4-1-2016

# Data Structures and Algorithms for the Identification of Biological Patterns

Marius Nicolae
*University of Connecticut - Storrs*, mariusmni@gmail.com

# Data Structures and Algorithms for the Identification of Biological Patterns

Marius Nicolae, PhD

University of Connecticut, 2016

This thesis studies the following problems:

**1. Planted Motif Search.** Discovering patterns in biological sequences is a crucial process that has resulted in the determination of open reading frames, gene promoter elements, intron/exon splicing sites, SH RNAs, etc. We study the $(\ell, d)$ motif search problem or Planted Motif Search (PMS). PMS receives as input $n$ strings and two integers $\ell$ and $d$. It returns all sequences $M$ of length $\ell$ that occur in each input string, where each occurrence differ from $M$ in at most $d$ positions. Another formulation is quorum PMS (qPMS), where $M$ appears in at least $q\%$ of the strings. We developed qPMS9, an efficient parallel exact qPMS algorithm for DNA and protein datasets.

**2. Suffix Array Construction.** The suffix array is a data structure that finds numerous applications in string processing problems for both linguistic texts and biological data. The suffix array consists of the sorted suffixes of a string. There are several linear time suffix array construction algorithms known in the literature. However, one of the fastest algorithms in practice has a worst case run time of $O(n^2)$. We developed an efficient algorithm called RadixSA that has a worst case run time of $O(n \log n)$ and is one of the fastest algorithms to date. RadixSA introduces an idea that may find independent applications as a speedup technique for other algorithms.

**3. Pattern Matching with Mismatches**. We consider several variants of the pattern matching with mismatches problem. Given a text $T = t_1 t_2 \cdots t_n$ and a pattern $P = p_1 p_2 \cdots p_m$, we investigate the following problems: 1) *Pattern matching with mismatches:* for every alignment $i, 1 \leq i \leq n - m + 1$ output the distance between $P$ and $t_i t_{i+1} \cdots t_{i+m-1}$, and 2) *Pattern matching with k mismatches:* output those alignments $i$ where the distance is at most $k$. The distance metric used is the Hamming distance. Variants of these problems allow for wild cards in the text or the pattern. For these problems we offer novel deterministic, randomized and approximation algorithms.

Source code relevant to these results is available at https://github.com/mariusmni/.

# Data Structures and Algorithms for the Identification of Biological Patterns

Marius Nicolae

Dipl. Ing., Politehnica University of Bucharest, 2009

M.Sc., University of Connecticut, 2011

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2016

APPROVAL PAGE

Doctor of Philosophy Dissertation

**Data Structures and Algorithms for the Identification of Biological Patterns**

Presented by

Marius Nicolae

Major Advisor _____

Prof. Sanguthevar Rajasekaran

Associate Advisor _____

Prof. Ion Măndoiu

Associate Advisor _____

Prof. Yufeng Wu

University of Connecticut

2016

# Acknowledgments

This thesis is based on the following publications: [63], [65], [77], [64] and [66]. I would like to thank my major advisor, Professor Sanguthevar Rajasekaran, who is also a co-author on the above papers, for his significant contribution to this thesis and for the experience of working together for the past few years. I would also like to thank my associate advisors Professor Ion Măndoiu and Professor Yufeng Wu for their support and suggestions. I thank also Professor Chun-Hsi (Vincent) Huang, Professor Mukul Bansal and Dr. Soumitra Pal for their participation in the review of this work. I am grateful to my colleagues from the Applied Algorithms Lab at the Booth Center for Advanced Technology for the collaborations and sharing of ideas and to the many excellent teachers in the Computer Science Department at UConn whose classes I was privileged to attend.

I would like to acknowledge the many teachers and colleagues that have influenced me in my undergraduate and high school years. In particular, my first steps into computer science were guided by Professor Doru Popescu Anastasiu from Radu Greceanu National College in Slatina, Romania. He was also the one who recommended the UConn grad school program to me. I am grateful for many other influences from professors and former students training the Romanian National Informatics Olympic team of which I was a part in my senior high school years. Finally, I would like to thank my family and friends for their patience and support while I undertook the grad school program many miles away.

