Toward Optimal Motif Enumeration

Patricia A. Evans[1] and Andrew D. Smith[1,2]

[1] University of New Brunswick, P.O. Box 4400, Fredericton N.B., E3B 5A3, Canada
[2] Ontario Cancer Institute, University Health Network, Suite 703
620 University Avenue, Toronto, Ontario, Canada, M5G 2M9 pevans@unb.ca,
asmith@uhnres.utoronto.ca

Abstract. We present algorithms that reduce the time and space needed to solve problems of finding all motifs common to a set of sequences. In particular, we give algorithms that (1) require time and space linear in the size of the input, (2) succinctly encode the output so that the time and space requirements depend on the number of motifs, not directly on motif length, and (3) efficiently parallelize the enumeration.

1 Introduction

The problem of discovering short strings occurring approximately in each member of a set of longer strings is important in computational biology. We refer to the short strings as *motifs*, and the longer strings as *sequences*[1]. By "occurring approximately", we mean that motifs must match a segment of each sequence with at most some specified number of mismatches.

The motif discovery problem abstracts many problems encountered in the analysis of biology sequence data, where the sequences are molecular sequences and motifs represent short biologically important patterns. A popular technique for finding motifs is to enumeratively test all strings over the sequence alphabet having length equal to the desired motif length. An advantage of the enumerative approach is that it does just that; enumerative algorithms produces all possible motifs for a set of sequences. This allows the discovered motifs, which posses a certain combinatorial property, to be evaluated according to other criteria. In this capacity, the enumerative algorithms can provide input to other algorithms that filter motifs based on other properties. Formally, we define the motif enumeration problem as follows:

Problem 1. The input is a set $\mathcal{F} = \{S_1, \ldots, S_m\}$ of strings over an alphabet Σ such that $|S_i| \leq n$, $1 \leq i \leq m$, and integers l and d such that $0 \leq d < l \leq n$. The solution is a set of motifs $\mathcal{M}_\mathcal{F} \subseteq \Sigma^l$ such that for each motif $\mathcal{C} \in \mathcal{M}_\mathcal{F}$ and each $S_i \in \mathcal{F}$, there exists a length l substring of S_i that is Hamming distance $\leq d$ from $\mathcal{C}$.

Hamming distance is defined, for equal length strings, as the number of mismatches between the strings. Note that it is sufficient, and often desirable, to

[1] This terminology may conflict with terminology used elsewhere.

produce a small encoding of $\mathcal{M}_{\mathcal{F}}$, from which the motifs can be efficiently extracted.

There are two major computational challenges to enumerating motifs. The first challenge is that the problem of deciding if $\mathcal{M}_{\mathcal{F}} = \emptyset$ is NP-hard; practical solutions are thus non-trivial. The second is that we are concerned with more than simply the decision, so we may have to produce output of exponential size.

The paper is organized as follows. Section 2 describes previous work on the problem. In Section 3 we describe the first algorithm, Census, that improves on an algorithm of Sagot [9] and establishes an upper bound on the time and space complexity that is linear in both the string length and the number of strings. We also discuss parallelizations of the algorithm. In Section 4, we describe the MotifIntersection algorithm, further reducing the upper bound on the time complexity. The algorithm is based on a data structure that succinctly encodes sets of motifs, and allows efficient set operations. In Section 5, we show the problem admits an FPP algorithm, placing the corresponding decision version in the subclass of fixed parameter tractable problems that are highly parallelizable [3].

2 Background

Many algorithms have used enumerative strategies to find motifs in sets of sequences (*e.g.* [2] [7] [10] [11]). These algorithms each approach the problem differently; most attempt to eliminate as much of the search space as possible. Each, however, attempts to enumerate all strings of length l over the sequence alphabet. This most naive form of search introduces a factor of $\Omega(|\Sigma|^l)$ into the time complexity. The benefit of this type of enumeration is that it requires space bounded by a linear function of the size of the input.

