Using Suffix Trees to Find Most Specific Substring

Tomáš Vinař

Project report for CS740C
Spring 2000

Contents

1 Introduction and Problem Statement

In this project we will explore possible use of suffix trees in approximate string matching. Motivation problem for our study is as follows.

Given is a string G (*gene*) of length l on the alphabet $\{A, C, G, T\}$ and the universe U, which is a very long string over the same alphabet. Also given is a number m.

Definition 1 *String P has a distance k to the universe U under given distance function d, if for some substring Q of the universe U $d(P, Q) = k$ and for all other substrings Q' of the universe U $d(P, Q) \geq k$.*

The problem is to find substring P of G of length m, which is the *most specific*. The *most specific* means that it has the largest distance from U among all substrings of G of length m.

The problem is motivated by studies in bioinformatics. In the design of oligonucleotide sequences, biologists need to determine short substrings of gene coding regions, that will be similar to the specified target (gene G), but not to anything else (other genes). The most of the work on this problem so far was focused on finding good model for description of similarity (see for example [12]). However, not much effort was spent in finding good algorithms to deal with the problem itself.

However, design of such algorithms is essential for this problem. Size of the universe U can be very large and brute force methods are becoming infeasible. Table 1 summarize the size of such real problem.

Description	**Approx. size**
Size of gene G	$l \approx 2000$
Size of universe U	$n \approx 100\,000\,000$
Size of subsequence we are searching for	$m \approx 50$
Size of alphabet	$\sigma = 4$

Table 1: Size of real life problem

In this project we will give overview of known methods using suffix tree technique, which were applied to similar problems. We will also present several ideas, how to apply these methods in our specific problem.

Notice, that so far we did not mention the distance function which we are going to use for this work. We will explore two models:

- **Edit distance model.** This model is well-studied and there are lots of interesting result based on this model. However, it does not model biology processes very well.
- **Weighted Hamming distance model.** This model is closer to real-life biology problems. However, as far as we know, little work was done on approximate string matching in this model.

We will usually explore following related problems instead of the originally proposed problem, where convenient:

- **Closest match problem.** Find closest match to the given string P.
- **Threshold version of the most specific substring problem.** Find, whether there exist a substring P of gene G of length m, such that its distance to any other substring of G is more than given threshold.
- **Threshold version of the closest match problem.** Find, whether distance of string P and universe U is larger than given threshold.

In the following sections we first give a short description of notation we use on suffix trees, then we will explore known results on edit distance model. After that we will explain Weighted Hamming distance model and its relation to the real life problems. Finally, we will present several ideas, how to apply results from edit distance model to Weighted Hamming distance model.

From now on, we denote a substring of string Q starting at ith position and ending at jth position by $Q_{i,j}$ and ith character of string Q by Q_i.

2 Suffix Trees

In most cases we will assume that we have already generated *suffix tree* of the universe U. This can be done in $O(n)$ time using one of the well known algorithms [18, 26].

For any node x in the suffix tree, we call string corresponding to the path from the root to the node x *label of node x*. The length of this label is *depth of the node x*.

In some cases we will also assume, that *suffix links* in the suffix tree are present. Suffix link is a pointer from node with label $L_{1,p}$ to the node with label $L_{2,p}$. Suffix links are by-product of construction of the suffix tree by any of the algorithms mentioned above. They will be extremely useful in some algorithms presented.

Finally, it is necessary to note, that in most of the algorithms we work with suffix tree structure as with ordinary trie (suffix tree without skip fields). It can be easily seen, that trie can be simulated on suffix tree (we just need to use a pointer to a suffix tree node plus an offset on the edge).

3 Edit Distance Model – Overview of Past Work

The edit distance model is the most studied model of approximate string matching to date. The problem of approximate string matching defined via edit distance model proved to be useful in molecular biology as well as in many other different areas.

Definition 2 *Two strings P and Q have* edit distance $d(P,Q) = k$ *if it is possible to get string Q from string P using at most k operations of insertion, deletion and replacement, and if k is the smallest such number.*

In this section we give a short overview of known solutions of the following question: Given a universe U of length n, string P of length m and threshold k. Does P *appear* in U, i.e. is there a substring Q of U for which $d(P,Q) \leq k$? We call one instance of such a problem the *query* and the resulting substring of U is called an *alignment*.

We assume, that we have already constructed a suffix tree T of U. As we have mentioned in the previous section, this can be done in $O(n)$ time as a preprocessing step. We can ask several types of questions about how fast we can perform a query on T.