This work has been supported in part by the following grants: NSF 0829916, NSF 1447711 and NIH R01-LM010101.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Planted Motif Search

## 1.1 Introduction

Motif searching is an important step in the detection of rare events occurring in a set of DNA or protein sequences. The Planted Motif Search (PMS) problem, also known as the $(l, d)$-motif problem, has been introduced in [69] with the aim of detecting motifs and significant conserved regions in a set of DNA or protein sequences. PMS receives as input $n$ biological sequences and two integers $\ell$ and $d$. It returns all possible biological sequences $M$ of length $\ell$ such that $M$ occurs in each of the input strings, and each occurrence differs from $M$ in at most $d$ positions. Any such $M$ is called a motif. Given two $\ell$-mers, the number of positions in which they differ is called their Hamming distance.

Buhler and Tompa [10] have employed PMS algorithms to find known transcriptional regulatory elements upstream of several eukaryotic genes. In particular, they have used orthologous sequences from different organisms upstream of four different genes: preproinsulin, dihydrofolate reductase (DHFR), metallothioneins, and c-fos. These sequences are known to contain binding sites for specific transcription factors. Their algorithm successfully identified the experimentally determined transcription factor binding sites. They have also employed their algorithm to solve the ribosome binding site problem for various prokaryotes. Eskin and Pevzner [31] used PMS algorithms to find composite regulatory patterns using their PMS algorithm called MITRA. They have employed the upstream regions involved in purine metabolism from three *Pyrococcus* genomes. They have also tested their algorithm on four sets of *S.cerevisiae* genes which are regulated by two transcription factors such that the transcription factor binding sites occur near each other. Price, et al. [72] have employed their PatternBranching PMS technique to find motifs on a sample containing CRP binding sites in *E.coli*, upstream regions of many

organisms of the eukaryotic genes: preproinsulin, DHFR, metallothionein, & c-fos, and a sample of yeast promoter regions.

A problem that is very similar to $(\ell, d)$ motif search is the Closest Substring problem. The Closest Substring problem is essentially the PMS problem where the aim is to find the smallest $d$ for which there exists at least one motif. These two problems have applications in PCR primer design, genetic probe design, discovering potential drug targets, antisense drug design, finding unbiased consensus of a protein family, creating diagnostic probes and motif finding (see e.g., [54]). Therefore, the development of efficient algorithms for solving the PMS problem constitute an active interest in biology and bioinformatics.

In a practical scenario, instances of the motif may not appear in all of the input strings. This has led to the introduction of a more general formulation of the problem, called quorum PMS (qPMS). In qPMS we are interested in motifs that appear in at least $q$ percent of the $n$ input strings. Therefore, when $q = 100\%$ the qPMS problem is the same as PMS.

The Closest Substring problem is NP-Hard [54]. The Closest Substring problem can be solved by a linear number of calls to PMS. Therefore, there is a polynomial time reduction from Closest Substring to PMS, which means that the PMS problem is also NP-Hard. Because of this, all known exact algorithms have an exponential runtime in the worst case. Thus, it is important to develop efficient algorithms in practice.

The practical performance of PMS algorithms is typically evaluated on datasets generated as follows (see [69, 27]): 20 DNA/protein strings of length 600 are generated according to the independent identically distributed (i.i.d.) model. Then, a random motif ($\ell$-mer) $M$ is similarly generated and "planted" at a random location in each input string (or in $q\%$ of the input strings for qPMS). Every planted instance of the motif is mutated in exactly $d$ positions.

**Definition 1.** *An $(\ell, d)$ instance is defined to be a* **challenging instance** *if $d$ is the largest integer for which the expected number of motifs of length $\ell$ that would occur in the input by random chance does not exceed a constant (500 in this thesis, same as in [63]).*

Intuitively, the more we increase $d$, the more we increase the search space. However, if we increase $d$ too much, we find many motifs just by random chance (spurious motifs). Hence, the challenging instances for PMS on DNA data, according to the above definition, are $(13, 4)$, $(15, 5)$, $(17, 6)$, $(19, 7)$, $(21, 8)$, $(23, 9)$, $(25, 10)$, $(26, 11)$, $(28, 12)$, $(30, 13)$, etc.

A PMS algorithm can be exact or approximate. An exact algorithm finds all the existing motifs. Note that in this chapter we only address exact algorithms. Namely, we will discuss two algorithms: PMS8

[63] and its successor qPMS9 [65].