New ground was broken when Sagot [9] introduced a different approach that enumerates only those strings that are potential motifs, letting information from the sequences guide the enumeration. This more intelligent search remains within the (l, d)-neighborhood of each sequence. In the following definition, d_H refers to the Hamming distance.

Definition 1. (neighborhood) *For a string $S \in \Sigma^n$, with $n \geq l$, the (l, d)-neighborhood of S is the set*

$$\{s' : s' \in \Sigma^l \wedge d_H(s, s') \leq d \text{ for some substring } s \text{ of } S \text{ with } |s| = l\}.$$

For any string S, we use $N_{l,d}(S)$ to denote the (l, d)-neighborhood of S. For a family $\mathcal{F}$ of strings, the (l, d)-neighborhood of $\mathcal{F}$ is the set

$$\{s' : s' \in \Sigma^l \wedge \forall S \in \mathcal{F}, s' \in N_{l,d}(S)\},$$

and is denoted $N_{l,d}(\mathcal{F})$.

We also define the value $N = \sum_{i=0}^{d} \binom{l}{i} (|\Sigma| - 1)^i$, and note that this value appears throughout our analysis. The significance of N is that for a string s with $|s| = l$, $N = |N_{l,d}(s)|$.

The method of Sagot has a time complexity of $O(lm^2nN)$, a space complexity of $O(lm^2n)$, and has proved successful in practice [13]. We note that the algorithm in [9] was actually designed with a "quorum" parameter, so that a motif is only required to be common to some $q \leq m$ of the sequences.

3 Improved Time and Space Complexity

In this section we make an initial improvement to the time and space complexity for the enumeration problem. We eliminate a factor of m from the requirements of the algorithm of Sagot [9]. This brings the time complexity to $O(lmnN)$, and the space complexity to $O(lmn)$. In Section 4 we further reduce the time complexity to $O(mnN)$.

3.1 The Census Algorithm

The algorithm in [9] employed a generalized suffix tree [6], and required that each node indicate the subset of strings having the node's label as a prefix. We eliminate the use of generalized suffix trees, and therefore eliminate the sets stored at each node. Analysis indicates that we have also eliminated the factor of m in the time complexity without increasing the influence of the remaining parameters.

Census begins with the construction, for each $S_i \in \mathcal{F}$, of the lexicographic tree T_i encoding all length l substrings of S_i, which requires $O(lmn)$ time [1]. The potential motifs of desired length l are not searched directly. The search process iteratively searches for each prefix of a given motif in order to take advantage of the fact that prefixes are shared by many potential motifs. This eliminates redundant processing, as will be shown in the complexity analysis.

Let $\mathcal{C}$ be a (length $\leq l$) motif for $\mathcal{F}$. Define the family of sets $F = \{F_1, \ldots, F_m\}$ with respect to $\mathcal{C}$ as $F_i = \{(v, k) : v$ is a node in the tree T_i, and $0 \leq k \leq d\}$, where k counts mismatches between the label of v and $\mathcal{C}$, for each $1 \leq i \leq m$. Think of F_i as the frontier of nodes in T_i whose path labels are of Hamming distance $\leq d$ from $\mathcal{C}$. For any $(v, k) \in F_i$, the path label of node v spells out an occurrence of $\mathcal{C}$ in S_i. For any $1 \leq i \leq m$ and element $(v, k) \in F_i$, the value of k is the number of mismatches between $\mathcal{C}$ and the path label of node v in T_i. Given a character $\alpha \in \Sigma$ and a frontier set F defined with respect to a motif $\mathcal{C}$, the family of sets $F^\alpha = \{F_1^\alpha, \ldots, F_m^\alpha\}$ is the frontier set defined with respect to the length $|\mathcal{C}| + 1$ motif $\mathcal{C}\alpha$.

While searching the space of possible motifs, if any $F_i \in F$ is found to be empty, the search space is pruned. The emptiness condition implies that there exists some member of $\mathcal{F}$ containing no occurrence of the motif presently being searched. Pseudocode for the Census algorithm is provided in Algorithm 1. For the initial call to Census, $\mathcal{C}$ is the empty string and F is the set of roots of the trees T_i, each given an error value of 0.

