# On the Efficient Evaluation of Relaxed Queries in Biological Databases

Yangjun Chen[1], Duren Che[2], Karl Aberer[3]
[1]Dept. of Business Computing, Uni. of Winnipeg, Canada R3B 2E9
[2]Dept. of Computer Science, Southern Illinois University, U.S.A. IL 62901
[3]Distributed Information Systems Laboratory EPFL-DSC, Lausanne, 1015 Switzerland

**Abstract** In this paper, a new technique is developed to support the query relaxation in biological databases. Query relaxation is required due to the fact that queries tend not to be expressed exactly by the users, especially in scientific databas s such as biological databases, in which complex domain knowledge is heavily involved. To treat this problem, we propose the concept of the so-called fuzzy equivalence classes to capture important kinds of domain knowledge that is used to relax queries. This concept is further integrated with the canonical techniques for pattern searching such as the position tree and automaton theory. As a result, fuzzy queries produced through relaxation can be efficiently evaluated. This method has been successfully utilized in a practical biological database - the GPCRDB.

**Categories & Subject Decriptors:** H.2.4

**General Terms:** Algorithms, Performance, Theory

**Key Words:** Query optimization, Biological databases, Query relaxation, Position trees, Automaton

## 1. Introduction

Biological databases are among the most important classes of scientific databases. A variety of biological databases have been developed that provide database support for both the research and the application in different biological disciplines. Well-known examples include GDB [PMFR92], GenBank [NCBI92, WZ98]], GSDB (Genome Sequence Data Base), GCRDb [GC98], GPCR mutant database [GP98] and GPCRDB [GPCR98]. Protein or DNA sequence data are the primary data that reside in these databases while various related data such as annotations, mutant information, physical-chemical characteristics are often added to the databases along with the main protein sequences. All these systems are equipped with information retrieval to help biologists to solve their own problems.

In this paper, we address an interesting issue: query relaxation and optimization. Query relaxation is important due to the fact that a query submitted to the system is usually domain knowledge related and a user often fails to appropriately formulate his/her problem to obtain all relevant sequences and relevant data. In such a case, it is desired that some assistance is offered by relaxing a query or providing helpful information to guide the user to express his/her problem more exactly. For example, consider a biologist who is issuing a key word query, say *ARCH*, to find all those DNA sequences related to it. If the result does not match what he/she wants, another key word, say *Adrenocorticotropin*, may be chosen to enquire the database once again. This behaviour is simulated by the query relaxation of key words along a *thesaurus hierarchy* built in the system. As another example, consider a pattern query, say P*S*E. If the search fails, he/she may use a different pattern, e.g., G*S*E, which is believed to be close to the original one according to biology. Also, this behaviour is modeled by the query relaxation of pattern queries in our design. Furthermore, the sequences obtained by evaluating a query can be explained by imposing a tree structure on them according to the *family tree* maintained in the system, which helps a biologist understand what has been obtained.

Although the above functionalities are important to biological research, less attention has been paid to them. We have recently checked several famous systems such as WPDB [WPDB02], NDB [NDB02] and SWISS-Prot [SWISS02]. All of them provide powerful sequence retrieval on different aspects of DNA structures; but the query relaxation has not been touched at all. We have also noticed the databases described in [AOA01, BS02], which are designed for DNA-binding protein motifs, and for bacterial lipoproteins and lipid modification, respectively. No discussion on query relaxation is conducted in them, either.

However, there are a lot of research on query relaxation or expansion in information retrieval research community with different methods from ours. In [YK98], a sort of query relaxation is carried out by loosening the searching condition in the sense of database theory. In [KJ98, JKN96], queries are expanded in terms of thesauri, by which narrower partitive concepts can be replaced with hierarchically broader generic concepts. A third kind of query expansion is based on an interactive approach, by which users select the terms of interests from a list to add to the initial queries [MSB98, Co00, Ef00]. In this paper, we consider the similarities among different concepts: similarity among key words and similarity among residue types. For the first kind of similarities, two hierarchies are constructed: *family tree* and *thesaurus hierarchy*. Then, the relaxation can be done bottom-up along the tree structures.


CIKM'02, November 4-9, 2002, McLean, Virginia, USA.


[1]The author is partly supported by UW-CIHR grant (CIHR - Canadian Institutes for Health Research).

This is similar to the methods proposed in [KJ98, JKN96]. For the second kind of similarities, a new concept of the so-called *fuzzy equivalence classes* is developed to capture the relationships among residue types. In particular, this concept can be integrated with the canonical techniques such as the position tree [We73, AHU74, CR94] and automaton theory [HU69] and therefore the relaxation can be performed in an efficient way. The contribution of this paper is the following:

(1) Family tree and thesaurus hierarchy are built to support the key word relaxation.

(2) A new concept of the fuzzy equivalence classes is proposed to capture the similarities among residue types.

(3) The technique of the position tree is extended to evaluate regular expressions, especially to evaluate queries expressed as fuzzy regular expressions. By simulating a Deterministic Finite Automaton (*DFA*) of a (fuzzy) regular expression on a position tree, high performance can be achieved. We argue that the cost of converting a regular expression into a *DFA* will not influence the time complexity of the pattern matching if the size of the regular expression is very small compared to the sequence considered.

(4) The technique mentioned above has been successfully used in our smart engine embedded in a practical database system GPCRDB [GPCR02], which can be invoked through the world wide web.

The remainder of the paper is organized as follows. First, in Section 2, the biological database special for protein sequences is formally described. Then, in Section 3, the relaxation processes for both key words and residue types are discussed. Next, we address the optimization problem in Section 4. Section 5 is devoted to the system implementation and Section 6 is on the experiment to compare different query evaluation strategies. Finally, a short conclusion is set forth in Section 7.

## 2. Biological databases

From an abstract point of view, a biological database is a set of protein sequences of letters from the protein alphabet $\Re$ = {A, ..., Z} (letters B, J, O, U, X and Z are excluded from $\Re$), in which each letter stands for a "residue type". A biologist may issue a query against the database to get knowledge on the relevance of subsequences to a protein structure.

Formally, a biological database can be defined as a triple of the form: $<S, H_F, H_G>$, where $S$ is a set of protein sequences belonging to $\Re^*$ ($\Re^*$ denotes the set of all finite-length sequences of letters from $\Re$), $H_F$ is a family tree imposed upon $S$ and $H_G$ is a thesaurus hierarchy.

For illustration, see a possible set of protein sequences shown below.

$s_1$ = ADCFGKLOHGK ... ...
$s_2$ = DDCHFGKLNHGK ... ...
... ...
$s_n$ = ... ...

In terms of the biological classification, $S$ can be partitioned into several subsets which, in turn, can be organized into a hierarchy, called the *family tree* and denoted $H_F(S)$. Fig. 1 shows a fragment of the family tree stored in our GPCR database [GPCR02].

