PRUNER: Algorithms for Finding Monad Patterns in DNA Sequences

Ravi VijayaSatya and Amar Mukherjee
(rvijaya, amar)@cs.cuf.edu
School of Computer Science, University of Central Florida, Orlando, FL 32816-2362

Abstract

In this paper, we present new algorithms for discovering monad patterns in DNA sequences. Monad patterns are of the form (*l*,*d*)-*k*, where *l* is the length of the pattern, *d* is the maximum number of mismatches allowed, and *k* is the minimum number of times the pattern is repeated in the given sample. The time-complexity of some of the best known algorithms to date is $O(nt^2l^d|\Sigma|^d)$, where *t* is the number of input sequences, and *n* is the length of each input sequence. The first algorithm that we present in this paper takes $O(n^2t^2l^{d/2}|\Sigma|^{d/2})$ and space $O(ntl^{d/2}|\Sigma|^{d/2})$, and the second algorithm takes $O(n^3t^3\ l^{d/2}|\Sigma|^{d/2})$ time using $O(l^{d/2}|\Sigma|^{d/2})$ space. In practice, our algorithms have much better performance provided the *d/l* ratio is small. The second algorithm performs very well even for large values *l* and *d* as long is the *d/l* ratio is small.

Keywords: Pattern discovery, regulatory patterns, k-mismatch patterns

1. Introduction

Discovering regulatory patterns in DNA sequences is a well known problem in computational biology. Due to mutations and other errors, the actual occurrences of these regulatory patterns allow for a certain degree of error. There fore, the actual regulatory pattern (or the consensus pattern) may never appear in a gene upstream region, but d-mismatch occurrences of this pattern might appear. The general approach to this problem is to take a set of *t* DNA sequences each of length *n*, at least *k* of which are guaranteed to contain the desired binding site, and look for patterns of a certain length *l* that occur in at least k out of the *t* sequences with at most *d* mismatches at each occurrence. The values of *l*, *d* and *k* can be determined either from prior knowledge about the binding site, or by trail and error, trying different values of *l* and *d*.

These single contiguous blocks of patterns are called *monad* patterns. In general, many regulatory signals are made up of a group of *monad* patterns occurring within a certain distance form each other [Eskin et. al, 2003, Eskin et. al. 2002, GuhaThakurtha et. al. 2001, van Helden et. al. 2000]. In such a case, the patterns are called *dyad*, *triad multi-ad*, or in general as *composite* patterns. Finding the composite patterns by finding the component monad patterns individually is significantly more difficult, since the composite monad patterns might be too subtle to detect. Eskin & Pevzner [Eskin et. al., 2002] present a simple transformation to convert a multi-ad problem into a slightly larger monad problem. In this paper, we present an algorithm to solve the monad problem. The same transformation as in [Eskin et. al., 2002] can be applied to transform a multi-ad problem into a monad problem that is handled by our algorithm.

Pevzner and Sze [Pevzner et. al, 2000] have put forward a challenge problem: to find the signal in a sample of sequences, each 600 nucleotides long, each containing an unknown pattern of length 15 with at most 4 mismatches. They presented the WINNOWER and SP-STAR algorithms that could solve this problem, which was not solvable by many of the earlier techniques. Many other approaches that can solve this problem have been proposed [Sagot 1998, Eskin et. al 2002, Liang 2003]. Time-complexity of the best known algorithms[Sagot, 1998, Eskin et. al. 2002] is $O(nt^2l^d|\Sigma|^d)$.

Most of these algorithms search the *d*-mismatch neighborhood of each *l*-gram in the sample. The size of the *d*-mismatch neighborhood of an *l*-gram in $O(l^d|\Sigma|^d)$. The main

motivation for our algorithms is that in most practical scenarios, it might be possible to limit the search to a small portion of the *d*-mismatch neighborhood. We refer to the set of patterns that mismatch in at most *d* positions with two *l*-grams as the *consistent* patterns of the two *l*-grams. The following observations form the basis for our algorithm:

Observation 1: At each *l*-gram, it is sufficient to search the consistent patterns of the *l*-gram with all other *l*-grams.

Observation 2: The number of other *l*-grams in the sample that are within *h* mismatches from the current *l*-gram reduces rapidly with decreasing *h*. When *h* is greater than 2*d*, this number is zero, as two *l*-grams that mismatch in more than 2*d* positions can not have any patterns that mismatch with both of them in at most *d* positions. This is illustrated in Figure-1 for a random sample of 20 sequences of 600 nucleotides each. The size of the average 2*d*-mismatch neighborhood is 571.395, where as the average size of the *d*-mismatch neighborhood is just 1.23.

