) is a finite set of states, \( q_0 \) is the initial state and \( \delta \) is a mapping \( \delta : Q \times L \rightarrow 2^Q \) associating to each pair \((q, a)\) \( \in Q \times L \) a regular expression over \( Q \), i.e., a regular language over the set \( Q \). We say that two states \( q_1 \) and \( q_2 \in Q \) are competitors if there exists a symbol \( a \in L \) and different rules \( \delta(q_1, a) = e_1 \) and \( \delta(q_2, a) = e_2 \); that is, label \( a \) can be processed by both states \( q_1 \) and \( q_2 \). A local tree automaton is a tree automaton without competing states.

Local tree automata have been introduced in (Takahashi 1975) and roughly corresponds to DTDs. For a comprehensive survey on different kinds of tree automata (including local ones), see (Murata et al. 2005). Local Automata are used in this paper, since they can specify a large class of constraints over trees and more important, it reduces the complexity of the mining algorithm. For instance, the automaton \( A \) described in Example 1 is not local, since the states \( q_3 \) and \( q_4 \) are competitors. However, the following tree automaton \( A' \) is a equivalent to \( A \) and is local.

**Example 2 (A local tree automaton)** The following tree automaton \( A' \) is local.

\[
\begin{align*}
\delta(q_0, \text{home}) &= q_1, q_2 \\
\delta(q_2, \text{Pr}) &= q_5^* \\
\delta(q_5, B) &= \epsilon \\
\delta(q_5, P_i) &= \epsilon \quad \text{for } i \in \{1, 2, 3\}
\end{align*}
\]

**Definition 3 (Valid and Prefix-Valid Tree Patterns)** Let \( A = (Q, q_0, L, \delta) \) be a tree automaton and \( T = (N_T, E_T) \) be a tree. A run of \( A \) on \( T \) is a mapping \( \lambda : N_T \rightarrow Q \) such that for every node \( v \in N_T \) with \( k \) children \( v_1, v_2, ..., v_k \), the word \( \lambda(v_1) \ldots \lambda(v_k) \) verifies the regular expression \( \delta(\lambda(v), l(v)) \), where \( l(v) \in L \) is the label of \( v \) in \( T \). We say that \( A \) accepts \( T \) (or that \( T \) is valid with respect to \( A \)) if there exists a run \( \lambda \) such that for each leaf \( v \) of \( T \) we have \( \delta(\lambda(v), l(v)) = \epsilon \). We say that \( T \) is prefix-valid (or p-valid for short) w.r.t. \( A \) if for each node \( v \) with children \( v_1, v_2, ..., v_k \), the word \( \lambda(v_1) \ldots \lambda(v_k) \) is a prefix of a word verifying the regular expression \( \delta(\lambda(v), l(v)) \).

For instance, the tree pattern depicted in Figure 1(b) is valid with respect to the tree automaton \( A \) of example 1. The accepting run is depicted in Figure 1(c). The pattern with string encoding (home)(Nb)(N1) - 1 - 1 is p-valid with respect to this automaton and w.r.t. the local automaton \( A' \) of Example 2. The mapping \( \lambda \) is given by \( \lambda(n_0) = q_0, \lambda(n_1) = q_1 \) and \( \lambda(n_2) = q_3 \). Notice that \( \lambda(n_1) = q_1 \) is a prefix of \( \delta(q_0, \text{home}) = q_1q_2 \), and \( \lambda(n_2) = q_3 \) satisfies \( q_5^* = \delta(q_1, Nb) \). Notice that, given a tree pattern \( T \) and a local tree automaton, if there is a run of \( A \) on \( T \) then it is unique.

Our algorithm CoBMiner generates new candidate patterns by expanding frequent and p-valid patterns level by level. During this expansion process, it stores information for each node of the expanded pattern (called the node trace) and uses it to continue the expansion at higher levels.

