

Algorithms for Motif Discovery based on Edit Distance^{*}

Shireesh Thota, Sudha Balla, and Sanguthevar Rajasekaran

Dept. of Computer Sci. & Eng., University of Connecticut, Storrs CT 06269, USA

Abstract. In this paper, we study the problem of identifying sequence patterns of length l in a database DB , consisting of n bio-sequences of average length m each, that have occurrences in at least t distinct sequences of DB , the occurrences being at an edit distance (also called the *Levenshtein distance*) of at most d from the pattern. We survey some algorithms for the problem from the literature and also present two improved algorithms for the same. An implementation and performance results of one of our algorithms is also presented.

1 Introduction

The edit distance based motif discovery problem can be formulated as follows:

Edit Distance Motif Problem (EDMP): Input is a database DB consisting of n sequences (S_1, S_2, \dots, S_n) of average length m from a fixed alphabet Σ . Input also are integers l, d , and t . Output should be all the patterns in DB such that each pattern is of length l and it occurs in at least t of the n sequences. If a pattern v is output, it will mean that v is a string of length l present in one of the input sequences and also that v occurs in at least t of the input sequences. (A string u is considered an occurrence of a pattern v as long as the edit distance between u and v is at most d . For a definition of edit distance see e.g., [5].)

Numerous papers have been written in the past on the topic of motif search, addressing several known variants of the motif discovery problem. An algorithm for EDMP has been given by Sagot [9] that has a run time of $O(n^2ml^d|\Sigma|^d)$ and a space requirement of $O(n^2m)$. An algorithm with an expected run time of $O(nm + d(nm)^{1+pow(\epsilon)}\log nm)$ where $\epsilon = d/l$ and $pow(\epsilon)$ is an increasing concave function has been proposed by Adebisi and Kaufmann [1]. The value of $pow(\epsilon)$ is roughly 0.9 for protein and DNA sequences. A deterministic algorithm is proposed in [8] called algorithm Deterministic Motif Search (DMS) that runs in time $O(n^2ml^d|\Sigma|^d)$ using $O(nmd + l^d|\Sigma|^d)$ space. The algorithm of [9] uses suffix trees whereas the DMS algorithm that employs arrays is expected to perform better in practice and also amenable to parallelization.

In this work, we present a short survey of known algorithms for EDMP from the literature, two new algorithms that promise improved performance over the existing algorithms and an implementation and performance results of one of the proposed algorithms. The rest of the paper is organized as follows. In section 2,

^{*} This research has been supported in part by the NSF Grant ITR-0326155.

we survey known algorithms from the literature. Section 3 gives the details of the two proposed algorithms, Section 4 gives the implementation and performance results of one of the new algorithms and Section 5 concludes the paper.

2 A brief survey of algorithms for EDMP

The suffix tree based algorithm of [9] is as follows. The preprocessing step involves constructing a generalized suffix tree (*GST*) of the input sequences. For a detailed description of suffix trees and their applications in computational biology, please refer [4]. Then, for every substring u of length l in the input sequences, strings that are at an edit distance of at most d from u (the d -neighborhood of u) are searched in the *GST* for their occurrence. If t or more of the input sequences have an occurrence of u , then u is output as a valid pattern in *DB*. The search step on *GST*, called *spelling*, is performed by generating the d -neighborhood of u one character at a time. The spelling of a pattern u on the *GST* would stop when no further extension is possible as the allowable edit distance d is reached or when the length l pattern desired is constructed. In [2], the authors show that the size of the d -neighborhood of a pattern of length l is $O(l^d|\Sigma|^d)$. There are $O(nm)$ substrings of length l in the input and to account for the number of occurrences of a pattern, the algorithm requires a $O(n/w)$ space (w being the length of the computer word) and $O(n)$ time per pattern on each internal node of *GST*. Thus, the time complexity of this algorithm is $O(n^2ml^d|\Sigma|^d)$ and its space complexity is $O(n^2m/w)$.

