The $ed\text{-}tree$: An Index for Large DNA Sequence Databases

Zhenqiang Tan Xia Cao Beng Chin Ooi Anthony K. H. Tung

Department of Computer Science
National University of Singapore
Email: {tanzhenq, caoxia, ooibc, atung }@comp.nus.edu.sg

Abstract

The growing interest in genomic research has caused an explosive growth in the size of DNA databases making it increasely challenging to perform searches on them. In this paper, we proposed an index structure called the $ed\text{-}tree$ for supporting fast and effective homology searches on DNA databases. The $ed\text{-}tree$ is developed to enable probe-based homology search algorithms like Blastn which generate short probe strings from the query sequence and then match them against the sequence database in order to identify potential regions of high similarity to the query sequence. Unlike Blastn however, the homology search algorithm we developed for $ed\text{-}tree$ supports more flexible probe model with longer probes and more relaxed matching. As a consequence, the $ed\text{-}tree$ is not only more effective and efficient than the latest Blastn(NCBI Blast2) when supporting homology search but also takes up moderate storage compared to existing data structures like the suffix tree. To index a DNA database of 2 giga base pairs(Gbps), $ed\text{-}tree$ only takes less than 3Gb of secondary storage which is easily handled by a desktop PC. Experiments will be shown in this paper to support our claim.

1 Introduction

Homology search on DNA sequence databases is an important function in genomic research. The result of searching homologous sequences can help biologists to do further analysis and detection on the protein structure and function. According to a survey of the of the tasks in bioinformatics, 35.2% [4] of the tasks are sequence similarity search, and 33.3% of them are related to DNA sequence search. The size of DNA sequence databases is growing exponentially in the past few years. As many existing search methods are based on a sequential scanning of the databases, this growth will adversely affect the efficiency. This motivates to develop new and more efficient search methods or to enhance existing methods for more scalability in terms of database size.

The problem of homology search on DNA databases can be described in a nutshell as follow. Given a query sequence Q and a target sequence database T, find a set of subsequences of Q such that each subsequence Q' in the set is highly similar to some subsequence T' of T. The similarity between Q' and T' is computed as a function of the **edit distance**, $edit(Q', T')$, which is defined as the minimum number of edit operations (insert, delete, replace) that transform Q' into T'. While the actual similarity function can vary depending on a biologist's preference model, if resemble to the simple one that we will use in this paper. In short, it means that our proposed algorithm and index structure will be robust with respect to changes in similarity function, defined as:

Definition 1.1 $ED\text{-}Similarity(Q', T')$
Given two sequences Q' and T', we measure the similarity between the two sequences as

$$\text{ED-Similarity}(Q', T') = \frac{|T'| - edit(Q', T')}{|T'|}$$

Intuitively, the measure defined above tries to estimate the maximum number of matches that occur between Q' and T' as $|T'| - edit(Q', T')$ and then normalizes the measure by dividing this difference by $|T'|$.

Our paper focuses on probe-based algorithms like the Blastn[9] which is arguably the most popular homology search tool. The latest version of Blastn works in two phases. Firstly, Blastn moves a sliding window of size w along the **query sequence** Q one letter at a time, generating $|Q| - w + 1$ probes. Secondly, a sequential scan is performed on the sequence data T to identify any portion that matches any one of the probes **completely**. These portions are then extended in a greedy fashion in both directions to identify Q' and T' that have a high similarity score. There are two problems with such an approach. The first and more obvious is that the whole sequence database must be scanned which is unacceptable for very large databases. The second is in the choice of the probe length, w, which is the major factor that affects the tradeoff between sensitivity and speed. The root of the problem here is the requirement for exact probe match which is too rigid for homology search. To find alternative way of probing, we define a model of probing as follow:

Definition 1.2 *Probe Model,* $pmodel(w, s, r)$
Given a target sequence T, let $T[i]$ denote its i^{th} letter and $T[i, i + w - 1]$ denote a substring of length w located at its i^{th} letter. We say that a homology search of sequence

Q is done on a target sequences T using **probe model,** $pmodel(w, s, r)$ with **skip interval,** s and **edit distance range,** r if we have the following three conditions:

1. the length of the probe is w
2. each probe is compared again all subsequences of T that are of the form $T[\alpha s, \alpha s + w - 1]$, $\alpha \in \mathcal{I}^+$
3. all subsequences of the form $T[\alpha s, \alpha s + w - 1]$ are extended if $edit(T[\alpha s, \alpha s + w - 1], Q') \leq r$, Q' being any one of the probes generated from Q

As an example, the default probe model for Blastn will be $pmodel(11, 1, 0)$ since it uses probes of length 11 by default and extends all substrings of length 11 in T if they match any of the probes exactly.

To investigate the querying effectiveness for varying configuration of the probe model, we randomly select 120,000 sequence pairs of equal length and with more than 50% $ED\text{-}Similarity$ from est_human genome. Given each sequence pair Q and T, we use Q as the query sequence and perform a homology search on T for each probe model to see whether at least one subsequence of T is extended (we will refer to this as a hit). Based on the result from the sequence pairs, we can now compute the probability of generating at least one hit from Q to T given a certain probe model. This probability will be used to evaluate the sensitivity of a probe model. Since typical homologous search is of length 20 to 200 bases [8], we select 64 and 128 as the fixed homology length.

Because Blastn has a probe length of 11 by default, we adopted it as a benchmark and will denote it as Blastn-11. Figure 1 and 2 illustrate the sensitivity for detecting 64-bases homologous sequence pair and 128-bases homologous sequence pair in different models [1]. For the sequences of 64 bases, we can see from Figure 1 that adopting $pmodel(18, 2, 3)$, $pmodel(18, 3, 3)$ will find all sequences that have ED-Similarity of 0.7 and above. This compares very well against Blastn-11 which only detects 80% of such sequences. Although, $pmodel(18, 2, 2)$ and $pmodel(18, 3, 2)$ are not as sensitive as Blastn-11 when detecting sequences with low ED-Similarity, their performance also becomes slightly better than Blastn-11 when we consider sequences with ED-Similarity of 0.75 and above. For sequences with 128 base pairs, Figure 2 essentially tells the same story. The only difference is that the curves level off when ED-Similarity is from 0.65 to 0.7 instead of from 0.75 onwards. This is due to the longer length of the sequences which make it less likely for Blastn to miss sequences that have higher ED-Similarity. From the experiments, we see that adopting a probe model with longer probe length and more relaxed matching can result in higher sensitivity. Our aim in this paper is to develop an index structure which will support such probe models efficiently.