Algorithm 1: Pseudocode for the CENSUS algorithm.
Input: A set of lexicographic trees $F = \{F_1, \ldots F_m\}$ and a string $\mathcal{C}$.
Output: All motifs for $\mathcal{F}$.

```
CENSUS($\mathcal{C}, F$)
1.      for each character $\alpha \in \Sigma$
2.          for each $F_i \in F$
3.              for each $(v, k) \in F_i$
4.                  if node $v$ has a child $v'$ labeled with $\alpha$
5.                      $F_i^\alpha \leftarrow F_i^\alpha \cup \{(v', k)\}$
6.                  if $k < d$
7.                      for each child $v'$ of $v$ that is not labeled with $\alpha$
8.                          $F_i^\alpha \leftarrow F_i^\alpha \cup \{(v', k+1)\}$
9.          if $\forall F_i^\alpha \in F^\alpha, F_i^\alpha \neq \emptyset$
10.             $\mathcal{C}' \leftarrow \mathcal{C}\alpha$
11.             if $|\mathcal{C}'| = l$ then output($\mathcal{C}'$)
12.                     else make the recursive call CENSUS($\mathcal{C}', F^\alpha$)
```

Theorem 1. *The time complexity of* CENSUS *is* $O(lmnN)$.

Proof. The time complexity of the algorithm is proportional to the number of motifs in the search space, multiplied by the size of the family of frontiers F that must be constructed for each point in the search space. In the worst case, for m strings of length n, there are $O(nN)$ potential motifs for $\mathcal{F}$. The maximum size of the (l, d)-neighborhood of a string S of length n is $(n - l + 1)N$, and this is achieved when the d-neighborhoods of all length l substrings of S are disjoint. Further observe that this upper bound on the number of motifs in the search space gives an upper bound on the search space traversed by the algorithm when all (l, d)-neighborhoods of members of $\mathcal{F}$ completely overlap. Attaining this limit on the search space requires that each $F_i \in F$ have exactly one element. For $1 \leq j \leq l$, let X_j be the space of all motifs $\mathcal{C}$ for $\mathcal{F}$ such that $|\mathcal{C}| = j$. Then under the condition of complete (l, d)-neighborhood overlap for members of $\mathcal{F}$:

$$|F|\textstyle\sum_{j \leq l}|X_j| < mn\sum_{j \leq l}\binom{j}{d}(|\Sigma| - 1)^d = O(lmnN).$$

Consider how the search space is affected should any F_i have more than one element. The (l, d)-neighborhoods of substrings of S_i would no longer be disjoint and the d-neighborhood of S_i would have at least one fewer member. Since each increase in the size of a set $F_i \in F$, with respect to any motif $\mathcal{C}$, decreases the size of the search space by an equal amount, the situation of total (l, d)-neighborhood overlap is the worst case. Hence, the overall running time of the algorithm is $O(lnmN)$. □

The space complexity of CENSUS is $O(lmn)$, exactly the space required for the lexicographic trees, since the frontier sets consist of nodes from the lexicographic trees, and no node exists simultaneously in more than one frontier set. We note that if CENSUS were modified to solve the quorum version as in [9], the time complexity would increase by a factor of $(m - q)$ for a quorum of q; the space complexity would not be altered.

The CENSUS algorithm was implemented in `C` and tested on a 1.4GHz Pentium 3 processor. Simulated data consisted of sequences generated uniformly at random from a 4 character alphabet. For (m, n, l, d) values of $(1000, 1000, 12, 3)$, CENSUS required 95 minutes and 370 megabytes of memory; 4.5 hours and 97 megabytes was required for values of $(100, 2000, 15, 4)$. The algorithm was also tested on a dataset taken from the *E. coli* genome, with values $(2645, 2000, 9, 2)$, and required 37 minutes and 991 megabytes of memory.