The algorithm of [1] uses the sublinear approximate matching algorithm of [6] to extend a set of maximal patterns identified in the input strings to discover the patterns of interest. Their algorithm has an expected run time of $O(nm + d(nm)^{1+pow(\epsilon)} \log nm)$ where $\epsilon = d/l$ and $pow(\epsilon)$ is an increasing concave function. The value of $pow(\epsilon)$ is roughly 0.9 for protein and DNA sequences. An implementation and comparison of their algorithm was shown to perform better than the algorithm of [9] on real biological datasets.

Now, we describe a sorting based algorithm for EDMP called DMS of [8] that has the same run time as that of [9] but has the potential of performing better in practice. The basic idea behind the algorithm is: We generate all possible l -mers in the database. There are at most $O(nm)$ such l -mers and these are the patterns of interest. For each such l -mer we want to determine if it occurs in at least t of the input sequences. Let u be one of the above l -mers. If v is a string such that the edit distance between u and v is at most d , then we say v is a neighbor of u . We generate all the neighbors of u . For each neighbor v of u we determine a list of input sequences in which v is present. These lists (over all possible neighbors of u) are then merged to obtain a list of input sequences in which u occurs (within an edit distance of d). Each pattern is represented as an integer, in $2l/w$ computer words and radix sorting is adopted to sort the list of patterns generated. Every l -mer of *DB* can have $O(l^d|\Sigma|^d)$ neighbors. Therefore, the time complexity of the generation of the neighbors and the sorting step is $O(nml^d|\Sigma|^d)$. Each unique pattern can occur in $O(n)$ distinct sequences of the

input and therefore to the time complexity of this algorithm is $O(n^2ml^d|\Sigma|^d)$. This algorithm has not been implemented yet.

3 New algorithms for EDMP

3.1 Algorithm EDMS1

In this section we describe algorithm EMDS1, another sorting based algorithm that eliminates the dependence of the run time of algorithm DMS on the size of the alphabet of the input sequences.

Let u be a substring of length l (l -mer), $|u| = l$. Let v be an occurrence of u in DB . Let i, d, s denote the number of inserts, deletes and substitutions, $k = (i + d + s)$ and $0 \leq k \leq e$, that are performed on v to transform it to u , e being the input edit distance parameter. Then, note that $|v| = (l - i + d)$. Also, an alignment of u and v , say $A_{u,v}$ is of length $(l + d)$. There are, $\Pi = \binom{l+d}{i} \binom{l+d-i}{d} \binom{l+d-i-d}{s} \binom{l+d-i-d-s}{d}$ possible permutations of $A_{u,v}$.

Let $A_{u,v}(\pi_j)$ be one such permutation. Note that the only matches in $A_{u,v}(\pi_j)$ are those positions of u that are not inserts and substitutions and those positions of v that are not deletes and substitutions. For example,

$u = AGACTGC$ and $v = AAGTCA$ and $A_{u,v}(\pi_j)$ is as follows:

```
AGACTGC_
A_AGT_CA
```

There are 2 inserts, 1 delete and 1 substitution operations performed on v to obtain u . The length of $v = 7 - 2 + 1 = 6$ and length of $A_{u,v}(\pi_j) = 7 + 1 = 8$. Note that the matches in $A_{u,v}(\pi_j)$ are $u[1], u[3], u[5], u[7]$, those positions of u that are neither inserts nor substitutions and $v[1], v[2], v[4], v[5]$, those positions of v that are neither deletes nor substitutions.

Let u' be the concatenation of $u[1], u[3], u[5], u[7]$, $\Rightarrow u' = AATC$, and v' be the concatenation of $v[1], v[2], v[4], v[5]$, $\Rightarrow v' = AATC$. Clearly, $u' = v'$.

Algorithm EDMS1 described below uses this observation to identify the patterns of interest in the DB . For each possible permutation, it generates a collection P representing u' of all patterns u from DB and a collection Q representing v' of all occurrences v from DB . P and Q are sorted and merged to identify the patterns of interest.