Figure 1: Variation of the size of *h*-mismatch neigborhood with *h*

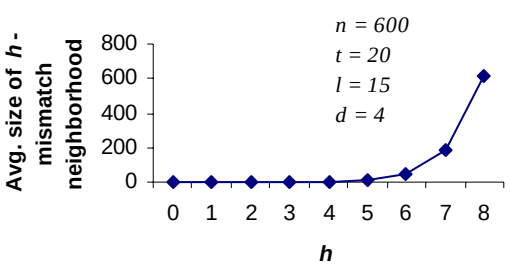
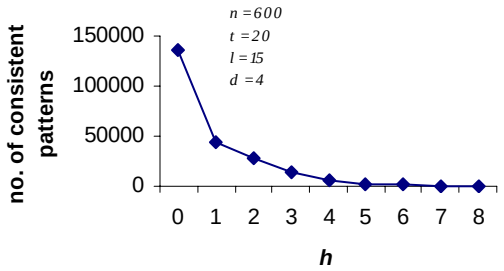


Figure2: Number of consistent patterns between two *h*-mismatch *l*-grams



Observation 3: The number of consistent patterns between two *l*-grams which mismatch in *h* positions decreases rapidly with increasing *h*. Therefore, as is illustrated in Figure-2, the number of consistent patterns between two *l*-grams which mismatch in more than *d* positions is quiet small.

2. Previous approaches to pattern discovery

The pattern discovery problem can be formally stated as follows: Given a set of DNA sequences (also referred to as the sample) $S = \{S_1, S_2, \ldots S_t\}$, and a set of parameters *l*, *d* and *k*, the problem is to find all length-*l* patterns that occur with up to *d* mismatches in at least *k* different sequences in the sample.

One of the earliest techniques to solve this problem, as presented in [Pevzner, 2000] is known as the pattern driven approach. The pattern driven approach searches all of the pattern space – it enumerates each possible pattern and checks if it meets the search criteria. If the pattern length is *l*, there are 4^l possible patterns, assuming a DNA alphabet. Pattern based approaches take each one of these patterns and compare them with all the *l*-grams in the sample. This approach takes exponential time in terms of *l*, and the problem quickly becomes practically unsolvable even for moderate values of *l*.

A faster approach, termed by [Eskin et. al 2002] as the Sample Driven Approach(SDA), searches a reduced search space of only the *l*-grams that occur in the sample and their d-mismatch neighbors. The SDA algorithm trades in space for time: it maintains a table of

size 4^l, each entry in the table corresponding to a pattern. For each *l*-gram in the input sample, the algorithm enumerates all the patterns that make up its *d*-mismatch neighbor hood. For each pattern in the neighborhood, the corresponding entry in the table is incremented. After all the *l*-grams have been processed, the patterns in the table that have a score greater than k are reported. The problem with SDA approach is that the memory requirements are huge, and increase exponentially with *l*. Therefore the SDA approach, like the PDA approach, becomes quickly unmanageable, even for moderate values of *l*.

The WINNOWER algorithm [Pevzner et. al. 2000] and the cWINNOWER algorithm [Liang 2003] are based on graph theory. In these algorithms, a graph is constructed in which each vertex is an *l*-gram in the input sequence. Two *l*-grams are connected by an edge if they mismatch in at most 2*d* positions. Now, the problem is mapped to the problem of finding *k*-cliques in the graph. The problem of finding *k*-cliques in graph, when k >3 is an np-complete problem. Therefore, WINNOWER and cWINNOWER try to apply some heuristics to arrive at a solution. In the first step, all the nodes that have a degree less than k-1 are removed. After that, different techniques are applied to try to remove the spurious edges in the graph that can not be part of a solution. The complexity of WINNOWER and cWINNOWER for the most sensitive versions of the algorithms are given by $O(t^3n^{2.66})$ and $O(t^4n^4)$, respectively. However, it is important to note even though most sensitive versions of these algorithms solve almost all practical problems, they are not guaranteed to solve a given problem.

Some of the other approaches include suffix tree –based approaches [Sagot 1998, Pavesi et. al. 2001]. The SPELLER algorithm presented in [Sagot 1998] first builds a suffix tree for the input sequence. It then examines all possible patterns traversing through the suffix tree. If the paths to *k* different leaves of length *l* mismatch with the pattern in at most *d* positions, then the pattern is reported. Starting with zero characters at the root, the pattern is extended one character at a time. At any time if there are less than *k* different paths in the suffix tree that mismatch in at most *d* positions with the current pattern, the search is stopped and the (alphabetical) next pattern of the same length, or the next pattern of a shorter length is searched. For example, we start with the pattern A. If there are *k* different paths in the suffix tree that mismatch with A in less than *d* positions, then we extend the path to AA. If there are *k* paths still, we extend it again, to AAA. Now, if the number of paths is less than *k*, we move on to the pattern AAC. Therefore, at a depth *i* in the suffix tree, we need to do $min(4^i, n)$ comparisons, since the tree can have *n* leaves at the most. In the worst-case scenario, we will be examining all *d*-mismatch neighbors of each *l*-gram in the sample, hence the complexity of the algorithm is given as $O(nt^2l^d4^d)$.