3.2 Parallelizations

The nature of the search in the algorithm makes it a prime candidate for distributed search. Practical aspects of such distributed searching are facilitated by the linear space requirements. The CENSUS algorithm can be parallelized to achieve $O(1)$ supersteps within the bulk-synchronous parallel (BSP) model [12]. BSP models parallelism using virtual processors that are mapped during execution to a smaller number of actual processors. An algorithm's computation is broken into *supersteps*, units of processing and communication that represent the necessary synchronization. All virtual processors must complete each specific superstep before any proceed to the next superstep. A parallel algorithm with a large superstep complexity is considered to be *fine grained*; it needs high synchronization and short time intervals. If the number of supersteps is small, the algorithm is *coarse grained* and requires little synchronization between virtual processors, which is desirable for parallel algorithms. The algorithm is modified as follows to partition the search space.

1. Each processor computes the prefixes of motifs for which it will search (these prefixes are distributed uniformly among the processors). Each processor searches for motifs, and when finished, broadcasts the number found.
2. With the information about the number of motifs each processor has found, processor P computes the starting address it will use to write the lexicographic tree into global memory. Then P writes the lexicographic tree encoding its motif set into global memory.

Theorem 2. *For all $1 \leq p \leq N$, the partitioned search space algorithm takes $O((N/p)mnl)$ time and requires $O(mn)$ space per processor, while performing $O(1)$ supersteps.*

Proof. The time complexity of the supersteps follows from the time complexity of CENSUS. This algorithm only needs to be synchronized after each step in the modification description, so the number of supersteps is constant. □

Another way to parallelize the algorithm is to assign each string in $\mathcal{F}$ to a unique processor. Since the data for each input string is indexed in a separate lexicographic tree, this would be feasible for systems without shared memory. This type of parallelization reduces the influence of the number of sequences (m) on the time complexity of Census.

1. Each processor constructs the lexicographic trees corresponding to its assigned sequences.
2. If the present motif prefix has length l, a motif has been found (a situation that can be handled in many ways). Otherwise, each processor makes the appropriate updates to the frontier sets based on the present motif prefix. When the frontiers have been updated, processors communicate whether or not to extend the present motif prefix, or backtrack. This step is repeated until the search is completed.

This is a fine grained parallelization, as the processors need to communicate after examining each extension. As such, we also consider the time for the communication required to direct the search.

Theorem 3. *Using p processors, the problem can be solved in $O(l(mn/p + \log p)N))$ time and $O(n)$ space per node.*

Proof. The factor of N in the time complexity of Census corresponds to the search space that is traversed; this factor remains untouched by the parallelization. Similarly, the factor ln corresponds to the frontiers that must be maintained. Since disjoint sets of m/p frontiers are associated with distinct processors, they are updated independently in parallel, thus eliminating a factor of p from the time complexity. The only additional work to be accounted for is that required to determine when to backtrack. This requires communication, and essentially computing a logical "or" of a value from each processor. Done carefully, this requires $O(\log p)$ time. □

4 Near Optimal Enumeration

In this section we describe how to eliminate a factor of l from the time complexity required by the enumeration. The result is an algorithm that requires $O(mnN)$ time, and suggests an efficient parallelization that will be described in Section 5. The algorithm is based on a new data structure, the neighborhood tree, that concisely encodes the (l, d)-neighborhood for a set of strings.

Definition 2. (neighborhood tree) *For any set $\mathcal{F}$ of strings, each of length $n \geq l$, the (l, d)-neighborhood tree $T_{l,d}(\mathcal{F})$ for $\mathcal{F}$ is a rooted directed tree satisfying four conditions: 1. each edge is labeled with a string; 2. any two edges out of the same node have labels beginning with distinct characters; 3. any internal node has out-degree at least 2; and 4. every string in $z \in N_{l,d}(\mathcal{F})$ maps to some leaf u of T such that the string formed by concatenating in order the labels on the path from the root to u exactly spell out z, and every leaf of T is mapped to some $z \in N_{l,d}(\mathcal{F})$. When $\mathcal{F} = \{S\}$, we write $T_{l,d}(\mathcal{F})$ as $T_{l,d}(S)$ for convenience.*