**Definition 4 (Trace)** Let \( A = (Q, q_0, L, \delta) \) be a local tree automaton and \( T = (N_T, E_T) \) be a p-valid tree pattern with respect to \( A \). For each node \( n \in N_T \) we associate a list \( tr(n) \) (called the trace of \( n \)) built as follows: (1) \( tr(n_0) = [] \); (2) for each node \( n_i \neq n_0 \), let \( v \) be the parent of \( n_i \) and \( n_1n_2...n_{i-1}n_k \) be the children of \( v \) (including \( n_i \)). Let \( \lambda \) be the run of \( A \) through \( T \) justifying the \( p \)-validity of \( T \) w.r.t. \( A \). So \( \lambda(n_1) \ldots \lambda(n_i) \ldots \lambda(n_k) \) is a prefix of a word satisfying the regular expression \( e = \delta(\lambda(v), l(v)) \). We define \( tr(n_i) = [e, \lambda(n_1) \ldots \lambda(n_i)] \).
For instance, let us take the tree pattern (home)(Nb)(N1) - 1 - 1 considered above, which is p-valid w.r.t. to the automaton A of Example 2. Then, its trace is: $tr(n_0) = [1]$; $tr(n_1) = [e_1, q_1]$; $tr(n_2) = [e_2, q_3]$, where $e_1 = q_1q_2$ and $e_2 = q_3^*$.

3. The Algorithm CoBMiner

At a high level, CoBMiner’s framework is similar in structure to the Tree Miner algorithm for mining tree patterns without constraints, introduced in (Zaki 2002). Concerning the handling of constraints in the mining process, it follows the ideas introduced in (Garofalakis et al. 1999) in the context of constraint-based sequential pattern mining. However, adapting these ideas in the tree pattern mining context is not a minor task, as we will see in this section. The algorithm Tree Miner for tree pattern mining uses a mining strategy based on two main principles:

**Depth-First Search**: like the general Apriori strategy, it works in passes, and at the end of each pass, it produces longer patterns than the ones already produced in previous passes.

But, differently from Apriori-like algorithms, which use a breath-first search strategy and produce, at each pass $k$, all frequent patterns of size $k$, TreeMiner uses a depth-first search strategy. At each step $k$ ($k \geq 2$), the set of frequent patterns of size $k$ is divided into a certain number $n_k$ of equivalence classes $C_1^k, ..., C_n^k$. In the generation phase of step $k$, pairs of patterns in the first equivalence class $C_1^k$ are combined to produce a set $C_{k+1}^1$ of candidates of size $k+1$. This candidates are tested for support counting and the remaining frequent patterns are also divided into equivalence classes $C_{1,1}^{k+1}, ..., C_{1,m_k}^{k+1}$, and so on. When no more frequent patterns are produced at some level $m$, the algorithm backtracks to the previous level $m - 1$ and repeats the whole process for the next equivalence class not yet considered at this previous pass.

**Embedded Storage for Support Counting**: The database is scanned for support counting only twice, namely, when producing the sets $F_1$ and $F_2$ of frequent tree patterns of size 1 and 2 respectively. When scanning the database for the second time for producing the set $F_2$, some essential information is collected for each pattern $P$ of size $k$ being tested. This information, the **scope-list** of the pattern $P$ (denoted by $\mathcal{L}(P)$), consists of a list whose elements are of the form $[t, m, s]$, where $t$ is the identifier of a tree where $P$ occurs, $m$ is a match label of an occurrence of $P$ in $t$ (in fact, the match label of the $(k - 1)$-prefix of $P$) and $s$ is the scope of the rightmost node of $P$. For instance, the scope-list of pattern $S$ with respect to the database $\{T_1, T_2, T_3\}$ depicted in Figure 2 is $\mathcal{L}(S) = [(1, (01), 5), (1, (02), 5), (3, (12), 4)]$. After step 2, no more database scans are performed. When generating a pattern $Q$ of size $k + 1$ by combining two patterns $P_1$ and $P_2$ of size $k$ in the same equivalence class, the scope-lists $\mathcal{L}(P_1)$ and $\mathcal{L}(P_2)$ of $P_1$ and $P_2$ respectively are joined producing the scope-list $\mathcal{L}(Q)$ of the new generated pattern $Q$ (details on this joining operation between scope-lists can be found in (Zaki 2002)). The support of $Q$ can be easily calculated using its scope-list $\mathcal{L}(Q)$.

By adding a post-processing phase to TreeMiner, we can obtain a first method to mine tree patterns with constraints specified by a tree automaton $A$: (1) use TreeMiner to produce the set $F$ of all frequent tree patterns and (2) Execute the given tree automaton $A$ over the set $F$ of tree patterns in order to filter those which is accepted by the automaton. There are two important differences between the algorithm CobMiner and the TreeMiner+Post-Processing (TreeMinerPP for short) method:

1. CoBMiner uses the constraint inside the mining process, that is, both in the candidate
generation and pruning phases. TreeMinerPP uses the constraint after the mining process, only to filter the frequent patterns which satisfies it.

