Similarity Based Retrieval from Sequence Data using Automata as Queries

A. Prasad Sistla Tao Hu
Vikas Chowdhry
Department of Electrical Engineering and Computer Science,
University of Illinois at Chicago,
Chicago, Illinois 60680
thu@eecs.uic.edu,sistla@surya.eecs.uic.edu

October 3, 2001

Abstract

Similarity based retrieval from sequence databases is of importance in many applications such as time-series, video and textual databases. In this paper, automata based formalisms are introduced for specifying queries over such databases. Various measures defining the distance of a database sequence from an automaton are defined. The distance of a database sequence from a query automaton is defined to be the minimum of its vector distances from each string accepted by the automaton.

Efficient methods for similarity based retrieval are presented for each of the distance measures. These methods employ indices for fast retrieval. They retrieve the k closest subsequences, of the database sequence, with respect to the query automaton. The proposed methods can be built on top of any database, which given an atomic query (i.e. a query on single items), outputs identities and similarity values of those items that closely satisfy the atomic query.

1 Introduction

Sequence Databases occur in many areas of research in Database Management Systems. For example, Temporal Databases, Time-series Databases and Video Databases are some examples of sequence databases. Many query languages and query processing algorithms have been developed in the literature for such databases. The web site [S] contains a survey of work on query languages for such database systems. All these works consider exact semantics for retrieval with respect to a given query; that is, a database subsequence is considered to be in the answer for a query only if it exactly satisfies a query predicate. In this paper we consider similarity based retrieval from sequence databases. Similarity based retrieval consists of retrieving those subsequences that closely satisfy the query based on a similarity measure. There has been some work done [AFS93, FRM94, ALSS95, APWZ95, RM97] on similarity based retrieval from sequence databases (see related works section for a detailed comparison to these works). Most of these works address the problem when the query is a single sequence of data values. The methods employed in these works cannot be used when the query is a predicate over sequences. In this paper, we consider a language based on finite state automata for specifying queries on sequences, and develop similarity based methods for retrieval. These methods employ indices for fast retrieval and can be implemented on top of an existing commercial database system. We have implemented the proposed methods on top of Sequel Server. Experimental results indicating the effectiveness of the methods are given in the paper.

The automata that we employ are more powerful than regular finite state automata. They allow us to specify many important properties including real-time properties that cannot be specified by ordinary automata. For example, we can express properties such as "the price of IBM stock raises by more than 10% in 1 hour" or the "price of IBM fluctuates between 100 and 120 during a certain period of time", etc. To the best of our knowledge, ours is the first work to develop similarity based techniques for retrieval using these automata.

In our formalism, a database sequence d is a sequence of database states denoting the states of the database at different instances of time as it evolves over time. For a query automaton $\mathcal{A}$, the input symbols represent atomic queries on individual database states. For a database sequence d and query automaton $\mathcal{A}$, we define a distance measure $dist(d, \mathcal{A})$ that denotes how closely the database sequence d satisfies the query specified by $\mathcal{A}$. This distance is defined to be the minimum value of the distance between d and each string of length $|d|$ accepted by $\mathcal{A}$. The distance between the database sequence $d = (d_1, ..., d_n)$ and a string $a = (a_1, ..., a_n)$ of length n accepted by $\mathcal{A}$, is defined using the similarity vector that, for each $i = 1, ..., n$, denotes how closely the atomic query a_i is satisfied in the database state d_i. The of value of $dist(d, \mathcal{A})$ is defined using the above similarity vectors and some standard vector distance functions. We consider a spectrum of vector distance functions (called norm distance measures) as defined by equations 1 and 2 in section 3. At one end of the spectrum is the average distance measure (denoted by F_1), followed by the mean square distance function, etc. At the other end of the spectrum is the infinity norm distance function (denoted by F_∞).

We consider the following problems for a given database sequence d and a specification automaton $\mathcal{A}$: (i) retrieval of k closest subsequences of d with respect to the automaton $\mathcal{A}$ (called "nearest neighbor query"); (ii) retrieval of all subsequences of d with in a given distance from $\mathcal{A}$ (also called "range query"). Here the closeness is measured by the distance with respect to the automaton as defined above. We first present algorithms for this problem assuming that the automaton is simple (i.e. it does not use any variables). We show later how these algorithms are extended to the general case when the automaton uses variables. For the distance function F_∞, we present two algorithms for the above problem. The first one does not employ any indices and is of complexity $O(kn|A|)$ where n is the length of the database sequence and $|A|$ is the size of the automaton including the number of states and transitions. The second algorithm employs indices on atomic queries and has worst complexity $O(krp^3 \log n)$ where r is the sum of the lengths of the similarity lists associated with the atomic queries, p is the number of automaton states; this algorithm will be useful when r is much smaller than n. For the F_1 distance function, we present an algorithm employing indices that is of complexity $O(r^2|A| + rp^3 \log n)$. The algorithm given for F_1 can be trivially extended to F_i for any i such that $1 < i < \infty$.

We also consider a restricted class of automata, called *cycle-restricted* automata. We present more efficient algorithms for these automata. Many standard queries are specified by such automata.

The complexity measures, as given above, only consider the internal processing time after the similarity lists for atomic queries are generated. We need to add the additional time (such as I/O time, etc.) for generating these similarity lists from the database. To do this, we have implemented a system on PCs using Sequel server for time-series databases. This system assumes that the time-series database is stored in a single relation. The implemented system has two components. The first component generates similarity lists for atomic queries by executing SQL queries through the Sequel server. The second component executes the previously mentioned algorithms for similarity based retrieval of subsequences. We have carried out various experiments using a real stock market database, and also an artificially generated database, showing the effectiveness of our methods. We also have built another similar system for textual databases. As expected, algorithms employing infinite norm distance are faster than the one that employ average block distance function. However these algorithms retrieve fewer subsequences as infinite norm is a more discriminating distance function. When I/O time is significant (i.e., when the database is large) then algorithms employing indices run faster.

The paper is organized as follows. In section 2, we discuss related work. In section 3, we give the definitions of the different automata and the distance measures with examples. Section 4 contains retrieval algorithms for the simple automata (without variables) under infinite norm distance function. Section 5 presents method for average block distance function. Section 6 shows how to extend the above algorithms for the automata with variables. Section 7 presents the experimental results. Section 8 contains conclusions and discussion.

2 Related Work

There has been much work done on querying from time-series and other sequence databases. For example, methods for similarity based retrieval from such databases have been proposed in [FRM94, AFS93, ALSS95, B97, RM97, NRS99]. These methods assume that the query is also a single sequence, not a predicate on sequences as we consider here.

In [FRM94] the problem of similarity based subsequence matching is studied. In their approach, the query sequence, of length m, is mapped into a point in k-dimensional space (called the transformed domain) by taking the first k coefficients of its Discrete Fourier Transform (DFT). The data sequence is transformed in to a trail of points in the transformed domain by taking the DFTs of successive subsequences of length m. Each such trail is divided in to sub-trails which are approximated by their minimum bounded rectangles (MBRs). These MBRs are organized into an index such as R^*-tree which is used to retrieve all rectangles that are with in a distance of ϵ from the query point. Thus, this method employs multi-dimensional indexing and assumes that the query is a single sequence. The work described in [RM97] also assumes that the query is a single sequence and the database is a set of sequences. In their method, the user also inputs a transformation function T together with the query sequence q. All database sequences d such that $T(d)$ is with in distance ϵ of $T(q)$ are retrieved. The transform functions is user specified and is general enough to allow consideration of moving averages etc. The paper [ALSS95] presents methods for discovering all similar sequences in a set of sequences. It can also be used to find all subsequences similar to a given query sequence. Their similarity relation permits scaling and shifting of sequences. Thus, this method can not be applied when the query is a predicate over sequences. The method given in [NRS99] employs wavelet transforms and is primarily used for retrieval from image databases (not sequences).

The paper [APWZ95] presents a language, called SDL, for specifying shapes of histories. This language is intended primarily for time-series databases. The language is defined using various operators and is equivalent to regular expressions in expressive power. The basic symbols in the alphabet denote how the values change from one instant to the next instance with in some limits (i.e., whether the value is going up or down, etc). The paper gives an index scheme for retrieving subsequences satisfying a SDL query. The retrieval is done based on exact match and is not similarity based retrieval like ours using a global distance measure.

There has also been much work done on data-mining over time series data [AS94, GRS99] and other databases. These works mostly consider discovery of patterns that have a given minimum level of support. They do not consider similarity based retrieval.

The papers [S, WS01] describes a parallel database management KDB for retrieval from huge time series databases (of hundreds of gigabytes in size) of stock quotes. It employs a query language (called KSQL, which is an extension of SQL) for specifying statistical queries over such databases. However, its retrieval is not similarity based.

A temporal query language and efficient algorithms for similarity based retrieval have been presented in [SYV97]. In that work, the query is specified by a formula in Temporal Logic (i.e., is a predicate over sequences). The logic considered in [SYV97] is less expressive than the extended automata considered in this paper. (For example, the query "price of IBM increases by 10% in one hour" can not be expressed in that language). Further more, the similarity measure defined there is syntax driven and is ad-hoc. On the contrary, the distance measures defined in the current paper is semantics based (i.e., the distance measure of a database sequence with respect to a query automaton is defined to be the minimum of its distance from each sequence accepted by the automaton). Also, these distance measures are based on well accepted distance measures over vectors (such as average block distance, Euclidean distance etc.).

There has been work done on approximate pattern matching (see [WM92] for references) based on regular expressions. Regular expressions are less powerful than our automata with variables. For example, the query "price of IBM increase by 10% in one hour" can not be expressed using regular expressions. Further more, the algorithms used for approximate pattern matching do not employ indices and they do not consider the same distance measures as ours. For example, they usually use the edit distance as a measure and look for patterns defined by a given regular expression with in a given edit distance. On the other hand, we consider average measures; for example, the distance function F_1 defines average block distance. In the area of bio-informatics much work has been done on sequence matching (see [D98] for references). Most of this work is based on probabilistic models (such Markov or extended Markov models). They do not employ techniques based on indices for the subsequence search.

Predicates on sequences have been employed in specifying triggers in Active Database Management Systems [C89, D88, GJS92, SW94]. However, there exact semantics is used for firing and processing the triggers.

3 Definitions and Examples

3.1 Basic Automata and Similarity values

Automata

An automaton $\mathcal{A}$ is 5-tuple $(Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is a finite set of symbols called the input alphabet, δ is the set of transitions, $I, F \subseteq Q$ are the set of initial and final states, respectively. Each transition of $\mathcal{A}$, i.e. each member of δ, is a triple of the form (q, a, q') where $q, q' \in Q$ and $a \in \Sigma$; this triple denotes that the automaton makes a transition from state q to q' on input a; we also represent such a transition as $q \ \rightarrow_a \ q'$. Each input symbol represents an atomic predicate (also called an atomic query in some places) on a single database state. For example, in a stock market database, $price(ibm) = 100$ represents an atomic predicate. In a textual database, each database state represents a document and a database sequence represents a sequence of documents; here an atomic predicate may state that the document contain some given key words. In a video database, which is a sequence of images or shots, each atomic predicate represents a condition on a picture such as requiring that the picture contain some given objects. Consider the automaton in figure 1. It has three states and three input symbols. Its input symbols are the atomic queries $time = 10AM, time = 4PM$ and $price(IBM) < 100$.

Let $a = a_0, a_1, ..., a_{n-1}$ be a sequence of input symbols from Σ and q, q' be states in Q. We say that the sequence a takes the automaton $\mathcal{A}$ from state q to q' if there exists a sequence of states $q_0, q_1, ..., q_n$ such that $q_0 = q$ and $q_n = q'$ and for each $i = 0, ..., n-1$, $q_i \ \rightarrow_{a_i} \ q_{i+1}$ is a transition of $\mathcal{A}$. For any state q, we let $T(q)$ denote the set of sequences that take the automaton from state q to a final state. We say that the automaton $\mathcal{A}$ accepts the string a if there exists an initial state q such that $a \in T(q)$. We let $L(\mathcal{A})$ denote the set of strings accepted by $\mathcal{A}$. The automaton in figure 1 accepts the string ($time = 10AM$, $pric(IBM) < 100$, $price(IBM) < 100$, $time = 4PM$).

