Similarity-Based Queries

H. V. Jagadish
AT&T Bell Laboratories
jag@research.att.com

Alberto O. Mendelzon
University of Toronto
mendel@db.toronto.edu

Tova Milo
University of Toronto and Tel-Aviv University
milo@math.tau.ac.il

Abstract

We develop a domain-independent framework for defining queries in terms of similarity of objects. Our framework has three components: a pattern language, a transformation rule language, and a query language. The pattern language specifies classes of objects, the transformation rule language defines similarity by specifying the similarity-preserving transformations, and the whole package is wrapped in a general query language. The framework can be "tuned" to the needs of a specific application domain, such as time sequences, molecules, text strings or images, by the choice of these languages.

We demonstrate the framework by presenting a specific instance on a specific domain – the domain of sequences. We start with sequences over a finite alphabet, and then consider sequences over infinite ordered domains. The basic pattern language we use is regular expressions, and the query language is calculus-based. We show that even when the pattern/query languages chosen are not too powerful, the approximation framework obtained is very strong. We study the properties of the framework, and in particular present expressive power and complexity results.

1 Introduction

There is a vast class of database applications where it is important to be able to pose queries in terms of *similarity* of objects, rather than equality or inequality. For instance, consider: searching a protein database for molecules similar to a given one; finding sets of stocks that reacted in a similar way to an interest rate increase; or checking an article against a text database to detect possible plagiarism.

The meaning of similarity may vary depending on the application domain and even the purpose of the query. Work in this area is usually specific to one particular domain and uses one specific notion of similarity. For example, Faloutsos et al. [FRM94] studied the problem of searching a database of time sequences for sequences similar to a given one. They reduce sequences to points in a low-dimensional space by using Fourier transforms and use the Euclidean distance in these spaces to measure similarity. Other domain specific work on approximate string searching in text, images, and biochemical data includes [AFS93, AGMML90, GCB92, J91]. Other authors have recently studied models and languages for databases containing sequences, (e.g. [CRSV94, GM95, GNU94, GW92, Ric92, SLR94]), but without taking into account notions of similarity or approximation.

In this paper we generalize this type of work in two directions. First, we develop a domain-independent framework for defining notions of similarity. Second, we go beyond specific questions such as "find all objects similar to a given object a" and provide a full-fledged query language to support queries such as "find all objects that are similar to some object in class A and are not similar to any object in class B."

Our framework has three components: a pattern language, a transformation rule language, and a query language. Since the kinds of data that we are interested in, such as time sequences, molecules, text strings, or images, are often described conveniently by pattern expressions, the first component of our framework is a *pattern language* $\mathcal{P}$. An expression in $\mathcal{P}$ (e.g., a regular expression on text strings) specifies a set of data objects. To specify objects that match a pattern approximately, we attach to patterns in $\mathcal{P}$ *transformation rules* defined in a transformation language $\mathcal{T}$. An object A is considered to approximate an object B, if B can be reduced to it by a sequence of transformations. A specific application of a transformation to a data object has a *cost*, which is a measure of the distance between object A and object B; for example, the edit distance between two character strings is such a cost. An expression e in the pattern language $\mathcal{P}$, augmented with a transformation rule in language $\mathcal{T}$, specifies a set of data objects that approximate some object satisfying the expression e, in the sense that they can constructed by applying the given transformation rule on some object satisfying e. Finally, a query language $\mathcal{L}$ (e.g. relational calculus) is obtained by wrapping logic or algebra around expressions built using $\mathcal{P}$ and $\mathcal{T}$.

The framework can be "tuned" to the needs of a specific application domain by the choice of $\mathcal{P}$, $\mathcal{T}$, and $\mathcal{L}$. We demonstrate the framework by presenting a specific instance on a specific domain – the domain of sequences. We start with sequences over a finite alphabet, and then consider sequences over infinite ordered domains. The basic pattern language we use is regular expressions, and the query language is calculus-based. We show that even when the pattern/query languages chosen are not too powerful, the approximation framework obtained is very strong. We study the properties of the framework and, in particular, present expressive power and complexity results.

In Section 1 we informally demonstrate our framework with an example. Section 2 formally presents the different components of the framework. The following three sections study the properties of the framework when applied to a specific domain – sequences over a finite alphabet. Section 3 studies transformation rules, Section 4 investigates the approximate pattern language, and Section 5 deals with the query language. Infinite ordered domains, and their effect on the properties of the framework components, are considered in Section 6. Finally, Section 7 is a conclusion.

2 Example – Sequence Data

To give some intuition on the framework, and its application to a specific domain, we start with an example that demonstrates informally the use of patterns, transformation rules, and queries for approximate search. Consider a database containing a set of sequences (e.g. stock price data), each describing a real-valued function $f(x)$ on some real interval. Assume that a function is encoded in the sequence using a piecewise linear approximation, i.e. a function $f(x)$ is described as a sequence of real number pairs. The first pair specifies the x and y coordinates where the graph of the function starts. Each of the following s, l pairs represent a linear segment. The s element is the slope and l is the x extent of the segment. The domain Σ of both slopes and extents is some set of floating point numbers.

Suppose we want to find functions that look like the one in Figure 1. Let our pattern language $\mathcal{P}$ consist of regular expressions, and our query language be the tuple relational calculus, with one unary predicate $R(u)$, meaning that a sequence u belongs to the database relation R.