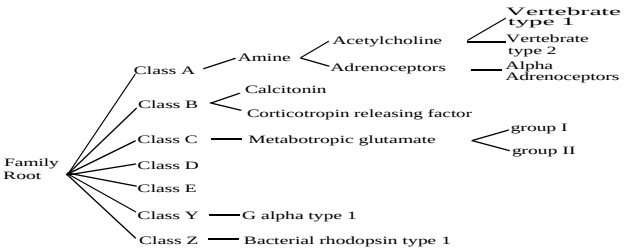



Fig. 1. A Fragment of the GPCR family tree

In the figure, each leaf node of a family tree is a type corresponding to a subset of $S$; and each internal node corresponds to a biological concept, representing a subclass which may further be partitioned into several smaller subclasses. Such a family tree helps a biologist understand the meaning of a subsequence.

In addition, we can also partition $S$ according to the thesaurus, leading to another hierarchy, called the *thesaurus hierarchy* and denoted $H_G(S)$. See Fig. 2 for illustration.

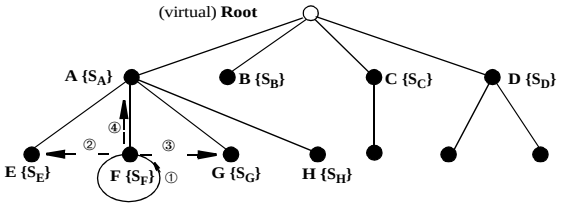



Fig. 2. Illustration of the GPCRDB thesaurus hierarchy

In Fig. 2, instead of giving a concrete example, we illustrate the generalization (thesaurus) hierarchy using an abstract representation. Let **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**, ... stand for the terms in this thesaurus, and $\mathbf{S_A}$, $\mathbf{S_B}$, $\mathbf{S_C}$, $\mathbf{S_D}$, $\mathbf{S_E}$, $\mathbf{S_F}$, $\mathbf{S_G}$, $\mathbf{S_H}$, ... for the synonym set of term **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**, ..., respectively. For each main entry in the thesaurus, there is a corresponding node in the generalization hierarchy, and a set of synonyms of the term are associated. Following are several term examples and their synonym sets:

**A** = *Melanocortin*
$\mathbf{S_A}$ = { };
**E** = *ARCH*
$\mathbf{S_E}$ = {*Adrenocorticotropic homone*, *Melanocortin-2*};
**F** = *Adrenocorticotropic homone*
$\mathbf{S_F}$ = {*ARCH*, *Adrenocorticotropin*};
**G** = *Melanocortin-2*
$\mathbf{S_G}$ = {*ARCH*};
**H** = *Adrenocorticotropin*
$\mathbf{S_H}$ = {*Adrenocorticotropic homone*};

A similarity relationship is defined between a pair of terms if one is a synonym of the other. In addition, a node in $H_G$ represents a generalized concept of the concepts represented by its child nodes. For the purpose of query relaxation, we may further indicate the similarities among the nodes that have the same parent. Obviously, the generalization hierarchy as a whole represents the similarities of all

terms in a biological database. It can be used to govern the first kind of query relaxation: the key word query relaxation.

In fact, organizing data as above covers main complex inter-relationships of data: aggregation (similar to the family tree), generalization (the thesaurus hierarchy when seen from an epistemological point of view) and set construction.

For the purpose of the biological research, a sequence $s_i \in S$ is typically partitioned into several subsequences with each reflecting one of $s_i$'s features; and very often only such subsequences are inquired. (How to partition a sequence is determined by biologists. In fact, such a subsequence corresponds to a so-called "secondary structure" of the protein sequence, to which a specific biological function is associated. In particular, it is also named using a character sequence with some biological meaning.)

Therefore, $s_i$ is represented as

$$s_i = s_{i1}s_{i2} \ldots s_{in_i},$$

where each $s_{ij}$ represents a subsequence.

Finally, due to the similarity among the residue types, we partition alphabet $\Re$ into several fuzzy equivalence classes (*FEC*s) $I_1$, ..., $I_m$ for some $m \geq 1$. The membership function of $I_k$ ($k$ = 1, ..., $m$) is defined as follows [Co93]:

$$f: 2^{\Re} \rightarrow [0, 1].$$

The following are several examples of *FEC*s.

$I_1$: {A}; $f$({A}) = 1.
$I_2$: {S, T}; $f$({S, T}) = 0.9.
... ...
$I_4$: {P, G}; $f$({P, G}) = 0.7.
$I_5$: {E, D, W}; $f$({E, D, W}) = 0.8.
... ...

Here, an equation such as $f$({E, D, W}) = 0.8 indicates that 'E', 'D' and 'W' are similar to each other to extent 0.8. By replacing each letter in a subsequence $s_{ij}$ with the fuzzy equivalence class, to which it belongs, we will obtain an *FEC-subsequence*, denoted $FEC(s_{ij})$. Accordingly, the *FEC-sequence* for $s_i$ can be represented as follows

$$FEC(s_i) = FEC(s_{i1}) \ldots FEC(s_{in_i}).$$

*FEC*s are used for the second kind of query relaxation: the pattern query relaxation.

In this paper, we consider two kinds of queries. They are

- key word queries and
- pattern queries expressed as a regular expression or as a fuzzy regular expression (see below).

For evaluating a query containing key words, $H_G$ will be searched and $H_F$ may be navigated by the user according to the query results for the relaxation purpose.

To find the sequences matching a pattern, we build a position tree [We73, AHU74, CR94] over each protein subsequence to speed up the evaluation.

An important issue is how the system reacts and offers any help to the user when a user cannot formulate his/her problem exactly.

To this end, we develop a relaxation technique in the sense that a key word in a query can be replaced with its synonym or its hypernym along $H_G$; and a fuzzy equivalence class is substituted for a letter (standing for a residue type) appearing in the regular expression.

From now on, we first concentrate on the relaxation of key words (along $H_G$) and residue types. Later, how to optimize the evaluation of fuzzy regular expressions will be discussed in detail.

## 3. Relaxation

In our system, the query is of the form: $(c_{11} \wedge \ldots \wedge c_{1i_1}) \vee \ldots \vee (c_{j1} \wedge \ldots \wedge c_{ji_j})$, where each $c_{kl}$ is a basic query form to be discussed. We distinguish between two kinds of query forms: those containing key words and those expressed as a regular expression. In this section, we discuss the relaxation technique for both of them. According to the structures of $H_F$, $H_G$ and *FEC* sequences, we developed two kinds of query relaxation techniques: relaxations of key word queries and pattern queries, which will be discussed in 3.1 and 3.2, respectively.

### 3.1 Relaxation of key word queries

The key word query form is expressed just as a key word *kw*.