The (l,d)-neighborhood tree has the important property that, given a query string s of length l, it is capable of answering whether s is contained in the (l,d)-neighborhood of $\mathcal{F}$, and can do so in time proportional to size of the query (*i.e.* $|s|$). Our goal is to build and represent these structures in time and space proportional to the number of leaves in the tree, which is exactly $|N_{l,d}(\mathcal{F})|$. To accomplish this we must avoid explicitly representing the edge labels, as doing so would require $\omega(1)$ space per node. Our method is inspired by the edge compression used in linear time suffix tree algorithms [8], which represent edge labels by indexing substrings of the underlying strings. The strategy does not transfer directly to neighborhood trees since not all substrings of members of $N_{l,d}(\mathcal{F})$ occur exactly as substrings of members of $\mathcal{F}$. The representation we use is based on an observation about $T_{l,d}(s)$ for a string s of length l.

Property 1. Let s be a string with $|s| = l$. For any edge $(u,v) \in T_{l,d}(s)$, if the string labeling edge (u,v) has length $x > 1$, then v is a leaf and the string labeling (u,v) may differ by at most 1 position from the suffix of s having length x. In addition, such a mismatch can only occur at the first position of the label.

Thus, in the restricted case of an (l,d)-neighborhood tree for a string s with $|s| = l$, any edge label may be represented in constant space. It is sufficient to index the beginning and end of a substring in s, and indicate the character occupying the first position of the label (which, by Property 1, is the only position where the character in the label may not match the character in the indexed substring of s).

We describe an algorithm to construct (l,d)-neighborhood trees for strings of length l. The reason we consider this restricted case is that the (l,d)-neighborhood tree for a string S of length $n > l$ may be obtained by taking the union of the (l,d)-neighborhood trees for each length l substring of S. The construction algorithm is based on the following recurrence. The symbol $\circ$ denotes concatenation and when applied to a set operates on each member of the set.

Property 2. Let s and s' be strings with $|s| = l$ and $|s'| = l-1$. If $s = \alpha \circ s'$ for some $\alpha \in \Sigma$, then

$$N_{l,d}(s) = \Big(\alpha \circ N_{l-1,d}(s')\Big) \bigcup \Big(\cup_{\beta \in \Sigma}\, \beta \circ N_{l-1,d-1}(s')\Big). \tag{1}$$

The recursive characterization of a neighborhood suggests a recursive algorithm for constructing a neighborhood tree. The information stored at each node in the tree includes numbers indexing a substring in the underlying string, and a modifier character that might override the first character indexed. We note that for this restricted case, the modifier alone is sufficient since edges with labels of length > 1 are incident on leaves, and their index is completely determined by the depth of the leaf. The reason for using the indexes is that they will be necessary later when constructing (l,d)-neighborhood trees for strings of length $> l$. We also anticipate the extension to neighborhood trees for sets of strings, and therefore assume the identity of each string is encoded along with the pair of indexes. The following algorithm is based on the recursion in Property 2.

Algorithm 2: Pseudocode for the BuildNeighborhood algorithm.
Input: The root of an $(l, 0)$-neighborhood tree for a string s. A distance parameter d.
Output: The root of an (l, d)-neighborhood tree $T_{l,d}(s)$ for s.

```
BuildNeighborhood($v, d$)
1.  if $d > 0$ and $v$ is not a leaf
2.      for each $\beta \in \Sigma \setminus \{\alpha\}$
3.          Create child $u_\beta$ of $v$ with index (depth($v$)+1, depth($v$)+1)
            and modifier character $\beta$.
4.          Create child $x_\beta$ of $u_\beta$ with index (depth($v$) + 2, $|s|$) and
            no modifier character.
5.          BuildNeighborhood($u_\beta, d - 1$)
6.      Let $x_\alpha$ be the original child of $v$. Create node $u_\alpha$ with index
        (depth($v$)+1, depth($v$)+1) and no modifier character. Incre-
        ment the start index of $x_\alpha$, insert $x_\alpha$ below $u_\alpha$, and replace
        $x_\alpha$ with $u_\alpha$ as child of $v$.
7.      BuildNeighborhood($u_\alpha, d$)
8.  return $v$
```