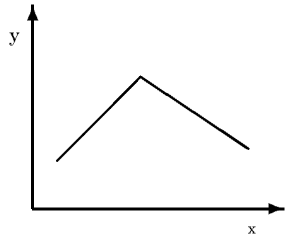

Figure 1

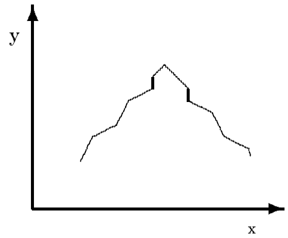

Figure 2

A first attempt could be to look for a sequence of segments of equal positive slopes, followed by a sequence of segments of equal negative slope.[1] The query computing this is

$$\{u \mid R(u) \wedge (\exists s_1, s_2)\ s_1 > 0 \wedge s_2 < 0 \wedge \mathrm{IN}(u, (\Sigma.\Sigma).(s_1.\Sigma)^+.(s_2.\Sigma)^+\ )\ \}.$$

The predicate $\mathrm{IN}(u, e)$ where e is a regular expression, means that string u is in the language defined by e. The symbol Σ in the regular expression stands for any symbol in the alphabet. The first two occurrences of Σ stand for the x and y coordinates where the function's graph starts, and s_1 and s_2 are the positive and negative slopes of the two pieces. Note that the only functions that will satisfy this query are those that have *exactly* the specified shape. Suppose we wanted instead to have something like Figure 2 be also considered a match. One possibility is to define a *refinement* transformation rule that collapses adjacent segments s_1 l_1 and s_2 l_2 to a single segment whose slope is the weighted average of the two slopes and whose length is the sum of the two lengths. Here is such a transformation rule r:

$$\langle(\Sigma.\Sigma)^+ : h\rangle.\langle\Sigma : s_1\rangle.\langle\Sigma : l_1\rangle.\langle\Sigma : s_2\rangle.\langle\Sigma : l_2\rangle.\langle(\Sigma.\Sigma)^* : t\rangle$$

$$\rightarrow h.(l_1 * s_1 + l_2 * s_2)/(l_1 + l_2).(l_1 + l_2).t$$

The meaning of the rule is that a function that matches the left hand side is transformed to the pattern in the right hand side. The left hand side is a concatenation of named regular expressions, and the names are used in the right hand side to define the transformed function. In the example, a string

[1]We have already made some choices here on what we want similarity to mean; for example, that the particular slopes of the two pieces of Figure 1 are irrelevant, that the length of the segments is irrelevant, etc. If we wished, we could change any of these assumptions and vary the query accordingly.

consisting of an arbitrary prefix h followed by two segments s_1, l_1 and s_2, l_2, followed by an arbitrary suffix t, is transformed by leaving h and t untouched and collapsing the two segments into one.

One application of the rule r refines a single pair of adjacent segments. We may want to apply a sequence of such refinement steps. For that we define a rule r' that allows applying r repeatedly any number of times.

$$r' : \quad r^*$$

To restrict the amount of refinement performed, we can use a cost measure. The rule r defines a ternary relation $\tau_r(s, s', c)$ where s is a sequence that can be transformed by r to a sequence s', with cost c. For example, we can define the cost of one application of rule r to be the change in the value of the integral under the function, i.e. $l_1 \times l_2 \times |s_1 - s_2|/2$. The cost of applying the rule r' to transform string s into string s' can be defined to be the minimum, over all possible sequences of applications of r that map u to u', of the sum of the cost of all the rule applications along the sequence.

Our goal is to find all functions in the database that can be transformed into the pattern in Figure 1, using the transformation rule r', with a cost less than some bound C. In general, given a pattern expression e and a rule r, the expression $e \approx r$ denotes all strings that can be obtained by applying the transformation rule r to some member of the set specified by e. In addition to the predicate $\mathrm{IN}(u, e)$, we use $\mathrm{IN}_c(u, e \approx r, C)$ to mean that u can be obtained with cost less than C by applying rule r to some member of the set specified by e. Thus the query becomes:

$$\begin{aligned} \{u \quad | \quad & R(u) \ \wedge \ (\exists u') \\ & \mathrm{IN}_c(u', \ u \approx r', \ C\ ) \wedge \\ & (\exists s_1, s_2) s_1 > 0 \wedge s_2 < 0 \wedge \\ & \mathrm{IN}(u' \ , \ (\Sigma.\Sigma).(s_1.\Sigma)^+.(s_2.\Sigma)^+ \ ) \ \}. \end{aligned}$$

The exact syntax and semantics of the different parts of the language (and in particular the built-in predicates IN and IN_c) are described in the following section.

3 Approximate Search

In this section we formalize our framework, and demonstrate it using sequences as an example. We first study strings over a finite alphabet Σ, and in Section 7 extend the work to infinite ordered domains, such as we would need to represent the piecewise linear approximations in the example above.

3.1 Pattern language

As the pattern language, we use regular expressions over Σ. (The framework does not constrain us to use any particular pattern language: other possibilities include context free grammars, temporal logic, etc.) For brevity, we use in an expression the symbol Σ to denote the set of all constants in the domain. We say that a string s *satisfies* a regular expression e (denoted $s \in e$) if it belongs to the class of strings defined by e.