In the sequence driven approach, each *l*-gram is searched separately. The Mitra-Count algorithm [Eskin & Pevzner, 2002] is based on the idea that if all the *l*-grams are searched concurrently, then only the information about those *l*-grams that meet the current search criteria need to be stored. This will reduce the memory requirements drastically. The MITRA algorithm searches the pattern search space in a depth first manner, abandoning the search whenever the search criterion is no longer met. For this it uses the mismatch tree data structure. The path from the root to a node at depth *m* in the mismatch tree represents a prefix of the pattern of length *m*. The list of *l*-grams from the sample whose m-length prefixes mismatch in at most *d* positions with the path label of the current node are stored at the node. The tree is built in a depth-first fashion.

Whenever the size of the list of *l*-grams at a node falls below *k*, the node is discarded, and the sub tree of the node is never searched. Whenever the search reaches a depth *l*, the pattern corresponding to the path label is reported. The algorithm is memory efficient, since only the nodes that lie in the current path need to be stored at any time.

An improved algorithm, Mitra-Graph, also presented in [Eskin & Pevzner, 2002] applies WINNOWER-like pair wise similarity information in order to maintain a graph at each node of the mismatch tree. Two *l*-grams L_1,L_2 that mismatch in d_1,d_2 positions with node label, and if their suffixes beyond the current depth mismatch in *q* positions, the two *l*-grams are connected by an edge if $d_1+d_2+q \leq 2d$. The nodes can be discarded if there is no possibility for a *k*-clique in the graph. Even though there is an extra overhead of maintaining the graph and extending the graph at each node, much smaller pattern sub-space needs to be searched in Mitra-Graph. Therefore, Mitra graph performs even better than Mitra-count. A detailed analysis of the time complexity for the Mitra-graph algorithm is not provided, but authors state that the theoretical complexity is the same as that of the SPELLER algorithm.

3. The PRUNER Algorithm

3.1 Our Contributions

Our approach is based on the WINNOWER algorithm [Pevzner & Sze, 2000, Liang, 2003]. As in WINNOWER, we build a graph based on pair-wise similarity information, and prune the graph eliminating vertices that can not be part of a solution. However, after this point, we employ a different approach. The algorithms try to successively remove edges from the graph, after checking all the patterns that mismatch in at most *d* positions from both the *l*-grams that are connected by the edge. We categorize the edges into two groups. Group1 consists of edges that connect *l*-grams that differ in *more* than *d* positions, and Group2 consists of edges that connect *l*-grams that differ in *less than or equal to d* positions. In the following sections, we will show that there will be relatively few patterns that mismatch in at most *d* positions from both the *l*-grams that are connected by a Group1-edge. Precisely, we will show that there will be at most $O(l^{d/2}|\Sigma|^{d/2})$ such patterns for every Group-1 edge. We present a technique which enumerates all the patterns corresponding to each Group1-edge, checks each one of them to see if they satisfy the search criteria, and removes the Group1-edge. We show that if at any vertex, after all the Group-1 edges are removed, if the degree of the vertex is less than *k*-1, then the vertex can be safely removed from the graph, without affecting any patterns that are not yet examined and reported. After all Group1-edges are removed, this leaves us with a graph in which each vertex has a degree of at least *k*-1, and all the edges that are incident on the vertex are Group2-edges. *Therefore, if the graph has any vertices left, there will be at least k vertices left.* Since the degree of each vertex is at least *k*-1 and each edge is a Group-2 edge, the *l*-gram corresponding to each vertex has at least *k*-1 other *l*-grams that mismatch with it in at most *d* positions. Therefore the *l*-gram corresponding to each vertex in the graph is itself a solution, and can be reported. Beyond this, there might be other patterns in the graph that meet the search criteria, but in a general case, we assume that there are fewer than *k* distinct monad patterns in the given sample. In the almost impractical scenario that there are more than *k* distinct monad patterns, the algorithms we present report at least *k* of them. Unlike WINNOWER and

cWINNOWER[Liang, 2003], our algorithm is guaranteed to find a solution in $O(n^2t^2l^{d/2}|\Sigma|^{d/2})$ time using $O(ntl^{d/2}|\Sigma|^{d/2})$ space.

3.2 Problem statement

In the discussion that follows, for convenience in illustration, we treat the input sample as a single sequence of size *n*. The time and space complexities are not affected by this simplification. In section 3.5, we explain the enhancements to handle *t* different sequences, instead of a single sequence. Therefore, the problem can be stated as follows: given a string *S* of length *n* over the alphabet $\Sigma = \{A,C,G,T\}$, the problem is to find a pattern *P* of length *l* that occurs at least *k* times in *S* with at most *d* mismatches in each occurrence.