[1]Note that while we have analyzed many pmodels for their sensitivity, we only show the more relevant ones here to avoid affecting the clarity of the graphs.

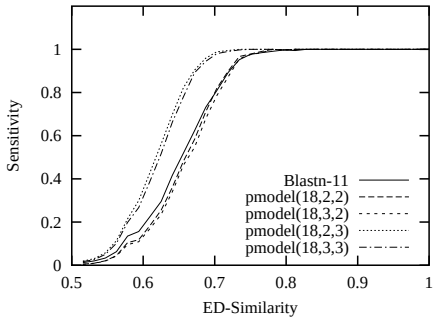

Figure 1. Sensitivity(64 bps)

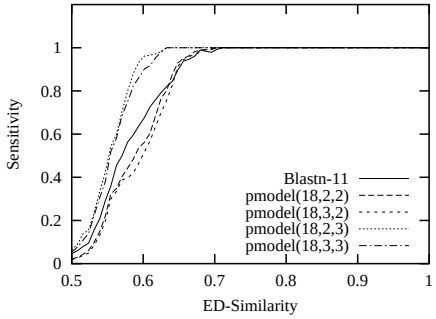

Figure 2. Sensitivity(128 bps)

Contribution:
In this paper, we propose an index structure called the $ed\text{-}tree$ which performs probe-based homology search on DNA databases with a longer probe length but more relaxed matching. The $ed\text{-}tree$ has the following strengths:

Scalablity. As we will see in the next section, previous work on indexing DNA sequences often results in an index that is much larger than the original sequence database. One contributing factor to this is that there is usually a need to store pointers into every position of the DNA sequences. For example, given a DNA sequence of 2 billion base pairs (i.e characters), each pointer will require around 4 bytes and the total size of the pointers alone will be 8GB. With the probe model adopted by $ed\text{-}tree$, this requirement is reduced by 2 to 3 times since not every position in the sequences needs to be indexed. Combined with some compression techniques which we will describe later, the index for 2 billion base pairs is around 2.28 to 2.97 GB which is acceptable consisting that the whole human genome is about 3 billion base pairs.

Efficiency. Unlike Blastn or other algorithms which perform a sequential scan on the database, the $ed\text{-}tree$ allows

us to directly access portions of the targets which are hit by probes. The ed-$tree$ is a multi-layer index and the first two layers of the index are sufficiently small for storage in the main memory. This enables us to quickly prune off disk accesses in memory. As mentioned earlier, the disk-resident portion of the index is also smaller than other sequence indexes and this reduces I/Os. The use of a large skip interval also means that there are fewer comparisons to be made in the search.

Sensitivity. The sensitivity of the ed-$tree$ does not shortchange the scalability and efficiency. The probe model of ed-$tree$ is more flexible and sensitive than Blastn since we allow certain mismatches between the target sequence and the probe. Experiments shown earlier have already illustrated this point.

The remainder of this paper as follow. In Section 2 some background knowledge and related work are introduced. In Section 3 the ed-$tree$ is proposed as an index structure for searching a DNA sequence database. In Section 4 we present an algorithm for performing homology search on DNA sequences based on the ed-$tree$. The performance of the ed-$tree$ will be evaluated in Section 5 and we will conclude our paper in Section 6.

2 Related Work

With the increasing interest on genomic research, various DNA sequence searching systems have been developed to support different objectives. Some methods locate similar regions in the sequence database by sequential scan while others index the databases by using novel data structures which can speed up homology searching processes.

2.1 Homology Search based on Sequential Scan

There have been many proposals on performing a full scan on the sequence database for homology search. The most fundamental method is the Smith-Waterman algorithm [11], which performs sequence alignment between query sequence and target sequence using a dynamic programming algorithm in $O(mn)$ time with m and n being the length of two sequences.

Blastn [9] is the most widely used biological homology search system since 1990. It regards exact match of w contiguous bases as candidates which are extended greedily towards both its's left and right side to obtain the final alignments. However, Blastn faces a difficulty in the choice of w since increasing w decreases sensitivity whereas decreasing w slows down computation.

Pattern Hunter [8] is an improvement on Blast both in speed and sensitivity by using non-consecutive k letters as model, where k is the weight of model. This model can achieve better sensitivity since it can better detect replacements in the sequences than Blastn. Unfortunately, Pattern Hunter is still essentially a sequential scan method which may not scale up for very large sequence databases.

SENSEI [10] is another sequential scanning method which selects hits by exact match. It outperforms Blastn by using compactly encoded scoring tables for k-tuples, encoding bases with single bits, removing the simple sequence repeats, and masking the some known repeats in the query sequence.

In [1], Buhler proposed a method, LSH-ALL-PAIRS, for finding longer seeds to improve efficiency, while maintaining sensitivity for weak similarity by using the technique of locality-sensitive hashing(LSH). However false drops and false hits cannot be completely avoided because the result is sensitive to the hashing functions being used. Furthermore, it may miss some short alignments in a collection of sequences.

2.2 Index Based Homology Search

There exist DNA sequence searching methods which pre-build indexes for searching the sequence database.

In SST[3], each sequence is partitioned into fragments according to the window size, and each window is mapped into a vector. Tree structured vector quantization is used to create its tree-structured index by a k-means clustering technique. SST is much faster than Blastn when searching for highly similar sequences. Unfortunately, since the distance between sequences in vector space does not correspond well with the actual edit distance, larger number of false dismissals may occur if the similarity between the query sequence and the target sequence is not sufficiently high.

Suffix tree is a well-studied data structure for handling string. QUASAR [2] applies a modification of q-tuple filtering implemented on top of suffix array. However, the performance deteriorates dramatically if the compared sequences are only distantly homologous since QUASAR does not consider regions that spanned across two consecutive blocks. A disk based suffix tree[5] and an in-memory compressed suffix array [7] are also used as the index structure for sequence similarity search. For a very large DNA sequence database, the index techniques using suffix tree or suffix array have two drawbacks. First, the index size can be much larger than the original sequence database. For example, the latest work[5] shows that the index of a 263Mbps DNA is about 18GB. Second, they are more suitable for dealing with precise matches, while awkward for handling mismatches.