- Is it possible to perform a query on T in *sublinear time* in the *worst case*?
- Is it possible to perform a query on T in *sublinear expected time*?

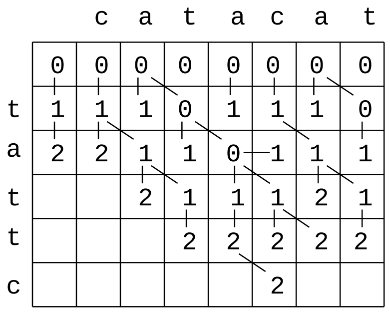
We will present some of the answers available for these questions. Our intention is to present different approaches known, therefore we will not go into details of time complexity analysis of presented algorithms, which can be found in corresponding papers. We will also omit linear time approximate string matching results and parallel algorithms available for this task (see [17]) and incremental algorithms (although these algorithms can be of interest in solving our original problem) [16].

Note, that for the sake of simplicity, the most of the time complexity results are presented only for size of alphabet $\sigma = 4$, which is of particular interest from biology point of view. Original results can be found in corresponding articles.

3.1 Dynamic Programming Algorithm

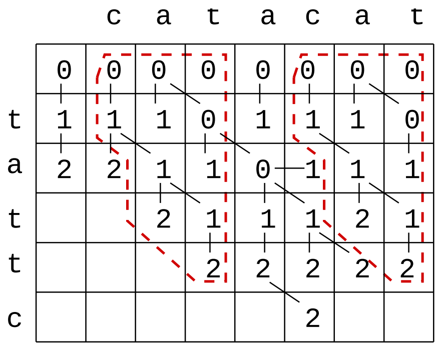
The simple solution for our task is to use dynamic programming to compute alignment. We will compute a matrix $D(0 \ldots m, 0 \ldots n)$, where $D(i,j)$ will represent a minimum possible edit distance of $P_{1,i}$ and a substring of U ending at position j. Solution for our problem are all entries $D(m,j)$, where $D(m,j) \leq k$.

Notice, that to get match with optimal number of errors, we can only either delete character P_i or U_j (reducing problem to compute $D(i,j-1)$ or $D(i-1,j)$) or leave last characters as a match (in the case that $P_i = U_j$) or mismatch (otherwise) (reducing problem to compute $D(i-1,j-1)$). Therefore, it is easy to see that following holds:



Threshold: $k = 2$
Optimal alignment:
```
cata--cat
  tattc
```

Figure 1: Example of dynamic programming algorithm matrix D for $k = 2$, where $U = catacat$ and $P = tattc$.



$V_0 = \{\epsilon_0\}$
$V_1 = \{t_0, \epsilon_1\}$
$V_2 = \{t_0, a_1, t_1, tac_1, \epsilon_2\}$
$V_3 = \{at_1, ta_1, tac_1, a_2\}$
$V_4 = \{at_2, ta_2, tac_2, taca_2\}$
$V_5 = \{tac_2\}$

Figure 2: Dynamic algorithm can perform the same work several times. See marked parts of the dynamic programming matrix. **Right:** Sets V_i for this dynamic programming matrix. Entries are of the form $v_{D(v)}$.

$$D(i,j) = \begin{cases} 0, & \text{if } i = 0, \\ D(i-1,j)+1, & \text{if } j = 0 \text{ and } i > 0, \\ \min\{D(i,j-1)+1, D(i-1,j)+1, D(i-1,j-1)+1\}, & \text{if } i,j > 0 \text{ and } P_i \neq U_j, \\ \min\{D(i,j-1)+1, D(i-1,j)+1, D(i-1,j-1)\}, & \text{if } i,j > 0 \text{ and } P_i = U_j. \end{cases} \tag{1}$$

With help of (1) we can easily design a dynamic programming algorithm. We can for each (i,j) also keep track of how we obtained the minimum value $D(i,j)$ (choose vertical edges over diagonal edges over horizontal edges whenever equal cost choices exist). From this information we can determine for every (i,j) the *shortest* substring $V(i,j)$ of U ending at position j for which $d(V(i,j), P_{1,i}) = D(i,j)$. In particular, exactly all strings $V(m,j)$ for which $D(m,j) \leq k$ are solutions of our query.

Figure 1 shows example of dynamic programming algorithm computation. Notice that cells $D(i,j)$ with value greater than k can be omitted.

The dynamic programming algorithm works in $O(mn)$ time per query.