3.3 Terms and Definitions

We denote a length-*l* substring(an *l*-gram) of *S* starting at position *i* in *S* by L_i. A score $h = D(L_i,L_j)$ indicates the number of positions in which the two *l*-grams L_i, L_j mismatch. We denote the set of patterns that mismatch with both L_i and L_j in at most d positions by $\rho(L_i,L_j)$. We refer to the set $\rho(L_i,L_j)$ also as the set of patterns that are *consistent* with L_i and L_j. We now describe, briefly, how to compute the size of the set $\rho(L_i,L_j)$. Let *P* be any pattern such that $P \in \rho(L_i,L_j)$. Now, it is important to note that $\rho(L_i,L_j) = \{\phi\}$ if $h > 2d$, as both $D(L_i,P)$ and $D(L_j,P)$ have to be less than or equal to *d*. We have to enumerate all the different possibilities for *P*. Also, let us divide each *l*-gram into two regions: M-region, consisting of positions in which L_i and L_j match with each other, and the H-region, consisting positions in which L_i and L_j mismatch with each other, as shown in Figure3(a). Both the regions are shown to be contiguous for simplicity in illustration. In reality, these regions need not be contiguous. Now, let us assume patterns L_i and L_j mismatch with *P* in d_c positions within the M-region. Additionally, let L_i mismatch with *P* in h_1 positions, and let L_j mismatch with *P* in h_2 positions, as shown in Figure 3(b). Again, none of these regions needs to be contiguous.

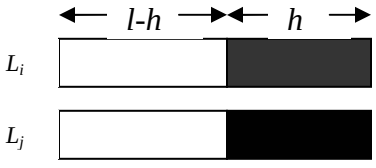


Figure 3(a) The matching and mismatching regions of L_i, L_j

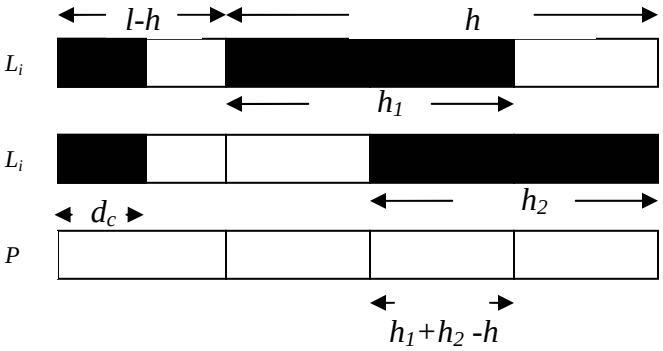


Figure 3(b) Different regions of the Pattern *P*. The shaded regions of L_i, L_j indicate the regions in which L_i, L_j mismatch with P.

Now, d_c mismatch positions can be chosen from *l*-*h* positions in $\binom{l-h}{d_c}$ ways. At each one of these positions, we have $|\Sigma| - 1 = 3$ symbols to choose from. Similarly, h_1 positions

in which L_i can mismatch with P can be selected from h positions in $\binom{h}{h_c}$ ways. The remaining h-h_1 positions in L_i have to match with P, and hence they mismatch with P in L_j (since we know that L_i mismatches with L_j in these positions). The remaining (h_1+h_2-h) positions in which L_j mismatches with P can be selected out of h_1 positions in $\binom{h_1}{h_1+h_2-h}$ ways. We have $|\sum|$-2 = 2 options at each one of these (h_1+h_2-h) positions, since P mismatches with both L_i and L_j. Therefore, the total number of patterns in $\rho(L_i,L_j)$ is given by:

$$|\rho(L_i,L_j)| = \sum_{d_c=0}^{\left\lfloor \frac{2d-h}{2} \right\rfloor} \left[\binom{l-h}{d_c} 3^{d_c} \sum_{h_1=h-d+d_c}^{d-d_c} \sum_{h_2=h-h_1}^{d-d_c} \binom{h}{h_1} \binom{h_1}{h_1+h_2-h} 2^{h_1+h_2-h} \right], \text{ if } h \le 2d \quad \text{.....(1)}$$

$$= 0 \qquad \text{otherwise}$$

In the above expression, $|\rho(L_i,L_j)|$ increases when h decreases. When $d < h \le 2d$, the maximum value of $|\rho(L_i,L_j)|$ occurs when $h = d+1$. When $h = d+1$, the maximum value that d_c can take is given by $d_c = (d-1)/2$ which is equal to $d/2$ when d is odd, and $(d/2)-1$ when d is even. Now, $\binom{l-h}{d_c} 3^{d_c}$ is in $O(l^{d_c} 3^{d_c})$. Therefore, on the whole, $|\rho(L_i,L_j)|$ is in $O(l^{d/2}4^{d/2})$.

3.4 The PRUNER-I and PRUNER-II algorithms

In both the algorithms, we construct a graph $G(L,E)$ where each vertex is an l-gram in the input sample, and there is an edge $e(L_i,L_j, D(L_i,L_j))$ connecting two l-grams L_i and L_j if $D(L_i,L_j)$ is less than or equal to $2d$. The procedure for building the graph is given in Figure-4. We then successively remove vertices representing l-grams from the graph $G(L,E)$ that have a degree less than k-1, and remove the edges that are incident on these vertices, following the algorithm given in Figure-5. Until this point, our algorithms are no different from WINNOWER. However, they differ from WINNOWER in the following steps.