Williams et al. [12] proposed a searching algorithm in a research prototype system, CAFE. CAFE uses an inverted index to select a subset of sequences that display broad similarity to the query sequence. The inverted index not only stores the number of sequences that contain the intervals, k-tuple subsequences, but also keeps the offset information

of the intervals. The experiments in [12] show that CAFE is in fact less sensitive than Blastn when searching for similar sequences.

In [6], a wavelet-based method is proposed to map the subsequence of the database into a 2σ dimensional integer vectors where σ is the alphabet size of the sequences. The coefficients of the integer vector are then indexed by MBR, an index structure storing a grid of trees. Range queries and nearest-neighbor queries are performed based on the index structure. Though this method avoids false dismissals, there are lots of false hits since the approximation of edit distance is not sufficiently tight. This increases the cost for refining the final result. In addition, this method tries to find the regions of the target sequence which are similar to the **whole** query sequence and **not part of the** query sequence. This is thus different from what we are trying to achieve in this paper.

3 The ed-$tree$

In this section, we present the structure of the ed-$tree$ and relevant concepts.

Definition 3.1 Segment Length Vector, $H = [h_1, ..., h_t]$
A segment length vector $H = [h_1, ..., h_t]$ is an integer vector in which each h_i is a positive integer. We say that a sequence S is segmented according to a segment length vector H (denoted as S^H) if we partition S into t segments $S^H[1]$,...,$S^H[t]$ such that each segment $S^H[i] = S[1 + \sum_{j=0}^{j=i-1} h_j, \sum_{j=0}^{j=i} h_j]$ with $h_0 = 0$ by default.

For example, if $H = [6, 6, 6]$ and S=AAAATTCGCGATAAGTAG, then we say that we segment S according to H by partitioning S into 3 parts, $S^H[1]$=AAAATT, $S^H[2]$=CGCGAT and $S^H[3]$=AAGTAG. As we can see, a sequence S can only be segmented by a segment length vector $H = [h_1, ..., h_t]$ if and only if $|S| = \sum_{j=1}^{t} h_j$.

Definition 3.2 ed-$tree(w, s, H), H = [h_1, ..., h_t]$
The structure of ed-$tree(w, s, H)$ defined on a target sequence database, T, will be a tree with $t + 1$ levels [2] *described as follow:*

- *it contains a virtual root node at level 0 which represents a null sequence (recall that $h_0 = 0$ by definition);*
- *each node t_i, at level i represents a sequence of length h_i;*
- *a path from the root to a leaf, $\{t_1, ..., t_n\}$ (note that we leave out t_0 which represents a null sequence) represents a sequence S that is segmented based on H with each t_i representing $S^H[i]$;*
- *the leaf node t_n at the end of the path $\{t_1, ..., t_n\}$ consists of the pointers to all subsequences in T which match S* **exactly** *with the form $T[\alpha s, \alpha s + w - 1]$, $\alpha \in \mathcal{I}^+$.*

For example, Figure 3 illustrates an ed-$tree(18, 2, [6, 6, 6])$ for a length 42 target sequence T below:

T=AGGTAGGTAGGTAGGTAGGTAGGGCTTACATTCAGTAC

Since we have $w = 18$ and $s = 2$ in this case, all subsequences in T of length 18 and located at even positions are indexed by the ed-$tree$. As such, there are a total of $\lfloor (42 - 18)/2 \rfloor$ positions being indexed (In Fig 3, we begin indexing from the second letter). The subsequences which start at location 2, 6, 10, 14, 18 and 22 share the common prefix of "GGTAGG" which are represented by the same level 1 node. At level 2 however, the subsequence is separated into 3 portions depending on which of the subsequences "TAGGGC", "GCTTAC" or "TAGGTA" they match for their second segment. Finally, if we follow any path from the root to a leaf node, the leaf node contains the pointers to all subsequences in T that match the subsequence represented by the path.

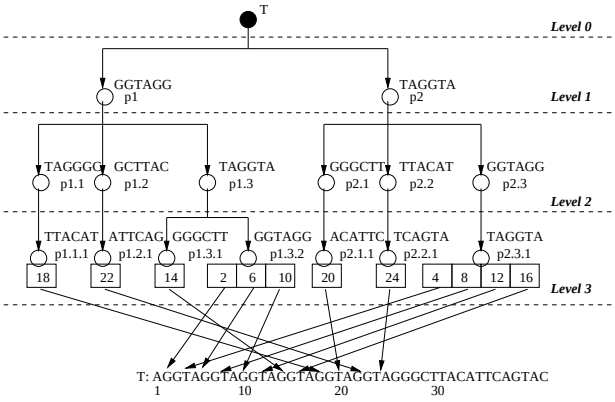

Figure 3. An example of ed-$tree$

Because the size of the leaf nodes could be rather huge for large sequence databases, to reduce this storage requirement we sort the pointers in each leaf node in increasing order and apply the frame-of-reference compression method to each set of pointers in the leaf nodes. We achieve a reduction in storage of around 40% to 50% with this approach. Considering the memory of our targeting hardware platform, normal PC, and the four major different nucleic acids, we set $w = 18$ and $H = [6, 6, 6]$ for the DNA sequences such that its efficiency and effectiveness are good in practice. Figure 4 gives the general algorithm for building an ed-$tree(w, s, H)$ with time complexity of $O(\frac{|D|}{s}t)$, $|D|$ is the sequence size and t is the number of the tree levels.

[2] For convenience of description, we consider the root of the tree to be at level 0. Thus we now have level 0 to level t for the ed-$tree$.

There are two points to highlight for the actual implementation. First, for the values of H, level 1 and 2 of the $ed\text{-}tree$ usually have a node that represents each possible sequence of length h_1 or h_2. As such, these two layers of the tree are implemented as a two-dimensional lookup table where the values along each dimension represent sequences and each element of the table is a pointer to the third level. To find the locations of a sequence Q, $Q^H[1]$ and $Q^H[2]$ will be used to map into a particular element in the lookup table and the third level of the tree can then be accessed. Second, for the last level of the tree, the pointers in the node are in fact **physically separated** from the node itself. This is because the pointers will only be accessed if the node is a matching one and we thus can avoid large amount of I/O by storing them separately. Figure 5 illustrates the physical implementation of a three level $ed\text{-}tree$.

