Published in Proc. of 8th International Conference and Workshop on Database and Expert Systems Application, DEXA 97.

Efficient Querying on Genomic Databases by Using Metric Space Indexing Techniques [†]

(extended abstract)

Weimin Chen and Karl Aberer

GMD-IPSI
Dolivostr. 15
64293 Darmstadt
Germany

{chen, aberer}@darmstadt.gmd.de

Abstract

A genomic database consists of a set of nucleotide sequences, for which an important kind of queries is the local sequence alignment. This paper investigates two different indexing techniques, namely the variations of GNAT trees [1] and M-trees [3], to support fast query evaluation for local alignment, by transforming the alignment problem to a variant metric space neighborhood search problem.

1 Introduction

Sequence databases are among the most important information repositories in molecular biology. Two types of sequences are found in sequence databases: nucleotide sequences with a four letter alphabet, and amino acid residue sequences with a twenty letter alphabet. A fundamental access mechanism to sequence databases is by sequence alignment. The relevance of sequence alignment derives from evolutionary relationships between sequences. Thus a typical query against a sequence database is to find for a given sequence x all sequences p in the database such that the (local or global) alignment score between x and p is greater equal to a given threshold σ.

Intensive research has been performed on efficient local alignment algorithms. The most efficient algorithms for aligning two sequences with total length F need time $O(F \log F)$. Most existing tools for evaluating alignment queries exhaustively compare the query sequence with each sequence in the database. Thus, for large databases, exhaustive search techniques become prohibitively expensive due to the volume of data to be processed for each query.

Conventional databases use indexing to provide efficient access to the data. Williams and Zobel [6] use an inverted-list indexing technique to support alignment queries on nucleotide databases. In their approach the processing of queries is partitioned in two phases:

- A coarse search based on an index structure to select candidates that are potentially good answers
- A fine search search through these candidates to select the desired result.

To support coarse searches, they propose to extract certain intervals, with given lengths of the sequences in the database, and then build inverted lists for these intervals. Thus, coarse search involves retrieval of the inverted lists corresponding to some intervals of the query string. This technique, however, becomes less sensitive (i.e., good answers may escape from the coarse search) for greater interval lengths. Conversely, indexes formed on shorter intervals are unlikely to be discriminating. Therefore, there is a trade-off between selectivity and sensitivity for this technique.

We attempt to provide a fully sensitive indexing technique for coarse search. The local alignment problem is first transformed into a metric space neighborhood search problem [1, 3, 5]. For indexing known algorithms for standard metric space neighborhood searching are used, namely GNAT trees [1] and M-Trees [3]. Let (X, d) be a metric space where X is the domain of points and d the metric distance function. Let $Y \subseteq X$ be a finite data set stored in the database. The standard neighborhood query supported by index structures for metric space searching is for the following problem (P):

---

† This work has partly been funded by the European Union within the framework of the ESPRIT Long Term Research Project HERMES, No. 9141 [http://www.ced.tuc.gr/hermes/hermes.html].

Given a point x ∈ X and radius $r > 0$, find all $y \in Y$ s.t. $d(x, y) \le r$.

For the alignment query, we need the index structure to support the query for the following problem (P*):

Let f be a function from X to the set of real numbers. Given a point $x \in X$ and a radius $r > 0$, find all $y \in Y$ s.t. $d(x, y) \le f(y) + r$.

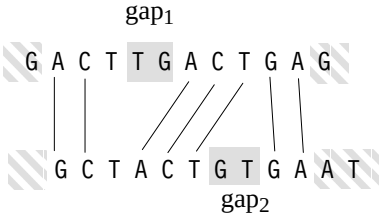
2 Transformation of Local Alignments to Edit Distance Evaluation

The score of local alignment for two sequences x and y is defined as

$$local\_alignment\_score(x, y) =_{\text{def}} \max\{c \times (\text{the total length of alignment}) - (\text{the penalty of the gaps})\}.$$

where $c > 0$ is a predefined constant, and the penalty of the gaps is a non-negative affine function with respect to the total length and the number of the gaps [2].