3.2 Simulation of Dynamic Programming Algorithm on Suffix Trees

The first algorithm we will present here was gradually developed in [15, 25, 7]. Notice, that one problem with dynamic programming is, that if U is a long string, some short substrings will repeat several times in it. Intuitively, our dynamic programming algorithm will perform the same work several times in such case (see Figure 2).

We will show how to avoid such a duplicated work. The main idea is to employ the same dynamic programming algorithm, but use suffix tree T instead of string U.

We can easily see, that if $V(i,j) = V(i,k)$, then also $D(i,j) = D(i,k)$. Therefore following notation is well-defined: let $D(V(i,j)) = D(i,j)$. It is sufficient to compute for every row i only entries $D(v)$, where

$v \in V_i$ and[1]

$$V_i = \{V(i,j) \mid 1 \le j \le n, D(i,j) \le k\}.$$

See Figure 2 for example of sets V_i.

Using suffix trees, we can compute set V_i from set V_{i-1}. After m iterations of such step, we will get V_m, which is desired result[2]. Cobbs [7] gave all details necessary for such an algorithm to be reasonably efficient. For alphabet of size $\sigma = 4$ his algorithm works in $O(4^k.m^{k+2} + t_o)$ *worst case* time, where t_o is the size of the output.

The algorithm give us an opportunity to answer query in time which is *independent of size* n. Unfortunately, although time complexity grows reasonably with length of query, it grows *exponentially* with number of errors k. Therefore the algorithm is suitable only for small values of k. The time is *sublinear* for $k < \frac{\log n - 2\log m}{2+\log m} + O(\log^{-1} m)$. For our values we can allow $k = 2$ to obtain sublinear running time.

It is necessary to note, that the time complexity analysis performed is *worst case* in contrast to other algorithms for which we will consider *expected case* time complexity. As far as we know, no average case analysis was performed on this algorithm. It would be interesting to perform average case analysis for this algorithm.

3.3 Partitioning To Exact Match

The previous algorithm used suffix tree data structure to perform dynamic programming algorithm on all substrings "at once". We will now present an algorithm by Baeza-Yates and Perleberg [3], which uses the suffix tree structure in completely different way.

The following lemma easily follows from pigeon hole principle:

Lemma 3 *Let P is a string of length m, which matches some string Q with edit distance at most k. Then if we partition string P into q substrings, at least one of substring must match string Q with edit distance at most $\left\lfloor \frac{k}{q} \right\rfloor$.*

In particular, for $q = k + 1$, it follows that at least one of the $q + 1$ substrings of P must match some substring of U *exactly*.

Therefore we will use following simple method for our task:

1: Partition query P into q equal size strings.
2: Perform exact matching search using suffix tree structure for every substring.
3: For every position i of exact match use dynamic programming algorithm on substring $U_{i-m-k,i+m+k}$.

In [3] expected case analysis is performed on this simple algorithm. The expected time complexity of algorithm is $O\left(\frac{1}{4}^{\frac{m}{k+1}} m^2 kn\right)$. Although the time complexity is dependent on n, it grows linearly with k. In particular, time complexity is $O(n^{1-\epsilon})$, if $k < m\frac{2}{3\log m + \epsilon \log n}$. For our values of m and n we can consider algorithm to be sublinear for $k < 0.11m$ (i.e. we can allow 11% error).

There are several possible improvements of this method. Baeza-Yates et al. propose in [1] instead of partitioning P into $k + 1$ equal size substrings, to use dynamic programming and additional information available in suffix tree to determine partitioning which gives least possible number of candidate positions which need to be checked. Shi in [23] suggests to split pattern P into $k + s$ parts instead of $k + 1$ parts. These directions need to be explored in the future.

Although partitioning to exact match is not best of the methods available, it is very easy to implement. Also, this technique is known to be very good for practical purposes.

3.4 Hybrid Methods

We will finish this section with very brief description of two hybrid methods. Recall the main problems we have encountered in previous sections. Methods, which explored suffix trees and tried to simulate dynamic

[1] In other words: for every two words $V(i,j)$ and $V(i,k)$ for which $v = V(i,j) = V(i,k)$ compute $D(v) = D(i,k) = D(i,j)$ only once.

[2] It is possible to imagine this algorithm as computation of dynamic programming on all paths in suffix tree in parallel.

programming on them, performed very well for low error rates. However, time grew exponentially with k. On the other hand, partitioning methods performed a little better, but on some strings (especially with often repeating substrings) they would spend too much time on performing dynamic programming on too many selected (and possibly similar) candidates.