Algorithm 1 *Algorithm Build* $ed\text{-}tree$
input: *Parameters* w, s, $H = [h_1, ..., h_t]$ *and the target sequence database* T
output: $ed\text{-}tree(w, s, H)$ *for* T
method:
1. $\mathcal{EDT} \leftarrow$ *EmptyNode ; /* root of the tree */*
2. Based on w *and* s*, slide along* T *and for each generated probe* p *do*
  i. Segment p *based on* H *into* $p^H[1]$,...., $p^H[t]$;
  ii. $pt \leftarrow \mathcal{EDT}$;
  ii. $i = 0$;
  iii. while ($i \leq t$ *)*
    If pt *has a child,* ch*, which represents* $p^H[i+1]$ *then*
      { $pt \leftarrow ch$;
      $i = i + 1$;}
    Else
      { *Add a child which sequence* $p^H[i+1]$ *to* pt;
      $pt \leftarrow$ *the new child;* }
  iv. Add a pointer to location of p *in* pt

Figure 4. Building an $ed\text{-}tree$

4 Homologous Sequence Search Using ed-tree

Our algorithm consists of two phases. The first phase is to search for candidate region in the target sequence which match the probes and the second phase is to extend these matching regions. Since the second phase is essentially the same as the other probe-based algorithms like Blastn, we focus our discussion on the first phase.

With a query sequence Q, a target sequence database T and the $ed\text{-}tree(w, s, H)$ built on T, given a probe, P (Note: $|P| = w$) that is generated from Q, we want to find the locations of all subsequences in T that are within a distance of r from P. In other words, we want to look at how the $ed\text{-}tree$ can be used to support a probe model, $pmodel(w, s, r)$.

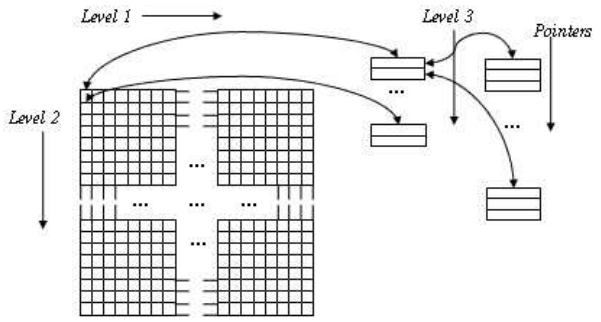

Figure 5. The 3-level $ed\text{-}tree$ **index**

To compare P efficiently against the segmented sequences in the $ed\text{-}tree$, we introduce **matching segment** and **length difference vector** as follow.

Definition 4.1 *Matching Segment,* $MS(P, S^H[i])$
Let S and P be two sequences of length w and $H = [h_1, ..., h_t]$ be a segment length vector. Assume that $edit(P, S)$ is computed based on alignment $\mathcal{L}$ and that we segment S based on H. We say that $MS(P, S^H[i])$ is the matching segment of P to $S^H[i]$ iff $MS(P, S^H[i])$ is the portion of P that can be transformed to $S^H[i]$ in the alignment $\mathcal{L}$.

Definition 4.2 *Length Difference Vector,* $\delta(P, S, H) = [\delta_1, ..., \delta_t]$
Let S and P be two sequences of length w and $H = [h_1, ..., h_t]$ be a segment length vector. We define the **length difference vector** between S and P based on H as an integer vector $\delta(P, S, H) = [\delta_1, ..., \delta_t]$ where $\delta_i = |MS(P, S^H[i])| - h_i$.

Example 1 Let us consider the two sequences
P=GGTAGCGGCTTACTTCAG and
S=GGTAGGGCTTACATTCAG.
Assume that $H = [6, 6, 6]$ is the segment length vector for sequence S. The alignment of sequence P and S is as follow:

P:GGTAGCG GCTTAC - TTCAG
S:GGTAG -G GCTTAC ATTCAG

Based on the alignment of S and P and the segment length vector H, P can be partitioned into 3 parts, $MS(P, S^H[1]) = GGTAGCG$, $MS(P, S^H[2]) = GCTTAC$, $MS(P, S^H[3]) = TTCAG$. Thus the corresponding length difference vector is $\delta(P, S, H) = [7 - 6, 6 - 6, 5 - 6] = [1, 0, -1]$.

Lemma 4.1 *Let S and P be two sequences of length w and H be a segment length vector. If $edit(S,P) \leq r$, we have the following properties for $\delta(P,S,H) = [\delta_1, ..., \delta_t]$:*

1. $\sum_{i=1}^{t} \delta_i = 0$;
2. $\sum_{i=1}^{t} |\delta_i| \leq r$;

Proof: Since P is transformed into S which is of the same length, the total change in length for P must be zero which implies that the sum of the elements in $\delta(P,S,H)$ must be zero. For the second property, we note that each edit operation increases or decreases the length of a sequence at most by 1. Since it takes r operations to transform P into S, this means that the absolute change in length of each segment which is computed as $\sum_{i=1}^{t} |\delta_i|$ should be less than or equal to r.

Theorem 4.1 Let S and P be two sequences of length w and $H = [h_1, ..., h_t]$ be a segment length vector. If $edit(S,P) \leq r$, then there exists another segment length vector $H' = [h_1 + \beta_1, ..., h_t + \beta_t]$ such all the followings are true:

1. $\sum_{i=1}^{t} \beta_i = 0$;
2. $\sum_{i=1}^{t} |\beta_i| \leq r$;
3. $\sum_{i=1}^{t} edit(P^{H'}[i], S^{H}[i]) \leq r$;

Proof: Let $\delta(P,S,H) = [\delta_1, ..., \delta_t]$. We set each value β_i to be δ_i which means that $P^{H'}[i]$ is in fact $MS(P, S^H[i])$. From Lemma 4.1, we will immediately know that the first two properties will be satisfied. For the third property, we should observe that $\sum_{i=1}^{t} edit(P^{H'}[i], S^{H}[i])$ is in fact the edit distance of P and S which is known to be less than r.