Example. Let the constant $c = 1$ and the penalty for each gap is the affine function $gap(t) = 1 + t$ where t is the length of the gap. Consider two sequences GACTTGACTGAG and AGCTACTGTGAAT, their maximal local alignment is as follows:



Thus, the total length of the alignment is 7, both penalties for gap_1 and gap_2 are 3. So the score of the above alignment is 7 – 6 = 1. Note, the local alignment does not count the penalties for the gaps in the left- and/or right-most corners, i.e., the gaps marked by pattern in the above figure. ∎

The general type of queries with respect to local alignment is, for a given sequence x and a score-threshold σ, to find all sequences y in the database such that

$$local\_alignment\_score(x, y) \ge \sigma \qquad (1)$$

Relation (1) guarantees that there is an alignment such that

$$c \times (\text{the total length of alignments}) \ge \sigma \qquad (2)$$

On the other hand, for a given alignment the number of characters in the gaps is equal to

$$len(x) + len(y) - 2 \times (\text{the total length of alignment}) \qquad (3)$$

Let $d(x, y)$ stand for the edit-distance between x and y (i.e., the minimal number of operations to change a string from x to y). Henceforth, $d(x, y)$ does not exceed (3). Thus, by (2), the relation (1) implies that

$$d(x, y) \le len(x) + len(y) - 2\sigma/c \qquad (4)$$

Our coarse searching scheme uses relation (4) to select candidates, for any given sequence x and score σ. Now for a given point x and score σ, we get the corresponding radius $r(x, \sigma) = len(x) - 2\sigma/c$ and hence relation (4) is equivalent to $d(x, y) \le r(x, \sigma) + len(y)$ which is an instance of problem (P*).

3 GNAT [1] and its Variation

At the top node of a GNAT , several distinct *split* points are chosen and the space is divided into Dirichlet domains based on those points. The remaining points are classified into groups depending on what Dirichlet domain they fall into. Each group is then structured recursively in the same manner.

In order to make full use of the distances calculated in GNAT, it is suggested to compute the range of distances for each pair of split points. More exactly, assume node p_0 has children $p_1, ..., p_k$. For each $j \in [1, k]$, let Y_j be the set of objects stored in the subtree rooted at p_j. Thus, for each $i \in [1, k]$, the node p_i stores

$$range(p_i, p_j) =_{\text{def}} [min\_d(\hat{p}_i, Y_j), max\_d(\hat{p}_i, Y_j)], \text{ for } j \in [1, k] - \{i\} \qquad (5)$$

where $min\_d(\hat{p}_i, Y_j)$ and $max\_d(\hat{p}_i, Y_j)$ are the minimal and maximal value of $d(\hat{p}_i, y)$ for $y \in Y_j$.

The GNAT is constructed by the call of *GNAT_construction*(*root*, *Y*).

GNAT_construction(p_0, Y_0)
{ choose k split points $y_1, ..., y_k \in Y_0$,
  where k is selected as an appropriate technical argument.
  create k nodes $p_1, ..., p_k$ such that $\hat{p}_i = y_i$
  children(p_0) ← $\{p_1, ..., p_k\}$
  partition Y_0 into $Y_1, ..., Y_k$,
  where Y_i is the Dirichlet domain of p_i in Y_0, for $i \in [1..k]$
  foreach $(i, j) \in [1, k] \times [1, k]$ **such that** $i \ne j$ **do**
  store $range(p_i, p_j) = [min\_d(\hat{p}_i, Y_j), max\_d(\hat{p}_i, Y_j)]$ **at** p_i
  foreach $i \in [1..k]$ **such that** $Y_i \ne \emptyset$ **do**
  GNAT_construction(p_i, Y_i)
}

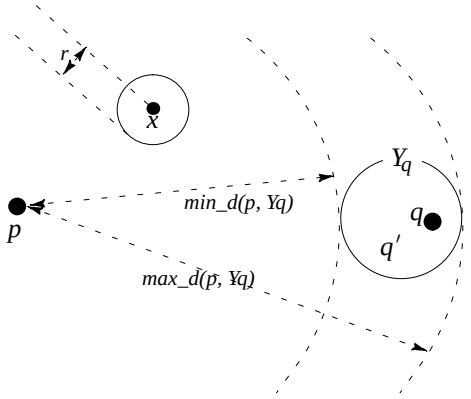


Fig. 1 The geometric property of GNAT

A search in a GNAT is performed by the call of *GNAT_search*(*x, r, root*), which is recursively defined as follows. Initially, all nodes in the GNAT are not marked.

In the following algorithm, the arguments *x*, *r*, and p_0 respectively indicate the given point, the radius, and the current node in GNAT.