Given a tuple of $\ell$-mers, the set of $\ell$-mers that have a Hamming Distance of no more than $d$ from any $\ell$-mer in the tuple is called the *common $d$-neighborhood* of the tuple.

There are many PMS algorithms in the literature. Most of the exact PMS algorithms use a combination of two fundamental techniques. One technique is sample driven and the other technique is pattern driven. In the sample driven stage, the algorithm selects a tuple of $\ell$-mers coming from distinct input strings. Then, in the pattern driven stage, the algorithm generates the common $d$-neighborhood of the $\ell$-mers in the tuple. Each neighbor becomes a motif candidate. The size of the tuple is usually fixed to a value such as 1 (see e.g. [75, 27, 76]), 2 (see e.g. [89]), 3 (see e.g. [86, 29, 7, 30]) or $n$ (see e.g. [69, 79]). The algorithms described in this chapter, PMS8 [63] and qPMS9 [65], utilize a variable tuple size, which adapts to the problem instance under consideration.

For tuples of size 3, qPMS7 [30] computes neighborhoods by using an Integer Linear Programming (ILP) formulation. A large number of ILP instances are solved and stored in a table, as a preprocessing step. This table is then repeatedly looked up to identify common neighbors of three $l$-mers. This preprocessing step takes a considerable amount of time and the look up table requires a large amount of memory.

In this chapter we state and prove necessary and sufficient conditions for 3 $l$-mers to have a common neighbor, therefore removing the requirement for a large look up table. These conditions generalize to necessary (but not sufficient) conditions for 4 or more $\ell$-mers to have a common neighborhood. These conditions are used as pruning techniques that form the basis for the efficiency of PMS8, along with several speedup techniques.

We have used PMS8 as the basis for the qPMS9 algorithm. The qPMS9 algorithm extends PMS8 in several ways. First, qPMS9 introduces a string reordering procedure which significantly increases performance by allowing for better pruning of the search space. Second, qPMS9 adds support for solving the qPMS problem, which was lacking in PMS8.

The first algorithm to solve the challenging DNA instance $(23, 9)$ has been the qPMS7 algorithm [30]. The algorithm in [28] can solve instances with relatively large $l$ (up to 48) provided that $d$ is at most $l/4$. However, most of the well known challenging instances have $d > l/4$. PairMotif [89] can solve instances with larger $l$, such as $(27, 9)$ or $(30, 9)$, but these are significantly less challenging than $(23, 9)$.

The first algorithm to solve $(25, 10)$, in a reasonable amount of time (no more than two days using commodity processors) has been TraverStringRef [86]. The TraverStringRef algorithm [86] is an algorithm for the qPMS problem, based on the earlier qPMS7 [30] algorithm. PMS8 can solve DNA instances

(25,10), on a single core machine, and (26,11) on a multi-core machine. Its successor, qPMS9, can solve $(28, 12)$ and $(30, 13)$ on a single core machine. Several of these algorithms are compared with PMS8 and qPMS9 in section 1.3.

## 1.2 Methods

We start by defining the PMS and qPMS problems more formally. A string of length $\ell$ is called an $\ell$-mer. Given two $\ell$-mers $u$ and $v$, the number of positions where the two $\ell$-mers differ is called their Hamming distance and is denoted as $Hd(u, v)$. If $T$ is a string, $T[i..j]$ denotes the substring of $T$ starting at position $i$ and ending at position $j$.

**Problem 1. PMS**: *Given $n$ sequences $s_1, s_2, \ldots, s_n$, over an alphabet $\Sigma$, and two integers $\ell$ and $d$, identify all $\ell$-mers $M$, such that $M \in \Sigma^l$ and $\forall i, 1 \leq i \leq n$, $\exists j_i, 1 \leq j_i \leq |s_i| - l + 1$, s. t. $Hd(M, s_i[j_i..j_i + l - 1]) \leq d$.*

**Problem 2. qPMS**: *same as the PMS problem, however the motif should appear in at least $q\%$ of the strings, instead of all of them. PMS is a special case of qPMS for which $q = 100\%$.*

Another useful notion is that of a $d$-neighborhood. Given a tuple of $\ell$-mers $T = (t_1, t_2, \ldots, t_s)$, the common $d$-neighborhood of $T$ includes all the $\ell$-mers $r$ such that $Hd(r, t_i) \leq d$, $\forall 1 \leq i \leq s$.