2- CoBMiner uses a relaxation of the automaton $A$, that is, a weaker (less restrictive) constraint. Instead of producing the valid patterns, it produces $p$-valid patterns. The main reason for which such a relaxation is considered in our approach is related to the way the constraints are pushed into the candidate generation phase: it is natural to expand a $p$-valid pattern $P$ of size $k$ in order to obtain another $p$-valid pattern $P'$ of size $k + 1$, following the rules of the automaton, instead to expand a valid pattern in order to obtain another valid one.

In order to describe the algorithm CoBMiner, we need to get some insight on the candidate generation phase of the algorithm TreeMiner, since our method is based on the same depth-first strategy of TreeMiner for traversing the search space of patterns.

**How the search space is divided into equivalence classes at each pass.** Let $P$ and $Q$ be two tree patterns of size $k$. We say that $P$ and $Q$ are equivalent if their respective prefixes of size $k - 1$ (considering their string encoding) are identical.

**Example 3** Let us consider the tree patterns $P_1 = (123-1-1)$ and $P_2 (12-14-1)$ of size 3 in Figure 3(a). They have the same prefix (12) of size 2. So, they are equivalent.

![Figure 3. (a) Equivalent patterns (b) Tree pattern extension](image)

The set of tree patterns of size $k$ is naturally divided into equivalence classes, according to the equivalence relation defined above. If $P$ is a prefix of size $k - 1$, we denote by $[P]_{k-1}$ the equivalence class containing all patterns of size $k$ sharing the same prefix $P$.

**Lemma 1 (Prefix Expansion)** Let $P$ be a tree pattern of size $k - 1$. Then the elements of the equivalence class $[P]_{k-1}$ are obtained by adding a new child (with label $x$) to a node $n_i$ of $P$ going from the root to the rightmost node (the rightmost path of $P$).

Figure 3(b) illustrates this construction. A pattern obtained in this way is denoted by $(x, i)$. So, $[P]_{k-1}$ is the set $\{(x, i) \mid x \in L \text{ and } n_i \text{ is a node lying on the rightmost path of the prefix } P\}$. The patterns $(x_1, i_1), \ldots, (x_n, i_n)$ in $[P]_{k-1}$ can be naturally ordered by considering the lexicographic order for pairs $(i, n)$. For instance, the patterns $P_1$ and $P_2$ illustrated in Figure 3 are denoted by $(3, 1)$ and $(4, 0)$ respectively. According to the ordering defined above we have $(3, 1) < (4, 0)$. In the description of the algorithm CoBMiner below, we will denote by $[P]_k^f$ (resp. by $[P]_k^{p}$) the set of frequent (resp. frequent and $p$-valid) patterns of size $k + 1$ in $[P]_k$. 


The operation \textit{Join}: How patterns of size \( k \) are combined to generate patterns of size \( k + 1 \). We only describe this construction for \( k \geq 2 \). From level 2 on, the search space is traversed recursively. Equivalent patterns of size \( k - 1 \) are combined in order to generate patterns of size \( k \). The joining operation between two patterns respects an order: each pattern \((x, i)\) of an equivalent class \( C \) is combined with all patterns \((y, j) \in C\), such that \( j \leq i \). In order to achieve optimality in pattern generation (that is, not obtain the same pattern in different ways), we do not join \((x, i)\) with patterns \((y, j)\) for \( j > i \). The rules for combining the patterns \((x, i)\) and \((y, j)\) of size \( k - 1 \) in order to obtain patterns of size \( k \) are the following:

1. If \( i = j \): in this case, two different patterns can be generated. \( \text{Join}((x, i), (y, j)) = \{(y, i), (y, l)\} \) where \((y, i)\) and \((y, l)\) have the same prefix \((x, i)\) of size \( k \). The pattern \((y, i)\) is obtained by adding a new child node labeled \( y \) to the node \( n_i \), parent of the node labeled \( x \) in \((x, i)\). And \((y, l)\) is the pattern obtained by adding a new child node labeled \( y \) to the node \( n_l \) of \((x, i)\) corresponding to the label \( x \).

2. If \( i > j \): in this case, a unique pattern is generated: \( \text{Join}((x, i), (y, j)) = \{(y, j)\} \), where \((y, j)\) is obtained by adding a new child node labeled \( y \) to the node \( n_j \) of \((x, i)\).