For any sequence s, let $|s|$ denote its length. Let $s = (s_0, ...,)$ be a sequence. For any i, j such that $0 \leq i \leq j < |s|$, we let `s[i,j]` denote the sequence $(s_i, s_{i+1}, ..., s_j)$ and `s(i)` denote the suffix of s starting from s_i. For any set Δ of symbols, let Δ^* denote the set of all finite sequences over Δ; note that the empty sequence is also in Δ^*. We identify vectors with sequences. We let $\vec{1}$ denote a vector all of whose components are 1. The length of the vector will be clear from the context.

Similarity Measure

A database sequence d is a finite sequence of database states. A database state my represent an image/shot (in case of video databases) or an assignment of values to database variables (in case of time-series databases) or a document in case of textual databases. As indicated earlier, each symbol in the input alphabet of an automaton represents an atomic predicate on a database state. For a database state c and an atomic query c', we let $sim(c', c)$ denote the similarity value with which c satisfies the query c'. Usually, we assume that there is an underlying mechanism for computing this value. We let $dist(c, c') = 1 - sim(c, c')$ represent the distance between c and c'.

Let $\mathcal{A}$ be the automaton as defined earlier. Now we define the similarity of a database sequence $d = (d_0, ..., d_{n-1})$ with respect to an automaton $\mathcal{A}$ as follows. First, we define a distance measure $dist(d, a)$ between d and an input sequence $a = (a_0, ..., a_{n-1})$ of equal length as follows. Let $sim\_vec(d, a)$ be the sequence $(s_0, ..., s_{n-1})$ where for each $i = 0, ..., n-1$, $s_i = sim(a_i, d_i)$. We assume that all similarity values and distances are normalized, i.e. they lie in the interval `[0, 1]`. Let F be a vector distance function which given two vectors $\vec{x}, \vec{y}$ as arguments, associates a positive real number lying in the interval `[0, 1]`. We define $dist(d, a) = F(sim\_vec(d, a), \vec{1})$. Now, we define a distance measure $dist(d, C)$ between the database sequence d and a set $C \subseteq \Sigma^*$ as follows. $dist(d, C)$ is the minimum of $dist(d, \alpha)$, where the minimum is taken over all $\alpha \in C$ such that $|\alpha| = |d|$; if there is no sequence $\alpha \in C$ such that $|\alpha| = |d|$ then we take $dist(d, C)$ to be equal to 1. Now we define the distance of d with respect to $\mathcal{A}$, denoted by $dist(d, \mathcal{A})$, to be $dist(d, L(\mathcal{A}))$. We define the similarity of d with respect to the automaton $\mathcal{A}$, denoted by $sim(d, \mathcal{A})$, to be $1 - dist(d, L(\mathcal{A}))$.

Clearly, the similarity value $sim(d, \mathcal{A})$ depends on the distance function F. We consider a spectrum of vector distance functions $\{F_m \ : m = 1, 2, ..., \infty\}$ defined as follows. Let $\vec{x}, \vec{y}$ be two vectors. For each $m = 1, ..., \infty$, if $\vec{x}, \vec{y}$ are of unequal length then $F_m(\vec{x}, \vec{y}) = 1$. Otherwise, let n be the length of both $\vec{x}$ and $\vec{y}$. In this case,

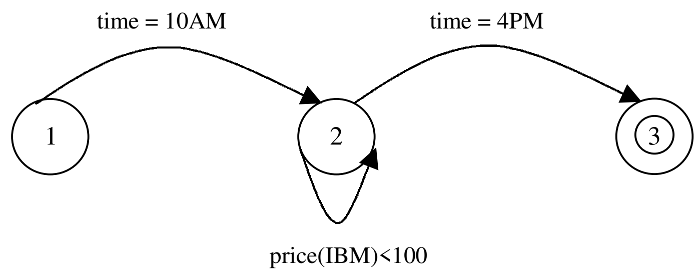

Figure 1:

$F_m(\vec{x}, \vec{y})$ is given as follows. For $m \neq \infty$,

$$F_m(\vec{x}, \vec{y}) = (\frac{\Sigma_{1 \leq i \leq n}(|x_i - y_i|)^m}{n})^{1/m} \tag{1}$$

$$F_\infty(\vec{x}, \vec{y}) = \max\{|x_i - y_i| \, : \, 1 \leq i \leq n\} \tag{2}$$

Note that F_1 is the average block distance function and F_2 is the mean square distance function, etc. It can easily be shown that $F_\infty(\vec{x}, \vec{y}) = \lim_{m \to \infty} F_m(\vec{x}, \vec{y})$. Note that $F_1(\vec{x}, \vec{y})$ gives equal importance to all components of the vectors; however, as m increases, the numerator in the expression for $F_m(\vec{x}, \vec{y})$ is dominated by the term having the maximum value, i.e. by $\max\{|x_i - y_i| \, : \, 1 \leq i \leq n\}$, and in the limit $F_\infty(\vec{x}, \vec{y})$ equals this maximum value. Thus we see that F_1 and F_∞ are the extremes of our distance functions. We call F_1 as the *average block distance* function and F_∞ as the *infinite norm* distance function. We give algorithms for computing the similarity values using these two distance functions. For other values of m, the algorithms are obtained by a simple modification to the algorithm for F_1. As we show later, it is easier to compute the similarity values using the distance function F_∞ as compared to F_1.

Example: Let $\mathcal{A}$ be the automaton in figure 1. Intuitively speaking, $\mathcal{A}$ states the following property: the value of *time* is $10AM$ and $4PM$ in the first and last database states respectively, and in all intermediate database states the value of $price(IBM)$ is less than 100. (Note that in the first and last database states the value of $price(IBM)$ can be any value.) Consider a database sequence $d = (d_0, d_1, d_2, d_3)$ such that the value of *time* is $10AM$ and $4PM$ in d_0 and d_3 respectively. Assume that $price(IBM)$ has values 104 and 109 in states d_1 and d_2 respectively. Suppose, we define the similarity value of the atomic predicate $price(IBM) < 100$ to be decreasing from 1 to 0 linearly as the value of $price(IBM)$ increases from 99 to 119. Then $sim(d_1, price(IBM) < 100) = .75$ and $sim(d_2, price(IBM) < 100) = .5$. If we use F_1 distance function, then $dist(d, \mathcal{A}) = \frac{.75}{4}$; on the other hand if we use F_∞, then $dist(d, \mathcal{A}) = .5$.

Wild Card Symbol

We assume that there is a special input symbol ϕ which denotes a wild card symbol, i.e. it denotes an atomic query which is always satisfied. The previous definition of the distance function $dist(d, \mathcal{A})$ and $sim(d, \mathcal{A})$ are extended with the following modifications. For a database sequence d and input sequence $a = (a_0, ..., a_{n-1})$, we define $sim\_vec(d, a)$ by considering only non-wild card symbols as follows. Let $i_0 < i_1 ... < i_{l-1}$ be all the indices such that for each $j = 0, ..., l-1$, $a_{i_j} \neq \phi$. Define $sim\_vec(d, a) = (s_0, s_1, ..., s_{l-1})$ where $s_j = sim(d_{i_j}, a_{i_j})$. The values $dist(d, \mathcal{A})$ and $sim(d, \mathcal{A})$ are defined as before using the above definition of $sim\_vec(d, a)$. Thus, we do not consider the wild card symbols when computing the similarity values.

Cycle-Restricted Automata

Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be an automaton. A *path* of the automaton is a sequence of transitions of the following form — $q_0 \rightarrow_{a_0} q_1, q_1 \rightarrow_{a_1} q_2, ..., q_{n-1} \rightarrow_{a_{n-1}} q_n$. We call such a sequence as a path from q_0 to q_n. The length of a path is the number of transitions appearing in it. The length of the above path is n. We call the path a ϕ-path if all input symbols appearing in it are wild cards, i.e., for each $i = 0, ..., n-1$, $a_i = \phi$. The above path is called a cycle if $q_n = q_0$ and $q_0, q_1, ..., q_{n-1}$ are all distinct. A ϕ-path which is also a cycle is called a ϕ-cycle. We say that an automaton is *cycle-restricted* if it has no ϕ-cycles of length greater than 1.

3.2 Extended Automata

In this subsection, we extend the basic automaton model. The extended automaton allows variables and is more expressive. For example, the query "the price of IBM stock increases by 10% in ten minutes" can not be expressed using the earlier automaton; this requires comparison of database values at different times; this is done in the extended automaton by the use of global variables.

We assume that the database consists of a finite set D of variables called *database variables*. A database state f is a mapping that associates values to the database variables. In addition to the database variables, we have another set X of variables called *global* variables. These variables will be used in the automaton for capturing values of the database variables in some state and comparing them to their values in a different state. An evaluation ρ is a mapping that associates appropriate values to the global variables.

We extend the automata defined in the previous subsection in the following way. First, recall that each input symbol to an automaton represents a predicate on a single database state. In our new model, the input symbols may be parameterized, i.e. they may have global variables appearing in them. For example, if $price(ibm)$ represents a database variable then $price(IBM) \geq 100$ and $price(IBM) \geq 2x$ represent input symbols; the former has no global variables appearing in it, while the later has the global variables x appearing in it. Recall that, previously we assumed that for each input symbol a and database state f, we have a similarity measure $sim(f, a)$ that denotes how closely the atomic predicate a is satisfied in the database state f. In our new model, we assume that the sim measure takes another parameter ρ, which is an evaluation; thus, $sim(f, a, \rho)$ denotes how closely the system state (f, ρ) satisfies a. (If a has no global variables appearing in it, then the value of $sim(f, a, \rho)$ is independent of ρ, i.e., it only depends on f, a). As usual we let $dist(f, a, \rho)$ be the distance measure given by $1 - sim(f, a, \rho)$. For the wild card symbol ϕ, we assume that $dist(f, \phi, \rho)$ is zero for all f and ρ. In our new automaton model, the automaton can change values of the global variables by assigning new values to some of these variables in each of its states; we do this by associating some assignments with each state of the automaton. An assignment is pair of the form (x, q) where x is a global variable and q is a query on a database state that retrieves a single value; such an assignment is represented by $x \leftarrow q$.

Formally, an automaton $\mathcal{A}$ is a tuple of the form $(Q, \Sigma, \delta, I, F, \psi)$ where Q, Σ, δ, I, F are as given before and ψ is a function that associates a set of assignments with each state such that the following condition is satisfied: for each $q \in Q$ and each global variable x, there is at most one assignment to x in $\psi(q)$. A path in Q is a sequence of automaton states $q_1, ..., q_n$ such that $q_1 \in I$ and for each $i = 1, ..., n-1$, there is a transition of the automaton from q_i to q_{i+1} on some input symbol. Note that a path always starts from an initial state. We require that the following condition be satisfied by the automaton for every transition $q \rightarrow_a q'$ in δ: for every path to the state q and for every global variable x appearing in a there should be some state on the path that has an assignment to the variable x; that is, for every path $q_1, ..., q_n$ such that $q_n = q$ and for every global variable x in a there exists an $i \leq n$ such that $\psi(q_i)$ has an assignment for x. Intuitively, this condition ensures that every global variable is assigned some value before it is referenced.

An extended input symbol is a pair of the form (a, γ) where a is an input symbol (i.e., $a \in \Sigma$) and γ is a set of assignments to the global variables such that each variable is assigned at most once. An input string is a sequence (a_1, γ_1), (a_2, γ_2), ..., (a_n, γ_n) of extended symbols. We say that the above string is accepted by the automaton $\mathcal{A}$ if the following condition is satisfied: there exists a sequence of states $q_1, ..., q_{n+1}$ such that $q_1 \in I$, $q_{n+1} \in F$ and for each $i = 1, ..., n$, $q_i \rightarrow_{a_i} q_{i+1}$ is a transition of the automaton and $\gamma_i = \psi(q_i)$ (i.e. γ_i is the set of assignments in q_i). We let $L(\mathcal{A})$ denote the set of strings accepted by $\mathcal{A}$.