Let C be a $n \times m$ matrix to maintain the count of occurrences of each l -mer in the DB . Let B be a $n \times m \times n$ Boolean matrix initialized to zero that will be used to keep track of the distinct number of sequences in DB that a pattern has occurred.

Algorithm EDMS1 {

 For each alignment $A_{u,v}(\pi_j)$ from $\Sigma_{k=0..e, (k=i+d+s)} \Pi$, do {

 Step: 1:

 Generate collection P of tuples $\langle u', seq, pos, 0 \rangle$
 from every l -mer of DB ,

```

|u'| = (l - i - s), of Au,v(πj).
Step: 2:
Generate collection Q of tuples < v', seq, pos, 1 >
from every (l - i + d)-mer of DB,
|v'| = ((l - i + d) - d - s), of Au,v(πj).
Step: 3:
Sort P and Q.
Merge P and Q, let the merged collection be R.
Step: 4:
Scan through R (similar to algorithm DMS).
For each occurrence detected,
say < u'x, seqx, posx, 0 >, < v'y, seqy, posy, 1 >,
  If B[seqx, posx, seqy] = 0, then,
    // recording the occurrence of l-mer at (seqx, posx) occurring in seqy.
    B[seqx, posx, seqy] = 1;
    // increment the occurrence count of l-mer at (seqx, posx).
    C[seqx, posx] + = 1;
  }
}
Scan through C and output x, y, C[x, y] for each C[x, y] >= t;
}

```

Analysis:

There are $O(l^e)$ possible alignments. In each possible alignment, generating collections P and Q requires $O(nml)$ time. Sorting and merging P and Q requires $O(nm(l/w))$ time using Radix sort. Step 4 of scanning through R and recording occurrences takes $O(n^2m)$ time. Therefore, time complexity of the algorithm is $O(n^2ml^e)$.

Space complexity of the algorithm is $O(n^2m)$, the space required by array B .

3.2 Algorithm EDMS2

We propose algorithm *EDMS2* which solves *EDMP* in $O((nm)^{1+\delta}l^2)$ where δ is a fractional value decided on empirical basis as is explained in more detail later. It requires a space of $O(n^2m)$. The bounds are clearly better than the existing ones and as shown experimentally later, it gives great results in practice as well.

The main idea of the paper lies in employing Myers [6] idea of dynamic programming approach to establish a d neighbor of a word while traversing a trie of $(l + d)$ mers.

We generate all nm , $(l + d)$ mers in the database DB and construct a trie based out of it. It is to be noted that a d neighbor of a l mer can have utmost length of $l + d$ (starting from $l - d$) and hence we will consider $(l + d)$ mers in the trie. Effectively, the trie will have nm leaves (paths) and the information of each $(l + d)$ mer (like sequence number, position number) in the sequence are stored

in the leaf of its path. Such a trie can be constructed in time $O(nml)$ and in space bounded by $O(nml)$. In practice, it would be much lesser as lot of paths ($(l+d)$ mers) have common prefix path and hence is represented only once.

We use Myers [6] idea to establish d neighbor property for two given words. It involves a dynamic programming matrix for calculating edit distance between two words. But unlike the traditional edit distance calculation, we fix one word (first word) and extend the other word (second word) character by character until we process all characters in it or until we decide categorically the d neighborhood property. Let's denote 'current word' as concatenation of characters considered till a 'current row' in the dynamic programming matrix. The current row directs the search as below:

1. If the last entry of the most current row is d , then the current word is a d neighbor to the first word.
2. If each entry in the most current row is greater than d , then any amount of extension in the second word would not yield a d neighbor to the first word.
3. If at least one entry of the most current row is lesser than or equal to d , then the second word has a potential to become a d neighbor to the first neighbor and hence we need to extend it by one more character to get a new 'current row'