Figure 4 illustrates the \textit{Join} operator for \( k = 4 \).

![Figure 4. Generating patterns of size 4 by joining patterns of size 3.](image)

Now, we are ready to present the general structure of the algorithm CoBMiner:

**Procedure CoBMiner** \((T, \alpha, A)\)

- \( T \) = tree dataset over a set of labels \( L \), \( \alpha \) = minimum support; \( A \) = tree automaton over \( L \).

\[
\begin{align*}
F_1 &= \text{set of all frequent patterns of size 1} \\
V_1 &= \text{set of all frequent and p-valid patterns of size 1} \\
F_2 &= \text{set of all frequent patterns of size 2} \\
V_2 &= \text{set of all frequent and p-valid patterns of size 2 with their traces} \\
\text{For each } P \in V_1 & \text{ do} \\
& \mathcal{L}(P) = \text{CreateScopeList}(\{P\}) \\
\text{For each } P \in V_1 & \text{ do} \\
& \text{Enumerate-Frequent-PValid-Patterns}([P], [P], \mathcal{L}(P), A) \\
\text{Enumerate-Frequent-PValid-Patterns}([P], [P], \mathcal{L}(P), A)
\end{align*}
\]
For each \((x, i) \in [P]^o\) do
\[
[P_x]^v = \emptyset; \\
[P_x]^f = \emptyset; \\
NewScopeList = \emptyset;
\]
For each \((y, j) \in [P]^f\) do
\[
R^f = Join((x, i), (y, j)) \\
L(R^f) = JoinScopeList(L(x, i), L(y, j)) \\
For each \(Q \in R^f\) do
\]
\[n_j := \text{node of } (x, i) \text{ where } y \text{ has been added as a child};
\]
If \(F_{\text{Freq}}(Q, L(Q); \alpha) = \text{true}\) then
\[
[P_x]^f = [P_x]^f \cup \{Q\} \\
R^v = \text{Expand}((x, i), Q, n_j, A) \\
[P_x]^v = [P_x]^v \cup R^v \\
NewScopeList := \text{NewScopeList} \cup L(R^v)
\]
Enumerate-Frequent-PValid-Patterns([P_x]^v, [P_x]^f, NewScopeList, A)

**Discussion.** CoBMiner assumes that the trees in the dataset are represented by their string encodings. In order to compute the set \(F_1\) of frequent patterns of size 1, we consider a 1D array of size \(|L|\). This array stores counters for each label \(i \in L\). For each \(s \in T\), we scan \(s\) and increment the \(i\)-count of each item in \(i \in s, i \neq -1\). Using the automaton \(A\), we also compute the set \(V_1\) of frequent and p-valid patterns in \(F_1\), corresponding to labels \(i\) where \(\delta(q_0, i)\) is defined in \(A\) \((q_0 = \text{initial state of } A)\). For computing \(F_2\), we compute the supports of each candidate \((ij - 1)\) by using a 2D array of size \(|V_1 \times F_1|\) where \(\text{cut}[i][j]\) gives the count of candidate pattern \((ij - 1)\). Using the automaton \(A\), we also compute the set \(V_2\) of frequent and p-valid patterns in \(F_2\). From level 3 on, the algorithm works recursively on the equivalence classes. Let \([P]\) be one equivalence class of patterns of size \(k - 1\), at which one arrived at level \(k - 1\) of the recursive procedure. At this point, the algorithm has identified two sets in \([P]: \[P]^f\) and \([P]^v\), standing for the set of frequent patterns in \([P]\) and the set of frequent and p-valid patterns in \([P]\). At the next level \(k\), each \(p\text{-valid and frequent}\) pattern \((x, i) \in [P]^o\) will be combined with \(frequent\) patterns \((y, j) \in [P]^f\) producing patterns \(Q\) which are p-valid and potentially frequent. When joining the patterns, their scope-lists are also joined using the routine \(\text{JoinScopeList}\). The details of this routine can be found in (Zaki 2002). We notice that the validation phases (support counting) in CoBMiner and in TreeMiner are identical, and follow the embedded storage strategy we have already mention in the beginning of this section. The routine \(F_{\text{Freq}}(Q, L(Q), \alpha)\) tests the frequency of a pattern \(Q\) with respect to the minimum support threshold \(\alpha\), using its scope-list \(L(Q)\).