Fortunately, when using Lemma 3, we are not forced to use $k+1$ or more partitions. Instead we can only reduce k - we can get k small enough to proceed with dynamic programming technique simulation and in this way get smaller number of candidates that need to be verified.

The best-known technique working along these lines was proposed by Myers [20]. The algorithm is too complex to describe here in detail, instead we give only main ideas.

The Myers method first partitions string P into subpatterns of length $O(\log_\sigma n)$ of approximately equal length, then finds candidates for match and combines results in sophisticated way to obtain final results. Search for candidate is done in exhaustive way — all substrings in edit distance k are checked (it is possible to use suffix tree for this, however Myers uses a simpler index). The analysis of time complexity of this algorithm is quite complicated, Method performs in expected time $O(kn^{1-\epsilon}\log n)$ for $k < 0.33m$ (for $\sigma = 4$).

Recently, Baeza and Navarro [2] came with a new method based on similar approach. They partition string P into j equal size substrings, find approximate matches with $\left\lfloor \frac{k}{j} \right\rfloor$ errors (using technique similar to one mentioned in Section 3.2) and check all resulting candidate sites by dynamic programming algorithm. They analyzed the algorithm and found, that when optimal value of j is set ($\Theta(m/\log_{\sigma n})$ with "complicated constant"), the algorithm has expected time complexity $O(n^{1-\epsilon})$ for values of $k < 1 - \frac{e}{\sqrt{\sigma}}$, which gives $k < 0.29m$ for $\sigma = 4$. Moreover, they claim that with their approach, they can replace suffix tree data structure with less memory expensive suffix arrays with addition $O(\log m)$ factor.

3.5 Summary

Usage of suffix trees in context of edit distance model of approximate string matching is well explored. We have presented several results to illustrate main techniques used in past, namely:

- **Simulation of sequential algorithm on suffix tree.** These methods performed well on small error rates, however, time complexity grows exponentially with value of k (error rate). On the other hand, algorithms had worst time complexity *independent* of n.
- **Partitioning to exact match.** These algorithms tried to use suffix trees in another way — they reduced approximate match problem to exact match problem. However, sometimes these algorithms generated too many candidates and it took a long time to perform dynamic programming algorithm to check them all. These algorithm can perform in expected time sublinear in n with error rates about 10% (computed for our values of n and m).
- **Hybrid methods.** Hybrid methods combined advantages of both of previously studied approaches. They were able to perform in expected time sublinear in n with error rates about 30% (error rate is independent of m and n).

4 Weighted Hamming Distance Model

The model, which we present here is motivated by effort of biologists to describe process of DNA hybridization. Models based on thermodynamic properties of hybridization proved to have enough predictive capability for practical purposes.

The main problem can be stated as follows: given two strings P and Q of the same length over the alphabet $\{A, C, G, T\}$. The question is, whether pieces of DNA represented by these two strings will bind, and if so, how strong such bound will be.

If two string have on corresponding positions only corresponding pairs of letters (i.e. $A-T$, $T-A$, $G-C$, or $C-G$), it is well known that strong bound will occur. However, the question is more complicated, when there are mismatches in these strings.

In [6] and [22], the *nearest-neighbor* model is presented. The basic idea is to compute free energy (the lower the free energy, the stronger bound will occur) using experimentally measured parameters.

Parameters for model consist of a table, where for every four-combinations (AB,CD) of characters a real value $\Delta G(A, B, C, D)$ is given. Also, for any combination of two letters additional value $\Delta G_{init}(A, B)$ is given.

The free energy is computed as follows:

$$\Delta G(P, Q) = \Delta G_{init}(P_1, Q_1) \sum_{i=1}^{|P|-1} \Delta G_(P_i, P_{i+1}, Q_i, Q_{i+1}) \tag{2}$$

Parameters for the model are gradually becoming available in different publications, the summary of parameters can be downloaded from [21]. At least two computer programs are available based on this model: HYBSimulator [12] and HyTher [5, 13].

The *weighted Hamming distance model* is simpler model than nearest neighbor model. For every pair of characters (A, B) we have a parameter $d(A, B)$. The distance of two strings is then defined by the sum of distances of corresponding characters, i.e.

$$d(P, Q) = \sum_{i=1}^{n} d(P_i, Q_i) \tag{3}$$

Note, that the nearest neighbor model can be embedded in weighted Hamming distance model. We will use 25 character alphabet, every character pair of characters A, C, G, T, s, where s means beginning of the sequence. The string P will be then rewrited as $[sP_1, P_1P_2, P_2P_3, \ldots, P_{n-1}P_n]$. Therefore weighted Hamming distance model is reasonable model for our task (although it would be probably better to adapt algorithms presented below to nearest neighbor model, than use such a simulation).