The above process helps to establish an occurrence of a given $(l+d)$ -mer for a given l -mer in utmost $O(l^2)$ time. We make use of an array $A[1:n, 1:m, 1:n]$ to hold the information for each l -mer as to which sequence does it occur in. At the end of the algorithm $A[SeqNum, Pos, j] = 1$ iff the l -mer $(SeqNum, Pos)$ occurs in the input sequence with index j ($1 \leq j \leq n$) at least once. Another array $B[1:n, 1:m]$ is used to store the number of occurrences of each l -mer. At the end of the algorithm, $B[SeqNum, Pos]$ gives us the value of the l -mer $(SeqNum, Pos)$'s occurrences in the whole database DB. Below is the whole algorithm :

```

Trie EDMS2()
{
    A[1:n,1:m:1:n];
    B[1:n,1:m];

    //Construct the trie based on all the l+d mers
    tn = ConstructTrie(l+d);
    for(each l mer, (1,1) : (n,m))
    {
        curRow = 1; //indicates which row of dynamic matrix we are at
        //starting from Root of the trie - get first child
        TrieNode tn = GetNextChild(tn);

        while(tn != null) //until the whole trie (all paths) processed
        {

```

```

cost = 0;    //the last entry in the current row
minVal = 0; //minimum entry in the current row

//Calculate the current row of dynamic matrix for l against tn->element
//Update the cost, minVal finally
ComputeRow(l, tn->element, curRow, cost, minVal);

if(cost == d) //match found
{
    //Updates arrays A, B as all the leaves in the subtree are d
    UpdateOccurrences(A, B, l, tn);

    //prunes the subtree as we have already found d neighbor
    //updates tn's position and curRow value as well
    PruneToGetNextChild(tn, curRow);
}
else
{
    if(minVal <= d)
        { //match not found, but a possibility of a d neighbor exists
            tn = GetNextChild(tn);
            UpdateRow(curRow);
        }
    else
        { //match not found and no possibility of d neighbor exists

            //prunes the subtree as no extension in this subtree
            //gives a d neighbor
            //updates tn's position and curRow value as well
            PruneToGetNextChild(tn, curRow);
        }
    }
}
} //All l mers have been checked

for(each lmer, i=(1,n) : j = (1,m))
{
    if(Sum(A[i,j, k]) > t) // 1<=k<=n
    {
        Output ("Pattern : " lmer);
        Output ("Number of Occurrences: " B[i,j]);
    }
}
}

```

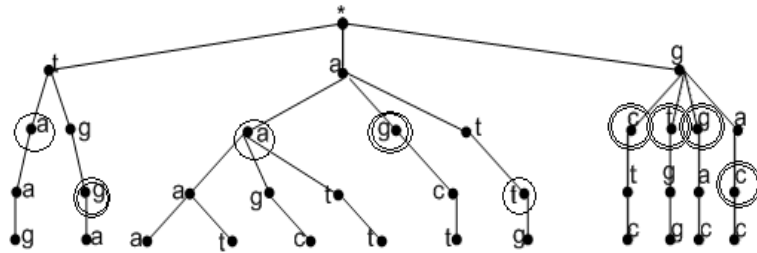
The most important advantage of using the trie lies in the capability to prune

the subtrees of a node. Please notice that for a given l mer, we try to traverse the whole trie establishing the occurrences. Once we reach a node where we can categorically decide that the extension of this node(path) gives us an occurrence, we need not calculate the dynamic programming matrix rows any further. But we need to traverse the whole subtree (linear time BFS, for instance) and update the array A with the indices stored at the leaves of each path. Finally, we jump from this intermediate node to the next node at its level or traverse upwards to determine the next potential candidates.

On the contrary, if we decide that the extension of the current node does not give us an occurrence, we still need not calculate the dynamic programming matrix rows any further. Also, we need not traverse the subtree as we would not need to update a non-occurrence. We should instead progress toward finding the next potential. This whole process is repeated for each l mer.