Both the PRUNER-I and the PRUNER-II algorithms process each vertex successively. The PRUNER-I algorithm enumerates the consistent patterns for every group1-edge (i.e., edges between l-grams which mismatch in more than $2d$ positions). It then computes how many times each pattern repeats. It does this by adding all the consistent patterns for each edge to a list, sorting and scanning the list Each time a pattern appears, it means that the pattern is within d mismatches from another l-gram. Hence, if a pattern repeats k-1 times, it means that the pattern is within d mismatches from k-1 other l-grams. However, since we have not yet processed the Group2-edges(i.e., edges connecting l-grams that mismatch in d or fewer positions), we can not yet discard the patterns that repeat less than k-1 times. We do not want to evaluate all the consistent patterns for the Group2-edges, as there are too many ($O(l^d 4^d)$) such patterns. Therefore, we will have to take each pattern in the list, and compare it with each l-gram that is connected to the current vertex through a

Group2-edge. Only then will we know how many times each one of those patterns has repeated. An efficient way of doing all this is presented below.

```
Algorithm BuildGraph()
Inputs: S, l, d, n
Output: G(L,E)
1. L ← {ϕ}, E ←{ϕ}          /*L is the set of l-grams or vertices, E is the set of edges*/
2.  for i ← 0 to n-l+1 do
3.          L_i ← S[i,i+l]          /* L_i is substring of S of length l starting positions i*/
4.          L ← L ∪ L_i
5. end for
6. for i ← 0 to n-l do
7.          for j ← i+1 to n-l+1 do
8.                  if D(L_i,L_j) ≤ 2d
9.                          E ← E ∪ (L_i,L_j, D(L_i,L_j))
10.                 end if
11.         end for
12. end for
```

Figure-4. **Procedure for building the graph G(L,E)**

```
Algorithm PruneGraph()
Inputs: G(L,E), l, d, k
1.  for i ← 0 to n-l+1 do
2.          if  Degree(L_i) < k-1
3.                  RemoveLGram(G(L,E), i, l, d, k)
4.          end if
5. end for
```

Figure-5. **Procedure for pruning the graph G(L,E)**

At each node L_i, we enumerate the consistent patterns $\rho(L_i,L_j)$ for all the Group1-edges, i.e., edges $(L_i,L_j, D(L_i,L_j))$, such that $d < D(L_i,L_j) \leq 2d$. We add these patterns to a list $\eta(i)$, and remove the edge $(L_i,L_j, D(L_i,L_j))$. Lemma 3.1 states that we can safely remove the edge $(L_i,L_j, D(L_i,L_j))$ after enumerating and adding $\rho(L_i,L_j)$ to $\eta(i)$.

Lemma 3.1: After a vertex L_i in $(L_i,L_j, D(L_i,L_j))$ is processed, there can be no new patterns in $\rho(L_i,L_j)$ that were not reported while processing L_i, but will be reported while processing the vertex L_j.

Proof: Let us assume that there is a pattern $P \in \rho(L_i,L_j)$ that was not reported while processing node L_i, but will be reported while processing node L_j. This means that there are a set of l-grams $\psi(P)$ other than L_i, such that for each $L_q \in \psi(P)$, there is an edge $(L_q, L_j, D(L_q,L_j))$ connecting L_q and L_j, and $D(L_q,P) \leq d$. additionally, since P will be reported while processing L_j, $|\psi(P)| \geq k$-2. Now, since for each $L_q \in \psi(P)$, $D(L_q,P) \leq d$ and $D(L_i,P) \leq$ d (as $P \in \rho(L_i,L_j)$ by definition), it implies that $D(L_i,L_q) \leq 2d$. Therefore, for each $L_q \in \psi(P)$ there is an edge $(L_i,L_q, D(L_i,L_q))$ connecting L_i and L_q. Since $|\psi(P)| \geq k$-2, and $P \in \rho(L_i,L_j)$, there are at least k-1 edges incident in L_i which contain P as one of their consistent patterns. Therefore, pattern P must have been reported while processing node L_i. Hence there can be no pattern $P \in \rho(L_i,L_j)$ that is not reported while processing L_i that can be reported while processing L_j. □

Algorithm *RemoveLGram*()
Inputs: $G(L,E)$, L_i
1. *DeleteList* ← {ϕ}
2. **for every** *j* such that (L_i,L_j, $D(L_i,L_j)$) $\in E$ **do**
3. $E \leftarrow E - (L_i,L_j, D(L_i,L_j))$
4. **if** *Degree*(L_j) = *k*-2 /* *The degree of L_j just fell below k-1*/
5. *DeleteList* ← *DeleteList* ∪ L_j
6. **end if**
7. **end for**
8. **for every** $L_j \in$ *DeleteList*
9. *RemoveLGram*($G(L,E)$,L_j)
10. **end for**

Figure-6. **Procedure for deleting an *l*-gram from the graph**

Now, we need to find out how many times each pattern is repeated in $\eta(i)$. An easy way of doing this will be to sort $\eta(i)$, and scan $\eta(i)$. As each pattern in $\eta(i)$ is a length-l string of a fixed alphabet, $\eta(i)$ can be sorted in linear time using radix sort. Let a pattern P repeat m times in $\eta(i)$. Let R be the degree of node L_i after processing and removing all Group1-edges. As explained in section1, R is expected to be very small. We do the following.