We now define the consensus $\ell$-mer and the consensus total distance for a tuple of $\ell$-mers. Given a tuple of $\ell$-mers $T = (t_1, \ldots, t_k)$ the **consensus $\ell$-mer** of $T$ is an $\ell$-mer $u$ where $u[i]$ is the most common character among $(t_1[i], t_2[i], \ldots, t_k[i])$ for each $1 \leq i \leq \ell$. If the consensus $\ell$-mer for $T$ is $p$ then the **consensus total distance** of $T$ is defined as $Cd(T) = \sum_{u \in T} Hd(u, p)$. While the consensus string is generally not a motif, the consensus total distance provides a lower bound on the total distance between any motif and a tuple of $\ell$-mers, as we show in section 1.2.3.

As indicated previously, most of the motif search algorithms combine a sample driven approach with a pattern driven approach. In the sample driven part, tuples of $\ell$-mers $(t_1, t_2, \ldots, t_k)$ are generated, where $t_i$ is an $\ell$-mer in $S_i$. Then, in the pattern driven part, for each tuple, its common $d$-neighborhood is generated. Every $\ell$-mer in the neighborhood is a candidate motif. In PMS8 and qPMS9, the tuple size $k$ is variable. By default, a good value for $k$ is estimated heuristically (see [63]) based on the input parameters, or $k$ can be user specified.

### 1.2.1 Generating Tuples of $\ell$-mers

In the sample driven part of PMS8 we generate tuples $T = (t_1, t_2, \ldots, t_k)$, where $t_i$ is an $\ell$-mer from string $s_i$, $\forall i = 1..k$, based on the following principles. First, if $T$ has a common $d$-neighbor, then every subset of $T$ has a common $d$-neighbor. Second, there has to be at least one $\ell$-mer $u$ in each of the remaining strings $s_{k+1}, s_{k+2}, \ldots, s_n$ such that $T \cup \{u\}$ has a common $d$-neighbor. We call such $\ell$-mers $u$ "alive" with respect to tuple $T$.

As we add $\ell$-mers to $T$, we update the alive $\ell$-mers in the remaining strings. Based on the number of alive $\ell$-mers, in PMS8 we reorder the remaining strings increasingly. This is a heuristic that speeds up the search because the first $\ell$-mers in the tuple are the most expensive so we want as few combinations of them as possible. However, in qPMS9 we used the following more efficient string reorder heuristic. Let $u$ be an alive $\ell$-mer with respect to $T$. If we add $u$ to $T$, then the consensus total distance of $T$ increases. We compute this additional distance $Cd(T \cup \{u\}) - Cd(T)$. For each of the remaining strings, we compute the minimum additional distance generated by any alive $\ell$-mer in that string. Then we sort the strings decreasingly by the minimum additional distance. Therefore, we give priority to the string with the largest minimum additional distance. The intuition is that larger minimum additional distance could indicate more "diversity" among the $\ell$-mers in the tuple, which means smaller common $d$-neighborhoods. If two strings have the same minimum additional distance, we give priority to the string with fewer alive $\ell$-mers.

The tuple generation is described in algorithm 1. We invoke the algorithm as $GenTuples(\{\}, k, R)$ where $k$ is the desired size of the tuples and $R$ is a matrix that contains all the $\ell$-mers in all the input strings, grouped as one row per string. This matrix is used to keep track of alive $\ell$-mers. To exclude tuples that cannot have a common neighbor we employ the pruning techniques in section 1.2.3.

### 1.2.2 Generating Common Neighborhoods

For every tuple that algorithm 1 generates we want to generate a common neighborhood. Namely, given a tuple $T = (t_1, t_2, \ldots, t_k)$ of $\ell$-mers, we want to generate all $\ell$-mers $M$ such that $Hd(t_i, M) \leq d, \forall i = 1..k$. To do this, we traverse the tree of all possible $\ell$-mers, starting with an empty string and adding one character at a time. A node at depth $r$, which represents an $r$-mer, is pruned if certain conditions are met (see section 1.2.3). The pseudocode for neighborhood generation is given in algorithm 2.