Now we define a distance measure between a database string and an input string of the same length. Let a denote the input string $(a_1, \gamma_1), ..., (a_n, \gamma_n)$ and d denote the database sequence $(d_1, ..., d_n)$. We define a distance measure $dist(d, a)$ as follows. Let $\rho_1, ..., \rho_n$ be any sequence of evaluations such that for each $i = 1, ..., n$ and for

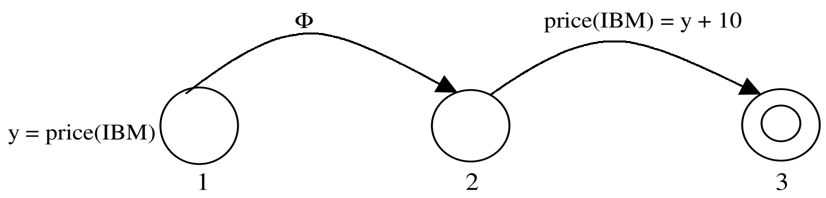

Figure 2:

each global variable x, the following condition is satisfied: if γ_i contains an assignment of the form $x \leftarrow query$ then $\rho_i(x)$ is the value of the query in the database state d_i, otherwise $\rho_i(x) = \rho_{i-1}(x)$ for $i > 1$. Thus the evaluations ρ_i (for $i = 1, ...n$) capture the values of different queries at different points on the database sequence. Let $sim\_vec(d, a)$ be the vector $(s_1, ..., s_n)$ where $s_i = sim(d_i, a_i, \rho_i)$, i.e. s_i denotes how closely the database state d_i satisfies the atomic query a_i with respect to the evaluation ρ_i. As before we define $dist(d, a)$ to be the value $F(sim\_vec(d, a), \vec{1})$ where F is one of the distance functions defined in section 3. We define $dist(d, \mathcal{A})$ to be $\min\{dist(d, a) : a \in L(\mathcal{A})\}$.

3.3 Examples

The following examples refer to a sequence database capturing instantaneous stock trades that occurred on various days during an entire year. In all these examples, state 1 is the initial state and the final state is denoted by double circle.

Now consider the automaton given in figure 2. It has three states 1,2,3. States 1,3 are the initial and final states respectively. In state 1, the global variable y is assigned the value of $price(ibm)$. There is a transition from state 1 to state 2 on the input symbol ϕ (wild-card symbol) and there is a transition from 2 to 3 on the input symbol $price(ibm) = y + 10$. Roughly speaking, this automaton states that $price(ibm)$ has increased by 10 from the first database state to the second database state. (Note: the sequence $((\phi, \{y = price(IBM)\}), (price(IBM) = y + 10, \{\}))$ is the only string accepted by this automaton.) Suppose we have a database sequence in which the price of IBM in the first two database states is $90, 105$, respectively. If we are using F_∞ then it is easy to see that the distance of this sequence with respect to the automaton equals the distance measure denoting how closely the predicate $price(ibm) = 100$ is satisfied by the second database state; this value is .05 if we take the percentage deviation as the distance measure.

Now, consider the automaton given in figure 3. It has three states 1,2,3. It uses two global variables y, t. Here $time$ and $price(ibm)$ are database variables. Here $time$ is an integer giving time in some base units such hours etc. The automaton states that with in ten units of time the price of IBM has increased by 10%. It states that if y, t denote the price of IBM and time at the beginning then there is a future state where the price of IBM is $1.1y$ and $time$ is less than $t + 10$.

Now consider the automaton in figure 4. Here states 1,5 are the initial and final states, respectively. This automaton states that from the second database state onwards the following pattern repeats until the last database state (excluding the last one). The price of IBM increases from one state to the next by 10%; then

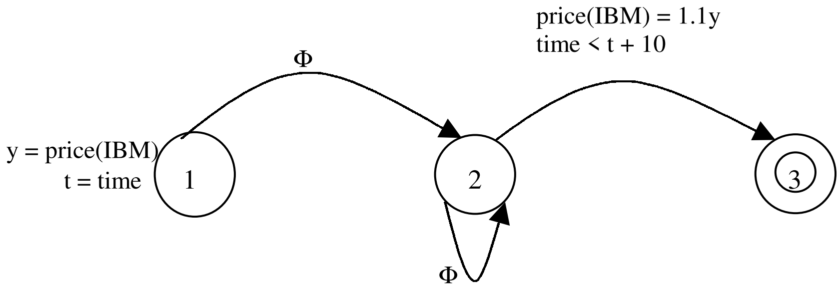

Figure 3:

the price of IBM decreases by 10% from the third state to the fourth state. This pattern among successive four states repeats until the last state (excluding the last state). It also states that this happens with in a span of 20 time units.

3.4 Nearest Neighbor and Range Queries

In this paper, we consider the evaluation of the following two types of queries assuming that we are given an automaton $\mathcal{A}$ and a database sequence d (i.e., the similarity values of atomic queries at the different states in d are given).

1. The first type of queries are called nearest neighbor queries. Here we have to retrieve k subsequences of d having the lowest distances with respect to $\mathcal{A}$ where k is an additional input which is a positive integer. Formally, the algorithm should output k pairs of the form (i, l) such that the k subsequences `d[i,i+l]` have the lowest distances with respect to the automaton $\mathcal{A}$. In the output, it is possible to have multiple subsequences having the same distance value.

2. The second type of queries are called range queries. Here we have to retrieve all subsequences of d whose distance with respect to $\mathcal{A}$ is less than or equal to ϵ, where ϵ is an additional input which is a positive fraction. Formally stated, the algorithm should output all pairs (i, l) such that $dist(\mathtt{d}[\mathtt{i}, \mathtt{i}+\mathtt{l}], \mathcal{A}) \leq \epsilon$.

An output to the query consists of a list of intervals, each denoting a subsequence of d. Although, we assume that we have a single database sequence, it can trivially be generalized to the case of many database sequences by adding an additional sequence id. In this section we present the algorithm for the case when the automaton does not use any variables and when infinity norm (i.e., F_∞) distance function is used .

4 Algorithms for Infinite norm distance

4.1 The basic algorithm using infinity norm distance function

In this subsection, we present the algorithm for the nearest neighbor query for infinite norm distance. It can be easily modified to answer a range query. Recall that in the nearest neighbor query we have to output k subsequences that are closest to the automaton. We present the algorithm for the case when $k = 1$ and describe

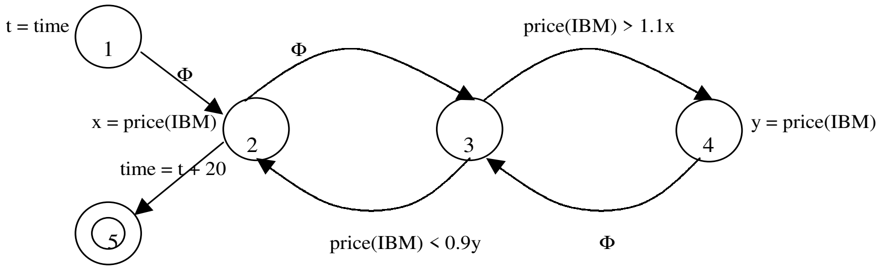

Figure 4:

how it can be extended for an arbitrary value of k. First, we need the following definitions and lemma. Let $d = (d_1, ..., d_n)$ be the database sequence. For any automaton state q and integer i $(1 \leq i \leq n)$, let $val(q,i)$ denote the value $\min\{dist(\sigma, T(q))\ : \sigma$ is a prefix of `d(i)`$\}$; that is, $val(q,i)$ is the shortest distance, from $T(q)$, of any subsequence of d beginning at d_i (including the empty subsequence); recall that `d(i)` is the suffix of d from d_i and $T(q)$ is the set of strings accepted by $\mathcal{A}$ starting from the state q. The algorithm computes the distance of d from $\mathcal{A}$ using the observation that this value is $\min\{val(q,i)\ : q \in I\ and\ 1 \leq i \leq n\}$, i.e., the minimum of $val(q,i)$ where the minimum is taken over all initial states q and over all i, $1 \leq i \leq n$. The following lemma shows how $val(q,i)$ can be computed from the set of values of the form $val(q', i+1)$, where q' is any node to which there is a transition from q.

LEMMA 3.1: Let q be any state in Q and i be an integer such that $1 \leq i \leq n$. Further, let $q \rightarrow_{a_1} q_1, ..., q \rightarrow_{a_m} q_m$ be all the transitions in δ from state q. Then the following properties hold.

1. For $q \in Q - F$ and $i < n$, $val(q,i) = \min\{x_1, ..., x_m\}$ where $x_j = \max\{dist(d_i, a_j), val(q_j, i+1)\}$ for $j = 1, ..., m$.
2. For $q \in Q - F$ and $i = n$, if there is at least one j such that $q_j \in F$ then $val(q,i)$ =$\min\{dist(d_n, a_j)\ : q_j \in F\}$, otherwise $val(q,i) = 1$.
3. For $q \in F$, $val(q,i) = 0$.

□

Proof: The proof of the part (3) of the lemma follows from the fact that the empty string is in $T(q)$ when $q \in F$ and hence the distance of empty prefix of `d(i)` from $T(q)$ is 0. Part (2) follows trivially from the definition of $val(q,i)$. Below we present the proof of part (1) of the lemma. Let $a_1, ..., a_m$ and $q_1, q_2, ..., q_m$ be as given in the statement of the lemma. Then $T(q) = a_1T(q_1) \cup a_2T(q_2) \cup ... \cup a_mT(q_m)$. Let $\alpha = (d_i, d_{i+1}, ..., d_{i+p})$ be any prefix of `d(i)` and β denote the sequence $(d_{i+1}, ..., d_{i+p})$. Thus, $\alpha = d_i\beta$. Clearly, $dist(\alpha, T(q)) = \min\{dist(\alpha, a_1T(q_1)), ..., dist(\alpha, a_mT(q_m))\}$. It is easy to see that $dist(\alpha, a_jT(q_j)) = \max\{dist(d_i, a_j),\ dist(\beta, T(q_j))\}$, for $1 \leq j \leq m$. From the above two equalities, it can be shown that
(*) $dist(\alpha, T(q)) = \min\{\max\{dist(d_i, a_j),\ dist(\beta, T(q_j))\}\ 1 \leq j \leq p\}$.
From the definition, we have $val(q,i) = \min\{dist(\alpha, T(q))\ : \alpha$ is a prefix of `d(i)`$\}$. Now using the property (*), it can be shown that $val(q,i) = \min\{x_1, ..., x_m\}$ where $x_1, ..., x_m$ are as given in the statement of the lemma. □

Using lemma 3.1, it is straightforward to present an algorithm that computes the distance of the subsequence of d (as well as its position) that has the lowest distance with respect to $\mathcal{A}$. To do this, we simply compute the values in $\{val(q,i) : q \in Q\}$ for decreasing values of i. For $i = n$, the values are computed by using parts 2 and 3 of lemma 3.1. Parts 1 of the lemma is used to compute the values for i from the values for $i+1$. The required minimum distance is computed to be the minimum of the values in $\{val(q,i) : 1 \leq i \leq n \text{ and } q \in I\}$. The subsequence having this minimum value is computed by saving the value of i that gives the minimum distance value and also the length of the subsequence. A formal description of the algorithm (called algorithm INF-NORM) is given below.

Formal Description of Algorithm INF-NORM

Let $d = (d_1, ..., d_n)$ be the database sequence and $A = (Q, \Sigma, \delta, I, F)$ be the automaton. We assume that, for each i and each $a \in \Sigma$, we can get the value of $dist(d_i, a)$; for the wild card symbol ϕ, $dist(d_i, \phi)$ is taken to be 0. The algorithm maintains an array *old*. This array is indexed by the states of the automaton. For each state q, each of the entries $old[q]$ have two values *length* and *val*. After processing the i^{th} database element, $old[q].length$ denotes the length of the prefix of `d(i)` that has the shortest distance to $T(q)$ among all prefixes of `d(i)`; $old[q].val$ denotes the distance of that prefix from $T(q)$. The algorithm also maintains a global structure *global*; this structure has three items *beg*, *end* and *val*. At anytime *global.beg* and *global.end* define an interval denoting a subsequence of the database sequence seen thus far that has the shortest distance to $L(\mathcal{A})$. For each final state q, $old[q].length$ and $old[q].val$ are both set to 0 and are left unchanged. This indicates that the empty prefix of `d(i)` (which is of length 0) has the shortest distance (which is 0) from $T(q)$ (this is because the empty string is in $T(q)$). For each non-final state q, $old[q].length$ and $old[q].val$ are initialized to $0, 1$ respectively. The algorithm updates the array *old* and the structure *global* using the procedure *Update* which takes *old*, *global* and i as parameters. The procedure *Update* maintains a local array *new* which holds the temporary values; this array is parallel to *old*.