Also note, that we are omitting effects of possible alignment with gaps and other interesting cases that can occur. We do so because of two reasons:

1. We want to keep this initial study simple and then extend results to other, more complicated models.
2. There is not enough experimental data to establish new model parameters for that task.

5 A Little Experimental Study

We have performed a little experimental study to see, whether methods from edit distance model can be successfully applied to weighted Hamming distance model. Results here are very preliminary.

5.1 Implementation

We tried to implement an algorithm for solving the following problem: Given universe U and pattern P. Find a substring of U, which is *closest* to P according to a given weighted Hamming distance function.

We used following algorithm for the task:

1: Build a suffix tree T for U.
2: Use A^* search algorithm (described below) to search for best approximate match of P in T.

For suffix tree construction and traversal we used library for suffix trees available in STRMAT package [11]. Unfortunately, we found out that the version of this package which is available contains several bugs in suffix tree library. Therefore, before performing other experiments, we will need to implement new suffix tree library. Library uses 24 to 28 bytes of memory per character of the input. It implements Ukkonen's [26] algorithm for suffix tree construction and therefore provides also suffix links.

The A^* search algorithm is used to find a leaf with optimal cost in a tree (see for example [4, p.64nn]) – in our case a suffix tree. It is assumed that all edges in the tree are weighted and cost of a node is a sum of tree edges from root to the node[3]. Denote cost of node x by $f(x)$. The algorithm uses so called *heuristic*

[3]For general state spaces it is assumed, that all edges have non-negative cost. However this assumption can be dropped in case of search trees.

function $h^*(x)$, which for every node x provides a lower bound of cost of the path from node x to the leaf with minimum cost.

At the beginning, we put a token on the root of the tree. In each step:

1: Token t with smallest value of $f(x) + h^*(x)$ is selected.
2: If t is positioned in the leaf l, the leaf l is the leaf with the optimal cost.
3: Otherwise token t is destroyed and new token is put to every children of token t (we have expanded token t)

Note that after every step of the algorithm there is exactly one token on each path from the root to a leaf of the tree.

The A^* search algorithm is *optimal* in the sense, that if A is the other searching algorithm and A^* is more informed than A (heuristic function), then A^* *will never expand* any node which will not be expanded by A [9].

The implementation uses a heap for storing token positions and to perform priority queue operations needed by A^* algorithm in $O(\log h)$ time (see for example [8, p.149nn]), where h is a size of the heap.

Notice, that since the number of leaves in a suffix tree is at most n, and also we examine maximum of m characters on each tree branch, the upper bound for the running time is $O(mh \log h)$, where $h = \min\{\sigma^{m+1}, n\}$. We conjecture, that time complexity in expected case is much better.

Also note, that trivial approach (checking every alignment) has time complexity $O(mn)$.

5.2 Results

As a universe U we have used subsets of 100, 200, 500, 1000, 2000 and 5000 gene coding regions from 19k gene expression array set[4] and we have processed queries of size 10, 15, 20, 25, 30, 40, 50 (subsequences of a gene from 19k set not used in U). We were interested in *number of characters*, which were explored during the computation of the closest match.

We used following weights for Hamming distance (motivated by 2-4 rule – see for example [24, p.106]):

- score -4 for matching G or C characters,
- score -2 for matching A or T characters,
- score 0 for GT mismatches (i.e. combinations AG, GA, CT, TC)
- score 10 otherwise

The results of our small experiment are presented on Figures 3, 4, 5. Several observations can be made from the graphs:

- For small sizes queries ($m = 10$ and $m = 15$ – see Figure 3) only a very small part of the whole sequence is explored in order to find the closest match. Except for small cases, the number of explored characters seem to be independent of n (size of U). This can be explained by the fact, that with large values of n several first levels of the suffix tree become saturated and therefore with larger n nothing is added to the portion of the suffix tree which is explored by the algorithm.
- For query of medium sizes (m between 20 and 30 – see Figure 4), the behavior of the algorithm becomes quite interesting. The number of explored characters grow linearly for smaller values of n. However at some point, number of characters continues to grow only very slowly ($m = 25$) or drops ($m = 30$ and $m = 20$).

  Remember, that in theoretical analysis of the edit distance case, time complexity depended very much on the error rate. With larger values of n, we get closer matches for our queries. If close match exists in the tree, we are able to find it very quickly.
- For large sizes of query ($m = 40$ and $m = 50$ – see Figure 5), growth seems to be linearly dependent on n. Notice, that even in these cases the number of explored characters is much less than the number of characters explored by trivial algorithm (mn).