GNAT_search(*x, r,* p_0)
{ **foreach** $p \in$ *children*(p_0) **such that** *p* is not marked **do**
{ **if** $d(x, \hat{p}) < r$ **then** output $\hat{p}$
foreach $q \in$ *children*(p_0) – {*p*} **such that**
q is not marked **do**
(i) **if** $[d(x, \hat{p}) - r, d(x, \hat{p}) + r] \cap range(p, q) = \emptyset$ **then**
mark *q*
/* marking *q* means to ignore the subtree root at *q**/
}
foreach $p \in$ *children*(p_0) **such that** *p* is not marked **do**
GNAT_search(*x, r, p*)
}

In the above algorithm, line (i) attempts to mark node *q* in order to prune the subtree rooted at *q*. In the following we explain why this is possible. Let q' be a descendant of *q*. The condition at line (i) ensures that either $d(x, \hat{p}) + r <$ min_d(p, Y_q) or $d(x, \hat{p}) - r >$ max_d(p, Y_q) where Y_q is the set of objects stored at the subtree rooted at *q* (see Fig. 1). Thus, in the former case,

$$\begin{aligned} & d(x, \hat{q}') \\ & \geq d(\hat{p}, \hat{q}') - d(x, \hat{p}) \\ & \geq \text{min\_d}(p, Y_q) - d(x, \hat{p}) \\ & > d(x, \hat{p}) + r - d(x, \hat{p}) \\ & = r. \end{aligned}$$

Alternatively, in the latter case,

$$\begin{aligned} & d(x, \hat{q}') \\ & \geq d(x, \hat{p}) - d(\hat{p}, \hat{q}') \\ & \geq d(x, \hat{p}) - \text{max\_d}(p, Y_q) \\ & > d(x, \hat{p}) - (d(x, \hat{p}) - r) \\ & = r. \end{aligned}$$

This means that $\hat{q}'$ can never be an answer and hence the subtree rooted at *q* can be ignored.

Variation of GNAT

In comparison to the original version, the variant GNAT changes the definitions of *range* as follows: Assume $p_1, ..., p_k$ are the children of p_0, then

$$range(p_i, p_j) =_{\text{def}} \quad (6)$$
$$[min\_d^-(\hat{p}_i, Y_j), max\_d^+(\hat{p}_i, Y_j)], \text{ for } j \in [1, k] - \{i\}$$

where

$$min\_d^-(\hat{p}_i, Y_j) =_{\text{def}} \min\{d(\hat{p}_i, x) - len(x) \mid x \in Y_j\}, \text{ and}$$
$$max\_d^+(p_i, Y_j) =_{\text{def}} \max\{d(\hat{p}_i, x) + len(x) \mid x \in Y_j\}$$

In other words, the variant GNAT is same as the original, after replacing (5) by (6). The detailed algorithm is as follows.

---

*GNAT_construction**(p_0, Y_0)
{ choose *k* split points $x_1, ..., x_k \in Y_0$,
where *k* is selected as an appropriate technical argument.
create *k* nodes $p_1, ..., p_k$ such that $\hat{p}_i = x_i$
children(p_0) ← $\{p_1, ..., p_k\}$
partition Y_0 into $Y_1, ..., Y_k$, where Y_i is the Dirichlet domain of p_i in Y_0, for $i \in [1..k]$
foreach pair (p_i, p_j) with $i \neq j$ **do**
store $range(p_i, p_j) = [min\_d^-(\hat{p}_i, Y_j), max\_d^+(\hat{p}_i, Y_j)]$ **to** *p*
foreach $i \in [1..k]$ **such that** $Y_i \neq \emptyset$ **do**
GNAT_construction(p_i, Y_i)
}

The search algorithm in the variant GNAT is as follows:

---

*GNAT_search**(*x, r,* p_0) /* *x*: the given point; *r*: the radius; p_0: a node in GNAT */
{ **foreach** $p \in$ *children*(p_0) **such that** *p* is not marked **do**
{ **if** $d(x, \hat{p}) < r + len(\hat{p})$ **then** output $\hat{p}$
foreach $q \in$ *children*(p_0) **such that** *q* is not marked **do**
(ii) **if** $[d(x, \hat{p}) - r, d(x, \hat{p}) + r] <> range(p, q)$ **then** mark *q*
/* marking *q* means to ignore the subtree root at *q* */
}
foreach $p \in$ *children*(p_0) **such that** *p* is not marked **do**
*GNAT_search**(*x, r, p*)
}

Notice that the searching radius r may be less than 0 in our application, and therefore the interval $[d(x, \hat{p}) - r, d(x, \hat{p}) + r]$, at line (ii), may make no sense. Thus we introduce relation <> defined as follows: Given two generalized intervals $[a_1, b_1]$ and $[a_2, b_2]$ where $a_1 > b_1$ and/or $a_2 > b_2$ is allowed, $[a_1, b_1] <> [a_2, b_2]$ iff "$b_1 < a_2 \vee b_2 < a_1$".