3.2 Transformation rules

Given a set O of objects, a transformation rule defines a new set O' containing objects that approximate the objects in O. In the Introduction, we spoke of a *transformation rule language* $\mathcal{T}$ used to define transformation rules. Given a pattern language $\mathcal{P}$, the transformation rule language is determined by two components: a composition language $\mathcal{C}$, that is used to define *atomic* transformation rules, and a sequence language $\mathcal{S}$, that is used to compose atomic rules into complex rules. We explain below how to define transformation rules for strings. Transformation rules for other domains of objects are defined similarly. We first consider atomic rules.

Atomic transformation rules

An atomic rule $r(k)$, when applied to an input string s, identifies substrings of s having certain patterns, and then uses these substrings (and the input parameter k) to construct a set of output strings. The substring patterns are described using the pattern language $\mathcal{P}$. The way in which these substrings are embedded in the input string s is described using some composition language $\mathcal{C}$. Usually $\mathcal{C}$ will consist of simple operators like concatenation. Formally, an atomic transformation rule $r(k)$ has the form

$$r(k) : e(\langle p_1 : x_1 \rangle, \ldots, \langle p_m : x_m \rangle) \rightarrow p(x_1, \ldots, x_m, k)$$

where $p, p_1, \ldots, p_m$ are expressions in the pattern language $\mathcal{P}$, e is an expression in C, and $k, x_1, \ldots, x_n$ are variable names.

The semantics of an atomic transformation rule $r(k)$ is as follows: Let s, t be some strings. The *set of transformations of s by $r(t)$*, denoted $s \approx r(t)$, is the set of all strings s' for which there exist some substrings $s_1, \ldots, s_m$ of s, such that $s = e(s_1, \ldots s_m)$, each s_i $(i = 1 \ldots m)$ satisfies p_i, and $s' \in p(s_1, \ldots, s_m, t)$. We say that a string s is **transformed** into s' by the rule $r(t)$, iff $s' \in (s \approx r(t))$.

A *cost* function $f_r(x_1, \ldots, x_m, k)$ may be associated with each transformation rule r. The cost of transforming a string s into s' using the rule $r(t)$, is defined to be the minimum of $f_r(s_1, \ldots s_m, t)$ over all the substrings $s_1, \ldots, s_m$ above.

For example, assume that $\mathcal{C}$ is a language containing a single operation – string concatenation. Consider the following two atomic transformation rules:

$$\begin{aligned} r_1(k) &: \ \langle \Sigma^* : x_1 \rangle . \langle a : x_2 \rangle . \langle \Sigma^* : x_3 \rangle \ \rightarrow \ x_1.k.x_3 \\ r_2 \quad &: \ \langle \Sigma^* : x_1 \rangle . \langle b : x_2 \rangle . \langle \Sigma^* : x_3 \rangle \ \rightarrow \ x_1.x_3 \end{aligned}$$

For every string s, $s \approx r_1(b)$ contains all strings constructed from s by replacing one occurrence of the letter a, by b. The set $s \approx r_2$ contains all strings constructed from s by deleting one b. Note that if s has no a's then $s \approx r_1(b)$ is empty, and if s has no b's then $s \approx r_2$ is empty.

Assume that the cost function $f_{r_2}(x_1, x_2, x_3)$ of the second rule is $f_{r_2}(x_1, x_2, x_3) = |x_1|$ (i.e. the number of letters up to the deleted b). When transforming a string $abbc$ into abc, the variables (x_1, x_2, x_3) can be assigned the values (a, b, bc) or (ab, b, c) resp. The transformation cost is 1 (since the assignment (a, b, bc) is the one for which f_{r_2} has minimal value).

For a set of strings S, $S \approx r(t)$ is the union of all the sets $s \approx r(t)$, such that $s \in S$. For example, $c^*.a.c^* \approx r_1(b)$ is the class of all strings of the form $c^*.b.c^*$. As another example, $(c^*.a.c^* \approx r_1(b)) \approx r_2$ is the class of all strings of the form c^*. Note that in general several strings in S can be transformed into the same string s'. We define the *cost of transforming S to s'* using $r(t)$ to be the minimum over all $s \in S$ of the cost of transforming s to s'.

Complex transformation rules

A complex transformation rule specifies *sequences* of atomic transformations to be applied to a string. These sequences are specified in a language $\mathcal{S}$. We will again use regular expressions as an example of such language, but the alphabet now is the set of names of atomic transformation rules. In general, the language $\mathcal{S}$ may differ from the pattern language $\mathcal{P}$, and we shall see an example of that in section 7.

A complex rule $r(k_1, \ldots, k_n)$ has the form

$$r(k_1, \ldots, k_n): \quad e(k_1, \ldots, k_n),$$

where e is an expression in S, and $k_1, \ldots, k_n$ are input variables. The semantics of such a rule is as follows.

Let $s, s_1, \ldots, s_n$ be some strings. $s \approx r(s_1, \ldots, s_n)$ is the set of strings s' such that, for some sequence of atomic transformations $r_1(s_{i_1}).r_2(s_{i_2}) \ldots r_n(s_{i_n})$ satisfying e, $s' \in ((((s \approx r_1(s_{i_1})) \approx r_2(s_{i_2})) \ldots) \approx r_n(s_{i_n}))$.

For example, the following is a complex transformation rule

$$r_3(k_1, k_2): \quad r_1(k_1).r_1(k_2).r_2^*.$$

For a string s, $s \approx r_3(t, d)$ contains all strings constructed from s by replacing one occurrence of the letter a by t, another occurrence by d, and then deleting some b's.