[4] Data was obtained from Dr. Aled Edwards, Department of Molecular Biophysics, University of Toronto.

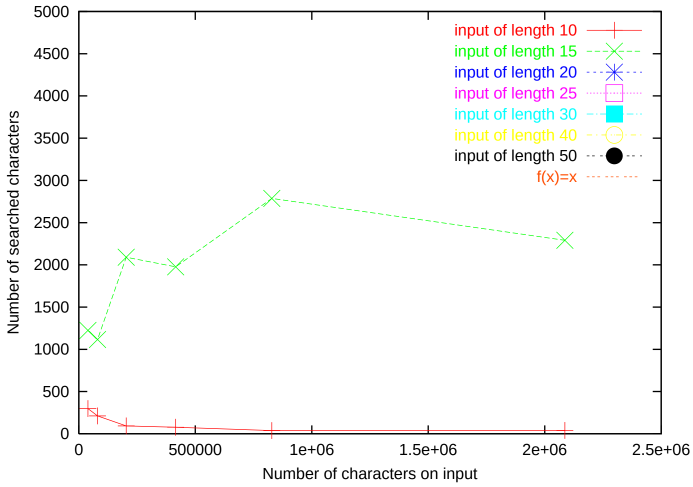


Figure 3: Behavior of the algorithm for small size queries ($m = 10$ and $m = 15$).

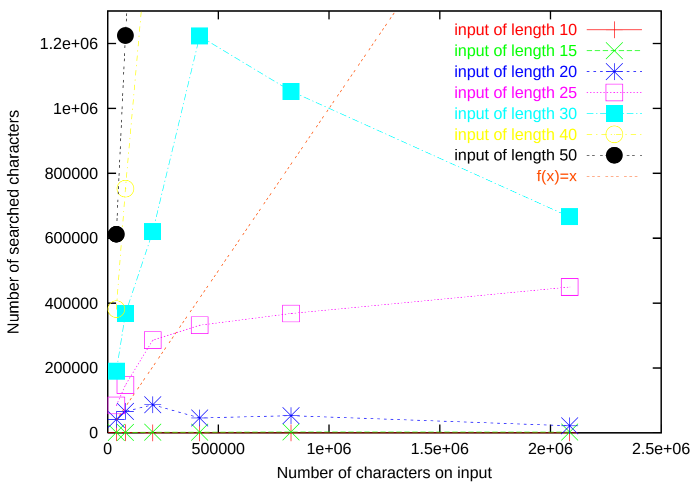


Figure 4: Behavior of the algorithm for medium size queries ($m = 20, 25, 30$).

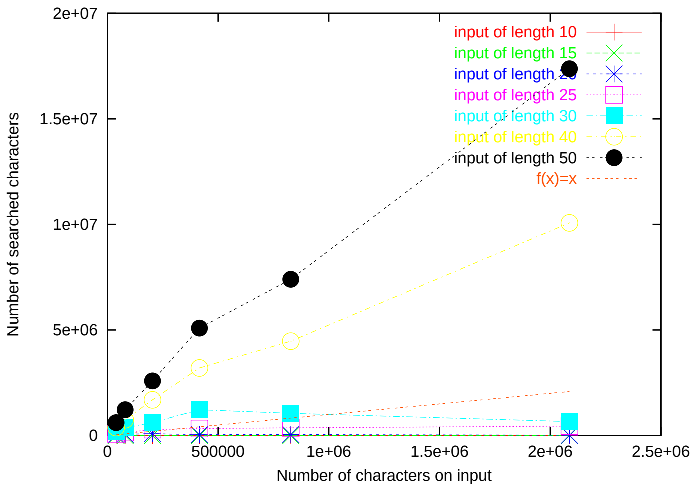


Figure 5: Behavior of the algorithm for large size queries ($m = 40, 50$).

5.3 Summary

We have implemented a simple algorithm for finding closest approximate match to the query using suffix trees and A^* search algorithm. We also performed a little experimental study on real biological data with interesting results and proposed several conjectures about the algorithm behaviour. These conjectures need to be yet explored by theoretical expected time analysis.

6 Further Work on The Problem

In this section we propose several extensions of this work which can be done. Some of them could lead to fast solution of our motivating problem.