Algorithm INF-NORM:

1 For each $q \in F$, $old[q].length := 0$, $old[q].val := 0$;
2 For each $q \notin F$, $old[q].length := 0$, $old[q].val := 1$;
3 $global.val := 1$;
4 For $i = n$ to 1
4.1 Update(old, global, i);

Procedure Update(old, global, i)
5 For each state $q \in Q - F$
6 $new[q].val := 1$;
7 For each transition $q \rightarrow_a q' \in \delta$
8 $x := \max(old[q'].val,\ dist(d_i, a))$;
9 If $x \leq new[q].val$ then
10 $new[q].val := x$, $new[q].length := 1 + old[q'].length$;
11 For each $q \in Q - F$
12 $old[q] := new[q]$;
13 For each initial state q (i.e. $q \in I$)
14 If $old[q].val \leq global.val$ then
15 $global.val := old[q].val$,
16 $global.beg := i$, $global.end := i + old[q].length$;
End procedure

It can be shown that after each iteration of the outer for loop at statement 4, $old[q].val = val(q,i)$. First, observe that, for $q \in F$, this property trivially holds as $old[q].val$ is initially set to zero and is never updated. For $q \in Q - F$, the above property is easily proven by downward induction on i. For $i = n$, this follows from part (2) of lemma 3.1 and the way the values are initialized and updated during the first iteration. By induction, from part (1) of lemma 3.1, we see that the property holds after each subsequent iteration because of the way the the values are updated at statement 7 of the *Update* procedure. Finally, it is trivial to see that after termination *global.val* contains $dist(d, \mathcal{A})$. By similar argument it is easy to see that *global.beg* and *global.end* define the subsequence of d that has the minimum distance to $\mathcal{A}$.

Let $|\mathcal{A}|$ denote the size of the automaton $\mathcal{A}$ which is defined as the sum of the number of states, number of transitions and the number elements in the input alphabet of $\mathcal{A}$. The main complexity of the algorithm is dictated by the number of times statements 8-10 (of the *Update* procedure) are executed which is bounded by $n|\mathcal{A}|$. Thus the complexity of the algorithm is $O(n|\mathcal{A}|)$.

Extensions to Algorithm INF-NORM

Algorithm INF-NORM can be generalized to output a set of k subsequences of d having the lowest distances with respect to $\mathcal{A}$ for any $k > 1$ as follows. The data structures $old[q], x$ and $global$ are all lists containing at most k entries and so is the data structure $new[q]$ of the *Update* procedure. In case of $old[q], new[q], x$, each entry of these lists has a length and a value field. Each entry of the $global$ list has three fields— beginning, ending and value. Initially, $old[q]$ has only one entry initialized as before (note that this initialization is different for final and non-final states) and $new[q]$ is empty.

The *Update* procedure is modified as follows. Statement 5 is executed for every state q including final states. At statement 6, $new[q]$ is set to the empty list. At statement 8, the list x is created by inserting one entry corresponding each entry e in $q'.old$; the value field of this entry is set as before (i.e. to the maximum of the value field of e and $dist(d_i, a)$); the length field is set to be one more than the length field of e. At statement 9, the list x is merged with $new[q]$ and the merged sequence of entries is stored in $new[q]$; only k entries having the lowest value fields are retained in $new[q]$; for a final state it is always ensured that there is an entry whose value and length fields are both zero. Statement 11 is executed for all states including the final states. At statement 13, the lists of all initial states are merged with the $global$ list and only those k items having the lowest value fields are retained in $global$; the beginning and end fields of the entries in $global$ are appropriately set.

It is fairly straightforward to show that the modified algorithm outputs k subsequences having the lowest distances with respect to $\mathcal{A}$. All the lists are maintained sorted on the value fields. Thus all the merge operations can be done in time linear in k. The complexity of the modified algorith can be shown to be $O(nk|A|)$.

4.2 Employing Indices for fast retrieval

In this subsection, we show how we can employ indices (i.e. inverted lists) for faster query processing. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be the automaton and $d = (d_1, ..., d_n)$ be the database sequence. Recall that each element of Σ denotes an atomic query. Let $\Sigma = \{a_1, ..., a_m\}$. We assume that there is an underlying query processing system which given an atomic query outputs a list of entries containing the indices of all the database states in d that satisfy the atomic query with some non-zero similarity value. More specifically, we assume that, for each $i = 1, ...m$, we can retrieve a list L_i of entries of the form (I, val) where I is an interval of the form `[u,v]` such that $1 \le u \le v \le n$ and and $0 \le val < 1$. The entry $(\texttt{[u,v]}, val)$ on the list L_i denotes that the the distance, with respect to a_i, of all database states whose indices fall with in the range `[u,v]` is val; that is, for all j such that $u \le j \le v$, $dist(d_j, a_i) = val$.

Notice that the values given in the list are distances and not similarity values; since $dist(d_j, a_i) = 1 - sim(d_j, a_i)$ and each entry on L_i corresponds to database states whose similarity values with respect to a_i is non-zero, it follows that the distance values in these lists are less than 1. It is also to be observed that each entry in L_i contains an interval; if the query processing system only returns pairs of indices and distances, we can convert such a list in to the above form by merging consecutive indices having the same distance value in to a single interval; if these intervals are very long compared to the number of states of the automaton $\mathcal{A}$, then by considering all the points in an interval together (rather than independently) we can speed up the similarity computation as we see later. It is to be noted that the list corresponding to the wild card query has a single entry whose interval is `[1,n]` and whose distance value is zero.

If many database states do not satisfy a_i at all, then the length of L_i is much smaller than n. Clearly the intervals appearing in L_i are all disjoint. We assume that L_i is sorted in increasing values of the left index of each interval. Note that successive intervals in L_i need not be consecutive; that is, if `([u,v],val)` and `([u',v'],val')` are successive entries then it is possible that $v + 1 < u'$; such gaps indicate that for all j, $v < j < u'$, $dist(d_j, a_i) = 1$; there can be such gaps at the beginning as well as the ending of the list. We fill up such gaps by introducing an additional entry for each such gap whose interval is the gap and whose distance value is 1; for example, in the above case, we introduce the entry `([v+1,u'],1)`. Let L'_i be the resulting list. It is easy to see that the length of L'_i is at most twice the length of L_i plus 1.

We merge all the lists $L'_1, ..., L'_m$ in to a single list L satisfying the following properties. Each entry on L is a pair of the form $(\texttt{[u,v]}, f)$ where `[u,v]` is an interval and f is a function from Σ to distances satisfying the following property: for each atomic query a and for each j, $u \le j \le v$, $f(a) = dist(d_j, a)$; that is, $f(a)$ is the distance of the database state d_j with respect to the atomic query a. The list is in increasing order of the left index of each interval. All the intervals in L are successive and they cover the entire interval [1,n], i.e., the entire database sequence. Further more, the length of L is at most one plus twice the sum of the lengths of $L_1, ..., L_m$. It is easy to see that L can be obtained from $L_1, ..., L_m$ in time proportional to the sum of the lengths of these lists.

Now we present an algorithm that employs L and outputs k subsequences of d having the lowest distances with respect to $\mathcal{A}$. The complexity of the algorithm will be linear in the length of L (and hence linear in the sum of the lengths of the lists L_i)and also linear in $\log n$. Thus, when the sum of the lengths of the lists is much smaller than n then this algorithm will run much faster than the previous one.

Let $e_1, ..., e_r$ be the entries in L in that order and for each j, $1 \leq j \leq r$, let $e_j = ([u_j, v_j], f_j)$. For each $j = 1, ...r$, let $IDIST(j)$ be the set of values given by $\{val(q, i) : u_j \leq i \leq v_j \text{ and } q \in I\}$; that is, $IDIST(j)$ is the set of distances (with respect to $\mathcal{A}$) of all those subsequences, of the database sequence d, starting at some point in the interval $[u_j, v_j]$. We will compute the minimum of all values in $IDIST(j)$ over all j; we will also find one subsequence having this distance. We achieve this, by finding the values in $IDIST(j)$ and the values in $\{val(q, u_j) : q \in Q\}$, in decreasing order of j. Let $p = |Q|$, i.e. p is the number of states of $\mathcal{A}$. The following lemma shows that the values in $IDIST(j)$ can be computed by considering only the last p points in the interval $[u_j, v_j]$.

LEMMA 3.2: For every $x \in IDIST(j)$, there exists a value of i such that $\max\{u_j, (v_j - p)\} \leq i \leq v_j$ such that $x = val(q, i)$ for some initial state q.

Proof: Let i' be some value in the interval $[u_j, v_j]$ such that $val(q, i') = x$ for some initial state q. This means that there exists a string α and a prefix γ of d(i) such that they are of the same length (i.e. $|\alpha| = |\gamma|$) and $dist(\gamma, \alpha) = x$, and α takes the automaton from q to a final state. Now we have two cases. In the first case, we assume that $|\alpha| \leq (v_j - i' + 1)$. If $|\alpha| \leq p$ then $i = i'$ itself satisfies the condition of the lemma. Otherwise, we can get a string α' of length less than p (using the pumping lemma argument) such that α' takes q to a final state and every symbol appearing in α' also appears in α. Since the similarity value of an atomic query is same at all points of the interval $[u_j, v_j]$, it follows that the suffix of the substring $d[u_j, v_j]$ of length $|\alpha'|$ has also distance x from $L(\mathcal{A})$. In the second case, $|\alpha| > (v_j - i' + 1)$ and the proof is similar to the first case and is left to the reader. □

Assume that we have already found the values of $val(q, u_{j+1})$ for each $q \in Q$. These values are enough to determine the values in $IDIST(j)$. Further more, from lemma 3.2, it should be clear that the minimum of the values in $IDIST(j)$ can be found by only considering the last p points in the interval $[u_j, v_j]$. This can be done by executing the *Update* procedure, of algorithm INF-NORM, exactly p number of times after initializing $old[q].val$ to $val(q, u_{j+1})$ for each $q \in Q - F$ and by setting $old[q].val = 0$ for each $q \in F$.

Now, we discuss how to compute the values in $\{val(q, u_j) : q \in Q - F\}$ using the values in $\{val(q, u_{j+1}) : q \in Q - F\}$. Consider any state $q \in Q - F$. Recall that $val(q, u_j)$ is the minimum of the distances from $T(q)$ of all subsequences of d starting at d_{u_j}, i.e., all prefixes of $d(u_j)$; also recall that $T(q)$ is the language accepted by $\mathcal{A}$ starting from state q. If $(v_j - u_j) \leq p$ then we simply execute the *Update* procedure, of algorithm INF-NORM, $(v_j - u_j + 1)$ times. Now assume that $(v_j - u_j) > p$. Using the same arguments as those of lemma 3.2, the following lemma can be proven.

LEMMA 3.3: Assume $(v_j - u_j) > p$. For every subsequence α of d starting from d_{u_j} whose length is less than or equal to $v_j - u_j + 1$, there exists a subsequence of length less than p starting from the same position whose distance from $T(q)$ is same as that of α from $T(q)$. □

In view of the above lemma, we take the following approach for computing $val(q, u_j)$. We first compute the minimum distance (from $T(q)$) of all subsequences of d starting from d_{u_j} whose length is less than p. Let this value be $x[q]$. Then we compute the minimum distances from $T(q)$ of all subsequences starting from d_{u_j} of length greater than $v_j - u_j + 1$. Let $y[q]$ be this value. Then we take $val(q, u_j)$ to be the minimum of $x[q]$ and $y[q]$.

Now, we show how to compute $y[q]$ in time proportional to $\log(v_j - u_j + 1)$. We assume that the states of the automaton are integers between 1 and p. Let Z be the $p \times p$ matrix defined as follows. The value of $z_{q,q'}$, which is the entry of Z in the row q and the column q', is given by $z_{q,q'} = \min\{f_j(a) : q \rightarrow_a q' \in \delta\}$; if the automaton $\mathcal{A}$ does not have any transitions from q to q' then we define $z_{q,q'} = 1$. Now let VAL_{j+1} denote the column vector of length p whose q^{th} component is $val(q, u_{j+1})$. For any two matrices U, V of appropriate dimensions, let UV denote their usual product with the change that the scalar multiplication and addition operations are replaced by the max and min operations; that is, for any two real numbers w, w', ww' is taken to be $\max\{w, w'\}$ and $w + w'$ is taken to be $\min\{w, w'\}$. The definition of the product of a matrix and a column vector is defined similarly. Now let $Z^{(v_j - u_j + 1)}$ denote the result obtained by multiplying Z with itself $(v_j - u_j + 1)$ times. The following lemma shows that the q^{th} entry of the column vector $Z^{(v_j - u_j + 1)} VAL_{j+1}$ is the value of $y[q]$.