**How the automaton is taken into account in the generation phase of CoBMiner: the routine Expand.** Expand verifies if the expansion \(Q\) of the p-valid pattern \((x, i)\) follows the rules of the automaton \(A\). This routine takes five parameters: a p-valid pattern \(P\) of size \(k\) (with its trace \(tr\)), a frequent pattern \(Q\) of size \(k + 1\) having the same prefix of size \(k - 1\) as pattern \(P\), a node \(n_i\) in \(P\), a label \(y\) and the tree automaton \(A\). First, it verifies if the node \(n_i\) has children in \(P\). If it is the case, let \(n_i\) be its last child and \(tr(n_i) = (e, q_1...q_g)\) be the trace of the node \(n_i\). Expand verifies if there is a state \(q\) of the tree automaton \(A\) such that \(\delta(q, y)\) is defined. If there is no such state, Expand returns the empty set. If there exists such state \(q\), it verifies that the string \(q_1...q_gq\) satisfies the regular expression \(e\). In case it does, Expand returns the set \(\{(Q, tr(Q))\}\), where \(tr(Q)\) is identical to \(tr(P)\) for all nodes except for its rightmost node \(n_{k+1}\), where \(tr(Q)(n_{k+1}) = [e, q_1...q_g]\). If it is the case that the string \(q_1...q_gq\) does not satisfy \(e\), then it returns the empty set. If it is
not the case that the node \( n_t \) has children in \( P \): let \( a = l(n_t) \) be the label associated to
node \( n_t \) in \( P \). \( \operatorname{Expand} \) verifies if there are states \( q'_1 \) and \( q'_2 \) of the tree automata \( A \) such
that \( \delta(q'_1, a) \) and \( \delta(q'_2, y) \) are defined. If one of these conditions is not verified, \( \operatorname{Expand} \)
returns the empty set. If both conditions are verified, let \( \delta(q'_1, a) = e' \). \( \operatorname{Expand} \) verifies if there is some string in
\( e' \) starting with \( q'_2 \). If it is this the case, \( \operatorname{Expand} \) returns the set \( \{(Q, tr(Q))\} \), where \( tr(Q) \) is identical to \( tr(P) \) for all nodes except for its rightmost node
\( n_{k+1} \), where \( tr(Q)(n_{k+1}) = [e', q'_2] \). If it is not the case, \( \operatorname{Expand} \) returns the empty set.

\textbf{Example 4} Let us consider the pattern \( P_1 \) and \( P_2 \) illustrated in Figure 5. Notice that \( P_1 \)
and \( P_2 \) are equivalent, since they share the prefix (12). So, \( P_1 \) and \( P_2 \) can be denoted
as (3, 1) and (4, 0) respectively. The pattern \( Q \) illustrated in this example is the result of
joining (3, 1) and (4, 0), or equivalently, the result of expanding (3, 1) by adding a new
child labeled 4 at node \( n_0 \) in \( P_1 = (3, 1) \). \( \operatorname{Expand} \) is executed over \( (P_1, Q, n_0, 4, A) \),
where \( A \) is the local automaton:

\[
\begin{align*}
\delta(q_0, 1) &= e_1 = q_1q_2 & \delta(q_1, 2) &= e_2 = q_3^4 & \delta(q_3, 3) &= e_4 = q_4q_5^3 \\
\delta(q_2, 4) &= e & \delta(q_1, 1) &= e & \delta(q_4, 2) &= e
\end{align*}
\]

The traces of nodes \( n_1 \) and \( n_2 \) in \( P_1 \) are \((e_1, q_1)\) and \((e_2, q_3)\) respectively. In order
to expand node \( n_0 \) with label \( y = 4 \), \( \operatorname{Expand} \) verifies if there is some state in the local
automaton, associated to 4. Such state is \( q_2 \). Then, \( \operatorname{Expand} \) verifies if \( n_0 \) has children
and which is its last child. Such child is node \( n_1 \). The trace of \( n_1 \) is \((e_1, q_1)\). Since \( q_1q_2 \)
satisfies the regular expression, then the expansion \( Q \) of \( P_1 \) is \( P \)-valid. Notice that the
automaton is used only in order to verify the existence of a rule associating some state to
the label \( y = 4 \). Thus, \( \operatorname{Expand} \) returns \( Q \).

\[ + \]

\textbf{Figure 5. Generating patterns of size 4 by joining patterns of size 3.}

\section{4. Experimental Results}

All experiments were performed on 3GHz Pentium 4 PC with 1GB memory running Suse
Linux 9.2.