**Algorithm 1:** GenerateTuples$(T, k, R)$

**Input**: $T = (t_1, t_2, \ldots, t_i)$, current tuple of $\ell$-mers;
$k$, desired size of the tuple;
$R$, array of $n - i$ rows, where $R_j$ contains all alive $\ell$-mers from string $s_{i+j}$;
**Result**: Generates tuples of size $k$, containing $\ell$-mers, that have common neighbors, then passes these tuples to the `GenerateNeighborhood` function;

```
begin
    if |T| == k then
        GenerateNeighborhood(T, d);
        return;
    outerLoop: for u ∈ R_1 do
        T' := T ∪ {u};
        for j ← 1 to n − i − 1 do
            R'_j = {v ∈ R_{j+1} | ∃ common d-neighborhood for T' ∪ {v}};
            if |R'_j| == 0 then
                continue outerLoop;
            minAdd := min_{v∈R'_j} Cd(T' ∪ {v}) − Cd(T');
            aliveLmers := |s_{i+j+1}| − |R'_j|;
            sortKey[j] := (minAdd, −aliveLmers)
        sort R' decreasingly by sortKey;
        GenerateTuples (T', k, R');
```

### 1.2.3 Pruning Conditions

In this section we address the following question. Given a tuple $T = (t_1, t_2, \ldots, t_k)$ of $\ell$-mers and a tuple $D = (d_1, d_2, \ldots, d_k)$ of distances, is there an $\ell$-mer $M$ such that $Hd(M, t_i) \leq d_i, \forall i = 1..k$? This question appears in algorithm 1 where $T$ is the current tuple and $D$ is an array with all values set to $d$. The same question appears in algorithm 2 where $T$ is a tuple of suffixes and $D$ is an array of remaining distances.

Two $\ell$-mers $a$ and $b$ have a common neighbor $M$ such that $Hd(a, M) \leq d_a$ and $Hd(b, M) \leq d_b$ if and only if $Hd(a, b) \leq d_a + d_b$. For 3 $l$-mers, no trivial necessary and sufficient conditions have been known up to now. We give simple necessary and sufficient conditions for 3 $\ell$-mers to have a common neighbor. These conditions are also necessary (but not sufficient) for 4 or more $\ell$-mers.

**Lemma 1.** *Let $T = (t_1, t_2, \ldots, t_k)$ be a tuple of $\ell$-mers and $D = (d_1, d_2, \ldots, d_k)$ be a tuple of distances, and $M$ be an $\ell$-mer. If $\sum_{i=1}^{k} Hd(M, t_i) > \sum_{i=1}^{k} d_i$ then, by the pigeonhole principle, at least one $l$-mer $t_i$ must have $Hd(M, t_i) > d_i$. Therefore, $M$ cannot be a common neighbor of the $l$-mers in $T$, under the given distances.*

Suppose we have a lower bound on the total distance $\sum_{i=1}^{k} Hd(M, t_i)$, and that lower bound is independent of $M$. If that lower bound is greater than $\sum_{i=1}^{k} d_i$ then there is no $M$ that is a common neighbor for $T$. One such lower bound is the *consensus total distance*. To prove this, we observe an

**Algorithm 2:** GenerateNeighborhood($T, d$)

**Input**: $T = (t_1, t_2, \ldots, t_k)$, tuple of $\ell$-mers;
$d$, maximum distance for a common neighbor;
**Result**: Generates all common $d$-neighbors of the $\ell$-mers in $T$;
**begin**
  **for** $i \leftarrow 1$ **to** $|T|$ **do**
    $r[i] = d$
  `GenerateLMers` $(x, 0, T, r)$;
**Procedure** `GenerateLMers`$(x, p, T, r)$
  **Input**: $x$, the current $\ell$-mer being generated;
  $p$, the current length of the $\ell$-mer being generated;
  $T = (t_1, t_2, \ldots, t_k)$, tuple of $(\ell - p)$-mers;
  $r[i]$, maximum distance between the (yet to be generated) suffix of $x$ and $t_i$, $\forall i = 1..k$;
  **Result**: Generates all possible suffixes of $x$ starting at position $p$ such that the distance between the suffix of $x$ and $t_i$ does not exceed $r[i]$, $\forall i = 1..k$;
  **if** $p == \ell$ **then**
    report $\ell$-mer $x$;
  **else**
    **if not** `prune(`$T$,$r$`)` **then**
      **for** $\alpha \in \Sigma$ **do**
        $x_p = \alpha$;
        **for** $i \leftarrow 1$ **to** $|T|$ **do**
          **if** $t_i[0] == \alpha$ **then**
            $r'[i] = r[i]$;
          **else**
            $r'[i] = r[i] - 1$;
          $t'_i = t_i[1..|t_i|]$;
        `GenerateLMers`$(x, p + 1, T', r')$;

alternative definition of the consensus total distance:

**Lemma 2.** *Let $T = (t_1, t_2, \ldots, t_k)$ be a tuple of $\ell$-mers. For every $i$, the set $t_1[i], t_2[i], .., t_k[i]$ is called the $i$-th column of $T$. Let $m_i$ be the maximum frequency of any character in column $i$, i.e., $m_i$ is the frequency of the consensus character for column $i$. Then the consensus total distance $Cd(T) = \sum_{i=1}^{\ell} k - m_i$.*

Now consider the total distance $\sum_{i=1}^{k} Hd(M, t_i)$ between any $l$-mer $M$ and the $l$-mers in $T$. For any $M$, column $i$ contributes at least $k - m_i$ to the total distance. Therefore, $\sum_{i=1}^{\ell} k - m_i$ is a lower bound for the total distance. In other words, $Cd(T)$ is a lower bound for the total distance for any $M$. Therefore, we have the following lemma:

**Lemma 3.** *Let $T = (t_1, t_2, \ldots, t_k)$ be a tuple of $l$-mers and $D = (d_1, d_2, \ldots d_k)$ be a tuple of non-negative integers. There exists an $l$-mer $M$ such that $Hd(M, T_i) \leq d_i, \forall i$, only if $Cd(T) \leq \Sigma_{i=1}^{k} d_i$.*

We are now ready for the main theorem:

**Theorem 1.** *Let $T = (t_1, t_2, t_3)$ be a tuple of three $l$-mers and $D = (d_1, d_2, d_3)$ be a tuple of three non-negative integers. There exists an $l$-mer $M$ such that $Hd(M, T_i) \leq d_i, \forall i, 1 \leq i \leq 3$ if and only if the following conditions hold:*

**i)** $Cd(t_i, t_j) \leq d_i + d_j, \forall i, j, 1 \leq i < j \leq 3$

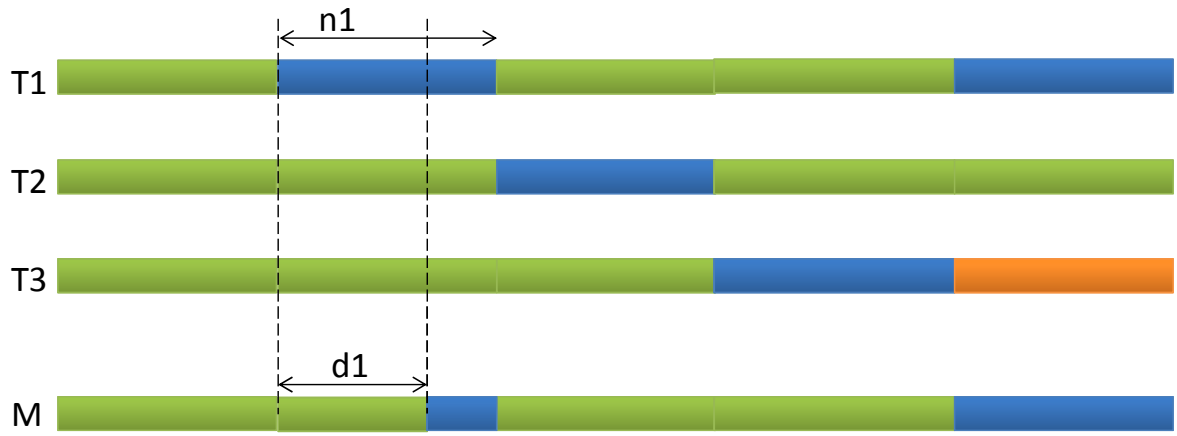
**ii)** $Cd(T) \leq d_1 + d_2 + d_3$

*Proof.* The "only if" part follows from lemma 3 and from the fact that if three $\ell$-mers have a common neighbor then any two of them must also have a common neighbor.