The cost of transforming a string s into s' by a sequence of applications of atomic transformations is obtained as a function (e.g. addition) of the costs of the individual transformations being performed. Note that a string s can be transformed into s' by several transformation sequences defined by $r(s_1, \ldots, s_n)$. We define the *cost of transforming s to s'* using $r(s_1, \ldots, s_n)$ to be the minimum of the cost over all such transformation sequences. For a set of strings S, $S \approx r(s_1, \ldots, s_n)$ is the union of all sets $s \approx r(s_1, \ldots, s_n)$ such that $s \in S$. As for atomic rules, we define the *cost of transforming S to s'* using $r(s_1, \ldots, s_n)$ to be the minimum over all $s \in S$ of the cost of transforming s to s'.

3.3 The approximate pattern language

A pattern language $\mathcal{P}$ defines sets of objects with a specific pattern. The language can be extended so that some variance from the pattern is allowed, by embedding transformation rules in the pattern language. We explain below how rules are embedded in regular expressions. A similar approach can be used for any other pattern language. Let $\mathcal{T}$ be a set of transformation rules. An *approximate regular expression* over $\mathcal{T}$ is a term formed as follows:

1. Every variable x, and every regular expression e are approximate expressions.

2. If e_1, e_2 are approximate expressions then so are $e_1 \cup e_2$, $e_1.e_2$, e_1^*, (e_1)

3. If e is an approximate expression and $r(k_1, \ldots, k_n) \in \mathcal{T}$ is a transformation rule, then $e \approx r(v_1, \ldots, v_n)$, where each v_i is a variable name or a constant, is an approximate expression.

4. If $e, e_1, \ldots, e_n$ are approximate expressions, and $v_1, \ldots, v_n$ are variable names that do not appear in $e_1, \ldots, e_n$, then $(\exists\ v_1 \in e_1, \ldots, v_n \in e_n \mid e)$ is an approximate expression.

To define the semantics of an approximate expression we use the standard notion of *free* and *bound* variables. The quantifier $\exists$ is used to bind variables. Let e be an approximate expression with no free variables, and let s be a string. The semantics of e is defined inductively as follows:

1. If $e = e'$ where e' is a regular expression then $s \in e$ iff $s \in e'$.

2. If $e = e' \approx r(v_1, \ldots, v_n)$, then $s \in e$ iff $s \in (s' \approx r(v_1, \ldots, v_n))$, for some string $s' \in e'$.

3. If $e = (\exists\ v_1 \in e_1, \ldots, v_n \in e_n \mid e')$, then $s \in e$ iff $s \in e''$ for some e'' obtained from e' by replacing all the free occurrences of v_i, $i = 1 \ldots, n$, in e' by some string $s_i \in e_i$.

4. The semantics of $e_1 \cup e_2$, $e_1.e_2$, e_1^*, (e_1), are defined in the standard way.

We say that a string s *satisfies* an approximate expression e iff $s \in e$. The following examples demonstrate the use of quantification to bind transformations performed in different parts of a formula. Consider the reduction rule

$$r_1(k): \quad \langle \Sigma^* : x_1 \rangle . \langle a : x_2 \rangle . \langle \Sigma^* : x_3 \rangle \quad \rightarrow \quad x_1.k.x_3$$

and the two approximate expressions

$$e_1 = (\exists k \in (d \cup e) \mid (a.b) \approx r_1(k))^*,$$

and

$$e_2 = (\exists k \in (d \cup e) \mid ((a.b) \approx r_1(k))^*).$$

In the first formula, each occurrence of $a.b$ is approximated independently. Thus the set of strings defined by e_1 is $((d \cup e).b)^*$. In the second formula, all the occurrences of $a.b$ are approximated in the same way. Thus the set of strings defined by e_2 is $(d.b)^* \cup (e.b)^*$

Let e be some approximate expression. The **construction cost** of an object, $s \in e$, can be defined as a function of the cost of transformations performed in the construction of s. The cost function is defined by induction on the structure of e, assuming the costs of strings constructed by non approximate expressions are 0, and then accumulating the costs of transformations performed in a possible construction.

3.4 The query language

Thus far we focused on languages to describe approximate patterns. In this section we consider how to embed an approximate pattern language, such as the one developed above, into a full-fledged query language.

We define below a calculus-based query language that is an extension of the tuple relational calculus with function symbols, and with some built-in predicates. An algebraic language can be defined similarly, and is briefly discussed in section 6. We assume that the attributes of relations in the database range over some *object* domain (e.g strings). We further assume a set of functions F over this domain. The built-in predicates are equality, and two special predicates IN and IN_c whose semantics are described below. We first define terms and atomic formulas, and then use them to define calculus formulas.

Object terms include variable names and constants over the object domain. If $t_1, \ldots, t_n$ are terms, and $f \in F$, then $f(t_1, \ldots, t_n)$ is an object term. *Atomic formulas* have the form $R(x_1, \ldots, x_n)$, $IN(x_1, e)$, $IN_c(x_1, e, c)$, and $e_1 = e_2$, where R is a relation name, $x_1, \ldots, x_n$ are variables or constants, e is an approximate pattern expression, e_1, e_2 are object terms, and c is a variable or a constant of type integer or real. *Calculus formulas* are constructed from atomic formulas using $\vee$, $\wedge$, $\neg$, $\exists$, $\forall$. The semantics of formulas is standard, except for the IN and IN_c predicates. Let θ be an assignment for the free variables in s and e. $\theta \models IN(s, e)$ iff $\theta(s) \in \theta(e)$, and $\theta \models IN_c(s, e, c)$ iff $\theta(s) \in \theta(e)$ with construction cost $\leq c$.