6.1 Applying Partition Techniques

Most of the partition techniques mentioned in context of edit distance model relied heavily on the fact, that error could be expressed as an integer number. Using this we were able to eliminate approximate string matching and replace it by exact matching techniques on suffix trees.

This cannot be done directly in the case of weighted Hamming distance. In general case, weights can be arbitrary numbers. Therefore, some other approach is needed to solve the problem.

Notice that in our experiment we still got very good results with decreasing *length* of the pattern, instead of error rate as it was in the case of edit distance model. Therefore the following algorithm could work reasonably well (threshold closest match problem):

1: Partition P into equal length substrings of size j.
2: Apply algorithm described in Section 5 to every substring with threshold k/j getting set of candidate sites.
3: Check all candidate sites by trivial algorithm.

Of course, implementation, experimental study and theoretical analysis is needed in order to evaluate this approach.

6.2 Incremental Computation

So far we only considered searching for *one* string in the universe U. However, our original motivation problem involved searching for *every substring of gene G of length m*. Trivial algorithm would take $O(lmn)$

time to accommodate this task. We will first show, how to speed up the trivial algorithm to $O(ln)$ time[5] and then we will give an idea, how to use this approach together with an algorithm described in Section 5.

The trivial algorithm proceeds as follows:

$ms = -1$; $mscost = -\infty$
for all starting positions i in gene G **do**
    $mincost = \infty$
    for all substring positions j in the universe U **do**
        $cost = d(G_{i,i+m-1}, U_{j,j+m-1})$ {takes $O(m)$ time}
        if $cost < mincost$ **then**
            $mincost = cost$
        end if
    end for
    {$mincost$ is now a cost of the closest match for string $G_{i,i+m}$}
    if $mincost > mscost$ **then**
        $mscost = mincost$; $ms = i$
    end if
end for
{$G_{ms,ms+m-1}$ is the most specific substring of length m}

There is one thing to note here. If we compare $d = d(G_{i,i+m-1}, U_{j,j+m-1})$ and $d' = d(G_{i+1,i+m}, U_{j+1,j+m})$, there will be not a big change. In fact, d' can be computed from d in constant time:

$$d' = d - d(G_i, U_j) + d(G_m, U_m). \tag{4}$$

Using (4) we can easily speed up the trivial algorithm and get time complexity $O(ln)$.

Similar method can be used also in connection with A^* algorithm on suffix trees as it was described in section 5. Note, that both Ukkonen [26] and McCreight [18] constructions of suffix tree provides us with additional information in the tree – suffix links. After we finish search for the closest match of substring $G_{i,i+m-1}$ we can "transfer" tokens to the new positions in the tree rather than starting search again for string $G_{i+1,i+m}$ from scratch.

There are three things which we need to change when "transferring" token. Let token be positioned on a node with label $Q_{1,p}$. Then:

- **New position of the token.** We will use suffix link to change position of the token to the node with label $Q_{2,p}$.
- **New cost.** Notice, that the old cost represented $d = d(Q_{1,p}, G_{i,i+p-1})$. We need to change it to $d' = d(Q_{2,p}, G_{i+1,i+p-1})$. However, this can be easily done in constant time, because $d' = d - d(Q_1, G_i)$.
- **New heuristic estimate.** Similarly, heuristic estimate can be easily adjusted in constant time.

The only problem, we get after "transferring" tokens, is that the property that on every path from root to a leaf there is exactly one token (which is essential to the A^* algorithm) is no longer maintained. If we try to clean up and remove some tokens from the tree, we will get too high time complexity for the algorithm to be useful.

However, we can get away very easily. Notice, that any of the four subtrees of the root will produce after "transferring" the complete set of tokens maintaining the desired property. Therefore we will choose only one subtree and we will throw away all other tokens.

To continue with A^* algorithm, we need to build a new heap of "transfered" tokens. However, this can be done in $O(h)$ time, where h is the number of tokens (see for example [8, p.145nn]).

Again, performance of this method need to be evaluated in practice and theoretical average case bound need to be developed.

[5]Acknowledgement: This part was done together with Broňa Brejová

6.3 Space Issues

So far we have ignored the problem of space. The usual representations of suffix trees take too much space (for example representation in [11] takes about 24-28 bytes per one character of the represented set of strings) to fit in the main computer memory when processing data of such scale as we introduced in the Section 1.

One possibility is to use succinct representation of suffix trees. See [14, 19]. However, there are two problems with these data structures:

- They support only limited set of operations. In particular, neither of those support suffix links, which seem to be quite useful in our context.
- Although these articles show nice and small representations for suffix trees, there is a question, how to construct these representation in this limited amount of memory provided that at the beginning we are give set of strings (not suffix tree itself) in $O(n)$ time.