For such a query form, the relaxation is very simple. We calculate the level numbers for all nodes in a hierarchy $H_G$. (The root is level 0.) If a key word *kw* is issued, all the sequences below it in $H_G$ will be returned. If the expected results do not come out, the query will be relaxed in the following way. Assume that *kw* is at level *k*. We will find a node *v* that has the same parent as *kw* in the hierarchy. (If the similarities among siblings are specified, we'll chose node that is the closest to *kw*.) Then, all the sequences below *v* will be output. In a next loop, another sibling (a sibling that is secondly closest to *kw*) will be considered. This process repeats until the expected results are obtained or all the siblings of *kw* are visited. The relaxation may be expanded to other nodes at level *k* but with different parents if necessary. During the relaxation, the results obtained in each step can be grouped in terms of the family tree $H_F$. That is, a tree structure is imposed on the results so that a user can traverse that tree to find what he/she wants from the results.

**Example 3.1.** Consider a key word query: 'ARCH'. We first search $H_G$ (see Fig. 2) to see whether this key word exists. If it is the case, all the sequences associated with this key word will be returned. If the user cannot find what he/she wants from the result returned, the first relaxation will be made, i.e., the synonym set of this key word will be checked. We may do the second relaxation by checking one of its sibling if necessary. This process repeats until the user finds what he/she expects, as depicted in Fig. 2: first ① is tried, then ②, next ③ and last ④. This method can be extended to ontologies; but more complicated control is needed because an ontology is normally organized into a directed graph [ABB00], where a node represents a concept and an edge from *a* to *b* represents that *b* is a direct sub-concept of *a*.

During the process, the sequences obtained are grouped in terms of $H_F$, which provides the user a classification of the sequences visited. For instance, if the sequences obtained belong to 'Alpha Adrenoceptros', 'group II' and 'G alpha type 1' (see Fig. 1), respectively, a sub-family-tree will be constructed automatically as shown in Fig. 3. This helps a user understand the meaning of the sequences obtained.

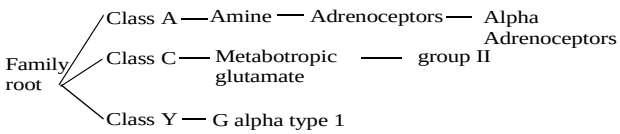



Figure 3. A sub-family-free automatically constructed

### 3.2 Relaxation of pattern queries

A pattern query form is represented as a pair of the form: <*ss*, *E*> or <_, *E*>, where *ss* is a subsequence name, *E* is a regular expression representing a pattern and "_" means "don't care". If the query form is <_, *E*>, the entire database will be searched to find those sequences containing *E*. If <*ss*, *E*> is submitted, *ss* will be checked to see whether it contains *E*. If so, all the sequences containing *ss* as a subsequence will be output. The following are two examples of pattern queries:

<gggg, P*S*E>,

<gfgh, (A+D)*CI*>,

where "gggg" and "gfgh" are two subsequence names and P*S*E and (A+D)*CI* are two regular expressions (see [AHU74]).

For the relaxation purpose, we define the following three concepts.

**Definition 3.1** (*fuzzy regular expression*) Let *E* be a regular expression. Let $l_i$ ($i$ = 1, 2, ..., $m$) be the different letters appearing in *E*. By replacing each $l_i$ with its fuzzy equivalence class $I_{k_i}$, we get another regular expression *E*', called the fuzzy regular expression (*FRE*).

**Definition 3.2** (*mixed regular expression*) Let *E* be a regular expression. Let $l_i$ ($i$ = 1, 2, ..., $m$) be the different letters appearing in *E*. By replacing some $l_{i_j}$'s ($j$ = 1, ..., $h$; $h < m$) with the corresponding fuzzy equivalence classes, respectively, we get another regular expression *E*', called the mixed regular expression (*MRE*).

Note that each fuzzy equivalence class *I* is associated with its membership function value *f*(*I*), based on which the value for a mixed (fuzzy) regular expression can be defined.

**Definition 3.3** (*value of mixed regular expression*) Let *E* be an *MRE* (*FRE*). Let $I_i$ ($i$ = 1, ..., $h$) be the fuzzy equivalences appearing in *MRE*(*FRE*). Then the value of the *MRE*(*FRE*) is defined to be min{$f(I_1)$, ..., $f(I_h)$}, denoted *v*(*MRE*) (*v*(*FRE*)).

Based on this definition, the relaxation of a pattern query can be specified as follows.

Let *M* be the set of all *MRE*s generated from a regular expression to be evaluated. We sort *M* such that all elements in *M* are non-increasingly ordered. Assume that $M_1$, ..., $M_h$ are ordered *MRE*s in *M*, which can be dynamically produced and evaluated during a relaxation process. Below is its simplified algorithm description.

**Algorithm** *pattern-query-relaxation*
{$i \leftarrow 1$; *success* $\leftarrow 0$;
**repeat**
evaluate $M_i$;
**if** result is O.K. **then** *success* $\leftarrow 1$
**else** $i \leftarrow i + 1$;
**until** *success* = 1
}

**Example 3.2.** Assume that we have a set of *FECs* as shown in Section 2. For the regular expression: P*S*E, we'll have the following *MRE* sequence: P*$I_2$*E, P*S*$I_5$, P*$I_2$*$I_5$, $I_4$*S*E, $I_4$*S*$I_5$, $I_4$*$I_2$*E, $I_4$*$I_2$*$I_5$ with $v$(P*$I_2$*E) = 0.9, $v$(P*S*$I_5$) = $v$(P*$I_2$*$I_5$) = 0.8 and $v$($I_4$*S*E) = $v$($I_4$*S*$I_5$) = $v$($I_4$*$I_2$*E) = $v$($I_4$*$I_2$*$I_5$) = 0.7. Using the above algorithm, the relaxation of the pattern query: P*S*E will be done along the above *MRE* sequence, i.e., first retrieve P*$I_2$*E, then P*S*$I_5$, and so on.

In the next section, we will discuss the evaluation optimization of the pattern queries expressed as *MREs* (*FREs*).

## 4. Optimization of pattern queries

Two kinds of the optimization have been made in our implementation. First, the semantic optimization is conducted in a conventional way. That is, the order of the conjunctive and disjunctive basic query forms in the original query will be changed so that the key word queries are always evaluated first. Then, the pattern queries with subsequence names will be executed. Next, on the results returned from the previous execution, the pattern queries with no subsequence names will performed. Since the relevant techniques are available, we do not discuss them any further here. Below we concentrate only on the (fuzzy) pattern matching and attempt to extend *position trees* for both the regular expression and the mixed (fuzzy) regular expression searching involving fuzzy equivalence classes. We first show the algorithm for the regular expression searching based on the position tree in 4.1. Then, in 4.2, the fuzzy regular expression searching will be addressed.

### 4.1 Regular expression searching

The position tree is an index structure built over long sequences. It is well-known that this technique can be used to expedite the simple substring matching. But it has never been discussed how to utilize it for a regular expression searching in the literature. Before the algorithm is described, two concepts concerning the string matching have to be defined: *position* and *position tree* (see [AHU74]).

**Definition 4.1** (*position*) A *position* in a string of length *n* is an integer between 1 and *n*. We say letter *c* occurs in position *i* of string *s* if $s = s_1 c s_2$, with $|s_1| = i - 1$. We say substring *u* identifies position in string *s* if $s = s_1 u s_2$, with $|s_1| = i - 1$, and cannot be written as $s_1' u s_2'$ unless $s_1' = s_1$. That is, the only occurrence of *u* within *s* begins at position *i*.