As an example, consider a database where attributes of relations may contain strings. Let $r_1(k)$ be the transformation rule defined in the previous subsection. The following query selects pairs of strings s, s' such that s' differs from s by at most one occurrence of the letter a.

$$Q = \{s, s' | R(s, s') \wedge IN(s, (\exists k \in \Sigma | s' \approx r_1(k)))\}$$

The framework presented above can be "tuned" to handle a specific domain by choosing an appropriate pattern language $\mathcal{P}$, a construction language $\mathcal{C}$ (for defining atomic transformation rules), a sequence language $\mathcal{S}$ (for defining complex rules), and a set of functions F (to be used in calculus formulas). To understand the capabilities and properties of the framework, we study below a specific instance. We concentrate on string approximations. For simplicity, we use regular expressions for $\mathcal{P}$ and $\mathcal{S}$, and assume that $\mathcal{C}$ and F contain a single operator – string concatenation. We show that although the languages chosen are not too powerful, the approximation framework obtained is very strong. We first study the properties of transformation rules. Then we study the approximate regular language that uses such rules. Finally, we study the calculus extended with the approximate regular language.

4 Transformation Testing

A transformation rule takes strings and transforms them into other strings. A fundamental problem is whether a string s can be transformed into another string s' by a transformation rule r. Unfortunately, it turns out that even when rules are specified by regular expressions, this is not always effectively testable. For a set of transformation rules T, a rule $r(k_1, \ldots, k_n) \in T$, and strings $s, s', s_1, \ldots, s_n$, we call the problem of checking if $s' \in (s \approx r(s_1, \ldots, s_n))$, the *transformation testing* problem.

Theorem 4.1 *The transformation testing problem is undecidable.*

The proof, by reduction from Turing machine acceptance, is based on the fact that the use of $*$ in defining sequences of atomic rules allows sequences of arbitrary length. If this is ruled out, transformation testing becomes decidable.

Definition 4.1 *An atomic transformation rule*

$$r(k) : e(\langle p_1 : x_1 \rangle, \ldots, \langle p_m : x_m \rangle) \rightarrow p(x_1, \ldots, x_m, k)$$

has bounded growth *if p does not contain $*$. The name comes from the fact that the rules with bounded growth are precisely those for which the length of the output string is bounded by some function of the input string.*

Definition 4.2 *A complex transformation rule*

$$r(k_1, \ldots, k_n): \quad e(k_1, \ldots, k_n)$$

is star-free *if e does not contain $*$ and all the atomic rules mentioned in e have bounded growth.*

Theorem 4.2 *For star-free rules, transformation testing is NP-complete.*

Proof: (Sketch) Membership in NP is based on the observations that (i) the length of any sequence of atomic rules defined by a star-free complex rule r is at most polynomial in the size of r, and (ii) the output of a bounded-growth atomic rule r is of size at most polynomial in the size of r and its input. Hardness is by reduction from 3CNF satisfiability. $\Box$

Note that testing whether a string s can be transformed into s' by a rule r, with a cost lower then some bound c, is not more difficult than simply testing for the feasibility of the transformation. This follows from the fact that the algorithm used in the above proof "guesses" a sequence of transformations to be performed, and the input/output for each transformation step. Thus all the information needed for computing the transformation cost is available.

It turns out that if we consider only atomic rules, then even if $*$ appears in the right-hand side, transformation testing is in NP.

Theorem 4.3 *For atomic transformation rules, transformation testing is in NP.*

It is open whether the problem is NP-complete for atomic transformation rules. Finding interesting classes of rules for which the problem is in PTIME is also an open problem.

5 Expressive Power and Complexity

We study the expressive power and complexity of the approximate regular language presented in section 3.3.

5.1 Complexity

For a set of transformation rules T, a string s, and an expression e over T, the **membership testing** problem is the problem of testing whether $s \in e$. From Theorem 4.1, it is clear that:

Proposition 5.1 *Membership testing is undecidable.*

Here again, the undecidability is due to the use of $*$ in transformation rules. If the transformation rules are star-free and have *bounded shrinkage*, the problem becomes decidable. Intuitively, transformation rules with bounded shrinkage decrease the length of their input by at most a constant.

Definition 5.1 *We say that an atomic transformation rule r has* bounded shrinkage *if there is some constant n such that for all strings s, s', t, if $s' \in (s \approx r(t))$ then $|s'| \geq |s| - n$. s is the length of string s.*

It is easy to show that there is a simple syntactic test for bounded shrinkage. The next theorem shows that membership testing for star-free rules with bounded shrinkage is decidable.