\textbf{Synthetic Datasets.} We have developed a data generator which produces tree datasets.
The generator operates in the following way: (1) It has a procedure responsible for
creating regular tree grammars (equivalent to tree automata) according to five parameters:
\( K, T, k, t, e \). We illustrate in Figure 4(a) a tree grammar obtained by taking
\( K = 3, T = 2, k = 2, t = 2, e = 2 \). The general idea of the generated tree grammars is the following:
the first rule is \( q_0 \rightarrow a_0r_0 \), where \( a_0 \) is a label in \( L \) and \( r_0 \) is a
regular expression over the alphabet of states. Only one state appearing in \( r_0 \) (let us call
it \( q_1 \)) will have a rule of the form \( q_1 \rightarrow a_1r_1 \). All the other states \( q \) appearing in \( r_0 \) have
corresponding rules of the form \( q_1 \rightarrow \epsilon \). The same process repeats for the rule \( q_1 \rightarrow a_1r_1 \)
(only one state in \( r_1 \) is chosen, etc). After \( K \) passes, we come to the initial state \( q_0 \) and
the cycle is closed. In Figure 4(a) each group of rules is delimited by horizontal lines.
The parameter \( T \) stands for the number of choices we have for the rules corresponding
to the first state in each group. The parameters \( k, t \) and \( e \) concern the general format of
the regular expressions appearing in the right side of the grammar rules. These regular
expressions are of the form \((B_1^* B_2^* \ldots B_k^*)\), where each block \(B_i\) is of the form \((T_1 \ldots T_t)\), and each \(T_j\) (called term) is of the form \((q_{i_1} + \ldots + q_{i_e})\), where each \(q_i\) is a state symbol (i.e., a non-terminal symbol). Thus, our generator can produce a broad spectrum of regular expressions, by varying the number of blocks \((k)\), the maximum number of terms in each block \((t)\) and the maximum number \(e\) of alternative states appearing in the terms. In Figure 4(b) we show two tree patterns generated by the tree grammar. In Figure 4(a).

(2) Given a tree grammar \(G\) (equivalent to a tree automata), the generator produces a number \(N\) of tree patterns, with average fanout \(F_{avg}\) and average depth \(P_{avg}\). The parameters \(K, T\) of the tree grammar are related to the depth of the generated patterns. Parameters \(k, t\) of the regular expression at the right side of the rules are related to the fanout of the generated patterns. (3) Given a set of \(N\) tree patterns, the generator produces a dataset of \(M\) trees where these patterns appear in a certain proportion given by statistical parameters. The parameter \(\mu\), \(0 \leq \mu \leq 1\), stands for the rate of valid patterns inserted into the dataset. For instance, if \(\mu = \frac{1}{2}\) then the dataset \(D\) surely contains the same proportion of valid and non-valid patterns.

For the tests presented in this paper, we used 13 synthetic datasets and 7 tree automata, generated by 7 distinct tree grammars. We use the notation DB-xy to denote an input dataset, where \(x\) corresponds to the unique parameter evaluated by a non-default value and \(y\) is the corresponding value \(x\). So, DB-M200 corresponds to the dataset with 200(x1000) trees and DB-K1 corresponds to the dataset where the tree grammar used for generating the “hidden” patterns has \(K = 1\). The values for the other parameters are set as the default ones. DB-DEF is a dataset generated by setting the parameters with their default values. We use a similar notation to denote the tree automaton A-zw.

Real Dataset. We have considered a real dataset DB-X consisting of subtrees of the XML document SigmodRecord.xml obtained from the XML Data Repository \(^1\). The roots of the subtrees correspond to the tag <article>. The whole database contains 1504 trees. A piece of such document is shown in Appendix A. In all the experiments, we used the automaton A-X for mining this dataset (given in Appendix B). It restricts the patterns structure in the following way: only articles having one unique author, for which the initial and final pages are given in the document and which satisfy some other restrictions concerning the contents of these items (authors and page numbers) are considered interesting to be mined. We used the default parameters shown in Table 1.

Performance Analysis. In Figure 7(a), we present the results of the experiments comparing the performaces of CobMiner and TreeMinerPP. When considering a minimum support of 5\%, CobMiner is 2 times more efficient than TreeMinerPP. The efficiency rate of CoBMiner with respect to TreeMinerPP increases considerably as the minimum support decreases. For a minimum support of 1\% CobMiner is 8 times more performing than TreeMinerPP. This difference between the performance of the two algorithms is due to the restriction on the search space of CoBMiner, during the generation phase. The mining process is directed by the tree automaton. If, at a given level \(k\), there is no more \(p\)-valid patterns, the algorithm terminates its mining phase and goes to the post-processing phase. Instead, TreeMinerPP continues the candidate generation phase, which is very time-consuming in the tree mining context, due to the great amount of generated patterns.