For example, the substring ADDA identifies position 1 of the string ADDACADD. But the substring ADD does not identify position 1.

To make each position in a string *s* over some alphabet ***A*** uniquely identifiable in terms of Definition 4.1, we attach a special symbol not in *A*, say $, to the end of the string. Then each position *i* of *s*$ can be identified by at least one string. We call the shortest string which identify position *i* in *s*$ the *substring identifier* for position *i* in *s*, denoted PI(*i*).

**Example 4.1** Consider the string ADDACADD$. The substring identifier for position 1 through 9 are shown in Fig. 4(b).

**Definition 4.2** (*position tree*) A position tree for a string $s\$ = a_1 \ldots a_n a_{n+1}$, where $a_i \in \boldsymbol{A}$ ($\boldsymbol{A}$ is an alphabet) ($1 \le i \le n$) and $a_{n+1} = \$$, is a tree $T$ such that:

1. For each internal node of $T$, the edges leaving it have distinct labels in $\boldsymbol{A}$.
2. $T$ has $n + 1$ leaves labeled 1, 2, ..., $n + 1$. The leaves of $T$ are in on-to-one correspondence with the positions in $s\$$.
3. The sequence of labels of edges on the path from the root to the leaf labeled $i$ is PI($i$), the substring identifier for position $i$.

**Example 4.2** The position tree for the string ADDACADD$ is shown in Fig. 4(a). We notice that the path from the root to the leaf labeled 6 spelled out ADD$, which is the substring identifier for position 6.

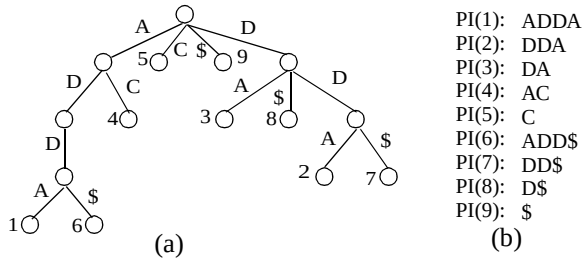



Fig. 4. Position tree and substring identifiers

(Position trees are also called *suffix trees* in literature [McC76, MM96, Uk92].) How to construct a position tree efficiently is not trivial and a sophisticated method should be used, which is discussed in detail in [We73, AHU74, McC76, CR94]. However, we can build the position trees off-line and prepare them beforehand. Besides, a linear space complexity (and thus a linear time complexity) can be achieved through constructing compact position trees [AHU74, CR94]. This can be obtained by avoiding one-way branching through including in each internal node the number of letters to skip over before making the next test as done for pat-trees [Mo68, Kn73, GBS92]. Therefore, no much storage overhead arises. (For the sake of exposition, however, only the normal position tree is shown here to make the main idea of our algorithm clear.)

In order to use the position tree to expedite a *DFA* matching, we change the position tree a bit. That is, each leaf node is not associate with a point, rather a pair of the form: ($po$, $h$), where $po$ is the point to a substring identifier as defined above and $h$ is the length of the path from the root to the node.

The main steps of the algorithm can be described as follows.

(1) Transform the regular expression $p$ issued as a query into a minimized deterministic automaton (*DFA*), which can be performed in two steps. First, we convert $p$ into a nondeterministic finite automaton (*NDFA*) $M$. Then, in the second step $M$ is transformed into $M'$, a minimized *DFA*. Next eliminate the outgoing transitions from the final states since they are useless for the pattern matching (see Example 4.3).

(2) Traverse the position tree of the considered protein substring $s$. On reaching a node $e$, some state $i$ of the *DFA* will be associated with it. Then, the label of an edge incident to $e$ will be checked against the label of a transition leaving $i$. At the very beginning, we associate the root of the position tree for $s$ with the initial state of the *DFA*; and for any internal node $e$, if it is associated with some state $i$ we associate any of its child nodes $g$ with state $j$ if the transition $i \rightarrow j$ (in the *DFA*) and the edge $e \rightarrow g$ (in the position tree for $s$) are labeled with the same letter.

(3) If a node $e$ is associated with an "unsuccessful" state (see below), stop the search of that subtree and make a backtracking. That is, another child of $e$'s father node will be checked.

(4) If a node of the position tree is associated with a final state, accept the whole subtree, terminate the search in that subtree and if desired, make a backtracking to try more matching (or report a "successful matching" and stop the search process.)

(5) If a leaf node $e$ is encountered, traverse the remainder of the *DFA* on the substring that begins at position $po + h$, where ($po$, $h$) is the pair associated with $e$. If a final state can be reached, report a "successful matching". Otherwise, a backtracking will be made. In other words, another child of $e$'s father node will be checked (against, of course, another transition in the *DFA*.)

**Example 4.3** Now we make a sample trace step by step to demonstrate how the algorithm works.

- pattern: (D*+C)A
- protein substring: ADDACADD (which will be changed to ADDACADD$.)

The first step of the algorithm is to transform (D*+C)A into a *DFA* which is shown in Fig. 5 (a). (See [AHU74] for *DFA* construction.)

In the figure, $t_0$ and $t_f$ are the initial and the final state, respectively. In addition, $t_3$ has no outgoing transition. It represents an "unsuccessful" state. All the outgoing transitions from the final states will be removed. Then, we have the *DFA* as shown in Fig. 5(b).

In the second step, both the *DFA* for the regular expression and the position tree for the string will be traversed. The position tree for ADDACADD$ is shown in Fig. 4(a).

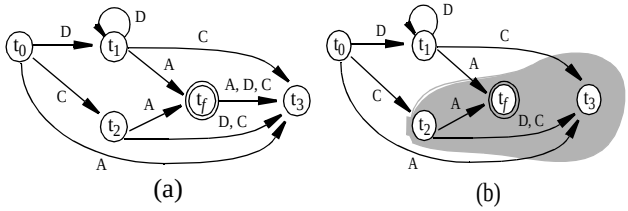



Fig. 5. *DFA* for (D*+C)A

The search begins from the position tree root *r* by associating it with the initial state $t_0$ of the *DFA*. Let *e* be a child node of *r*. Let $t_i$ be a state reachable directly from $t_0$. If $r \rightarrow e$ *and* $t_0 \rightarrow t_i$ are labeled with the same letter, then associate *e* with $t_i$. In a next step, we examine one of *e*'s children and try to find a subsequent state of $t_i$ to which it can be associated in the same way as for *e*. This process repeats until one of the following three cases is encountered.

(i) an "unsuccessful" state is met.

(ii) a final state of the *DFA* is met.

(iii) a leaf node of the position tree is reached.

If case (i) happens, the third step of the algorithm will be executed, by which a backtracking will be made. For instance, if we traverse the edge (labeled with A) incident to *r*, reaching to a node which will be associated with $t_3$, the "unsuccessful" state. In this case, the control will be switched over to *r* to try another edge, say the edge labeled with C.