Lemma 1. *For any string s, such that $|s| = l$, the (l, d)-neighborhood tree of s can be built in $O(N)$ time.*

Proof. First, notice that the total number of leaves in the tree is $O(N)$, since they are in 1-to-1 correspondence with members of $N_{l,d}(s)$. Because we use indexes instead of explicitly representing edge labels, the size of each node is constant, as is the time to create each node. Finally, since all nodes have out-degree 2 or 0, the total number of nodes is proportional to the number of leaves in the tree. □

After constructing $T_{l,d}(s_{ij})$ for each length l substring s_{ij} of each $S_i \in \mathcal{F}$, we take the union of these structures to obtain each $T_{l,d}(S_i)$. The intersection of all $T_{l,d}(S_i)$ gives $T_{l,d}(\mathcal{F})$. For two (l, d)-neighborhood trees $T_{l,d}(s)$ and $T_{l,d}(s')$, corresponding to strings s and s', the union $T_{l,d}(s) \cup T_{l,d}(s')$ is defined as the (l, d)-neighborhood tree encoding $N_{l,d}(s) \cup N_{l,d}(s')$. The intersection of neighborhood trees is defined similarly with respect to the intersection of the neighborhoods. The union and intersection operations are accomplished by recursively applying the operations to subtrees. This requires being able to determine the extent to which two nodes are identical, which requires determining the length of the longest identical prefix between two edge labels. As an example, let $s = abcd$ and $s' = abba$, and consider $T_{4,0}(s)$ and $T_{4,0}(s')$ which each have two nodes. The union $T_{4,0}(s) \cup T_{4,0}(s')$ has four nodes (a root with one child that has two children), and three edge labels (*ab*, *cd* and *ba*). In order to determine the length (and label) of the edge labeled with *ab* while taking the union, we are required to determine the length of the longest prefix of s and s'. Our goal is to do this in constant time regardless of the length of the edge labels, so simply matching the strings does not suffice. The problem is handled using *longest common extension* queries, as explained in the proof of the next lemma.

Lemma 2. *The union and intersection operations for neighborhood trees can be performed in time bounded by a linear function of the size of the input structures.*

Proof. In a neighborhood tree resulting from a union or intersection operation, the number of nodes is bounded by a linear function of the total number of nodes in the trees being operated on. The union and intersection algorithms proceed by recursively doing unions and intersections on the appropriate subtrees of the input structures, and each node in the input structures need only be visited once. When the length of each edge label is $O(1)$, we need only spend constant time at each node in the input structures to determine the identity of the nodes to be created in the resulting structure. So for this restricted case, the set operations take linear time.

The only complication arises when two edge labels of length $\omega(1)$ must be compared to determine the length of their longest common prefix (as illustrated by the example above). Sequentially matching individual characters requires time proportional to the length of the shorter of the two edge labels. We use longest common extension queries to speed up the comparison. Given a pair of start indexes (i, j) for two substrings from (not necessarily distinct) strings S and S', the longest common extension for (i, j) is the length of the longest prefix of suffix i of S that matches a prefix of suffix j of S'. The longest common extension for the starting indexes of two edge labels is equal to the length of the longest prefix that is identical in the two labels.

After linear time preprocessing, longest common extension queries can be done in constant time. This is implemented by (1) creating a generalized suffix tree for the strings, which can be done in linear time by a number of methods, and (2) augmenting the tree for lowest common ancestor queries, which can also be done in linear time (for details see [6]). After creating and augmenting the generalized suffix tree, lowest common ancestor queries can be answered in constant time. The depth of the lowest common ancestor for two leaves gives the length of the longest common extension for the corresponding suffixes.

Therefore, even when arbitrary length edge labels are allowed, the edge labels can be compared in constant time during union and intersection operations. So linear time is sufficient for union and intersection operations in the general case. □

While enumerative algorithms must have their running times dependent on the output size, we avoid this by encoding the set of motifs in a structure instead of producing each motif. The following algorithm is the best known that solves this modified problem; it also provides the best known upper bound on the complexity of the corresponding decision problem.