LEMMA 3.4: Let Z be the matrix and VAL_{j+1} be the column vector as defined above. Then the value of $y[q]$ is given by the q^{th} entry of the column vector $Z^{(v_j - u_j + 1)} VAL_{j+1}$ where scalar multiplication and addition operations are taken to be the max and min operations, respectively.

Proof: Let G be a complete labeled graph defined as follows. The nodes of G are the states of $\mathcal{A}$, i.e. integers between 1 and p. For each q', q'', the edge from q' to q'' is labeled with $z_{q',q''}$. For any path p in G, let $weight(p)$ denote the maximum of all the labels of edges in p. For any pair of nodes q', q'', let $W(q', q'')$ denote the minimum of $weight(p)$, over all paths p of length $v_j - u_j + 1$ from node q' to the node q''. It can be shown that $y[q]$ equals the minimum, over all nodes q', of the value $\max\{W(q, q'), val(q', u_{j+1})\}$. The later value is the q^{th} entry of the column vector $Z^{(v_j - u_j + 1)} VAL_{j+1}$. □

Using the techniques given in [AHU], $Z^{(v_j - u_j + 1)}$ can be computed using $O(log(v_j - u_j + 1))$ multiplications. The complete algorithm, called algorithm INF-NORM-INDX is given below.

Description of Algorithm INF-NORM-INDX and its extensions

Algorithm INF-NORM-INDX uses the same data structures as algorithm INF-NORM. It also uses the following additional temporary data structures: arrays $temp, X, Y$ which are similar to *old*; *tglobal* which is similar to *global*; matrices Z, W which are two dimensional $p \times p$ array. It also uses the *Update* procedure of algorithm INF-NORM.

Algorithm INF-NORM-INDX:

1 $Initialize(old, global)$;
2 For $j = r$ to 1
3   If $(v_j - u_j + 1) < p$
4   For $i = v_j$ up to u_j
5             $Update(old, global, i)$
6   Else
7         $temp := old$;
8   For $i = v_j$ up to $v_j - p + 1$
9             $Update(temp, global, i)$;
10         $Initialize(X, tglobal)$;
11   For $i = u_j + p - 1$ up to u_j
12             $Update(X, tglobal, i)$;
13   For each $q \in Q$ and for each $q' \in Q$
               $Z[q, q'] = \min\{f_j(a) : q \to_a q' \in \delta\}$
14   Compute matrix $W := Z^{v_j - u_j + 1}$
15   Intialize(Y)
16   For each $q \in Q$ and each $q' \in Q$
17   If $Y[q].val > \max\{Z[q, q'], old[q'].val\}$ then
18                 $Y[q].val := \max\{Z[q, q'], old[q'].val\}$
19                 $Y[q].length := old[q'].length + v_j - u_j + 1$
20   For each $q \in Q - F$
21             $old[q].val := \min\{X[q].val,\ Y[q].val\}$
21   If $X[q].val > Y[q].val$
22                 $old[q].length := Y[q].length$
23   Else
24                 $old[q].length := X[q].length$

Algorithm INF-NORM-INDX works as described in subsection 4.2. In statement 1, the array *old* and the structure *global* are initialized as in algorithm INF-NORM (lines 1-3). In statement 2, each entry of the list L is examined (recall that r is the length of L). If the length of the interval of the entry is less than p then the values of *old* and *global* are updated as in algorithm INF-NORM using the *Update* procedure; otherwise statements 7 thru 24 are executed (Note that inside the *Update* procedure, $dist(d_i, a)$ is taken to be $f_j(a)$). In statements 8 and 9 the *global.val* is set to the minimum of all the values in $IDIST(j)$ and the previous value of *global.val*. In statements 11 and 12 the value of $x[q]$ is computed and is assigned to $X[q].val$. In statement 14, the value of $Z^{v_j - u_j + 1}$ is computed. In statements 15 thru 19, the value of $y[q]$ is computed and is assigned to $Y[q].val$. (Note that we do not need to compute the vector VAL since $VAL[q] = old[q].val$.) In statements 20 thru 24, $val(q, u_j)$ is computed and is assigned to $old[q].val$. Note that the lengths fields in the different data structures are appropriately computed. After the termination of the above algorithm, *global.val* contains the value of the minimum distance of all subsequences of d (with respect to $\mathcal{A}$), and *global.beg*, *global.end* contain the end points of the a subsequence having the minimum distance value.

The complexity of a single iteration of the for loop at statement 2 is bounded as follows. Statements 3 thru 5 take time $O(p|A|)$. Recall that p is the number of states of $\mathcal{A}$ while $|A|$ is the size of $\mathcal{A}$ including its transitions; clearly $|A| > p$. Statements 8 thru 12 also take time $O(p|A|)$. Statements 13 and statements 15 thru 24, takes time $O(p^2)$. Statement 14 requires the computation of $Z^{v_j - u_j + 1}$. This can be computed in time $O(p^3 \log(v_j - u_j + 1))$ by a technique called successive squaring (see [AHU]); since $v_j - u_j + 1 \leq n$, this complexity is $O(p^3 \log n)$. This dominates the complexities of the other statements. Thus the complexity of the above algorithm, over all iterations of the for loop at statement 2, is $O(rp^3 \log n)$.

The above algorith can be extended to compute the k subsequences of d having the smallest distances to $\mathcal{A}$. As in the extension of the algorithm INF-NORM, we extend all appropriate data structures to be lists of length k. These include the elements of the arrays $old, new, X, Y, temp$, the data structures *global*, etc. They are updated as in the previous case. The elements of the matrix Z also are lists of length k. The value of $Z[q, q']$ is taken to be a list of the k elements in $\{f_j(a) \ : q \rightarrow_a q' \in \delta\}$ having the lowest values. In defining the matrix multiplication, scalar multiplication of two matrix elements ρ, ρ' is defined to be the lowest k values in the set $\{\max t, t' \ t \in \rho, t' \in \rho'\}$; the addition of two such elements is defined to be the lowest k values in their union. It is not difficult to see that these operations can be carried out in time $O(k)$. The details are left out to the reader. The complexity of the resulting algorithm is $O(krp^3 \log n)$.

Algorithm INF-NORM-RESTR-AUT for Cycle-restricted Automata.

In this subsection we present algorithm INF-NORM-RESTR-AUT which is a modification of algorithm INF-NORM for restricted automata.

Roughly speaking, algorithm INF-NORM-RESTR-AUT is a modified version of algorithm INF-NORM. However, its input is like the input of algorithm INF-NORM-INDX. That is, its input is a list L of entries where each entry is a pair of the form $([\mathtt{u}, \mathtt{v}], f)$ where $[\mathtt{u}, \mathtt{v}]$ is an interval and f is a function that assigns a distance with each input symbol of the automaton; for each input symbol a and for each i such $u \leq i \leq v$, $f(a) = dist(d_i, a)$.

Let $e_1, ..., e_r$ be the entries on the list L. As before we assume that the list is sorted in increasing values of the starting points of the intervals. For each $j = 1, ..., r$, let $e_j = ([\mathtt{u_j}, \mathtt{v_j}], f_j)$. We say that e_j is a gap entry, if $f_j(a) = 1$ for all input symbols a; this means all the database states in the interval satisfy all the atomic queries with zero similarity (i.e. they do not at all satisfy them). Essentially, algorithm INF-NORM-RESTR-AUT processes long gap entries more efficiently than algorithm INF-NORM. Let p be the number of states of the automaton. A state q of the automaton is called a ϕ-loop-state if $q \rightarrow_\phi q$ is a transition of the automaton, i.e. there is a ϕ-cycle of length 1 from it.

Algorithm INF-NORM-RESTR-AUT uses the same data structures as algorithm INF-NORM, i.e., arrays old, new, etc.; it initializes them similarly. For each $j = r$ up to 1, it processes e_j as follows. If e_j is not a gap entry or $v_j - u_j \leq p$ then it executes the procedure $Update(old, global, i)$ for decreasing values of i from v_j up to u_j; i.e. all database states d_i such that $u_j \leq i \leq v_j$ are processed as algorithm INF-NORM. Otherwise, $v_j - u_j > p$, and for each automaton state q, $old[q].val$ and $old[q].length$ are updated as follows. Let $Reach(q)$ be the set of all ϕ-loop-states that are reachable from q using ϕ-paths. (Note q is a member of $Reach(q)$ iff q is a ϕ-loop-state). If $Reach(q)$ is non-empty then $old[q].val$ is set to the minimum of $\{old[q'].val \ q' \in Reach(q)\}$ and $old[q].length$ is incremented by $v_j - u_j + 1$. Otherwise, $old[q].val$ is set to 1 and $old[q].length$ is set to 0.

It can be shown that algorithm INF-NORM-RESTR-AUT works correctly.

4.3 Processing range queries

Processing range queries is simpler. Here we need to retrieve all subsequences of d whose distances from $\mathcal{A}$ is less than or equal to ϵ. For each atomic query $a \in \Sigma$, we retrieve the list of all pairs of the form $(i, dist(a, d_i))$ such that $dist(a, d_i) \leq \epsilon$.

Let $L_1, L_2, ..., L_m$ be all such lists. In each list we merge all consecutive elements (i.e., elements whose index values are in increments of one) having the same distance value in to a single element to obtain lists where each entry is of the form $([u, v], x)$ where $[u, v]$ is an interval and x is the distance value. We further process and merge these lists in to a single list L, as in the previous subsection. On this list we execute the modified algorithm INF-NORM-INDX where we retrieve k subsequences of having the lowest distance; in this algorithm we take k to be ∞, i.e. unbounded. Note that in the worst case the number of retrieved subsequences can be as much as $O(n^2)$ where n is the length of the database sequence. Thus the worst case complexity of the algorithm can be $O(n^3|A|)$. But average complexity may be much lower.

5 Algorithms for Average Block Distance

In this section, we present the algorithm for the average block distance function (i.e. F_1) for the simple automata, i.e. when no variables are used. As before, we present the algorithm for the case when $k = 1$ and show how it can be extended for arbitrary values of k. First, we need some definitions. Let $d = (d_1, ..., d_n)$ be the database sequence and $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be an automaton.

For any subsequence $\sigma = (d_i, ..., d_{i+l-1})$ of d and any string $a = (a_1, ..., a_l) \in \Sigma^*$ of the same length, let $bd(\sigma, a)$ be the sum $\Sigma_{j=0,...,l-1} dist(d_{i+j}, a_{j+1})$; it denotes the block distance between σ and a. For any string $a \in \Sigma^*$, let *count*(a) denote the number of non-wild card symbols in a. We call *count*(a) as the pseudo length of a. For any automaton state q and integers i, r such that $1 \leq i \leq n$ and $r \leq n - i + 1$, let $val(q, i, r) = \min\{bd(\sigma, a) : \sigma$ is a subsequence of d starting from d_i and a is any string in $T(q)$ which is of the same length as σ whose pseudo length is r $\}$. If there are no strings of pseudo length r in $T(q)$ then we define $val(q, i, r)$ to be ∞. Now, it should be easy to see that the minimum of the distances from $\mathcal{A}$, of all subsequences of d starting at d_i, equals $\min\{\frac{val(q,i,r)}{r} : q \in I, 1 \leq r \leq n - i + 1\}$. We take the minimum of these minima for different values of $i = 1, ..., n$ and this is the required value. The following lemma is easily seen.

LEMMA 4.1: For each q, i, r such that $q \in Q$, $1 \leq i \leq n$ and $1 \leq r \leq n - i + 1$ the following properties hold.

1. If there is a transition from q to some final state on a non-wild card symbol then $val(q, i, 1) = \min\{dist(d_i, b) : q \rightarrow_b q', \, q' \in F, \, b \neq \phi\}$; otherwise, $val(q, i, 1) = \infty$.
2. If $r > 1$ then $val(q, i, r) = \min\{x, y\}$ where $x = \min\{dist(d_i, b) + val(q', i+1, r-1) : q \rightarrow_b q' \in \delta, b \neq \phi\}$ and $y = \min\{val(q', i+1, r) : q \rightarrow_\phi q' \in \delta\}$.