If (ii) is the case, the fourth step of the algorithm will be performed, by which "successful matching" is reported. For the given pattern and the string, if the path from the root to the leaf node labeled with 2 is followed, the leaf node will be associated with the final state of the *DFA*, which indicates that the pattern matches the string.

In case (iii), the fifth step of the algorithm will be conducted, by which the substring from the position indicated by the leaf node will be searched against the remainder of the *DFA*. For the given pattern and the string, if the edge (labeled with C) incident to *r* is visited, the node labeled with 5 will be associated with state $t_2$. Since this node is a leaf, the substring from position $p + h$ (= 5 + 1) will be searched against part of the *DFA* as shown in Fig. 5(b) (the portion marked gray in Fig. 5(b) will be traversed.) The letter at position 6 is A. It matches the label of the transition $t_2 \rightarrow t_f$ and thus "successful matching" will be reported.

The following is the formal description of the algorithm, by which a stack is used. Each item of *stack* is a pair of the form: (*v*, *t*), where *v* is a node of the position tree and *t* is a state of the *DFA*.

**Algorithm** *pattern-matching*
{ transform the pattern *p* into a *DFA*;
  let *r* be the root of the position tree;
  let $t_0$ be the initial state of the *DFA*;
  push (*r*, *t*) into *stack*;
  **while** *stack* is not empty **do**
    {(*v*, *t*) ← pop(*stack*);
      **if** *t* is an unsuccessful state then **do** nothing
      **else if** *t* is a final state **then** report "successful matching"
        **else if** *t* is a leaf node **then**
          {traverse the remainder of the *DFA* on the subsequence that begins at *po* + *h*, where (*po*, *h*) is the pair associated with *v*;
          **if** a final state is encountered **then** report "successful matching";}
          **else**
            {let $v_1$, ..., $v_i$ be child nodes of *v*;
              let $t_1$, ..., $t_j$ be child nodes of *t*;
              put all ($v_{i_k}$, $t_{j_l}$) into *stack* if $v \rightarrow v_{i_k}$ and $t \rightarrow t_{j_l}$ have the same label;
            }}}

The main part of the algorithm consists in a depth-first search of the position tree. During a traversal, if the *DFA* contains no cycles each edge of the *DFA* is visited at most once. Thus, in this case, the time complexity of the above algorithm is bounded by O(|*DFA*|). If the *DFA* contains back edges (a back edge is an edge like $t_1 \rightarrow t_1$ shown in Fig. 5(a)) and therefore cycles, the time complexity of the worst case is $O(2^{|P|} + |s|)$, where |*s*| represents the length of sequence *s*, and |*P*| represents the number of letters in the pattern *P*. (Even if it is the case, for the short regular expressions, the scanning using the *DFA* is still better than the sequence scanning using a normalized nondeterministic automaton [CR94], which requires $O(|P| \cdot |s|)$ time.) To mitigate the time complexity, we use the following methods:

1. Using Tarjan's algorithm for identifying strong connected components [Ta72] to check the cycles appearing in the *DFA*.
2. Using the word periodicity theory [GM95] to mark the cyclic words for each path of the position tree (and the subsequences) so that the back edges appearing in the *DFA* can be skipped over.

Thus, the time complexity of the above algorithm can be reduced to O(|*DFA*|) if the time spent for converting a regular expression into a *DFA* is not taken into account. In fact, if the regular expression is small (compared to the sequence involved), which happens very often in the GBCRDB, the transformation will not degrade the entire time complexity although it may take exponential time theoretically.

### 4.2 Fuzzy regular expression searching

The technique discussed in the previous subsection can be easily extended to treat fuzzy (mixed) regular expressions. Concretely, only two simple modifications of the strategy shown in 4.1 are needed.

First, for relaxing a pattern, the letters labeling some transitions of the corresponding *DFA* will be replaced with their respective fuzzy equivalence classes. The resulting *DFA* is called the *relaxed DFA*.

Secondly, the third step of the above algorithm should be accordingly changed. That is, by simulating a transition *a* of the *DFA* on an edge *b* in the position tree, we check whether the label $\alpha$ of *a* contains the label $\beta$ of *b* if $\alpha$ is a fuzzy equivalence class. Otherwise, the equivalence between $\alpha$ and $\beta$ will be checked.

The following example helps for illustration.

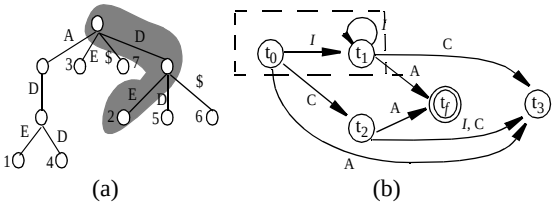
**Example 4.4** Now we consider a new data setting as follows:

- pattern: (D*+C)A
- protein sequence: ADEADD$

Since the pattern does not match the protein sequence, the system will answer "no". The position tree for ADEADD$ is shown in Fig. 6(a). In this case, the relaxation may be invoked.

Assume that $I$ = {D, E} with $f(I)$ = 0.7 and at a certain step

during the relaxation (see Example 3.2 to understand a relaxation step) the mixed regular expression: (*I**+C)A is produced (automatically) and issued to the query evaluation component. Then, the *DFA* of (D*+C)A will be changed as shown in Fig. 6(b), in which each D is replaced with *I*.




(a) (b)

Fig. 6. Position tree and relaxed *DFA*

While simulating transitions in the portion enclosed by the broken rectangle in the *DFA* shown in Fig. 6(b) on the part marked gray in the position tree shown in Fig. 6(a), the system will check whether *I* contains D or E. Since the checkings are successful, the final state of the *DFA* will be eventually reached and the system will report a positive answer with credibility 0.7.

From this example, we can see that the method works efficiently since we needn't reconstruct the relaxed *DFA* for the mixed regular expression. To achieve a constant time complexity for replacing a letter with its corresponding fuzzy equivalence class, we build an array $a[\mathrm{A} .. \mathrm{Z}]$ (not containing entries for B, J, O, U, X and Z), indexed by the letters in $\Re$. Initially, set $a[x] = x$ for each $x \in \Re$. We label the transitions of the *DFA* with $a[x]$ ($x \in \Re$ ) when it is constructed. During the relaxation, if a letter appearing in the regular expression, say A, should be replaced with its fuzzy equivalence class *I*, we put simply *I* into $a[\mathrm{A}]$. No extra operations are needed. Therefore, it requires only a constant time for this task.

Since the number of letters in $\Re$ is a constant, the checking whether a subset of $\Re$ contains a letter requires only a constant time. Therefore, the time complexity of a relaxation operation is $O(|I_1| + |I_2| ... + |I_m| + |DFA|)$.

## 5. System implementation

In this section, three important aspects concerning the implementation are presented. They are: system architecture, interface and sequence storage structure.