Algorithm 3: Pseudocode for the MotifIntersection algorithm.
Input: A set of strings $\mathcal{F}$ and two integers $d < l$.
Output: An edge-compressed neighborhood tree $T_{l,d}(\mathcal{F})$ encoding $\mathcal{M}_{\mathcal{F}}$.

```
MotifIntersection($\mathcal{F}, l, d$)
1.   for each $S_i \in \mathcal{F}$
2.       for each substring $s_{ij}$ of $S_i$ such that $|s_{ij}| = l$
3.           construct $T_{l,d}(s_{ij})$ using BuildNeighborhood
4.       $T_{l,d}(S_i) \leftarrow \cup_{1 \le j \le n-l+1} T_{l,d}(s_{ij})$
5.   $T_{l,d}(\mathcal{F}) \leftarrow \cap_{1 \le i \le m} T_{l,d}(S_i)$
6.   return $T_{l,d}(\mathcal{F})$
```

Theorem 4. *The time complexity of* MotifIntersection *is* $O(mnN)$.

Proof. By Lemma 1, each call to BuildNeighborhood requires $O(N)$ time. There are $O(m)$ union operations, which by Lemma 2, each require $O(nN)$ time. Also by Lemma 2, the intersection operation requires at most $O(mnN)$ time. □

Once the structure has been built, the complete list of motifs can be extracted in $O(lN)$ time, so the enumeration can be done in $O(mnN + lN) = O(mnN)$ time. The space requirements of the algorithm are the same as the time requirements.

5 An FPP Algorithm

Of more theoretical interest is the complexity of the problem when we are allowed a polynomial number of processors. The class of algorithms that achieve a logarithmic running time using polynomial number of processors is called NC [4]. Two analogues of NC have been defined within the context of parameterized complexity [3]. For a problem Π with parameter k, the class PNC contains problems with algorithms requiring at most $O(f(k)(\log|x|)^{g(k)})$ time, using $O(h(k)|x|^c)$ processors, on instance $x \in \Pi$, for some constant c and arbitrary functions f, g, h. The definition of the class FPP modifies the allowed time to be $O(f(k)(\log|x|)^c)$, so FPP $\subseteq$ PNC $\subseteq$ FPT (see [5] for definitions of the basic concepts of parameterized complexity).

The algorithm can be seen as an adaptation of the MotifIntersection algorithm, where the set operations proceed from the leaves to the root of a binary processor tree. For any processor p at an internal node in this tree, subscripts L and R indicate data held by the left and right children of p. Leaf processors are assigned substrings of the strings in $\mathcal{F}$, and all processors assigned a substring from the same string are arranged consecutively. For the purpose of illustration, we assume both m and $n - l + 1$ are powers of 2. Our algorithm requires that the neighborhood trees do not have compressed edges, so that construction of the suffix trees required for longest common extension queries may be avoided.

The simple lexicographic trees used instead have a number of nodes bounded by $f(|\Sigma|, l) = O(|\Sigma|^l)$.

Algorithm 4: Pseudocode for the FPPMotifs algorithm.
Input: A set $\mathcal{F}$ of strings.
Output: A lexicographic tree encoding $\mathcal{M}_{\mathcal{F}}$.

```
FPPMotifs($\mathcal{F}$)
1.      for each leaf processor $p$ (in parallel)
2.          ($p$ computes) $T_{ij} \leftarrow$ BuildNeighborhood($s_{ij}$)
3.      for $k \leftarrow 2$ to $\log(n-l+1)$
4.          for each processor $p$ (in parallel)
5.              if $p$ is at level $k$ then ($p$ computes) $T \leftarrow T_L \cup T_R$
6.      for $k \leftarrow \log(n-l+1)+1$ to $\log(m(n-l+1))$
7.          for each processor $p$ (in parallel)
8.              if $p$ is a level $k$ then ($p$ computes) $T \leftarrow T_L \cap T_R$
9.      return $T$
```

The above algorithm establishes the following result concerning the parallel parameterized complexity of the motif enumeration problem.