The proof of the above lemma easily follows from the definition of $val(q, i, r)$ and is left to the reader. The lemma gives a way of computing $val(q, i, r)$. Now we present an algorithm that outputs the minimum of all the distances of the subsequences of d from $\mathcal{A}$ and a subsequences of d having that distance. As before, we assume that for each atomic query $b \in \Sigma$ such that $b \neq \phi$, we have a list of entries of the form $(i, dist(d_i, b))$ where $1 \leq i \leq n$ and $0 \leq dist(d_i, b) < 1$. Note that the distance in every entry is less than 1. Let $a_1, ..., a_m$ be all the non-wild card atomic queries in Σ. Let L_i be the list corresponding to a_i, for $i = 1, ..., m$. We merge all these lists in to another list L which is a sequence of entries of the form (i, C) where $1 \leq i \neq n$ and C specifies the distance of atomic queries at position i; that is, C is a set of elements of the form (j, v) such that the element (i, v) is an entry on the list L_j, i.e. $v = dist(d_i, a_j)$. Let r be the length of L. Clearly r is bounded by the sum of the lengths of $L_1, ..., L_m$. We assume that the elements in L are sorted in increasing values of the index value. Let $L = e_1, e_2, ..., e_r$ where $e_i = (u_i, C_i)$

A formal description of the algorithm, called algorithm AVG-DIST, is given below. (Note that, unlike the input list for algorithm INF-NORM-INDX, the first entry in each element in the input list L is a single index and not an interval.) This algorithm, like algorithms INF-NORM and INF-NORM-INDX, also maintains the following data variables: *global* having three entries *beg*, *end*, *val*. We have an array *old* indexed by the states of the automaton. For each $q \in Q$, *old*$[q]$ is a list of entries where each entry contains three items—

length, *sum*, *count*. After processing the entry e_i in the list L, $old[q]$ satisfies condition X given below:

Condition X: For each $r = 1, ..., n-u_i+1$ such that $val(q, u_i, r) < \infty$, there is an entry in $old[q]$ whose *count* and *sum* fields have values r and $val(q, u_i, r)$, respectively; in this case, the *length* field of the corresponding entry gives the length of a subsequence σ of d starting at d_{u_i} such that $val(q, u_i, r) = bd(\sigma, a)$ where a is a string in $T(q)$ whose pseudo length is r. For each r, there is at most one entry in $old[q]$ whose *count* field equals r.

The algorithm maintains additional variable *last* in which it stores the index field of the previous entry processed. It also maintains a set $S[q]$ and a list $new[q]$ for each q. These are used as temporary variables.

Algorithm AVG-DIST:

1 For each $q \in F$, set $old[q]$ to the single entry all of whose fields are 0;
2 For each $q \notin F$, set $old[q]$ to the empty list;
3 $global.val := 1$; $last := n$;
4 For $i = r$ to 1
5   If $last > u_i + 1$
6   For each $q \in Q$
7             $S[q] := \{q' : q$ can reach q' in $\mathcal{A}$ using a string of length $last - u_i - 1$ containing only wild card symbols$\}$
8   For each $q \in Q$
            $new[q] := MergeUpdate(S[q], last - u_i - 1)$
9   For each $q \in Q$ $old[q] := new[q]$
10   For each state $q \in Q$
11   Set $new[q]$ to the empty list
12   For each transition $q \rightarrow_a q' \in \delta$
13   If a is not the wild card symbol then
14                 $x := \{(length + 1, sum + dist(d_{u_i}, a), count + 1) :$ the element $(length, sum, count)$ is in $old[q']\}$
15   If a is the wild card symbol then
16                 $x := \{(length + 1, sum, count) :$ the element $(length, sum, count)$ is in $old[q']\}$
17             $Merge(new[q], x)$
18   If $q \in F$ then add the entry with all zeros to $new[q]$
19   For each $q \in Q$
20         $old[q] := new[q]$;
21   For each initial state q (i.e. $q \in I$)
22         $Updateglobal(q)$
23     $last := u_i$

Algorithm AVG-DIST given above works as follows. In statements 1 thru 3, the different data structures are initialized. For each non-final state q, $old[q]$ is set to the empty list, and for each final state it is set to contain the single entry all of whose fields are set to 0. In statement 4, the entries in L are examined starting with the last entry. After each iteration of this loop, $old[q]$ (for $q \in Q$) and *global* are updated. In statements 5 thru 9, the following operation is performed. If $last > u_i + 1$ then this denotes that for all database items d_j such that $u_i + 1 \leq j < last$ and for all $a \in \Sigma$ such that a is not a wild card symbol, $dist(d_j, a) = 1$. In this case, for each q, the following is done. We first find all q' that can be reached from q (in the automaton $\mathcal{A}$) using a string of wild card symbols of length $last - u_i - 1$; all such q' are stored in $S[q]$. The lists associated with all such q' (i.e. $old[q']$ lists) are merged and updated in to a single list which is stored in $new[q]$ and later assigned to $old[q]$. First the lists are merged; if multiple entries have the same value for the *count* field then, among these the one with least *sum* value is retained and all other are discarded. In the resulting list, the *length* fields of all the entries is incremented by $last - u_i - 1$. All this operation is carried out by the $Mergeupdate$ procedure. If the set $S[q]$ is empty, i.e., there is no q' satisfying the above property, then $Mergeupdate$ acts as follows: if q is a non-final state then it returns an empty list; otherwise it returns the list containing the single entry with zero values, i.e. all its fields are zero.

In statements 11 thru 17, the entry e_i is processed and $old[q]$ is updated. In statement 12, each transition from q is considered. If the transition is to state q' then a temporary list x is created and is later merged with $new[q]$. The $Merge$ operation in statement 17 acts as follows. The list x is merged with $new[q]$; if $new[q]$ contains multiple entries with the same $count$ fields then an entry with the lowest sum field among them is retained and others are discarded. The procedure $Updateglobal(q)$ acts as follows. For each entry in $old[q]$ whose count field is non-zero, it computes the average value of the entry (which is $\frac{sum}{count}$) and if it is less than $global.val$ then it is assigned to $global.val$; in this case, $global.beg$ and $global.end$ are set to u_i and $u_i+length-1$, respectively.

Using lemma 4.1, by induction on i, it is not difficult to see that after processing entry e_i of L, the list $old[q]$ satisfies condition X given above; further more, $global.val$ contains the minimum of the values $\{\frac{val(q,j,r)}{r} : r > 0, u_i \leq j \leq n\}$.

Now we analyze the worst case complexity of algorithm AVG-DIST. First we bound the length of the lists $out[q]$ for $q \in Q$. Observe that the $count$ fields are incremented in statement 14. They are initialized to zero, for final states, at the beginning and in statement 18. By induction on i, it is easy to show the maximum value of a $count$ field in any entry after the i^{th} iteration of the for loop at statement 4, is at most i. Hence the $count$ field is bounded by r, the length of L. All the lists are maintained in sorted order, sorted on $count$ fields. It is not difficult to see that merging the lists can be carried out in linear time. As a consequence, it is not difficult to see that statements 10 thru 23 contribute a complexity of $O(|A|r)$ in a single iteration of the for loop at statement 4.

In statement 7, we compute the sets $S[q]$ for each q. To compute these, we define a matrix Z, which is a $p \times p$ matrix (recall that p is the number of states of the automaton $\mathcal{A}$). For each $q, q' \in Q$, the entry $Z[q,q']$ is set to 1 if there is a transition in $\mathcal{A}$ from q to q' on the wild card symbol; otherwise $Z[q,q']$ is set to 0. Now consider the product Z^{last-u_i-1} in which scalar multiplication and addition are the regular operations on integers. As we have indicated in the previous subsection, this product can be computed in $\log(last - u_i - 1)$ matrix multiplications. It is not difficult to see that the entry in the q^{th} row and q'^{th} column of the product is non-zero iff q' can be reached from q using a string of wild card symbols of length $last - u_i - 1$; thus $S[q]$ is the set of all q' such that the q'^{th} entry in the q^{th} row of the above product is non-zero. Since each matrix multiplication has complexity $O(p^3)$, it follows that all these sets can be computed in time $O(p^3 \log(last-u_i-1))$; this complexity is $O(p^3 \log n)$ since $last - u_i - 1 \leq n$. Since each $S[q]$ has at most p elements, the $Mergeupdate$ operation can be implemented in time $O(p^2)$ since there are at most p such lists; statement 7 has complexity $O(p^3)$. Thus, statements 7 thru 9 contribute a complexity of $O(p^3 \log n)$

Over all the iterations of the for loop at statement 4, statements 5 thru 23 contribute a complexity of $O(r^2|A| + rp^3 \log n)$. Thus the over all complexity of the algorithm is $O(r^2|A| + rp^3 \log n)$. Thus we see that the dependence on n is through the $\log n$ term. If r is much smaller than n this algorithm will be practical.

The above algorithm can be modified to compute the k subsequences of d having the smallest distance to $\mathcal{A}$. Essentially, we will keep up to k entries in each $old[q]$ list having the same $count$ field. This requires modification of the merge routines at statements 8 and 17. Also, $Updateglobal$ will be modified to keep up to k values. The complexity of the resulting algorithm will be $O(kr^2|A| + krp^3 \log n)$.

For answering range queries, i.e. for retrieving all subsequences whose distance from $\mathcal{A}$ is less than or equal to ϵ where ϵ is a given value we do as follows. We run the above algorithm that retrieves k subsequences of d (having the lowest distance values) by taking k to be ∞ (i.e., unbounded).

Algorithm AVG-DIST-RESTR-AUT

Now, we show how algorithm AVG-DIST can be modified to get algorithm AVG-DIST-RESTR-AUT for cycle-restricted automata. It runs faster than algorithm AVG-DIST for these types of automata. Algorithm AVG-DIST-RESTR-AUT is identical to algorithm AVG-DIST excepting for the way in which the set of automaton states $S[q]$ at line 7 is computed. In algorithm AVG-DIST-RESTR-AUT, the set $S[q]$ is computed as follows. Let p be the number of automaton states. If $(last - u_i - 1) \leq p$ then $S[q]$ is assigned the set of states reachable from q using ϕ-paths of length $last - u_i - 1$. Otherwise, $S[q]$ is set to $Reach(q)$, where $Reach(q)$ is defined as above. It should be easy to see that $S[q]$ can be computed using an algorithm whose complexity is linear in the size of the automaton. It is not difficult to show that the modified algorithm works correctly.

6 Algorithms for the extended Automata

6.1 Extensions to the Algorithm INF-NORM

Now we describe the extensions to the algorithms for handling automata with variables. First observe that for atomic predicates with parameters, the distance of the atomic predicate with respect to a database state also depends on the values of the global variables appearing in it. For this reason, we represent $dist(d_i, a)$ as a table which has $k+1$ columns if there are k global variables appearing in it; the first k columns correspond to the k variables and the last column gives the distance value. We call this table as the distance table. Each row in the table gives the distance value for the given values of the global variables. Actually, we represent the values of global variables as ranges (i.e. intervals). For example, if the atomic predicate a has global variables $y_1, ..., y_k$ appearing in it and $(I_1, ..., I_k, v)$ is a row in the distance table then for each j, I_j is an interval denoting a range of values for variable y_j; further more, this row indicates that the distance of a with respect to the database state is v for all values of the variables $y_1, ..., y_k$ in the ranges given by $I_1, ..., I_k$, respectively. We assume that the ranges for each variable are fixed a-priori.

Let $A = (Q, \Sigma, \delta, I, F, \psi)$ be the given automaton and l be the total number of global variables. We modify algorithm INF-NORM, given in section 4, as follows. Recall that variable $old[q]$ has two values $old[q].val$ and $old[q].length$. After processing the database entry d_i, the value of $old[q].val$ equals the smallest distance with respect to $T(q)$ of any subsequence of d starting at d_i, and $old[q].length$ denotes the length of the corresponding subsequence. In our new setting, this value may depend on the values of the global variables. For this reason, we represent $old[q]$ also as a table with at most $l+2$ columns where the last two columns give the distance value and length respectively; the remaining columns represent global variables. In actual implementation, the number of columns may be less than $l+2$; the missing columns indicate that the distance value is independent of the values of those global variables whose columns are missing. However, for ease of exposition, we assume that there is one column for each global variable. As before, each row indicates the distance values and lengths for ranges of values to global variables specified by the entries in the variable columns. If the distance value is independent of a variable, then all the entries in that column have the *universal* interval as their value (the universal interval is the interval denoting the whole range of possible values for the variable). The variables $new[q]$ and x are also tables like $old[q]$.