Based on Theorem 4.1, we derive a method for finding all the substrings of length w that are within an edit distance of r from a sequence P given $ed\text{-}tree(w,s,H)$. Our approach is to generate all possible values of $\delta(P,S,H)$ such that the first two properties in Theorem 4.1 are satisfied and then use these values to segment P for comparison to the subsequences indexed in the $ed\text{-}tree$. We call the set of all such possible values the **cover generator** and denote it as $cover\_gen(r,t)$. Note that the size of $cover\_gen(r,t)$ is only dependent on r, the edit distance range and t, the number of segments. Combinatorial analysis shows the cardinality of the cover generator is:

$$|cover\_gen(r,t)| = 1 + \sum_{j=1}^{\lfloor \frac{r}{2} \rfloor} \sum_{i=1}^{t-1} \binom{t}{i} \binom{j-1}{i-1} \binom{j+1}{i-j+1}$$

Figure 6 shows how the cardinality of the cover generator varies with r and t. We note that the cardinality increases quickly with increasing t but only moderately for increasing r. For example, if t, the number of the segments is set to 3, then $|cover\_gen| = 7$ for both the cases where $r = 2$ and $r = 3$.

Example 2 Consider a sequence P=GGTAGCGGCTTACTTCAG and an $ed\text{-}tree$ with segment length vector H=[6, 6, 6]. When r=2, searching the $ed\text{-}tree$ will result in a cover generator, $cov\_gen(2,3)$={[−1, 0, 1], [−1, 1, 0], [0, −1, 1],[0, 0, 0],[0, 1, −1],[1, 0, −1],[1, −1, 0]} . By adding each element in the cover generator to H, we will generate 7 different segment vectors $H_1, ..., H_7$ which will be used to generate 7 ways of segmenting P for searching in the $ed\text{-}tree$. Figure 4 shows how P is segmented based on the different elements in cov_gen. Let us now illustrate Theorem 4.1 by assuming that a sequence, S=GGTAGGGCTTACATTCAG is indexed by the $ed\text{-}tree$. In this case, since $edit(P,S)$=2, readers can verify that H_6 will be able to segment P such that all the three conditions in Theorem 4.1 are satisfied[3].

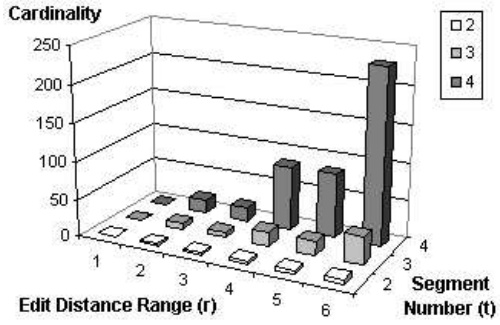

Figure 6. Cardinality of Cover Generator

i	$[\beta_1, \beta_2, \beta_3]$	H_i	$P^{H_i}[1]$	$P^{H_i}[2]$	$P^{H_i}[3]$
1	$[-1, 0, 1]$	$[5, 6, 7]$	GGTAG	CGGCTT	ACTTCAG
2	$[-1, 1, 0]$	$[5, 7, 6]$	GGTAG	CGGCTTA	CTTCAG
3	$[0, -1, 1]$	$[6, 5, 7]$	GGTAGC	GGCTT	ACTTCAG
4	$[0, 0, 0]$	$[6, 6, 6]$	GGTAGC	GGCTTA	CATTCAG
5	$[0, 1, -1]$	$[6, 7, 5]$	GGTAGC	GGCTTAC	TTCAG
6	$[1, 0, -1]$	$[7, 6, 5]$	GGTAGCG	GCTTAC	TTCAG
7	$[1, -1, 0]$	$[7, 5, 6]$	GGTAGCG	GCTTA	CTTCAG

Figure 7. Segmenting P=GGTAGCGGCTTACTTCAG

Figure 8 presents the homology search algorithm. For example, for a given input probe, P=GGTAGCGGCTTACTTCAG, we are going to find all the locations of the subsequences S indexed by $ed\text{-}tree(18, 2, [6, 6, 6])$ of the target sequence T, shown in Figure 3, such that $edit(P,S) \leq 2$. Step 1 initializes a set, $PSet$, to store the information of **active nodes**. In $ed\text{-}tree$, **active nodes** are nodes not pruned off based on the search condition. Each tuple in $PSet$ includes 4 elements, $< v, \beta, l, d >$, where v denotes an active node in the tree, β is a member $cover\_gen(r,t)$, l denotes the level of the active node v, and d is the sum of the edit distances from the root to the level l of the active node v (i.e $\sum_{i=1}^{l} edit(a[i], P^{H+\beta}[i])$ where $a[i]$ is the ith level

[3]edit(GGTAGCG,GGTAGG) + edit(GCTTAC,GCTTAC) + edit(TTCAG,ATTCAG) = 2

Algorithm 2 *Algorithm* $Probe\_Search$
input: *probe* P, *edit distance* r, *target sequence* T, $ed\text{-}tree(w, s, H = [h_1, ..., h_t])$ *built on* T
output: *All locations of the subsequences* S *indexed by* $ed\text{-}tree(w, s, H)$ *such that* $edit(P, S) \leq r$
method:

1. $PSet \leftarrow \emptyset$; */* a set for storing active nodes */*
2. For each $\beta \in cover\_gen(r, t)$
$PSet \leftarrow PSet + \{< \mathcal{EDT}, \beta, 0, 0 >\}$; */* $\mathcal{EDT}$ is the root of the input ed-tree */*
3. $ResultSet \leftarrow \emptyset$;
4. while($PSet \neq \emptyset$ *)*
i.Select the first tuple $< v, \beta, l, d >$ *from* $PSet$;
ii.$QSet \leftarrow \{< v', \beta', l', d' > | v' = v \bigwedge$
$< v', \beta', l', d' > \in PSet\}$;
iii. $PSet \leftarrow PSet - QSet$;
iv. for each child node y *of* v
for each tuple $< v', \beta', l', d' >$ *in* $QSet$
if $edit(y.sequence, P^{H+\beta'}[l'+1]) + d' + |\sum_{i=l'+2}^{t} \beta_i| \leq r$ *then*
if $l' + 1 < t$ *then*
$PSet \leftarrow PSet +$
$\{< y, \beta', l'+1, d' + edit(y.sequence, P^{H+\beta'}[l'+1]) >\}$;
else
for each pointer pt *in* y */*y is t-th level node */*
$ResultSet \leftarrow ResultSet + \{pt\}$;
5. Return $ResultSet$;

Figure 8. Homology search in $ed\text{-}tree(w, s, H)$

ancestor of v).