### 5.1 System architecture

Now we briefly outline the system architecture of GPCRDB query system. As is depicted in Fig. 7, the GPCRDB query system is designed to run on the world wide Web. On the client side, users formulate their queries by simply filling out the query forms embedded in HTML pages. When submitted, the queries are transmitted to the IUS Webdriver or to the smart query engine through the Web server on the server side over the Internet. (Here IUS stands for "Informix Universal Server"; see [IUS97].) The IUS Webdriver is provided by the Informix Inc. as a web datablade that works as a gateway between Informix databases and the Web server while the task of the smart query engine is to do the pattern matching and the query relaxation for both key words and regular expressions as described in the previous sections.

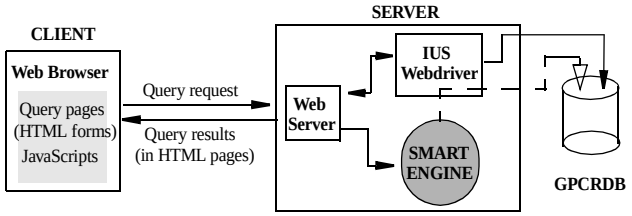



Fig. 7. GPCRDB Query System Architecture

The ordinary queries submitted to the Web server can be interpreted by the IUS Webdriver that accesses the GPCRDB database and returns evaluation results as Web pages to the client's Web browser. However, if a user activates the smart query engine explicitly, the control will be taken over by it from the IUS Webdriver, by which the following steps will be performed:

1. For the key word involved in query, the hierarchies $H_G$ will be taken from the database and the relaxation will be done in a way as described in 3.1.
2. For the pattern expressed as a regular expression, the substring and the corresponding tree will be taken from the database and the relaxation will be made as described in 4.1 and 4.2.

During the relaxation, the interference of users is allowed so that they can conveniently choose among multiple alternatives and provide the information on how to relax a query if the system default method is not desired.

### 5.2 Interface

The GPCRDB query system allows users to query the GPCR database through a variety of ways: "family", "keyword", "secondary structure" (subsequence), "ligant" (binding information), "mutants". In addition, an advanced (compound) query page is provided for formulating query expressions consisting of up to six different conditions that can be logically interconnected with "$\wedge$" or "$\vee$", constituting an "and-or" normal form. For example, if $c_1 \vee c_2 \wedge c_3$ is input, it will be treated as $c_1 \vee (c_2 \wedge c_3)$. That is, "$\wedge$" takes precedence over "$\vee$". The interface shown in Fig. 8 contains a query:

*Find GPCR sequences that have the keyword 'ARCH' (contained in any one of the annotation fields; to each sequence a number of annotation fields are attached) and contain the pattern 'R*SS'; furthermore, the secondary structure (subsequence) "N-terminus" of the sequences contain the pattern 'PP'.*

The first response of the query system to an input query is a summary page that presents to the user some links to various pages containing more details about all hits of the query (the number of the sequences satisfying the query). Fig. 9 is an example of such summary pages. This summary page also contains a button at the bottom that the user can use to invoke the smart query engine to relax a query if he/she wants more relevant results. Users can activate the relaxation process of their queries by making additional selections predefined by clicking an option button that allows them to specify one or two facets (of their queries) that should not be relaxed. Currently, three kinds of relaxation (operating on different facets of a query) are supported, keyword (concept) generalization, residue pattern

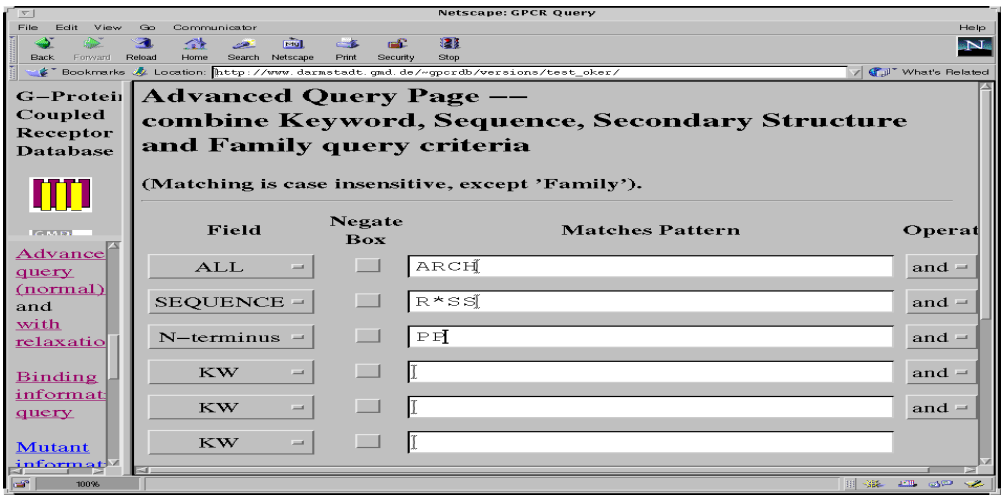



Fig. 8. The advanced query page (form) of GPCRDB

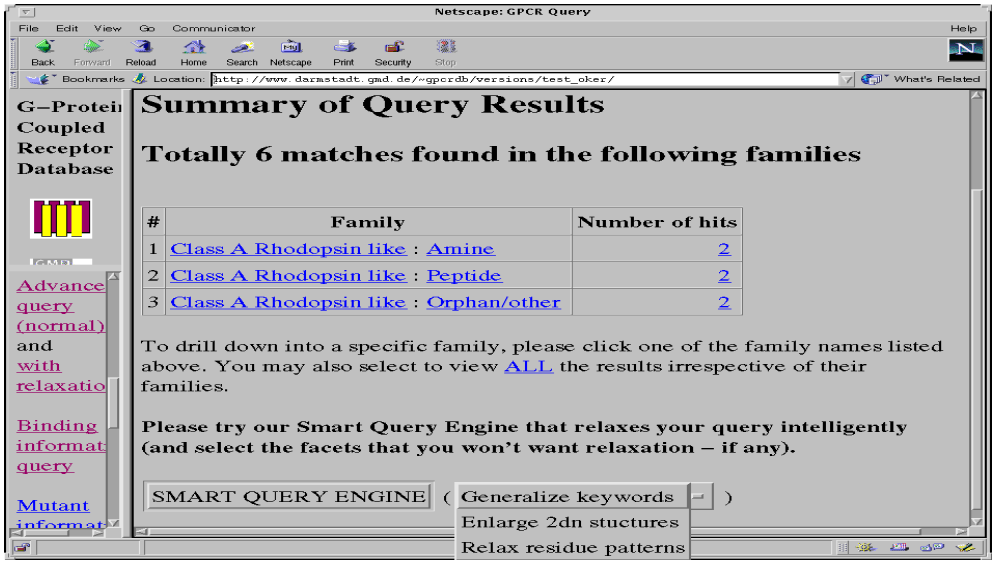



Fig. 9. Result page of ordinary query (without relaxation)

relaxation and secondary structure's border expansion (which is not discussed since the technique involved is relatively trivial.) By default, all the three kinds are to be performed on a query. When the user does activate the smart query engine, the system will return a first alternative (query results) of relaxation of his/her query, as is illustrated in Fig. 10.