Below, we present an example:

Consider the sequences - *taagctc*, *gtggacc*, *aaaattg* and the other values as: $l = 3$, $d = 1$ and $t = 2$. The trie constructed based on the sequences is as below As an



example, the trie is used for calculating the occurrences of a l -mer 'taa' and as per the algorithm, dynamic programming for calculating edit distance is pursued with *taa* as the first word and a dynamically changing second word. The nodes with a single oval are the ones which give a d neighbor to *taa* while the ones with the double ovals don't. In either case, we need not travel the subtree of the oval nodes as at that node, we have established/denied the d neighborhood property. The order of the second words is 'ta', 'tgg', 'aa', 'ag', 'att', 'gc', 'gt', 'gg' and 'gac'.

The complexity of the algorithm is measured by

- Construction of the trie - $O(nml)$
- Dynamic Programming for edit distance - each pair consumes $O(l^2)$ and there exists $O((nm)^2)$ pairs to be matched against in the worst case. Due to the pruning as explained above, most of the trie entries are not matched, thereby reducing the number of second words to $O((nm)^\delta)$ where δ is given by the amount of pruning and the nature of the input provided. It is noteworthy to

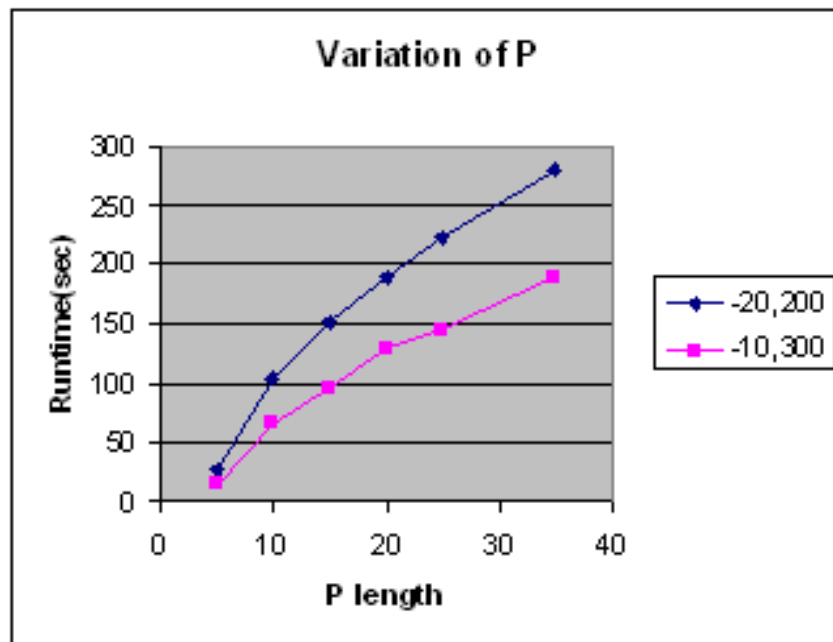
realize that the cost of dynamic program for every pair would not be $O(l^2)$ as lot of rows are common between second words depending on the prefix match between them.

So, overall the run time of this algorithm is $O((nm)^{1+\delta}l^2)$.

4 Implementation and Performance Results

We implemented the code for the edit distance based motif search in C++. Primarily, we conducted two experiments to verify the theoretical bounds.

Impact of l (length of motif) The impact of the length of the motif l (given as P in the figure) is quadratic as given in the bound. In the figure below, we vary the value of l from 5 till 35 and map it against the running time. The same experiment is repeated for two sets of n, m values - (10, 300), (20, 200). In both