In Step 2, a tuple $(\mathcal{EDT}, \beta, 0, 0)$ is inserted into $PSet$ for each $\beta \in cover\_gen(r, t)\}$. Here $\mathcal{EDT}$ is the root of the input $ed\text{-}tree$. In the example, $PSet$ becomes $\{(\mathcal{EDT}, [-1, 0, 1], 0, 0)$, $(\mathcal{EDT}, [-1, 1, 0], 0, 0)$, $(\mathcal{EDT}, [0, -1, 1], 0, 0)$, $(\mathcal{EDT}, [0, 0, 0], 0, 0)$, $(\mathcal{EDT}, [0, 1, -1], 0, 0)$, $(\mathcal{EDT}, [1, 0, -1], 0, 0)$, $(\mathcal{EDT}, [1, -1, 0], 0, 0)\}$ after Step 2.

Step 3 initializes the $ResultSet$ which will contain the final output.

Step 4 iteratively goes through the following four substeps until $PSet$ becomes empty. In each iteration, the next available tuple $< v, \beta, l, d > \in PSet$ is retrieved in Step 4(i) and all tuples of the form $< v, \beta', l', d' > \in PSet$ are moved from $PSet$ into $QSet$ from Step 4(ii) to 4(iii). In step 4(iv), each child y of the node v is compared against each tuple $< v, \beta', l', d' >$ in $QSet$ to determine whether y will lead to a leaf in the solution set with respect to $< v, \beta', l', d' >$. This is done by checking whether the total sum of the edit distance between the sequence represented by y and $P^{H+\beta'}[l'+1]$, i.e. $edit(y.sequence, P^{H+\beta'}[l'+1])$, d' and $|\sum_{i=l'+2}^{t} \beta_i|$ is less than or equal to r. If the condition fails, then the leaf nodes under y is not a active node with respect to $< v, \beta', l', d' >$ and the next pair of node-tuple comparison will be processed. If the condition holds and y is a leaf node, then all the pointers at y will be added to the result set. Otherwise, a new tuple $< y, \beta', l'+1, d' + edit(y.sequence, P^{H+\beta'}[l'+1]) >$ will be inserted in the front of $PSet$ for the next round of processing where the children of y will be searched for the next level of active nodes. Figure 9 depicts the iterations in step 4 of the example. In iteration 1, node $p2$ is pruned and there exist 7 values of β which can make $p1$ a active node. In iteration 2, the only active node is $p1.2$ while $p1.1$ and $p1.3$ are pruned. This is because we can see that for node $p1.2$,

$edit(p1.2.sequence, P^{H+\beta'}[2]) + d' + |\beta'_3|$
$= edit(\text{GCTTAC,GCTTAC}) + edit(p1.sequence, P^{H+\beta'}[1]) + 1$
$= edit(\text{GCTTAC,GCTTAC}) + edit(\text{GGTAGCG,GGTAGG})) + 1$
$= 0 + 1 + 1 = 2$

In iteration 3, $p1.2.1$ is again verified to be a active node and the result leaf containing a pointer to location 22 of the sequence is appended to $ResultSet$. Finally, in Step 5, $ResultSet$ which contains all the valid locations will be returned.

Level	ed-tree(18,*,(6,6,6)) r=3	ed-tree(18,*,(6,6,6)) r=2
1	63.5082%	90.8696%
2	99.4152%	99.7760%
3	99.8828%	99.7555%

Note: skip interval does not influence the experiment result here.

Figure 10. Pruning Rate

We carry out an experiment to analyze the effect of the sequence pruning in each level of the $ed\text{-}tree$. 2,000,000 sequences pairs with the length of 18 are randomly selected from the esthuman.z database. In the pruning of level l, sequence pair (S_1, S_2) will be remained only if there exists $\beta = [\beta_1, ..., \beta_t]$ to satisfy $\sum_{i=1}^{l} edit({S_1}^H[i], {S_2}^{H+\beta}[i]) \leq r$. For level l, $1 \leq l \leq 3$, the pruning rate, μ_l, can be formalized as follow:

$$\mu_l = \frac{the\ number\ of\ the\ sequence\ pairs\ remained\ in\ level\ l}{the\ number\ of\ the\ sequence\ pairs\ checked\ in\ level\ l}$$

Figure 10 shows the pruning rates in the different levels. After the processing of the first two levels, 1-(1-μ_1)*(1-μ_2) of sequence pairs are pruned. For example, we consider the pruning rate after the first two levels. Only fewer than 0.03%(0.22%) sequence pairs needed to be checked in the level 3 for $r = 2(r = 3)$. It greatly speeds up the homology searching. The time complexity is $O(\sum_{i=1}^{3} \prod_{j=0}^{i-1} \mu_j \cdot 4^{\beta_{j+1}})$ where $\mu_0 = 1$.

5 Performance Analysis

We implemented the $ed\text{-}tree$ in C. All our experiments are done on a PC with a Pentium 4 1.6Ghz CPU, 256MB

Iteration	PSet	QSet	ResultSet
0	$(\mathcal{EDT}, [-1,0,1], 0, 0), (\mathcal{EDT}, [-1,1,0], 0, 0)$ $(\mathcal{EDT}, [0,-1,1], 0, 0), (\mathcal{EDT}, [0,0,0], 0, 0)$ $(\mathcal{EDT}, [0,1,-1], 0, 0), (\mathcal{EDT}, [1,0,-1], 0, 0)$ $(\mathcal{EDT}, [1,-1,0], 0, 0)$	$\emptyset$	$\emptyset$
1	$(p1, [-1,0,1], 1, 1), (p1, [-1,1,0], 1, 1)$ $(p1, [0,-1,1], 1, 1), (p1, [0,0,0], 1, 1)$ $(p1, [0,1,-1], 1, 1), (p1, [1,0,-1], 1, 1)$ $(p1, [1,-1,0], 1, 1)$	$(\mathcal{EDT}, [-1,0,1], 0, 0), (\mathcal{EDT}, [-1,1,0], 0, 0)$ $(\mathcal{EDT}, [0,-1,1], 0, 0), (\mathcal{EDT}, [0,0,0], 0, 0)$ $(\mathcal{EDT}, [0,1,-1], 0, 0), (\mathcal{EDT}, [1,0,-1], 0, 0)$ $(\mathcal{EDT}, [1,-1,0], 0, 0)$	$\emptyset$
2	$(p1.2, [1,0,-1], 2, 1)$	$(p1, [-1,0,1], 1, 1), (p1, [-1,1,0], 1, 1)$ $(p1, [0,-1,1], 1, 1), (p1, [0,0,0], 1, 1)$ $(p1, [0,1,-1], 1, 1), (p1, [1,0,-1], 1, 1)$ $(p1, [1,-1,0], 1, 1)$	$\emptyset$
3	$\emptyset$	$(p1.2, [1,0,-1], 2, 1)$	$\{22\}$