With the same input query as in Fig. 9, this page contains a much enlarged set of matches (68 hits *vs.* the original 18 hits in Fig. 9). This page also presents the original query predicate and the relaxed one (just executed) as an explanation to the achieved relaxation. This helps the user not only to understand result of the query relaxation but also to formulate more pertinent queries for query relaxation in a subsequent step. In the presented relaxed query predicate (see Fig. 10), expressionrp('*R*SS*', '*R*[ST][ST]*', 2) stands for that input residue pattern '*R*SS*' is relaxed as'*R*[ST][ST]*' using the fuzzy equivalence class numbered 2 (i.e., residue type 'S' is replaced in this pattern by its fuzzy equivalence class '[ST]'), and ss(DOMAIN1) indicates that each of the two borders of secondary structure DOMAIN1in this query is expanded by one residue position (while ss(ss(DOMAIN1)) means the relaxation is performed two times).

### 5.3 Sequence storage structure

The data to be handled in the GPCRDB involve various, domain-specific and non-standard data types. These include protein sequences, secondary structure motifs (subsequence variants), and hierarchical classifications of the GPCR proteins (i.e., the so-called family trees), mutant and ligant (binding information), and other chemical/physical characteristics. These data can be classified as two types: the primary data, i.e., the GPCR sequence data, and GPCR-related data. Most of the GPCR related data are annotations made by the domain experts, such as keywords, descriptors, ligand (binding information), and mutant data. Typical queries issued by the intended users indicate a search in the database for particular sequences, e.g., those containing a given residue pattern in the sequences or in a specific range (secondary structure) of the sequences. GPCR related data may be queried also with imposed conditions as additional search criteria for sequences. But as long as the primary data, the sequences, have been identified, other related data of the sequences can be easily obtained through the IDs (identifiers) of the identified sequences. Therefore, in the following, we only show the mains storage structure of sequences which is depicted in Fig. 11.

For each sequence, a unique Sequence ID, a number of keywords, the primary sequence data, and 16 substructures (subsequences) are logically defined. A sequence is physically decomposed as 16 substructures or subsequences (called secondary structures in biology). They are separately stored in 16 buckets, S1, S2, ..., S16. This matrix storage structure can efficiently support the two primary types of sequence matching indicated by the queries that are most frequently issued by the intended users of GPCRDB:

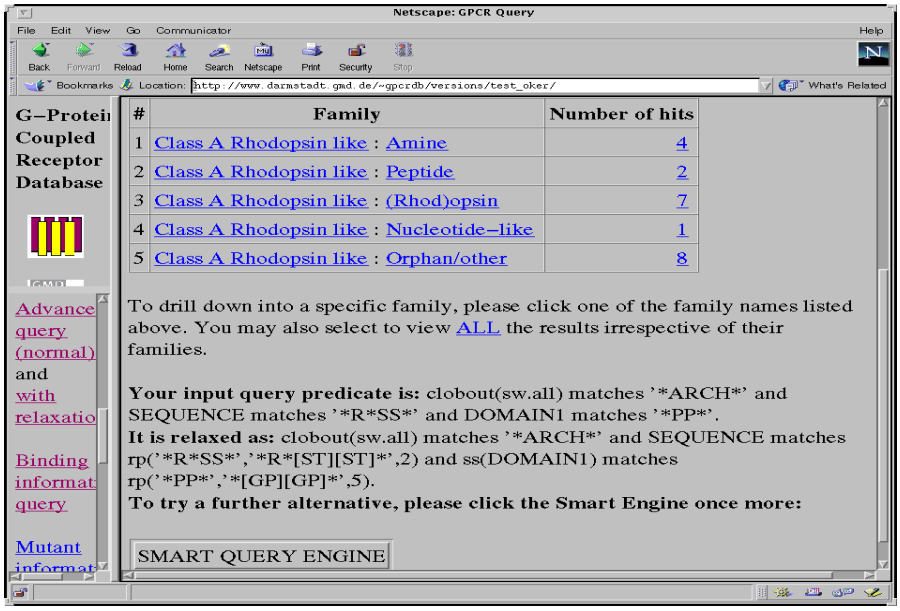



Figure 10. Result page of a relaxed query returned by GPCRDB Smart Query Engine

1. Search for sequences that contain a given residue pattern (specified as a regular expression).
2. Search for sequences that contain a given residue pattern in a specific substructure (a secondary structure) of the sequences.

In the main database table (see Fig. 11), the "sequence" field simply holds the internal sequential number assigned by the system for each sequence. Using this number, the system can easily locate the 16 subsequences and quickly construct the sequence data (as a whole) using the 16 subsequences when those are needed to perform entire sequence matching. The matrix storage structure greatly facilitates quick pattern-matching of sequences as a whole and any of the 16 subsequences. In our system, pattern search in subsequences is even more frequently required for the evaluation of user queries.

If a query initiates a pattern matching with entire sequences, the 16 subsequence buckets are in parallel read out; whole sequences are constructed on the spot, and pattern matching is then carried out sequentially against each of the sequences. Once a match is found, the relative position of any of the involved subsequence in its subsequece stream is mapped to the Sequence ID of the corresponding sequence that is unique in our database. (In the above discussion, number '16' is decided by biologists. This number is designed as a system parameter and can be changed.)

If a pattern matching is against a subsequence, only the corresponding

bucket of the subsequence need to be read out, and pattern matching is then carried out in a similar way as patter matching against entire sequences.

For border expansion of a subsequences, the predecessor and successor (if exist) of the subsequences are additionally read out for the construction of expanded subsequences' stream.

## 6. Experiments

To see our algorithm works efficiently, we did an off-line test, i.e., a test without involving the Webdriver. The algorithms are implemented in C, running on a SUNstation under UNIX with Informix database system. In our database, there are about 5000 DNA sequences and each is about 1000 characters long. For the test purpose, we evaluate a pattern query: (D*+C)A* with relaxation. For simplicity, however, only the *FEC* of 'D' is used. That is, to relax the query, only 'D' in (D*+C)A* is replaced with $I_D$. In addition, we change the cardinality $|I_D|$ of $I_D$ from 2 to 10 to observe the performance of different query evaluation strategies. (Note that in practice $|I_D|$ is determined by biologists and cannot be changed as one will.)

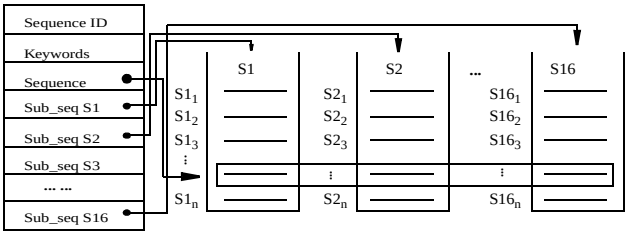



Fig. 11. Storage Structure of GPCR Sequence Databases

In the experiment, three strategies are tested against (D*+C)A* with relaxation.