- If $m <$ k-1- R, we discard P. The number of times P repeats can increase by at most R, by comparing P with each one of the Group2-edges. If $m <$ k-1-R, there is no way that P can repeat k-1 times. So we can discard P.
- If $m \geq$ k-1, report P, since it is clear that P has already occurred at least k-1 times.
- If k-1- $R \leq m < k$-1, we compare P with all l-grams that are still connected to L_i. For each such l-gram that mismatches P in at most d locations, we increment the repeat count of P. If the repeat count reaches k-*1*, we report P. Other wise, we discard P.

Before we leave L_i and proceed to process the next vertex, we can do one more thing – we can remove the vertex L_i from the graph if $R<k$-1, without ever enumerating the consistent patterns for these edges. Lemma 3.2 proves this.

Lemma 3.2: If the residual degree R of vertex L_i is less than k-1 after processing and removing all Group1-edges of L_i, there can be no new patterns that will be reported by processing the Group2-edges.

Proof: Let us assume that there is a pattern P that was not reported while processing the Group1-edges, but will be reported while processing the Group2-edges. Since we will be reporting P, and since $R<k$-1, there should have been at least one Group1-edge (L_i, L_q, $D(L_i, L_q)$ such that $P \in \rho(L_i,L_q)$. Therefore, P was checked and reported while processing vertex L_q. Hence there can be no new patterns that will be reported by processing the Group2-edges. □

```
Algorithm ProcessLGram()
Inputs: G(L,E), i, l, d, k
Output: Reports patterns in the d-mismatch neighborhood of L_i that satisfy the search criteria
1.  PatternList ← {ϕ}
2.  for every j such that (L_i,L_j, D(L_i,L_j)) ∈ E do
3.          if D(L_i,L_j) > d
4.                  PatternList ← PatternList ∪ ρ(L_i,L_j)
5.          end if
6.          E ← E - (L_i,L_j, D(L_i,L_j))
7.   end for
8.   RadixSort(PatternList)
9.   Cnt ← 0                    /* Cnt is the number of times the current pattern has repeated */
10.  for j ← 1 to |PatternList|-1
11.         if  PatternList_j = PatternList_j-1          /* if the current pattern is the same as the
12.                 Cnt ← Cnt+1                                  previous patterns*/
13.         else if Cnt ≥ k-1-Degree(L_j)   /*if there are enough edge to push Cnt beyond k-1 */
14.                 for every r S.T. (L_i,L_r, D(L_i,L_r)) ∈ E do
15.                         if D(PatternList_j, L_r) ≤ d      /*compare the current pattern with each such
16.                                 Cnt ← Cnt +1                      l-gram */
17.                         end if
18.                 end for
19.                 if Cnt > k -1
20.                         Report(PatternList_j)          /* PatternList_j is an (l,d)-k pattern*/
21.                 end if
22.                 Cnt ← 0
23.         end if
24. end for
```

Figure-7. **Procedure for deleting an *l*-gram from the graph**

We are now left with a graph in which the score of each edge is at most *d*, and degree of each remaining vertex is at least k-1. In practice, we do not expect any vertices to remain at this stage, as our assumption is that there are not too many patterns that meet the search criteria. All the *l*-grams that do remain until this stage are valid solutions, since they mismatch in at most *d* positions with at least k-1 other *l*-grams. Hence, we report all the remaining *l*-grams.

```
Algorithm SearchForPatterns()
Inputs: S, l, d, k, n
1.  BuildGraph(S, l, d, n)
2.  PruneGraph(G(L,E), l, d, k, n)
3.  for i ← 0 to n-l+1
4.          ProcessLGram(G(L,E), i, l, d, k, n)
5.          if(Degree(L_i) < k-1)
6.                  RemoveLGram(G(L,E),i)
7.  end for
8.  PruneGraph(G(L,E), l, d, k, n)
          /* Prune once again, to remove l-grams with degree < k-1
9.  for i ← 0 to n-l+1                  /*check if any l-grams are still remaining */
10.         if Degree( L_i) > k-1
11.                 Report(L_i)          /* report all remaining l-grams */
12.         end if
13. end for
```

Figure-8. **The PRUNER-I algorithm**

The PRUNER-II algorithm is very similar to the PRUNER-I algorithm in concept. However, the PRUNER-II algorithm attempts to eliminate the potentially huge memory

requirements of the PRUNER-I algorithm. While processing each node L_i, the PRUNER-I algorithm maintains a list $\eta(i)$ that contains all the patterns that are consistent with each one of the Group1-edges. When the number of such edges is huge, the amount of memory required for $\eta(i)$ may be too big. Especially, this might be the case when d is large and the d/l ratio is large, in which case the graph $G(L,E)$ will be highly connected.