Theorem 5.2 *Let T be a set of transformation rules. The membership testing problem is NP-complete for T containing only star-free rules, where all the atomic rules have bounded shrinkage.*

Proof: (Sketch) The NP-hardness follows from the proof of Theorem 4.2. (All the transformation rules used there have bounded shrinkage). To show that the problem is in NP-TIME we present an algorithm, based on the following observations. (i) The shrinkage factor of complex rules can be computed using the factors of the atomic rules, and (ii) if $s \in e$, then all the strings involved in the construction of s are at most of length $|s| + |e| \times n$, where n is the maximal shrinkage factor of the rules in e. $\Box$

Note that testing if $s \in e$ with a construction cost lower then some bound c, is not more difficult than simply testing for membership. This is because the algorithm used in the above proof "guesses" how the string was actually constructed, the transformations performed in the construction, and the input/output for each transformation step. Thus all the information needed for computing the construction cost is available. It is open if the problem stays in NP-TIME for star-free rules without bounded shrinkage. As in the previous section, it can be shown that testing whether $s \in e$ with a construction cost lower then some bound c is not more difficult than simply testing for membership.

Finally, even with rules that are not star-free and do not have bounded shrinkage there are restrictions that guarantee decidability. For example:

Proposition 5.3 *If f is a regular expression and r is an atomic transformation rule where each variable is used in the left hand side only once then the question whether $s \in (f \approx r)$ is decidable in exponential time.*

5.2 Expressive power

We showed above that adding approximation increases the complexity of the pattern language. Does approximation also add expressive power? The answer is affirmative; in fact, approximate regular expressions can express any computable language.

Theorem 5.4 *Approximate regular expressions can express any computable language.*

The expressiveness is due to three features: (1) the use of $*$ in transformation rules, (2) multiple occurrences of variables, and (3) quantification. If all three are disallowed, the restricted approximate language has the same expressive power as the standard regular language.

Theorem 5.5 *Every approximate regular expression f where*

1. *the transformation rules are star-free,*
2. *in the atomic transformation rules, each variable*[2] *appears in each side of the rule at most once, and*
3. *no quantification is used in the approximate expressions,*

has an equivalent regular expression f'.

All the restrictions imposed on the languages are essential for staying within the expressiveness of regular expression. Indeed, if any of them is removed, the language can express non-regular sets.

Proposition 5.6 *If any one of the restrictions (1)–(3) in Theorem 5.5 is dropped, then regular approximate expressions can express non-regular sets.*

Proof: (sketch) For (1), the proof follows from the proof of Theorem 5.4. In all the rules in the proof the variables are used only once in each side of the rule, and the expression defining the language is quantifier free. For (2) and (3), we prove the proposition by showing that the language $b^n.c.b^n.c.b^n$ is expressible. □

Note that there are some restricted cases where quantification does not add expressive power. For example if all the quantifiers range over finite sets (e.g if they range over strings defined by star-free regular expressions) then the quantification can be eliminated and replaced by a union over all the possible assignments to the quantified variables, that is:

$$(\exists v_1 \in e_1, \ldots, v_n \in e_n \quad | \quad e(v_1, \ldots, v_n)) = \bigcup_{c_1 \in e_1, \ldots, c_n \in e_n} e(c_1, \ldots, c_n)$$

6 The Calculus

We focused above on languages for describing approximate patterns. In this section we consider the calculus-based query language that uses such patterns. We study here a specific instance: (i) the calculus uses approximate regular expressions as the underlying approximate language, and (ii) the set of functions F used in calculus formulas contains a single operator – string concatenation.

[2] including the input variable k

To ensure computability of queries, we restrict our attention to a **safe** portion of the calculus. We follow the standard approach to safety, but generalize it to ensure that all the strings tested for membership in approximate expressions are constructed from substrings of strings that appear in the database.

Definition 6.1 *A range formula restricting the variables $x_1, \ldots, x_n$ is a formula were $x_1, \ldots, x_n$ are free, having the structure defined below:*
basis:

a. *$R(x_1, \ldots, x_n)$ is a range formula restricting $x_1, \ldots, x_n$.*

b. *$s = x_1.x_2 \ldots x_n$, where s is a constant string, is a range formula restricting $x_1, \ldots, x_n$.*

construction: *if ϕ_1, ϕ_2 are range formulas restricting $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$ resp., then:*

1. *$\phi_1 \vee \phi_2$, where ϕ_1 and ϕ_2 have the same free variables $x_1, \ldots, x_n$, is a range formula restricting $x_1, \ldots, x_n$.*
2. *$\phi_1 \wedge \phi_2$ is a range formula restricting both $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$.*
3. *$\phi_1 \wedge (e_1 = e_2)$, where all the variables in e_1, e_2 are restricted by ϕ_1, is a range formula restricting $x_1, \ldots, x_n$.*
4. *$\phi_1 \wedge \neg\phi_2$, where all the free variables in ϕ_2 are restricted by ϕ_1, is a range formula restricting $x_1, \ldots, x_n$.*
5. *$\phi_1 \wedge y = e_1$, where all the variables of the term e_1 are restricted by ϕ_1, is a range formula restricting $y, x_1, \ldots, x_n$.*
6. *$\phi_1 \wedge x = y_1.y_2 \ldots y_m$, where x is restricted by ϕ_1, is a range formula restricting $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$.*
7. *$\phi_1 \wedge IN(s, e)$, where s and all the free variables in e are restricted by ϕ_1, is a range formula restricting $x_1, \ldots, x_n$.*
8. *The same as above, but instead of $IN(s, e)$ we have $IN_c(s, e, c)$ for some constant c.*
9. *$\exists x_i(\phi_1)$, where x_i is restricted by ϕ_1, is a range formula restricting the variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$.*

We call a calculus query $Q = \{x_1, \ldots, x_n | \phi\}$ **safe**, iff ϕ is a range formula that restricts $x_1, \ldots, x_n$. We refer to the calculus, restricted to safe queries as the **safe calculus**.