Figure 9. Processing for the example in step 4

of SDRAM and a 7200rpm 20GB harddisk running Windows XP. Two databases are used for our experiments, esthuman.z and estother.z which were both downloaded from the NCBI website. The databases are composed from the alphabets A,C,G,T,N with N representing a wildcard i.e. N can be any one of the four alphabets. After removing the wildcard character, the estother.z database contains 2.07G bases while the esthuman.z database contains 1.55G bases.

Figure 11 shows the sizes of various ed-$tree$s that are built on the two databases with the different parameters. We note that the first two levels of the ed-$tree$ take up at most 96MB of storage and could easily be stored in the main memory. The third level of the ed-$tree$ on the other hand will require storage of up 3GB.

s	H	Dataset	Level-1,2(MB)	Level-3(MB)
2	[6,6,6]	est_human	96	2720
2	[6,6,6]	est_other	96	3036
2	[6,5,7]	est_human	24	2713
2	[6,5,7]	est_other	24	3003
3	[6,6,6]	est_human	96	2075
3	[6,6,6]	est_other	96	2317
3	[6,5,7]	est_human	24	2068
3	[6,5,7]	est_other	24	2304

Figure 11. ed-$tree$ **Index Sizes,** $w = 18$

5.1 Comparison of ed-tree and Blastn

Our focus here is to compare the efficiency of ed-$tree$ against the latest version of Blastn(NCBI Blastn2) which is available from the NCBI website. The sensitivity of the probe models that are supported by the ed-$tree$ is in fact comparable to Blastn as demonstrated in an earlier section. We conducted two sets of experiments to compare the efficiency of the ed-$tree$ against Blastn by varying the database size and the query length. In the experiments, the length of probe sequence w is set to 18 and H is set to $[6, 6, 6]$.

The first set of experiments is designed to evaluate how the performance of Blastn and ed-$tree$ varies with the database size. Three databases of varying sizes are used. The first two are the esthuman.z and estother.z databases while the third is a 'hybrid" dataset containing 2.3G bases, which is created by combining the whole esthuman.z and a part of estother.z. 1000 query subsequences, each with length of 250 bases, are randomly selected from the DNA of yeast(Saccharomyces cerevisiae) and the average time for answering the query is taken. As shown in Figure 12, the increase in query time for Blastn is much more significant than the ed-$tree$ algorithms as Blastn suffers from its high I/O cost in large sequence databases. For $r = 2$, the ed-$tree$ can be faster than Blastn by a factor of up to 6. For $r = 3$, the ed-$tree$ still outperforms Blastn when the database size increased. We believe that the ed-$tree$ will be much more capable in handling large DNA sequence databases than Blastn.

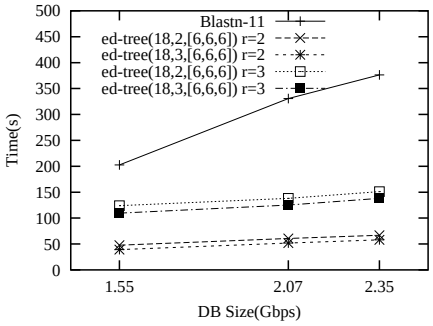

Figure 12. Speed vs DB Size (Query length=250)

Another set of the experiments is carried out to investigate how the query length influences the efficiency of the ed-$tree$ compared to Blastn. We randomly select 1000 subsequences with length varying from 30 to 250 bases from the yeast DNA. For each query length, we take the average query time and plot them against the query length. Figure 13 and 14 depict the relationship between the query time of the various algorithms versus the query length for the two databases. The query time of ed-$tree$ increases linearly with the query length since the number of probes in

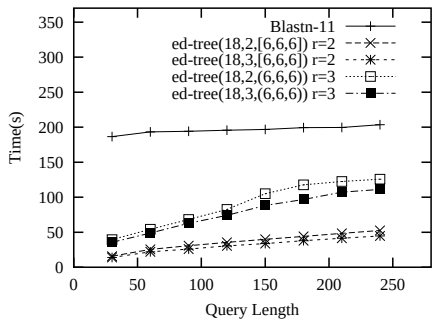

Figure 13. DB:est_human 1.55Gbps

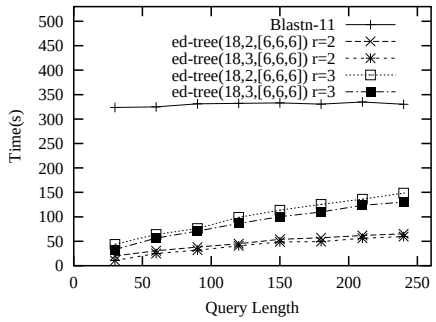

Figure 14. DB:est_other 2.07Gbps

the ed-$tree$ grows linearly. As such, there are likely to be more hits on different parts of the databases which will incur more I/Os for the search. We note that in general, indexes will not be useful if most parts of the database have to be accessed and this is also applicable in the case of the ed-$tree$.

Blastn's performance is not significantly affected by the length of the query since its running time is dominated by the I/O time of its sequential scan which will not increase substantially for the longer queries. Notwithstanding, Blastn's performance is typically an order of magnitude slower than the ed-$tree$'s.

We will next look at the effect of parameter settings on the ed-$tree$'s performance. Since there exist a large number of combinations for the parameter values of the ed-$tree$, we can only provide more insight for the more important ones.