\(^1\)http://www.cs.washington.edu/research/xmldatasets/data/sigmod-record/SigmodRecord.xml.
CobMiner’s performance superiority can also been verified in Figure 7(b), which shows the executions of the two algorithms over the real dataset DB-X. In these experiments the minimum support varied from 0,5% to 10%. Notice that TreeMinerPP presented a better performance when executed over the synthetic dataset containing 100,000 trees than when executed over the real dataset DB-X of 1504 trees. This apparent discrepancy is explained by the fact that DB-X trees have siblings nodes with descendants having maximum fanout 21, while in DB-DEF dataset (generated by our synthetic tree grammars) only one child node has descendants and the maximum fanout is 10. In Figure 8(a) we show an example of frequent pattern mined by both algorithms in the DB-X dataset. We also show, in Figure 8(b) the number of candidates generated, the number of frequent patterns mined by both algorithms before the post-processing phase and the total number of frequent patterns obtained by both algorithms after the post-processing phase.
Figure 7. Performance and Scalability Results

(a) A mined pattern; (b) CoBMiner efficiency (c) Pruning Effect

Figure 7(d) shows the variation of the execution times of both algorithms with respect to the parameter $\mu$. When $\mu$ varies from 0.1 to 0.50, the execution times decreases for both algorithms. For $\mu = 0.1$ (many valid “hidden” patterns) CoBMiner is 3.5 times more efficient than TreeMinerPP. With $\mu = 0.75$ (few valid patterns) the execution times of both algorithms verify a similar proportion. Figures 7(e) and (f) show the behaviour of both algorithms when the grammar parameters $k$ and $K$ vary. These parameters affect the size of the “hidden” patterns in the dataset. As $k$ and $K$ increase, the average fanout and average depth of the hidden patterns increase as well. As expected, the execution time of both algorithms increase, since the number of levels in the recursive execution is related to the size of the “hidden” patterns. Notice that for $k = 4$ (bigger fanout) CoBMiner is nearly 6 times more efficient than TreeMinerPP.

Scalability Analysis Figure 7(c) shows how both algorithms scale when the number of trees in the database is increased from 50,000 to 600,000. With a minimum support of 5% and constraints given by the automaton A-DEF, we notice that CobMiner is 2 times more efficient than TreeMinerPP.

Pruning Effect. We have implemented a pruning strategy for CoBMiner, based on the following: at each step $k$, after p-valid candidates have been generated, we prune those candidate patterns $Q$ for which there exists a subpattern $P$ which is p-valid and does not appear in the set of frequent patterns produced so far. We evaluated the effect of pruning on the performance of CobMiner and TreMinerPP. CoBMiner outperforms TreeMinerPP when both algorithms are implemented with a pruning phase. We notice that TreeMinerPP
always benefits from pruning. On the other hand, the benefits of pruning on CobMiner is
minor, sometimes it is even disadvantageous. One reason for this is that the number of
candidates to prune is low and there are an execution cost in performing the pruning test to
all candidates, affecting the performance of CobMiner. Figure 8(c) shows the number of
candidates generated and the corresponding execution times for CobMiner, on DB-DEF
dataset with no pruning and with pruning step, with support varying from 1% to 10% and
restriction given by A-DEF. Analyzing CobMiner executions where minsup = 1%, only
255 candidates have been pruned and the pruning test increased the execution time of
CobMiner.

5. Further Research
Currently we are developing a Web Mining system based on CoBMiner. Our future re-
search focus on the implementation of a third algorithm (CoBMinerV) which consists in
modifying the generation phase of CoBMiner in order to produce, at each pass, valid pat-
terns instead of p-valid ones. Since pruning phase is almost irrelevant for the performance
of CoBMiner, we think that forcing a more restrictive constraint (patterns must be valid,
not only p-valid) at the generation phase and so, eliminating the post-processing phase,
may produce better results.