Initially, $old[q]$ is initialized to contain a single row whose value column and length column are zero and whose other columns have the universal interval. At statement 6, $new[q]$ is initialized to contain a single row whose value column has 1 and whose variable columns have the universal interval.

At statement 8, the table for x is computed as follows. First, a *temp* table is created by merging $old[q']$ with the table $dist(d_i, a)$ as follows. We call this procedure as $Maxmerge$. First, pairs of rows r', r'', such that r' is from $old[q']$ and r'' is from the $dist(d_i, a)$, satisfying the following condition are identified: for every global variable y, if r'' has a y column then the intervals in r', r'' in the y column should have non-empty intersection. For each such pair, a row s, defined as follows, is generated for the table *temp*. For each variable y, the y column in s is the intersection of the y columns in r', r'' if there is a y column in both; otherwise the y column in s is taken from r'. The value column of s is the maximum of the value columns of r' and r''; the length column is taken from the same row as the value column is taken after incrementing it. If there are no assignments in the automaton state q then table for x is taken to be the table *temp*. Otherwise the table for x is calculated as follows. Let $y_1 := q_1, ..., y_t := q_t$ be all the assignments associated with the automaton state q (i.e. $\psi(q)$ consists of these assignments). The values $v_1, ..., v_t$ of the queries $q_1, ..., q_t$ in the the database state d_i are computed. Then a selection on the table *temp* is performed; those rows such that, for each $j = 1, ..., t$, the value v_j is in the range of values defined by the interval of the y_j column are retrieved; for these selected rows the columns for the $y_1, ..., y_t$ variables are changed to contain the universal interval (this change indicates that the value columns of the table are independent of the values of $y_1, ..., y_t$) ; the set of all such rows is taken to be the table x.

Statements 9 and 10 are replaced by a procedure that merges table x with table $new[q]$ as follows. We call this operation as $Minmerge$. First pairs of rows r', r'', such that r' is from table x and r'' is from $new[q]$ and such that they have non-empty intersection on the same variable columns, are identified. For each such pair, a row s is created whose variable columns are intervals obtained by intersecting the corresponding variable columns in r', r''; the $value$ column of s is the minimum of the $value$ columns of r', r''; the $length$ column is taken from the $length$ column of the row from which the value column is taken. After the merge operation,

$new[q]$ will contain the set of rows s created as above.

For every initial state q, it can be shown that the table $old[q]$ contains exactly one row; all the variable columns of this row have universal interval indicating that the *value* and *length* column of the are independent of the values of the global variables. Statements 13 thru 16 are modified to take $old[q].val, old[q].length$ to be *value* and *length* columns, respectively, of the single row in $old[q]$ table.

Notice that the operations $Maxmerge$ and $Minmerge$ given above can be defined in terms of the standard relational join, selection and projection operations. The overall complexity of the resulting algorithm is dependent on the complexity of $Maxmerge$ and $Minmerge$ operations; if these operations can be implemented in time T then the overall complexity of the modified algorithm is $O(n|A|T)$ where n is the length of the database sequence, $|A|$ is the automaton size. Note that the value T depends on the sizes of various relations which in turn depend on the ranges we have chosen for various global variable values. The above changes can be extended to retrieve k subsequences having the smallest distances to $\mathcal{A}$. The complexity gets multiplied by the factor k.

6.2 Modification to Algorithm AVG-DIST

The algorithm AVG-DIST is modified as follows. Recall that algorithm AVG-DIST uses the average block distance measure. It employs inverted lists (also called indices). It assumes that a list L of r entries is first generated. Now, we assume that L is a sequence $e_1, ..., e_r$ of entries where $e_i = (u_i, C_i)$ where u_i is an index in to the database sequence and C_i is pairs of the form (j, V) where V is the distance table for the atomic predicate a_j (recall that previously V is a single value;as explained at the beginning of the previous subsection, the table V gives the distance values of a_j with respect to d_{u_i} for different values of the global variables).

Now, the data structure $old[q]$ is a table having a column corresponding to each global variable; it also has a *count*, *sum* and *length* columns. At the end of an iteration of the for loop at statement 4, the table $old[q]$ signifies the following. Each row denotes ranges of values of global variables; the *sum* column gives the minimum block distance of a subsequence starting at d_{u_i} having *count* number of non-wild card symbols. For different ranges of values, the *count*, *sum* and *length* fields may be different. The variable $new[q]$ is also a table like $old[q]$.

At statement 1, $old[q]$ is assigned to have a single row, with all variable columns having universal intervals and other columns having zeros. At statement 2, $old[q]$ is assigned an empty table. At statement 8, $Mergeupdate$ is performed as follows. First, two tables $old[q'], old[q'']$ corresponding two states q', q'' in $S[q]$ are merged to obtain *temp* as follows. Pairs of rows r' in $old[q']$ and row r'' in $old[q'']$ such that the columns for the same variables in r', r'' have non-empty intersection and such that the *count* column have equal values are identified. A single row whose variable columns contain the intersection of the corresponding variables columns in r', r'' and whose *sum* column is the minimum of the two sums is created; the *length* is incremented by $last - u_i - 1$. The set of all such rows is placed in *temp* relation. Also, for each row r' in $old[q]$, additional rows that capture the non-overlapping portions (i.e. non-overlapping with any in $old[q''])$ of the variable columns are created; the *count*, *sum* are taken from r' and the *length* column is appropriately incremented. After creating *temp* as given above, it is merged with $old[q']$ corresponding to other $q' \in S[q]$ by using the above procedure; each time the result is placed back in $new[q]$.

At statement 11, $new[q]$ is set to the empty table. At statement 4, x is created as a table. The procedure is similar to the $Maxmerge$ procedure used in the previous section; the only difference is that the *sum* column is obtained by adding the *sum* column in $old[q']$ and the *value* column in the distance table are added. In statement 15, x is taken to be same as $old[q']$ after incrementing the length column and after processing the assignments also (using the selection operation) as in the previous case. In statement 17, the table x is merged with $new[q]$. During the merge the minimum value of the *sum* columns is taken. This operation is similar to the $Minmerge$ specified in the previous subsection. At statement 22, *Updateglobal* is done by computing the average value in the single row of the table $old[q]$.

The over all complexity of the algorithm is as that of algorithm AVG-DIST with the change that the table size becomes another factor in the complexity measure.

7 Experimental Results

We have implemented all the algorithms INF-NORM,INF-NORM-INDX,AVG-DIST,INF-NORM-RESTR-AUT and AVG-DIST-RESTR-AUT. In our implementation, the input query is specified as an automaton together with a length constraint. The length constraint is given by a pair of numbers giving the lower and upper bounds on the lengths of the strings. The automaton together with the length constraint define the query. The distances of the database subsequences are defined with respect to those strings, accepted by the automaton, whose lengths satisfy the length constraint, i.e., whose lengths lie with in the lower and upper bounds. (It is to be noted that one can certainly obtain another automaton which exactly accepts all such strings. However the automaton may be too big). The algorithms are easily modified (without much changes) to take the length constraint in to consideration. Also the algorithms are implemented to work on many database sequences.

7.1 Implementation for Time-Series Databases using Sequel Server

We have developed a number of systems (corresponding to the different algorithms presented in the paper) on PCs using Sequel server. They consider the database sequence to be stored in a single relation. This relation has a value attribute and a time attribute. It also may have other attributes. The value attribute in the different tuples defines the time-series at the different times specified by the time attribute. A primary index on the time attribute and a secondary index on the value attribute are defined for the relation. The implemented systems take an automata specification and output the identities of subsequences of the database sequence that closely satisfy the predicate specified by the automaton. Each of the implemented systems has two components. The first component generates the similarity lists for the atomic queries by executing appropriate SQL queries. The second component executes one of our algorithms and identifies the closest subsequences. The different options allow them to execute either range queries or nearest neighbor queries. The implementation corresponding to algorithm INF-NORM retrieves the database relation tuple by tuple, thus examining all the tuples. The other implementations execute an SQL query with appropriate WHERE clauses for each atomic query in the automaton. In the later case, the Sequel server employs the appropriate indices to execute the query. The similarity lists for the atomic queries are generated as follows. If the atomic query is a range query on the value attribute with bounds x and y (where $x \leq y$), then the similarity value for any number with in the bounds is 1; the similarity value for any number in the range y and $y + d$ goes down linearly from 1 to zero; correspondingly the similarity value goes down in the range x and $x - d$; for any number above $y + d$ or below $x - d$, the similarity value is zero. Here the value of d is given as percentage of the mean of x and y and this percentage is given as an input parameter; in most of our experiments this percentage is ten. (An exact atomic query on the value attribute is taken as a range atomic query whose upper and lower bounds are equal).

We ran the above implementations on a stock market database. The database stored the end-of-day Dow-Jones Industrial averages over the last 98 years giving a database sequence of length 26,716 (the length is the total number of trading days during that period). Our experiments were carried out using range queries.

1. Our first query automaton requires retrieval of subsequences starting on the 20^{th} day of a month and ending on the last day of the month such that Dow-Jones value is about 1100 during this whole period. This query is specified by an automaton that accepts the language given by the regular expression ab^*c where a, c denote the 20^{th} day of the month and the beginning of the next month respectively; b states that the value is 1100. Algorithm INF-NORM took one second and retrieved two subsequences; the first one starts on 20^{th} of July 1984 and has a similarity value of .8623; the second one starts on 20^{th} June 1984 and has a similarity value of .7081. Algorithms B and INF-NORM-RESTR-AUT gave same answers. Algorithm AVG-DIST and AVG-DIST-RESTR-AUT gave lot more than 30 subsequences with similarity value greater than 0.7; they took 44 seconds and 12 seconds respectively. Their I/O time is only about one second.

2. Our second query requires similar query for the beginning of the month. That is, it requires retrieval of subsequences starting with the beginning of a month and ending by the tenth day of the month such that the Dow Jones Average is about one thousand during the whole period. For this algorithm INF-NORM gave one such sequence with similarity value greater than 0.8 and four subsequences with similarity value greater than 0.7; it took one second. Algorithms INF-NORM-INDX and INF-NORM-RESTR-AUT gave

similar results and times. Algorithms AVG-DIST and AVG-DIST-RESTR-AUT gave nine subsequences with similarity value greater than 0.8. They took 102 and 34 seconds respectively, most of it processing time.

3. Our third and last query requires the retrieval of subsequences that start with a Dow Jones value between 1000 and 1080 and stay in that range until the value goes in to the range between 1100 and 1180 and stays in that range until it crosses 1200. This can be specified by an automaton that accepts strings represented by the regular expression a^+b^+c where a, b, c denote the three atomic predicates (here a^+ denotes one or more occurrences of a). For this query, algorithms INF-NORM,INF-NORM-INDX and INF-NORM-RESTR-AUT all gave one subsequence whose similarity value is greater than 0.5. This subsequence starts on July 24, 1984 and ends with Aug. 6, 1984. It has a similarity value of 0.7. Algorithm INF-NORM took two seconds, while algorithms INF-NORM-INDX and INF-NORM-RESTR-AUT took about one second. Algorithms AVG-DIST and INF-NORM-RESTR-AUT retrieved more than thirty subsequences with similarity greater than 0.9. They took 15 and 5 seconds, respectively.

The following table summarizes the results for the stock market database for the three queries and four algorithms. Each entry has three numbers (separated by /) — the number of retrieved subsequences, the threshold similarity value and the total time (in seconds), respectively. An entry 30+ denotes a number greater than 30. The results for algorithm INF-NORM-RESTR-AUT is same as that for algorithm INF-NORM-INDX and so omitted.

	INF-NORM	INF-NORM-INDX	AVG-DIST	AVG-DIST-RESTR-AUT
Query 1	2/.7/1	2/.7/1	30+/.7/44	30+/.7/12
Query 2	4/.7/1	4/.7/1	9/.8/132	9/.8/34
Query 3	1/.7/2	1/.7/1	30+/.9/15	30+/.9/5

Table 1: Results for Stock Market Database

We have also implemented the extended version of algorithm INF-NORM for automata with variables. We used an automaton with variables that specified the following query: "the *dow_index* went up by 1000 points from the beginning to the end of the subsequence". Specification of this property requires variables. We divided the previous sequence in to ten sequences, each one corresponding to a decade. We used a range query with a similarity threshold of 0.74. The algorithm took 1034 seconds. As to be expected, all the subsequences it retrieved are after 1980.