At each vertex L_i, the PRUNER-II algorithm processes edges one by one. For each edge (L_i, L_j, $D(L_i,L_j)$), it enumerates the set of consistent patterns $\rho(L_i,L_j)$. For each consistent pattern $P \in \rho(L_i,L_j)$, if we compare P with all the l-grams that are directly connected with vertex L_i, we can determine if P mismatches in at most d positions with at least k-1 of them. However, a deeper analysis reveals that it not necessary to compare P with all the l-grams that share an edge with L_i. For any l-gram L_q, if $D(L_q,P) \leq d$, then $D(L_q,L_j)$ will be less than or equal to $2d$. This means that the l-grams L_q and L_j will also be connected. Therefore, we only need to compare P with all vertices L_q such that the edge (L_q, L_j, $D(L_q,L_j)$) $\in E$. If at least k-2 of them mismatch with P in fewer than d positions, it reports P. Otherwise, P is discarded. As in the PRUNER-I algorithm, it removes the edge (L_i,L_j, $D(L_i,L_j)$) after checking all the patterns in $\rho(L_i,L_j)$.

3.5 Extending the algorithm to handle multiple sequences:

When the input sample is made of t sequences of length n each, and the problem is find an (l,d) motif that occurs in at least k of them, the graph G(L,E) will be a t-partite graph. At each vertex in the graph, we need to maintain and update another variable, which we call t-degree. The variable t-degree stores the number of distinct sequences in t that the current vertex is connected to. In the algorithms that we discussed above, whenever we are referring to the degree of a vertex, we will be using t-degree instead of the actual degree of the vertex. Whenever we are checking for a pattern P, it is no longer sufficient to check if the pattern is within d mismatches from k-1 other l-grams. We need to make sure that the l-grams are derived from k-1 distinct sequence in the sample. The implementation typically involves maintaining a bit-vector of length t for the pattern that is being considered. Whenever the pattern is within d-mismatches from an l-gram, the bit corresponding to the sequence from which the l-gram is derived is set to 1. P satisfies the search criteria if at least k-1 (or whatever is necessary at that point in the algorithm) bits are set to 1.

3.6 Complexity analysis:

Building the graph involves calculating the mismatch count for each l-gram pair (L_i, L_j) such that L_i and L_j are derived from different input sequences. There are (n-l+1) l-grams for each input sequence, and $n(t$-1) other l-grams for each l-gram in the input sequence. Therefore, building the graph takes $O(n^2t^2)$. Pruning the graph involves removing all the edges incident on each vertex whose degree is less than k-1. In the worst case, we might have to delete all the nodes, so the maximum number of edges that need to be removed is ((k-1)*nt -1), which is O(ntk). This time is common for both PRUNER-I and PRUNER-II.

In the PRUNER-I algorithm, each l-gram can have up to $n(t$-1) $2d$-mismatch neighbors. Therefore, at each l-gram, we might have to enumerate the consistent patterns with $n(t$-1) other l-grams. The maximum number of these consistent patterns as discussed in section

3.3, is $O(l^{d/2}4^{d/2})$. Hence the worst-case time complexity at each node is given by $O(ntl^{d/2}4^{d/2})$. We need to store all these patterns in a list, so we need $O(ntl^{d/2}4^{d/2})$ space. In the worst case, we will have to process $(t\text{-}k\text{+}1)n$ l-grams, since no new patterns can be discovered after removing all l-grams from $(k\text{-}1)$ l-grams. Therefore, the overall complexity is given by $O(n^2t(t\text{-}k\text{+}1)l^{d/2}4^{d/2})$. If k is small w.r.t. t, this will be $O(n^2t^2l^{d/2}4^{d/2})$. When $k = t$, the complexity of the PRUNER-I algorithm is $O(n^2tl^{d/2}4^{d/2})$.

In case of the PRUNER-II algorithm, each edge is processed separately. All the patterns consistent with each edge (L_i,L_j) have to be compared with all the l-grams that are connected to both L_i and L_j. In the worst case, there can be $n(t\text{-}2)$ vertices that are connected to both L_i and L_j. The total number of the edges could be $n^2t(t\text{-}1)$ in the worst case. The edge can have $O(l^{d/2}4^{d/2})$ patterns that are consistent with it, so the total time taken will be $O(n^3t^3\ l^{d/2}4^{d/2})$. Each pattern could be compared separately; therefore the space needed is approximately the same as that necessary for the graph.

4. Results

The algorithms were tested on generated samples containing 20 sequences of 600 nucleotides each. The sequences are implanted with randomly mutated patterns at randomly chosen positions. In order to make the patterns as realistic as possible, each occurrence of the pattern is allowed to have up to d mismatches, as against allowing exactly d mismatches in each pattern. The tests were carried out on a Pentium-4 3.2 GHz PC with 2GB of memory, running Redhat Linux 9.0. The time/memory results are illustrated in Table 4.1.