References
Efficient substructure discovery from large semi-structured data. In SDM ’02: Proceedings of the Second
ACM SIGMOD Int. Conf. on Management of Data, pages 682–682, New York, NY, USA. ACM Press.
Recomendation.
from Web Data. PhD thesis, Department of Computer Science, University of Minnesota.
with regular expression constraints. In 20th Brazilian Symposium on Databases, Uberlândia, Brazil,
pages 280–294.
of frequent tree structured patterns in semistructured web documents. In PAKDD ’01, pages 47–52,
pages 271–274.
New York, NY, USA. ACM Press.
Appendix A: A piece of the XML document SigmodRecord.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SigmodRecord>
  <issue>
    <volume>11</volume>
    <number>1</number>
    <articles>
      <article>
        <title>Annotated Bibliography on Data Design.</title>
        <initPage>45</initPage>
        <endPage>77</endPage>
        <authors>
          <author position="00">Anthony I. Wasserman</author>
          <author position="01">Karen Botnich</author>
        </authors>
      </article>
      <article>
        <title>Architecture of Future Data Base Systems.</title>
        <initPage>30</initPage>
        <endPage>44</endPage>
        <authors>
          <author position="00">Lawrence A. Rowe</author>
          <author position="01">Michael Stonebraker</author>
        </authors>
      </article>
      <article>
        <title>Database Directions III Workshop Review.</title>
        <initPage>8</initPage>
        <endPage>8</endPage>
        <authors>
          <author position="00">Tom Cook</author>
        </authors>
      </article>
      <article>
        <title>Errors in 'Process Synchronization in Database Systems'.</title>
        <initPage>9</initPage>
        <endPage>29</endPage>
        <authors>
          <author position="00">Philip A. Bernstein</author>
          <author position="01">Marco A. Casanova</author>
          <author position="02">Nathan Goodman</author>
        </authors>
      </article>
    </articles>
  </issue>
</SigmodRecord>
```
Appendix B: Tree Automaton A-X = (Q, q_0, Σ, δ)

Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{20}, q_{21}, q_{22}, q_{23}, q_{24}, q_{25}, q_{26}, q_{27}, q_{28}, q_{29}, q_{30}\}


\(\delta(q_0, article) = q_1.q_2.q_3;\)
\(\delta(q_1, initPage) = q_4.q_5.q_6.q_7.q_8.q_9.q_{10}.q_{11}.q_{12};\)
\(\delta(q_2, endPage) = q_{13}.q_{14}.q_{15}.q_{16}.q_{17}.q_{18}.q_{19};\)
\(\delta(q_3, authors) = q_{20};\)
\(\delta(q_4, 9) = \epsilon;\)
\(\delta(q_5, 19) = \epsilon;\)
\(\delta(q_6, 20) = \epsilon;\)
\(\delta(q_7, 25) = \epsilon;\)
\(\delta(q_8, 30) = \epsilon;\)
\(\delta(q_9, 31) = \epsilon;\)
\(\delta(q_{10}, 33) = \epsilon;\)
\(\delta(q_{11}, 36) = \epsilon;\)
\(\delta(q_{12}, 37) = \epsilon;\)
\(\delta(q_{13}, 40) = \epsilon;\)
\(\delta(q_{14}, 44) = \epsilon;\)
\(\delta(q_{15}, 45) = \epsilon;\)
\(\delta(q_{16}, 52) = \epsilon;\)
\(\delta(q_{17}, 55) = \epsilon;\)
\(\delta(q_{18}, 340) = \epsilon;\)
\(\delta(q_{19}, 355) = \epsilon;\)
\(\delta(q_{20}, author) = q_{21}.q_{22}.q_{23}.q_{24}.q_{25}.q_{26}.q_{27}.q_{28}.q_{29};\)
\(\delta(q_{21}, Jeffrey D. Ullman) = \epsilon;\)
\(\delta(q_{22}, Oumlzguumlr Ulusoy) = \epsilon;\)
\(\delta(q_{23}, Arnon Rosenthal) = \epsilon;\)
\(\delta(q_{24}, James H. Burrows) = \epsilon;\)
\(\delta(q_{25}, Richard L. Nolan) = \epsilon;\)
\(\delta(q_{26}, Peter Thanisch) = \epsilon;\)
\(\delta(q_{27}, Ulrich Schiel) = \epsilon;\)
\(\delta(q_{28}, Ahmed K. Elmagarmid) = \epsilon;\)
\(\delta(q_{29}, Edwin McKenzie) = \epsilon;\)
\(\delta(q_{30}, Michael Stonebraker) = \epsilon;\)