For the "if" part we show how to construct a common neighbor $M$ provided that the conditions hold. We say that a column $k$ where $t_1[k] = t_2[k] = t_3[k]$ is of type $N_0$. If $t_1[k] \neq t_2[k] = t_3[k]$ then the column is of type $N_1$. If $t_1[k] = t_3[k] \neq t_2[k]$ then the column is of type $N_2$ and if $t_1[k] = t_2[k] \neq t_3[k]$ then the column is of type $N_3$. If all three characters in the column are distinct then the column is of type $N_4$. Let $n_i, \forall i, 0 \leq i \leq 4$ be the number of columns of type $N_i$. Consider two cases:

Case 1) There exists $i, 1 \leq i \leq 3$ for which $n_i \geq d_i$. We construct $M$ as illustrated in figure 1.1. Pick $d_i$ columns of type $n_i$. For each chosen column $k$ set $M[k] = t_j[k]$ where $j \neq i$. For all other columns set $M[k] = t_i[k]$. Therefore $Cd(t_i, M) = d_i$. For $j \neq i$ we know that $Cd(t_i, t_j) \leq d_i + d_j$ from condition **i)** (condition **i** is assumed to be true at this point because we are proving the "if" part). We also know that $Cd(t_i, M) + Cd(M, t_j) \leq Cd(t_i, t_j)$ from the triangle inequality. It follows that $Cd(M, t_j) \leq d_j$. Since $Cd(M, t_j) = Hd(M, t_j)$ it means that $M$ is indeed a common neighbor of the three $l$-mers.

Figure 1.1: Proof of theorem 1, case 1




Proof of theorem 1, case 1: There exists $i, 1 \leq i \leq 3$ for which $n_i \geq d_i$. Without loss of generality we assume $i = 1$. The top 3 rows represent the input $l$-mers. The last row shows a common neighbor $M$. In any column, identical colors represents matches, different colors represent mismatches.

Case 2) For all $i, 1 \leq i \leq 3$ we have $n_i < d_i$. We construct $M$ as shown in figure 1.2. For columns $k$ of type $N_0, N_2$ and $N_3$ we set $M[k] = t_1[k]$. For columns of type $N_1$ we set $M[k] = t_2[k]$. For any $i, 1 \leq i \leq 3$ the following applies. If $n_i + n_4 \leq d_i$ then the Hamming distance between $M$ and $t_i$ is less than $d_i$ regardless of what characters we choose for $M$ in the columns of type $N_4$. On the other hand, if $n_i + n_4 > d_i$ then $M$ and $t_i$ have to match in at least $n_i + n_4 - d_i$ columns of type $N_4$. Thus, we pick $max(0, n_i + n_4 - d_i)$ columns of type $N_4$ and for each such column $k$ we set $M[k] = t_i[k]$. Now we prove that we actually have enough columns to make the above choices, in other words $\Sigma_{i=1}^{3} max(0, n_i + n_4 - d_i) \leq n_4$. This is equivalent to the following conditions being true:

**a)** For any $i, 1 \leq i \leq 3$ we want $n_i + n_4 - d_i \leq n_4$. This is true because $n_i < d_i$.

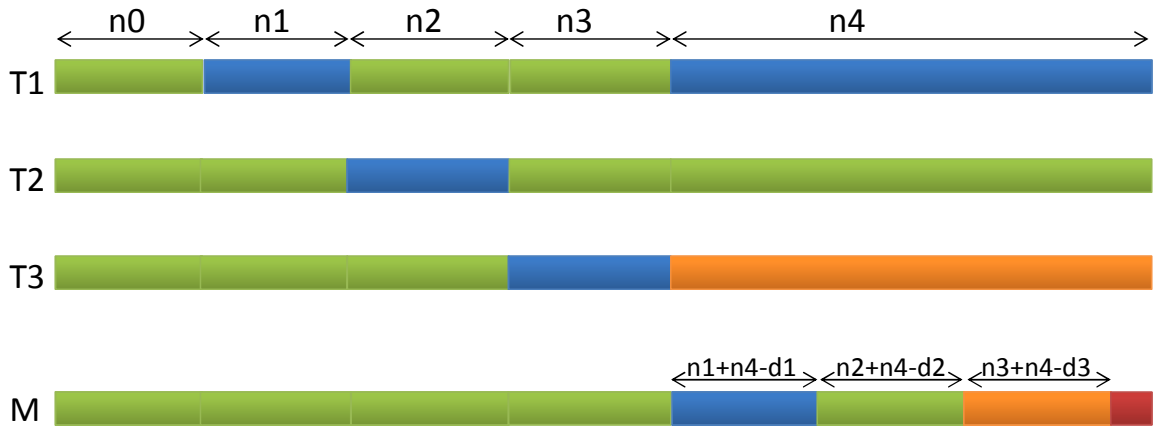
**b)** For any $i, j, 1 \leq i < j \leq 3$ we want $(n_i + n_4 - d_i) + (n_j + n_4 - d_j) \leq n_4$. This can be rewritten as $n_i + n_j + n_4 \leq d_i + d_j$. The left hand side is $Hd(t_i, t_j)$ which we know is less or equal to $d_i + d_j$.