From Figure 13 and 14 which are shown earlier, we make the following observations on the effect of r and s:

1. Query time of the ed-$tree$ increases with increasing r
   Increasing r, the edit distance range means that a more relaxed query constraint is being specified. Naturally, this means that a larger result set will be returned. From the difference between the result of $r = 2$ and $r = 3$, we can tell that a difference of one edit operation can change the size of the result set substantially causing a significant increase in I/Os.

2. Query time of ed-$tree$ decreases with increasing s
   Since increasing the skip interval s decreases the number of subsequences being indexed in the database, the number of I/Os operations for the search will also decrease. The effect of varying s is however less significant than r as it can be seen from Figure 13 and 14.

5.2 Effect of Parameters

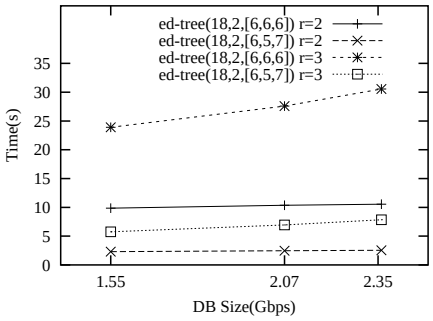

Figure 15. Level 1,2 Pruning time vs DB Size

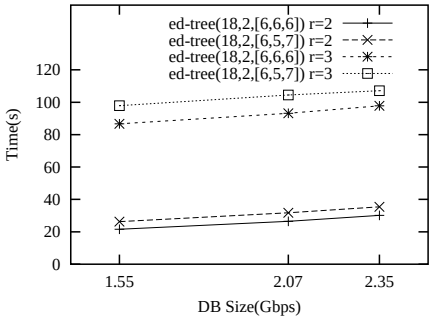

Figure 16. Level 3 Pruning time vs DB Size

We look at how the segment length vector H affects the performance of ed-$tree$, more specifically the pruning processes at the first and second level of the ed-$tree$. Figure 15 shows that the pruning time at the first two levels grows moderately with the database size for a query length of 250. One interesting observation here is that the ed-$tree(18, 2, [6, 6, 6])$ took more time for pruning than ed-$tree(18, 2, [6, 5, 7])$ regardless of the value of r. This is because there are significantly less nodes in the second level when H is set to $[6, 5, 7]$. For $H = [6, 6, 6]$, there are 4^{6+6}(=16, 777, 216) nodes in level 2, compared

to 4^{6+5}(=$4,194,304$) nodes for $H = [6, 5, 7]$. At the third level however, r plays a more important role as shown in Figure 16 where the search time for $r = 3$ is always significantly higher than that for $r = 2$. In this case, the I/Os required for retrieving a larger result set dominate the cost.

Likewise, Figure 17 and 18 confirm that H has a more significant effect on the first two level of pruning while r has more effect on the third level pruning. Overall, since the I/Os at the third level dominate the total pruning time for the three levels, it makes sense to try to adjust H such that a small increase in pruning time at the first two levels causes a substantial decrease in the I/O time at the third level. In fact, this is the main principle we adopted in fine tuning H to achieve the best performance.

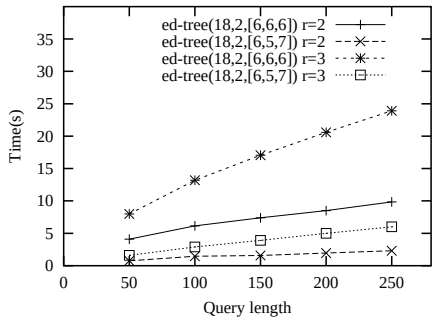

Figure 17. Level 1,2 Pruning time vs Query Length

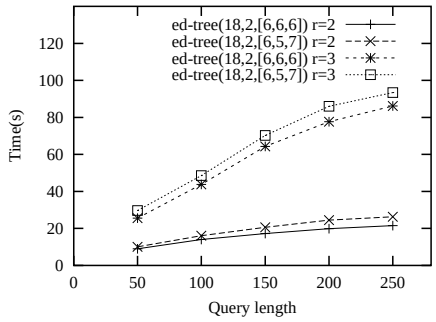

Figure 18. Level 3 Pruning time vs Query Length

6 Conclusion

In this paper, we proposed the $ed\text{-}tree(w, s, H)$, a new tree-structured index, for searching meaningful alignments in large DNA sequence databases. The index structure is constructed by sliding a window w on the target sequence database with the skip interval s. In each window, the subsequences are segmented based on a segment length vector, H.

According to our experimental results, the query time using $ed\text{-}tree$ is up to 6 times lower than Blastn for large DNA sequence databases and this performance gap grows with the size of the DNA sequence database. Unlike previous sequence indexes, the size of the $ed\text{-}tree$ is at most 3Gb for a sequence database of 2 billion base pairs. Considering that the mapped human genome contains around 3 billion base pairs, we believe that the $ed\text{-}tree$ is well positioned for searching large DNA sequence database on a desktop computer.

References

[1] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[2] S. Burkhardt, A. Crauser, P. Ferragina, H. P. Lenhof, and M. Vingron. q-gram based database searching using a suffix array (quasar). In *Int. Conf. RECOMB*, Lyon, April 1999.

[3] E. Giladi, M. Walker, J. Wang, and W. Volkmuth. Sst: An algorithm for searching sequence databases in time proportional to the logarithm of the database size. In *Int. Conf. RECOMB*, Japan, 2000.

[4] R. S. C. Goble, P. Baker, and A. Brass. A classification of tasks in bioinformatics. *Bioinformatics*, 17:180–188, 2001.

[5] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *International Journal on VLDB*, pages 139–148, Roma, Italy, September 2001.

[6] T. Kahveci and A. Singh. An efficient index structure for string databases. In *Int. Conf. VLDB*, Roma, Italy, 2001.

[7] K.Sadakane and T.Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, pages 175–183, 2001.

[8] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[9] S.F.Altschul, W.Gish, W.Miller, E.W.Myers, and D.J.Lipman. Basic local alignment search tool. *Molecular Biology*, 215:403–410, 1990.

[10] D. J. States and P. Agarwal. Compact encoding strategies for dna sequence similarity search. In *ISMB*, 1996.

[11] T.F.Smith and M.S.Waterman. Identification of common molecular subsequences. *Molecular Biology*, 147:195–197, 1981.

[12] H.E. Williams and J.Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14:63–78, 2002.