SS: (Simple Strategy) It uses the pattern matching function in C language. The relaxation is done in a step-by-step fashion. That is, each time 'D' is replaced with a different letter in $I_D$ for a relaxation.

PSS: (Position-tree Supporting Strategy) It uses the position trees as indexes; but the relaxation is done in a step-by-step fashion as by SS strategy

PRS: (Position-tree with Relaxation Strategy) It is the method proposed in this paper.

The test result is shown in Fig. 12.

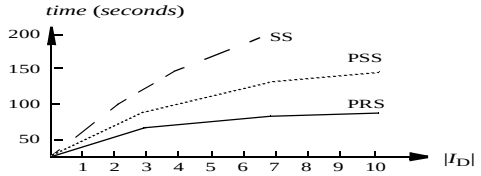



Fig. 12. Test results

The result shows that PRS outperforms SS and PSS uniformly, indicating that the elaboration of the searching process is worthwhile.

## 7. Conclusion

In this paper, a new method is proposed to do the query relaxation. Two kinds of relaxations can be handled: key word relaxation and residue type relaxation. For the first kind of relaxation, two hierarchies: family tree and thesaurus have been established to govern the relaxation process of the key words. For the second kind of relaxation, we introduce the concept of fuzzy equivalence classes to capture the similarities among residue types. This concept is used to develop a technique for fuzzy pattern searching. In addition, the position tree is extended to evaluate regular expressions and fuzzy regular expressions. In this way, high performance can be achieved.

## References


ABB00 Ashburner, M., Ball, C.A., Black, J.A. and so on, Gene Ontology: tool for the unification of biology, *Nature Genetics*, Vol. 25, 2000, pp. 25-29.

AHU74 Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Com., London, 1969.

AOA01 Aguilar, D., Oliva, B., Aviles, F.X. and Querol, E., TranScount: prediction of gene expression regulatory proteins from their sequences, *Bioinformatics*, Vol. 18. no. 4, 2002, pp. 597-607.

BMR96 Brajnik, G., Mizzaro, S., Rasso, C., Evaluating User Interfaces to Information Retrieval Systems: A Case Study on User Support, in *Proc. of the 19th Anual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1996), pp. 128-136.

Co00 Coverson, C., Query Expansion Using and Interactive Concept Hierarchy, Master's dissertation, Department of Information Studies, University of Sheffield, 2000.

Co93 Cottawald, S., *Fuzzy Sets and Fuzzy Logic: the foundations of application - from a mathematical point of view.* Braunschweig, Wiesbaden: Vieweg, 1993.

CR94 Crochemore, M. and Rytter, W., *Text Algorithms*. Oxford University Press, New York, 1994.

Ef00 Efthimiadis, E.N., Interactive Query Expansion: A usernased evaluation in a relevance feedback environment, *Journal of the American Socienty for Infromation Science*, 51(11):989-1003, 2000.

GC98 GCRDb, URL: http://www.gcrdb.uthscsa.edu/.

GM95 Giancarlo, R. and Mignosi, F., Generalization of the periodicity theorem of Fine and Wilf, in Proc. CAAP 94, Lecture Notes in Computer Science, vol. 78, Springer, Berlin, 1994, pp. 130-141.

GP98 GPCR mutant database, URL: http://mgddk1.niddk.nih.gov:8000/GPCR.html.

GPCR02 GPCRDB, URL: http://www.sander.embl-heidelberg.de/7tm/.

GBS92 Gonnet, G.H., Baeza-Yates, R.A. and Snider, T. New Indices for Text: PAT trees and PAT arrays, *Information Retrieval*, ed.: Frakes, W.B., Baeza-Yates, R.A., Prentice Hall, New Jersey, 1992, pp. 66-83.

HU69 Hopcroft, J.E. and Ullman, J.D., *Formal Language and Their Relation to Automata*, Addison-Wesley Publishing Com., London, 1969.

Kn73 Knuth, D.E., *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Publishing Com., London, 1973.

IUS97 *Informix-Universal Server - Informix Guide to SQL*, Informix Press, Menlo Park, CA, USA, 1997.

KJ98 Kekakainen, J. and Jarvelin, K., The impact of query structure and query expansion on retrieval performance, In: *Proc. of the 21th Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval*, Melbourne, Australia, August 23-28, 1998.

JKN96 Jarvelin, K., Kristensen, J., Niemi, T., Sormunen, E., and Keskustalo, H., A deductive data model for query expansion, In: *Proc. of the 19th Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval*, Zurich, Switzerland, 1996, pp. 235-243.

McC76 E.M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, Vol. 23, No. 2, 1976, pp. 262-272.

Mo68 Morrison, D.R., PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of Association for Computing Machinary*, Vol. 15, No. 4, Oct. 1968, pp. 514-534.

MM93 U. Manber and E. Myers, "Suffix arrays: a new method for on-line string searches," *SIAM J. Comput*. 22 (OCT 1993), pp. 935-948.

MSB98 Mitra, M., Singhal, A., and Buckley, C. Improving Automatic Query Expansion. In *Proceedings of the 21 st Annual International ACM-SIGIR*, 1998, pp. 204-214.

NCBI92 National Center for Biotechnology Information. ENTREZ: Sequences User's Guide, *National Library of Medicine*, Bethesda, MD, 1992, Release 1.0.

NDB02 Nucleic Acid Database Project (NDB), URL: http://ndbserver.rutgers.edu/NDB/structure-finder/tutorials, 2002.

PMFR92 Pearson, P., Matheson, N., Flescher, N., and Robbins, R.J., The GDB huan genome data base anno 1992, *Nucleic Acids Research* 20 (1992), 2201-2206.

SWISS02 SWISS-PROT Protein knowledgebase, URL: http://www.expasy.ch/sprot, 2002.

Ta72 R. Tarjan, Depth-first Search and Linear Graph Algorithm, *SIAM J. Comput*. Vol. 1. No. 2. June 1972.

Uk92 E. Ukkonen, "Constructing suffix trees on-line in linear time," in *Proc. of IFIP'92*, pp. 484-492.

We73 Weiner P., Linear Pattern Matching Algorithms. *Conf. Recorder, IEEE 14th Annual Symposium on Switching and Automata Theory*, 1973, pp. 1-11.

WPDB02 The Protein Data Bank Through Microsoft Windows, URL: http://www.sdsc.edu/pb/wpdb/wpdb.htm, 2002.

WZ96 Williams, H. and Zobel J., Indexing Nucleotide Databases for Fast Query Evaluation, in *Proc. of 5th Int. Conf. on Extending Database Technology*, Avignon, France, March 1998, pp. 275-288.

YK98 Yoon, J. and Kim S.-H. A., Three-Level User Interface of Multimedia Digital Libraries with Relaxation and Restriction, in *Proc. of IEEE International Forum on Research and Technology Advances in Digital Libraries* (*ADL'98*), April 22-24, 1998, Santa Barbara, California, pp. 206-215.