The second example we used was for a time-series database denoting temperatures on different days. On each day, the minimum, maximum and average temperatures are given. The sequence is of length one million and is generated randomly. The minimum and maximum temperatures of a day are chosen randomly between 20 and 120 such that the minimum is less than the maximum. The average is taken as the mean of these two values. This time series is stored as a relation with one million tuples. We ran our algorithms for the following queries.

1. The first query requires the retrieval of subsequences starting with maximum temperature 60 and ending with maximum temperature greater than 80 and such that the average temperature is between 45 and 55 throughout the subsequence. Algorithms INF-NORM, INF-NORM-INDX and INF-NORM-RESTR-AUT retrieved five sequences with similarity value greater than 0.7. Algorithm INF-NORM took more than two seconds while algorithms INF-NORM-INDX and INF-NORM-RESTR-AUT only took one second each. The times for INF-NORM-INDX and INF-NORM-RESTR-AUT are smaller because they employ indices for retrieval. Algorithms AVG-DIST and AVG-DIST-RESTR-AUT retrieved 12 subsequences with similarity greater than 0.7. Algorithm AVG-DIST-RESTR-AUT took seven seconds while algorithm C took 22 seconds.

2. Our second query required retrieval of subsequences in which the average temperature starts with value 50, and alternates between 50, 60 and ends with 70. It also required the subsequence to be of length at least four. Algorithms INF-NORM,INF-NORM-INDX and INF-NORM-RESTR-AUT all retrieved two

subsequences with similarity value greater than 0.5. Algorithm INF-NORM took two seconds, while INF-NORM-INDX and INF-NORM-RESTR-AUT took one second each. Here again, INF-NORM-INDX and INF-NORM-RESTR-AUT took lower time since they are using indices. Algorithms AVG-DIST and AVG-DIST-RESTR-AUT retrieved more than two subsequences. They took 42 and 16 seconds respectively.

7.2 Textual Database Example

We employed our methods for retrieval from textual databases. In our experiment we considered books. We used 83 books. Each book is modeled as a sequence of chapters and each chapter is considered as a single document. Altogether they had 2690 chapters. We used the query automaton $\mathcal{B}$ as defined below. It has four states 1,2,3,4. It uses three atomic predicates a_1, a_2, a_3. For each $i = 1, 2, 3$, it has a transition from state i to $i + 1$ on input symbol a_i. States 2,3,4 have self loops labeled with the wild card symbol. States 1 and 4 are the initial and final states, respectively. This automaton accepts all strings specified by the regular expression $a_1\phi^*a_2\phi^*a_3$; that is, it accepts strings starting with a_1, later followed by a_2 and ending with a_3 and having zero or more wild card symbols in between.

We took a_1, a_2, a_3 to be sets of key words taken from chapters 3,13 and 33 of the book "Pride and Prejudice". The similarity values of each document (i.e. chapter) for these atomic predicates are calculated using standard text retrieval methods (i.e. using term frequency and inverse document frequency taken from [Y98]). We ran algorithm INF-NORM requiring it to retrieve top 30 subsequences with in a distance range of 0.2. It ran in 0.04 seconds and found two books which together have less than 30 subsequences. As to be expected, most of the subsequences were from the book "Pride and Prejudice" itself. We ran algorithm AVG-DIST on the same query with the same parameters. It terminated in half second. It gave more books and subsequences than algorithm INF-NORM. This is to be expected as average block distance is less discriminating.

7.3 Comparison of Internal Processing Times

The experiments given in the previous subsections consider the total time (including the I/O time). We have also conducted experiments to compare the internal processing times of the different algorithms. The performance of the algorithms depends on the length of the intervals, length of the gaps in the similarity lists. We ran the algorithms on five different databases with respect to a typical query which is a cycle-restricted automaton. We got the following results. All the five database sequences are of length 2000, i.e. they have 2000 database states. Each sequence is divided into intervals. All database states with in an interval have identical similarity values with respect to the atomic queries. We define a *gap* to be an interval in which every state has zero similarity value with respect to all the atomic queries.

Database sequence 1 has 100 intervals each of length 20. One fourth of the intervals are gaps. Database sequence 2 is similar excepting that half the intervals are gaps. Database sequence 3 has 50 intervals half of which are gaps; each normal interval is of length 20, while each gap interval is of length 60. Database sequence 4 has 10 intervals each of length 200; half the intervals are gaps. Database sequence 5 has 5 intervals each of length 400; three of the intervals are gaps. The timings for the different algorithms are given by table 2.

	INF-NORM	INF-NORM-RESTR-AUT	INF-NORM-INDX	AVG-DIST	AVG-DIST-RESTR-AUT
Seq 1	2	1.5	16	2,906	618
Seq 2	2	1.5	15.5	1638	411
Seq 3	2	1	9	563	207
Seq 4	2	1	1.5	1469	573
Seq 5	2	1	0.5	1092	612

Table 2: Running times for different algorithms in tens of milliseconds

Algorithm INF-NORM has the same running time on all the sequences, since its running time depends only on the number of states in the sequence, which is 2000. Algorithm INF-NORM-RESTR-AUT performs better than algorithm INF-NORM since the query is a cycle-restricted automaton. It processes gaps faster. Algorithm INF-NORM-INDX takes advantage of the fact that, in each interval, all database states have identical similarity values; it uses a matrix method employing successive squaring. It treats gap and non-gap intervals similarly. When the intervals are large as in database sequence 5, it performs better than algorithm INF-NORM. Note that Algorithms INF-NORM, INF-NORM-RESTR-AUT and INF-NORM-INDX use infinite norm. Their complexity is linear in the length of the sequence. Algorithms AVG-DIST and AVG-DIST-RESTR-AUT use average block distance and they have complexity quadratic in the length of the sequence. That is why, their running times are higher. Algorithm AVG-DIST-RESTR-AUT performs better than algorithm AVG-DIST since the input automaton is cycle-restricted. In the first three database sequences, the length of the normal intervals remained same while the total gap size increased; as a consequence, the running times of algorithms AVG-DIST and INF-NORM-RESTR-AUT decreased. However, in the last two sequences the length of the normal intervals increased, and hence their running times increased.

From the above table, as expected, we see that using average block distance is more expensive in terms of running time. However, average block distance is less discriminating. Also, algorithms INF-NORM-RESTR-AUT and AVG-DIST-RESTR-AUT are better for cycle-restricted automata. Algorithms INF-NORM-INDX is preferable to algorithm INF-NORM if the lengths of the intervals is very large, i.e. the similarity values of atomic queries remain constant for long periods. More specifically, if all the intervals are of length greater than 200 then algorithm INF-NORM-INDX is preferable to algorithm INF-NORM.

8 Conclusion

We have introduced a powerful formalism based on automata for expressing queries on sequence databases. We have defined various distance measures denoting how closely a database sequence satisfies queries expressed in this formalism. We also have given efficient algorithms for similarity based retrieval that employ indices. We have implemented the algorithms for time-series databases on PC using Sequel server. We have also implemented them for retrieval from text databases. Experimental results showing the effectiveness of our methods are presented. We can also use the methods given in the paper for approximate firing of active database triggers when they are specified in the automata formalism.

The work given in the paper can also be used when the queries are given by regular expressions, by converting them in to automata. As far as we know ours is the first work in database area to consider similarity based retrieval from sequence databases using predicates over sequences in a comprehensive manner.

The primary application of our algorithms is when the query is given as a predicate. However, we may be able to use them effectively sometimes when the query is a single sequence that is periodic and that has a small number of atomic queries appearing in it. For example, consider a query sequence of the form $ab^{100}c$ where a, b and c are atomic queries. We can represent this sequence using an automaton with three states and with a length constraint. Our algorithms use indices on atomic queries. In many cases, these may be already available. We only need to make three accesses (corresponding to the three atomic queries) on these scalar indices. Unlike the approaches of [FRM94], we don't need to construct any special multi-dimensional access structures just for this purpose. Of course, our approach for a single sequence query will not work when the number of atomic queries is large and is as much as the length of the query sequence.

In this paper, we have not considered the complexity of computing the similarity values of atomic queries with respect to individual database states. We assumed that these are given to us or are easily computable (as in the time-series and text databases). For certain databases, such as video databases, computing these may incur substantial time. All these issues need to be further investigated as part of future work. Also, more faster techniques need to be explored for the average distance functions. Other similarity measures, apart from the ones given in the paper need to be investigated. It will also be interesting to see if and how the techniques of the paper can be extended for data mining over sequences.

References

[AFS93] Agarwal R., Faloutsis C., Swami A: *Efficient Similarity Search in Sequence Databases*, In FODO Conference, Evanston, Illinois, Oct. 1993.

[AHU] Aho A., Hopcroft J., Ullman J. : *The Design and Analysis of Computer Algorithms*, Addison Wesley 1974.

[APWZ95] Agarawal R., Psaila G., Wimmers E., Zait M., *Querying Shapes of Histories*, VLDB 95.

[AS94] Agarwal R., Srikant R.: *Mining Sequential Patterns*, In Proc. of the 11th Intnl. Conf. on Data Engineering, March 1995.

[ALSS95] Agarwal R., Lin K., Sawhney H., Shim K.: *Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases*, Proc. 21st Intnl. Conf. on Very Large Databases, Zurich, Switzerland, 1995.

[B97] Berchtold S., Bohm C., Braunm B., Keim D., Kriegel H.: *Fast Parallel Similarity Search in Multimedia Databases*, SIGMOD Conference 1997: 1-12.

[C89] Chakravarthy S., et al., *HiPAC: A Research Project in Active, Time-Constrained Database Management*, TR XAIT-89-02, Xerox Advanced Information Technology.

[D88] Dayal U., *Active Database Management Systems*, Proc. of 3rd Intnl. Conf. on Data and Knowledge bases — Improving usability and Responsiveness, Jerusalem, June 1988.

[D98] Durbin R., et al.: *Biological Sequence Analysis*, Cambridge University Press, 1998.

[FRM94] Faloutsos F., Ranganathan M., Manolopoulos: *Fast Subsequence Matching in Time-Series Databases*, Proc. of the 1994 ACM SIGMOD Intnl. Conf. on Management of Data, Minneapolis, MN, May 1994.

[GJS92] Gehani N., Jagadish H., Shmueli O.: *Composit Event Specification in Active Databases: Models and Implementation*, Proc. 18th Intnl. Conference on Very Large Databases, Aug. 1992.

[GRS99] Garofalakis, M. N., Rastogi R., Shim K.: *SPIRIT: Sequential Pattern Mining with Regular Expression Constraints*, Proc, of the 25th Intnl Conf. on Very Large Databases, Edinburgh, Scotlanad,UK, 1999.

[NRS99] Natsev A., Rastogi R., Shim K: *WALRUS: A Similarity Retrieval Algorithm for Image Databases*, ACM SIGMOD Conference 1999: 395-406.

[RM97] Rafiei D., Mendelzon A.: *Similarity based Queries for Time Series Data*, SIGMOD Conference 1997: 13-25.

[S] Shasha Dennis, *Time Series in Finance: the array database approach*, http://cs.nyu.edu/cs/faculty/shasha/index.html.

[SYV97] Sistla A. P., Yu C., Venkatasubramanian R.: *Similarity based Retrieval of Videos*, 13th International Conference on Data Engineering, April, 1997, Birmingham, U.K.

[SW94] Sistla A. P., Wolfson O.: *Temporal Conditions and Integrity Constraints in Active Database Systems*, Proc. of 1995 ACM SIGMOD Intnl. Conf. on Management of Data, 1995.

[WM92] Wu S., Manber U.,: *Fast Text Searching: Allowing Errors*, CACM Oct. 1992, pp 83-91.

[WS01] Whitney A., Shasha D.: *Lots O' Ticks: Real-time High Performance Time Series Queries on billions of trades and quotes*, SIGMOD01.

[Y98] Yu C., Meng W.: *Principles of Database Query Processing for advanced applications* Morgan Kaufmann, 1998.