The correctness of the above searching algorithm can be shown as follows: Let q' be a descendant of q. The condition at line (i) ensures that either $d(x, \hat{p}) + r < \text{min\_d}^-(p, Y_p)$ or $d(x, \hat{p}) - r > \text{max\_d}^+(p, Y_q)$ holds, where Y_q is the set of objects stored at the subtree rooted at q. Thus, in the former case,

$$\begin{aligned} & d(x, \hat{q}') \\ & \geq d(\hat{p}, \hat{q}') - d(x, \hat{p}) \\ & = (d(\hat{p}, \hat{q}') - len(\hat{q}')) + len(\hat{q}') - d(x, \hat{p}) \\ & \geq \text{min\_d}^-(p, Y_p) + len(\hat{q}') - d(x, \hat{p}) \\ & > d(x, \hat{p}) + r + len(\hat{q}') - d(x, \hat{p}) \\ & = r + len(\hat{q}'). \end{aligned}$$

Alternatively, in the latter case,

$$\begin{aligned} & d(x, \hat{q}') \\ & \geq d(x, \hat{p}) - d(\hat{p}, \hat{q}') \\ & = d(x, \hat{p}) + len(\hat{q}') - (d(\hat{p}, \hat{q}') + len(\hat{q}')) \\ & \geq d(x, \hat{p}) + len(\hat{q}') - \text{min\_d}^+(p, Y_p) \\ & > d(x, \hat{p}) + len(\hat{q}') - (d(x, \hat{p}) - r) \\ & = r + len(\hat{q}'). \end{aligned}$$

This means $\hat{q}'$ can never be an answer in any case.

4 M-Tree [3] and its variation

M-tree uses a different strategy to maintain the index structure with the following properties:

(a) The tree is balanced. That is, all paths from the root to the leaves have the same length.

(b) In an M-tree, each node n stores the radius, $r(n)$, such that $r(n) \geq d(\hat{n}, \hat{m})$, where m is any descendant of n. Especially, $r(root) = \infty$ and $r(n) = 0$ for any leaf n.

Based on the above properties, the search on M-tree is done by the call of *M_tree_search*(x, r, *root*).

In the following algorithm, the arguments x, r, and p_0 respectively indicates the given point, the radius, and the current node in M-tree.

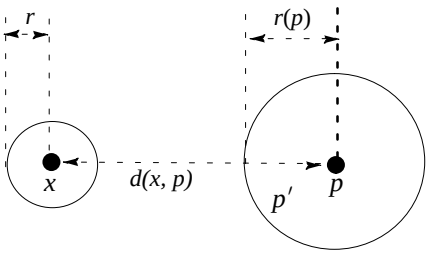


Fig. 2 The geometric property of M-tree

```
M_tree_search(x, r, p0)
{   foreach p ∈ children(p0) do
    {   if d(x, p̂) ≤ r then output p̂
        else if d(x, p̂) > r(p) + r then mark p
    }
    foreach p ∈ children(p0) such that p is not marked do
        M_tree_search(x, r, p)
}
```

In the above algorithm, once a node p is marked, it is sure that no answer can be found in the subtree rooted at p. This is because $d(x, \hat{p}) > r(p) + r$ implies that, for any descendant of p, say p', we have $d(x, \hat{p}') \geq d(x, \hat{p}) - d(\hat{p}, \hat{p}') \geq d(x, \hat{p}) - r(p) > r$ (see Fig. 2).

The construction of an M-tree is performed by means of a series of node insertions. Below we give an algorithm for the following problem: Given an M-tree and a new object, how to construct a new M-tree that accommodates the inserted object.

The algorithm is divided into two phases. The first phase finds the leaf (of the M-tree) that accommodates the inserted object. The second phase handles the possible overflow caused by the insertion.

```
insert(x, p0)
{       if p0 is a leaf then
        {   insert a new node p as a sibling of p0 such that
                p̂ = x and r(p) = 0
            overflow_handing(parent(p))
                return
        }
        select p ∈ children(p0) such that H(d(x, p̂), r(p)) minimal
            where H(x, y) =def 0 if x ≤ y, or x – y otherwise
(i)     r(p) ← max{r(p), d(x, p̂)}
        insert(x, p)
}
```

Variation of M-Tree

The variant of M-tree holds the following properties:

(a) The tree is balanced.