Theorem 5. *The motif enumeration problem can be solved in $O(f(|\Sigma|, l)\log(mn))$ time using $O(mn)$ processors.*

6 Discussion

We have presented an algorithm for enumerating all motifs for a set of sequences. This algorithm, presented as Census and MotifIntersection in two stages of incremental improvement, improves over the best previously known algorithm, in that the running time has been reduced by eliminating a factor of m, the number of strings, and of l, the length of the motifs sought. The space requirements have also been reduced by eliminating a factor of m. These improvements advance the frontier of which parameter values are usable for motif enumeration.

Given the potentially exponential size of the output, this algorithm must be close to optimal in its running time for the worst case. Consider the task of any enumerative algorithm that tries to obtain all motifs for a set of sequences. In the worst case, there are $N = \Theta(n\binom{l}{d}|\Sigma|^d)$ motifs, so no algorithm can eliminate the factor $nN = n\binom{l}{d}|\Sigma|^d$ from the time complexity. Also, the factor mn represents the size of the input, so those two cannot be separated; any algorithm must thus take at least $\Omega(mn + nN) = \Omega(mn + n\binom{l}{d}|\Sigma|^d)$ time. It seems unlikely that the factor m can be separated from $N = \binom{l}{d}|\Sigma|^d$ through some sort of preprocessing without introducing a factor of $|\Sigma|^l$ into the time complexity.

This algorithm can be parallelized in a variety of ways. It admits both a coarse and a finer grained parallelism. The coarse grained parallel algorithm reduces the time by a factor of p, for any number of processors $1 \leq p \leq N$, though it requires $O(mn)$ space for each processor. The finer grained algorithm, on the other hand, runs in $O(l(mn/p + \log p)N)$ time and $O(n)$ space for each of p processors. Our motif enumeration can also be parallelized to run in $O(f(|\Sigma|, l) \log(mn))$ time, using a linear number of processors, which positions the problem in FPP.

Acknowledgments. The authors thank Todd Wareham for helpful comments and discussions.

References

1. A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 257–300. MIT Press/Elsevier, 1990.
2. M. Blanchette, B. Schwikowski, and M. Tompa. An exact algorithm to identify motifs in orthologous sequences from multiple species. *Proceedings of the Annual International Conference on Computational Molecular Biology*, pages 37–45. ACM Press, 2000.
3. Marco Cesati and Miriam Di Ianni. Parameterized parallel complexity. Technical Report 4(6), *Electronic Colloquium on Computational Complexity (ECCC)*, 1997.
4. Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–21, 1985.
5. R. Downey and M. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer-Verlag, New York, 1999.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
7. J. van Helden, B. Andre, and J. Collado-Vides. Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies. *Journal of Molecular Biology*, 281:827–842, 1998.
8. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, 1976.
9. Marie-France Sagot. Spelling approximate repeated or common motifs using a suffix tree. In C. L. Lucchesi and A. V. Moura, editors, *Latin '98: Theoretical Informatics*, volume 1390 of *Lecture Notes in Computer Science*, pages 374–390. Springer, 1998.
10. S. Sinha and M. Tompa. A statistical method for finding transcription factor binding sites. *Proceedings of the Annual International Symposium on Intelligent Systems for Molecular Biology*, 2000. AAAI Press, pages 344–344.
11. Martin Tompa. An exact method for finding short motifs in sequences, with application to the ribosome binding site problem. *Proceedings of the Annual International Symposium on Intelligent Systems for Molecular Biology*, 1999. AAAI Press, pages 262–271.
12. L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the Association for Computing Machinery*, 33(8):103-111, 1990.
13. A. Vanet, L. Marsan, A. Labigne and M.-F. Sagot. Inferring regulatory elements from a whole genome. An application to the analysis of the genome of Helicobacter pylori σ^{80} family of promoter signals. *Journal of Molecular Biology*, 297:335–353, 2000.