The PRUNER-I algorithm ran out of memory for the (18,6) and the (28,7) cases. The implanted pattern was detected in all the test cases were allowed to run to completion.

Table 4.1: Performance of the algorithms

Test Case(l,d)	d/l	PRUNER-I		PRUNER-II	
		Time	Memory(MB)	Time	Memory(MB)
13,3	0.23	17s	43	14s	43
13,4	0.31	12m26s	166	29m58s	278
14,4	0.28	5m35s	198	7m09s	178
15,4	0.27	2m28s	122	1m34s	91
16,4	0.25	56s	51	56s	43
16,5	0.31	69m	540	4h44m59s	247
17,5	0.29	29m54s	315	36m58s	161
18,5	0.28	13m19s	174	13m19s	92
18,6	0.33	*	Out of memory	48hours(expected)	457
24,6	0.25	1m12s	720	16s	11
28,7	0.25	*	Out of memory	36s	640

6. Conclusion

We have presented two new algorithms for finding the monad patterns. Both the algorithms perform extremely well on the challenge problem of (15,4) on 20 input sequences of 600 nucleotides. As d increases in comparison to l, i.e., when the d/l ratio increases, The PRUNER-I algorithm takes a longer time and a larger memory. The PRUNER-I algorithm runs out of memory for large values of l and d. The PRUNER-II algorithm, on the other hand, reacts very sharply to the d/l ratio. In this case, the memory requirements are not a bottle neck as much as the speed. However, as long as the d/l is

around 0.25, the PRUNER-II algorithm performs very well, independent of the actual values of *l* and *d*.

Binaries for Mitra-Count and Mitra-Graph were requested, but were not available for comparison. We intend to provide comparisons with our own implementations of Mitra-Count and Mitra-Graph in the final version of this paper. We will also be presenting results on actual gene upstream regions in the final version of the paper.

Unlike Winnower and cWinnower, the algorithms we presented here are not sensitive to *k*. Our algorithms will be able to detect patterns even for very small values of *k*. The only concern when dealing with very small values of *k* is that there could be random signals in the input sample that meet the search criteria.

An interesting observation from the test cases is that graph itself starts consuming more and more space as the ratio *d*/*l* gets bigger. This is because there are more and more edges in the graph, as there are a larger number of *l*-gram pairs that mismatch in less than 2*d* positions. In the future, we plan to investigate compact representations for the graph. Another approach may involve using a two-pass algorithm. WINNOWER or cWINNOWER can be used initially in order to remove some spurious edges. Our algorithms can be applied in the second pass. As the graph has much fewer edges now, PRUNER-I or PRUNER-II may have very good performance. For the first pass, we can use a low sensitivity version of WINNOWER or cWINNOWER in order to maximize the speed.

References

Buhler J. and Tompa B. (2001) Finding Motifs Using Random Projections. In Proceedings of the Fifth Annual International Conference on Computational Molecular Biology (RECOMB01), 2001 pp.69-76

Eskin E., Keich U., Mikhail S. G., Pavel and Pevzner. P. A. (2003) Genome-Wide Analysis of Bacterial Promoter Regions. *In Proceedings of the Pacific Symposium on Biocomputing (PSB-2003)*. Kaua'i, Hawaii: January 3-7, 2003

Eskin E., Pevzner P. A. (2002) Finding Composite Regulatory Patterns in DNA Sequences.' *In Proceedings of the Tenth International Conference on Intelligent Systems for Molecular Biology (ISMB-2002)*. Edmonton, Canada: August 3-7, 2002

Guha Thakurtha D. and Stormo, G. D. (2001). Identifying target sites for cooperatively binding factors. Bioinformatics, 15, 563-577

Hertz, G. and Stormo, G. (1999). Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. Bioinformatics, 10, 1205-1214.

Liang S. (2003) cWINNOWER Algorithm for finding fuzzy DNA Motifs. In Proceedings of the 2003 IEEE Computer Society Bioinformatics Conference Bioinformatics Conference Aug 11-14 2003 pp. 613-614.

Marsan L. and Sagot M. (2000). Algorithms for extracting structured motifs using suffix tree with applications to promoter and regulatory site consensus identification. Journal of Computational Biology, 7, 345-360.

Pavesi, G., Mauri, G. & Pesole, G. (2001). An algorithm for finding signals of unknown length in DNA sequences. In proceedings of the Ninth international Conference on Intelligent Systems for Molecular Biology.

Pevzner, P. A. & Sze, S. (2000). Combinatorial approaches to finding subtle motifs in DNA sequences. In Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology. pp. 269-278.

Price A., Ramabhadran S. and Pevzner A. Finding Subtle Motifs by Branching from Sample Strings. Bioinformatics. 19, 149-155

Sagot, M. (1998). Spelling approximate or repeated motifs using a suffix tree. Lecture Notes in Computer Science, 1380, 111-127.