(b) In the tree, each node n stores the radii $r^+(n)$ and $r^-(n)$, such that

$$r^+(n) \geq d(\hat{n}, \hat{m}) + len(\hat{m}) \text{ and}$$
$$r^-(n) \geq d(\hat{n}, \hat{m}) - len(\hat{m}),$$

where m is any descendant of n. Especially, $r^+(root) = r^-(root) = \infty$; $r^+(n) = len(n)$ and $r^-(n) = -len(n)$ for any leaf n. Note, $r^-(n) < 0$ is possible.

Based on the above properties, the searching can be done by the call of *M_tree_search**$(x, r, root)$ where $root \in X - Y$. Again, $r < 0$ is possible. Note, with term $r^-(n)$ we can also determine the additional inclusion relation (see line (i) of the following algorithm).

In the following algorithm, the arguments x, r, and p_0 respectively indicate the given point, the radius, and the current node in M-tree.

---

*M_tree_search**(x, r, p_0)
{ **foreach** $p \in children(p_0)$ **do**
  { **if** $d(x, \hat{p}) < r + len(\hat{p})$ **then** output $\hat{p}$
(ii) **if** $d(x, \hat{p}) \leq r - r^-(p)$ **then**
    output all $\hat{p}'$ where p' is any descendant of p
(iii) **else if** $d(x, \hat{p}) > r^+(p) + r$ **then** mark p
  }
  foreach $p \in children(p_0)$ **such that** p is not marked **do**
    *M_tree_search**(x, r, p)
}

Here is why the algorithm is correct. The condition at line (ii) yields that

$$\begin{aligned} & d(x, \hat{p}') \\ & \leq d(x, \hat{p}) + d(\hat{p}, \hat{p}') \\ & \leq r - r^-(p) + d(\hat{p}, \hat{p}') \\ & \leq r + len(\hat{p}'), \quad (\text{as } r^-(p) \geq d(\hat{p}, \hat{p}') - len(\hat{p}')) \end{aligned}$$

Thus, none of $\hat{p}'$ can be an answer. On the other hand, the condition at line (iii) ensures that

$$\begin{aligned} & d(x, \hat{p}') \\ & \geq d(x, \hat{p}) - d(\hat{p}, \hat{p}') \\ & > r + r^+(p) - d(\hat{p}, \hat{p}') \\ & \geq r + len(\hat{p}'), \quad (\text{as } r^+(p) \geq d(\hat{p}, \hat{p}') + len(\hat{p}')) \end{aligned}$$

Thus, marking the node p ensures that no answer can be found in the subtree rooted at p.

The following is the algorithm for node insertion. Notice the labelled lines, which are comparable to the corresponding labelled lines in the same name procedure for the original M-tree insertion.

---

insert(x, p_0)
{ **if** p_0 is a leaf **then**
  { **insert** a new node p as a sibling of p_0 **such that**
      $\hat{p} = x$ and $r(p) = 0$
    overflow_handing(*parent*(p))
    return
  }
  select $p \in children(p_0)$ **such that** $H(d(x, \hat{p}), r(p))$ minimal where $H(x, y) = 0$ if $x \leq y$, or $x - y$ if $x > y$.
(i)-1 $r^+(p) \leftarrow \max\{r^+(p), d(x, \hat{p}) + len(x)\}$
(i)-2 $r^-(p) \leftarrow \max\{r^-(p), d(x, \hat{p}) - len(x)\}$
  insert(x, p)
}

References

1. S. Brin. **Near Neighbor Search in Large Metric Space** *Proc. of VLDB'95.*
2. K.-M. Chao and W. Miller. **Linear-Space Algorithms that Build Local Alignments from Fragments**. *Algorithmica. 13(1/2)*, pp 106–134.
3. P. Ciaccia, M. Patella, P. Zezula **M-tree: An Efficient Access Method for Similarity Search in Metric Spaces** *Proc.* of *VLDB'97.*
4. C. Faloutsos, M, Ranganathan, and Y. Manolopoulos. **Fast Sequence Matching in Time-Series Databases**. *Proc. of SIGMOD'94.*
5. J. K. Uhlmann. **Satisfying General Proximity/similarity Queries with Metric Trees** *Information Processing Letters 40*(1991) 175-179.
6. H. Williams and J. Zobel. **Indexing Nucleotide Databases for Fast Query Evaluation** *Proc. of EDBT' 96.*