It is easy to give an algebraic version for the safe calculus. The algebra is constructed from the standard relational algebra by (1) allowing relations to have attributes of type string, (2) adding a *substring* operation that given a 1-ary relation R containing a set of strings, returns a a 1-ary relation containing all the substrings of strings in R, (3) having a general MAP operator (instead of simple projection), where the operators used in the expressions applied by MAP are attribute projection, tuple construction, and string concatenation, (4) extending the selection operator with three types of selection criteria:

$$e_1(att_1, \ldots, att_n) = e_2(att_1, \ldots, att_n),$$

$$IN(att_i, e(att_1, \ldots, att_n)),$$

and

$$IN_c(att_i, e(att_1, \ldots, att_n), c),$$

where e_1, e_2 are string expressions that may use the concatenation operator, e is an approximated regular expression, and att_i are attribute names.

The equivalence of the calculus and algebra is easily established.

Proposition 6.1 *The extended algebra and the safe calculus are equivalent.*

An important observation is that each algebra operation expands the size of its input to at most a polynomial. It follows that

Proposition 6.2 *A safe calculus query can be evaluated in time $\mathcal{O}(P(\text{approx}))$, where P is a polynomial, and* approx *is the complexity of the membership problem for the underlying approximation language.*

7 Extensions

In the previous sections we demonstrated our framework on a specific domain – strings over a finite alphabet. Let us now go back to the example presented at the beginning of the paper – sequences representing linear approximation of functions – and the approximation problems related to them, e.g. refinement of functions. In this section we extend the pattern language $\mathcal{P}$ and the set of functions F used in the previous sections to be able to handle approximate search problems in this domain.

The pattern language

We represent piecewise linear approximations to real functions, as discussed in the Introduction, by strings over an infinite ordered domain Σ. The domain may be continuous or not. (We shall note where this makes a difference). We need a pattern language $\mathcal{P}$ suitable for describing such strings. We assume some basic arithmetic operations on elements in the domain, e.g. $+, -, \times, /$. Let e be some arithmetic expressions, and $c_1, \ldots, c_n$ be some constants in Σ. A possible language (that will be used in the rest of this section) is obtained by extending regular expressions with atomic terms of the form $c_1 < \Sigma$, $\Sigma < c_1$, $c_1 < \Sigma < c_2$, (each defining a possibly infinite set of constant in the corresponding range), and $e(c_1, \ldots, c_m)$ (that defines the constant $c = e(c_1, \ldots, c_m)$). For example, take Σ to be the domain of integers. The expression $(1 < \Sigma < 5)^*.(1{+}2{\cup}9).(3 < \Sigma)$ defines strings that start with a sequence containing 2's and 3's, followed by either 3 or 9, and ending with an integer greater than 3.

Transformation rules

Atomic transformation rules use the pattern language $\mathcal{P}$, and a composition language $\mathcal{C}$. Complex transformation rules define sequences of atomic transformations using a language $\mathcal{S}$. We use the extended regular language described above as the pattern language $\mathcal{P}$, and leave $\mathcal{C}$ and $\mathcal{S}$ as before. i.e. $\mathcal{C}$ contains just string concatenation, and $\mathcal{S}$ is regular expressions. This combination is powerful enough to define useful transformations such as uniform and non-uniform refinement, scaling, rotation, etc. For example, the refinement transformation rules r and r' discussed in section 2 were defined using this combination.

Approximate Patterns and Queries

The approximate pattern language is obtained by embedding the transformation rules in the extended regular language. This is done exactly as in Section 3. The approximate language can then be embedded in the calculus to define queries similar to the example in section 2. Note that the query there is not safe. Nevertheless, a safe equivalent query can be defined using an additional transformation rule, as follows:

$$\{\ u \mid R(u) \wedge \mathrm{IN}_c(1.1.1/2.1.{-1}/2.1,\ u \approx r' \approx r_1 \approx r_2,\ C\ )\ \}$$

where r_1 joins together adjacent pairs with the same slope and r_2 transforms a sequence of the form $x.y.s_1 > 0.l_1.s_2 < 0.l_2$ to the sequence $1.1.1/2.1. - 1/2.1$

7.1 Complexity

We now consider how the complexity is affected by the above extensions. Most of the results in the previous sections carry over, with some restrictions, emphasized below by setting the additional conditions in bold font.

A key observation used in the proofs is that although the extended regular language deals with an infinite domain, the expressions in the language still correspond to finite state automata over a finite domain. [3] We first reconsider the complexity of the *transformation testing* problem – testing whether $s' \in (s \approx r(s_1, \ldots, s_n))$, for strings $s, s', s_1 \ldots s_n$ and a transformation r (defined in the extended framework).

[3] This is because each expression actually uses only a finite number of constants $c_1 < c_2 < \ldots < c_n$, and all the constants $c_i < c < c_{i+1}$ get a uniform treatment.

Theorem 7.1 (4.2) *Let T be a set of transformation rules. The transformation testing problem is NP-complete for T with star-free rules, where Σ* **does not appear on the rhs of atomic rules**.