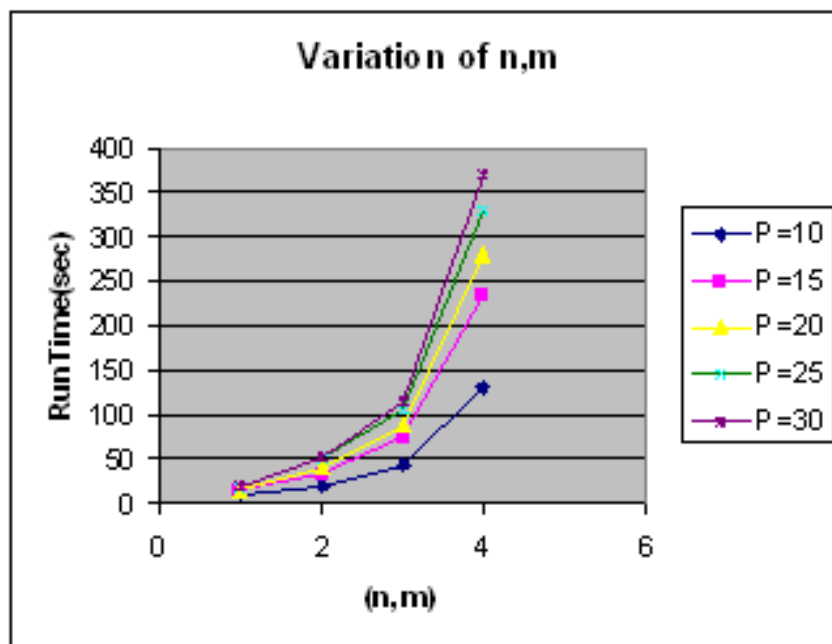


the cases, the curve does not behave exactly like a quadratic curve but rather much better. As we have discussed in the algorithm, most rows of the dynamic programming matrix are re-used and hence not every pair spend an $O(l^2)$ time.

Impact of n, m (DB size) In this experiment, we attempt to see the effect of varying n, m values. The same experiment is run by varying l values over a wide range. The x-axis is abstracted for various sequences as below-

- 1 - (10, 100)
- 2 - (10, 150)
- 3 - (15, 150)
- 4 - (20, 200)

As given in the bound, it is expected to behave better than quadratic but worse than linear. As seen in the above figure, it does follows in an expected way. For



a given value of l , the curve is closer to quadratic .

Challenging Instances We have run the code against so called *challenging instances* of the Planted (l, d) -motif Problem discussed in [3], [7] and below are the results. In all the case, we use 20 sequence each with a size of 600.

- $l = 15, d = 5$: time taken = 2004 sec
- $l = 17, d = 7$: time taken = 4613 sec
- $l = 19, d = 9$: time taken = 5946 sec

5 Conclusion

In this paper, we presented algorithms for the problem of identifying motifs based on edit distances. An implementation of one of our proposed algorithms performance well in practice on random datasets for several values of the input parameters. We expect that an implementation of all the algorithms discussed in this paper and a comparative performance of the same on real biological datasets would help in identifying the most efficient technique to discover motifs based on edit distances, which we anticipate as our future work.

References

1. Adebisi EF, Kaufmann M.: Extracting common motifs under the Levenshtein measure: Theory and experimentation. Proc. Workshop on Algorithms for Bioinformatics (WABI). Springer-Verlag LNCS **2452** (2002) 140–156
2. M. Crochemore and M.-F. Sagot. Motifs in sequences: localization and extraction. In Handbook of Computational Chemistry, Crabbe, Drew, Konopka, eds., Marcel Dekker, Inc., 2001.
3. Davila, J., Balla, S., Rajasekaran, S.: Space and Time Efficient Algorithms For Planted Motif Search. Proc. 6th International Conference on Computational Science (ICCS 2006)/ 2nd International Workshop on Bioinformatics Research and Applications (IWBRA 2006) LNCS **3992** (2006) 822–829
4. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press (1997)
5. Horowitz E, Sahni S, Rajasekaran S.: Computer Algorithms. W.H. Freeman Press, 1998.
6. Myers EW.: A sublinear algorithm for approximate keyword searching. Algorithmica 1994; 12: 345374.
7. Rajasekaran, S., Balla, S., Huang, C-H.: Exact Algorithms for Planted Motif Problems. Journal of Computational Biology **12(8)** (2005) 1117–1128
8. Rajasekaran S, Balla S, Huang C-H, Thapar V, Gryk M, Maciejewski M, Schiller M: High-performance Exact Algorithms for Motif Search. Journal of Clinical Monitoring and Computing (Springer) 19(4-5), 319-328 (2005).
9. Sagot M. F.: Spelling approximate repeated or common motifs using a suffix tree. Springer-Verlag LNCS **1380** (1998) 111–127