7 Conclusion

In this project we have explored use of suffix trees in approximate string matching. In particular, we dealt with the problem of finding most specific substring of given string. We investigated the problem under two models: **edit distance model**, which is very simple and there are lots of powerful methods applicable, however sometimes it does not suit real life problems well; and **weighted Hamming distance model** which seem to be good model for hybridization processes in biology – however, there is not much known about approximate string matching under this model.

We have presented several ideas on how to apply methods used in edit distance model into weighted Hamming distance model. However, our presentation lacks theoretical results on average time complexity of proposed methods. Such analysis need to be yet performed.

References

[1] Ricardo Baeza-Yates. Indexing methods for approximate text retrieval. Technical report, Department of Computer Science, University of Chile, 1997.

[2] Ricardo A. Baeza-Yates and Gonzalo Navarro. A new indexing method for approximate string matching. In Maxime Crochemore and Mike Paterson, editors, *Combinatorial Pattern Matching, 10th Annual Symposium*, volume 1645 of *Lecture Notes in Computer Science*, pages 163–185, Warwick, England, 22-24 July 1999. Springer.

[3] Ricardo A. Baeza-Yates and Chris H. Perleberg. Fast and practical approximate string matching. *Information Processing Letters*, 59(1):21–27, 1996. Early version CPM 1992.

[4] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence: Volume 1*. William Kaufmann, Los Altos, California, 1981.

[5] S. Bommarito, N. Peyret, and J. SantaLucia, jr. Thermodynamic parameter for DNA sequences with dangling ends. *Nucleic Acids Research*, 28(9):1929–1934, May 2000.

[6] Kenneth J. Breslauer, Ronald Frank, Helmut Blöcker, and Luis A. Marky. Predicting DNA duplex stability from the base sequence. *Proceedings of National Academy of Sciences of USA*, 83:3746–3750, June 1986.

[7] Archie L. Cobbs. Fast approximate matching using suffix trees. In Zvi Galil and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium*, volume 937 of *Lecture Notes in Computer Science*, pages 41–54, Espoo, Finland, 5-7 July 1995. Springer.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[9] David Gelperin. On the optimality of A*. *Artificial Intelligence*, 8(1):69–76, 1977.

[10] Dan Gusfield. *Algorithms on Strings, Tree, and Sequences*, chapter 12.2, 12.3. Cambridge University Press, New York, 1997.

[11] Dan Gusfield, Jim Knight, and Jens Stoye. STRMAT: A collection of string matching and pattern discovery algorithms. Available from `http://www.cs.ucdavis.edu/~gusfield/strmat.html`.

[12] D. Hyndman, A. Cooper, S. Pruzinsky, D. Coad, and M. Mitsuhashi. Software to determine optimal oligonucleotide sequences based on hybridization simulation data. *BioTechniques*, 20(6):1090–1097, June 1996.

[13] HyTher: DNA hybridization prediction program. Available from `http://jsl1.chem.wayne.edu/Hyther/hythermenu.html`.

[14] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, Research Triangle Park, North Carolina, 30 October–1 November 1989. IEEE.

[15] Petteri Jokinen and Esko Ukkonen. Two algorithms for approximate string matching in static texts. In Andrzej Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248, Kazimierz Dolny, Poland, 9–13 September 1991. Springer.

[16] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, April 1998.

[17] Gad M. Landau and Uzi Vishkin. Approximate string searching. In Alberto Apostolico and Zvi Galil, editors, *Pattern Matching Algorithms*. Oxford University Press, New York, 1997.

[18] Edward M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, April 1976.

[19] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science*, pages 118–126, Miami Beach, Florida, 20–22 October 1997. IEEE.

[20] Eugene W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, October/November 1994.

[21] Parameters for nearest neighbour model. Available from `http://www.hybsimulator.com/download.html`.

[22] John SantaLucia, jr. A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbour thermodynamics. *Proceedings of National Academy of Sciences of USA*, 95:1460–1465, February 1998.

[23] Fei Shi. Fast approximate string matching with q-blocks sequences. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 257–271, Recife, Brazil, 1996. Carleton University Press.

[24] Tom Strachan and Andrew P. Read. *Human Molecular Genetics, 2nd edition*. Wiley, New York, 1999.

[25] Esko Ukkonen. Approximate string-matching over suffix trees. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242, Padova, Italy, 2-4 June 1993. Springer.

[26] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.