Remark: Note that the refinement rule r' of Section 2 is not star-free, and thus the above theorem does not apply. Nevertheless, the transformation testing for it is in NP-TIME. (Each application of the rule r shortens the string, thus it can be applied at most $|s|$ times). □

As before, the problem is decidable for all atomic rules, even when they are not star-free.

Theorem 7.2 (4.3) *Transformation testing is in NP for all atomic transformation rules.*

We next reconsider the membership testing problem – testing whether $s \in e$ for a string s and an approximate expression e. In Section 5.1 we showed that the problem is in NP for star-free rules with bounded shrinkage. This shrinkage refers to the *length* of strings. To get the same complexity result for the extended language, we further require that the rules have *bounded domain shrinkage.* Intuitively, bounded domain shrinkage requires that the range of the constants appearing in the output of a rule be smaller than the range of the constants in the input string by at most a constant.

Definition 7.1 *For a string s, let $max(s)$ denote the highest constant in s, and $min(s)$ the smallest one. We say that an atomic rule r has* bounded domain shrinkage *if there is a constant n, such that for all strings s, s', t, if $s' \in (s \approx r(t))$ then $max(s') \geq max(s) - n$. and $min(s') \leq min(s) + n$.*

The following theorem applies to **discrete** domains.

Theorem 7.3 (5.2) *The membership testing problem for the extended language over a discrete domain is NP-complete, for star-free transformation rules with length and domain shrinkage bounded by some constants n_l and n_d resp.*

Proof: (Sketch) The proof is similar to that of Theorem 5.2, and is based on the fact that all constants involved in the construction of a string $s \in e$ are from a finite domain, bounded below by $min(s) - |e| \times n'_d$ and above by $max(s) + |e| \times n'_d$. (Where n'_d is the maximal domain shrinkage of all the rules in e.) □

Recall that it is possible to analyze rules syntactically and decide if they have bounded length shrinkage. It is open whether bounded domain shrinkage is decidable.

8 Conclusion

We conclude by briefly illustrating the framework on another domain. Consider shapes represented as a collection of black pixels, identified by location in a (rectangular) coordinate system. Assume that the pattern language $\mathcal{P}$ has a rectangular region of pixels as a basic unit, and union and difference as compositional operators. This allows defining expressions to describe shapes, and transformation rules to shift/scale objects along the x or y dimensions. The transformations, when embedded in the pattern language, permit portions of an object to be shifted/scaled in arbitrary ways, to create a variety of similar objects. With appropriate quantification on the permissible shift/scale parameters, one obtains a language for querying similar shapes, compatible with the similar shape retrieval techniques of [J91].

Acknowledgments

Parts of this work was done while the second and third authors were visiting AT&T Bell Laboratories. The second and third authors were partially supported by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre.

References

[AB87] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. To appear in the Journal on Very Large Databases - VLDB. INRIA research report n. 846, 1987.

[AFS93] Rakesh Agrawal, Christos Faloutsos and Arun Swami. Efficient Similarity Search in Sequence Databases. In *Proc. Foundations of Data Organization and Algorithms (FODO) Conference*, Evanston, Illinois, Oct. 1993.

[AGMML90] S.F. Altschul, W. Gish, W.Miller, E.W. Myers and D.J. Lipman. Basic local alignment search tool. J. Mol. Biol. (England), 215(3):403-410, Oct. 1990

[BM92] C. Beeri and T. Milo. Functional and predicative programming in oodb's. In *Proc. 11th Symp. on Principles of Database Systems*, pages 176–190, San Diego, 1992.

[CRSV94] L.S. Colby, E.L. Robertson, L.V. Saxton and D. Van Gucht. A query language for list-based complex objects. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 179–189, Minneapolis, May 1994.

[FRM94] C. Faloutsos, M. Ranganathan, Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. Int. Conf. on Management of Data - SIGMOD '94*, pages 419-429, Minneapolis, May 1994.

[GCB92] G.H. Gonnet, M.A. Cohen and S.A. Benner. Exhaustive matching of the entire protein sequence database. Science, Vol. 256, 5 June 1992.

[GM95] S. Grumbach and T. Milo. An algebra for pomsets. In *Proc. Int. Conf. on Database Theory*, Springer Verlag, pages 191-207, Prague, 1995.

[GNU94] G. Grahne, M. Nykanen and E. Ukkonen. Reasoning about strings in databases. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 303–312, Minneapolis, May 1994.

[GW92] S. Ginsburg and X. Wang. Towards a unified approach to querying sequenced data. In *Proc. 11th ACM Symp. on Principles of Database Systems*, pages 293–300, San Diego, 1992.

[J91] H.V. Jagadish. A retrieval techniques for similar shapes. In *Proc. Int. Conf. on Management of Data – SIGMOD '91*, pages 208–217, Denver, Colorado, 1992.

[Ric92] J. Richardson. Supporting lists in a data model (a timely approach). In *Proc. 18th Intl. Conf. on Very Large Databases*, Vancouver, 1992.

[SLR94] P. Seshadri, M. Livny and R. Ramakrishnan. Sequence query processing. In *Proc. Int. Conf. on Management of Data – SIGMOD '94*, pages 430–441, Minneapolis, May, 1994.

[U88] J.D. Ullman. *Database and Knowledge Base Systems.* Computer Science Press, 1988.