**c)** We want $\Sigma_{i=1}^{3} n_i + n_4 - d_i \leq n_4$. This can be rewritten as $n_1 + n_2 + n_3 + 2n_4 \leq d_1 + d_2 + d_3$. The left hand side is $Cd(T)$ which we know is less than $d_1 + d_2 + d_3$.

□

One of our reviewers kindly pointed out that the above proof is similar to an algorithm in [38].

Figure 1.2: Proof of theorem 1, case 2




Proof of theorem 1, case 2: $n_i < d_i$ for all $i$, $1 \leq i \leq 3$. The top 3 rows represent the input $l$-mers. The last row shows a common neighbor $M$. In any column, identical colors represents matches, different colors represent mismatches.

### 1.2.4 Adding Quorum Support

In this section we extend the above techniques to solve the qPMS problem. In the qPMS problem, when we generate tuples of $\ell$-mer, we may "skip" some of the strings. This translates to the implementation as follows: in the PMS version we successively try every alive $\ell$-mer in a given string by adding it to the tuple $T$ and recursively calling the algorithm for the remaining strings. For the qPMS version we have an additional step where, if the value of $q$ permits, we skip the current string and try $\ell$-mers from the next string. At all times we keep track of how many strings we have skipped. The pseudocode is given in algorithm 3. We invoke the algorithm as $QGenerateTuples(n - Q + 1, \{\}, 0, k, R)$ where $Q = \lfloor \frac{qn}{100} \rfloor$ and $R$ contains all the $\ell$-mers in all the strings.

### 1.2.5 Parallel Algorithm

PMS8 and qPMS9 are parallel algorithms. Processor 0 acts as both a master and a worker, the other processors are workers. Each worker requests a subproblem from the master, solves it, then repeats until all subproblems have been solved. Communication between processors is done using the Message Passing Interface (MPI).

In PMS8, the subproblems are generated as follows. The search space is split into $m = |s_1| - \ell + 1$ independent subproblems $P_1, P_2, \ldots, P_m$, where $P_i$ explores the $d$-neighborhood of $\ell$-mer $s_1[i..i + \ell - 1]$.

In qPMS9, we extend the previous idea to the $q$ version. We split the problem into subproblems $P_{1,1}, P_{1,2}, \ldots, P_{1,|s_1|-\ell+1}$, $P_{2,1}, P_{2,2}, \ldots, P_{2,|s_2|-\ell+1}$, $\ldots$, $P_{r,1}, P_{r,2}, \ldots, P_{r,|s_r|-\ell+1}$ where $r = n - Q + 1$

**Algorithm 3:** QGenerateTuples($qTolerance, T, k, R$)

**Input**: $qTolerance$, number of strings we can afford to skip;
$T = (t_1, t_2, \ldots, t_i)$, current tuple of $\ell$-mers;
$i$, last string processed;
$k$, desired size of the tuple;
$R = (R_1, \ldots R_{n-i})$, where $R_j$ contains all alive $\ell$-mers in $s_{i+j}$;

**Result**: Generates tuples of size $k$, containing $\ell$-mers, that have common neighbors, then passes these tuples to the `GenerateNeighborhood` function;

**begin**

- **if** $|T| == k$ **then**
  - `GenerateNeighborhood`$(T, d)$;
  - **return**;
- outerLoop: **for** $u \in R_1$ **do**
  - $T' := T \cup \{u\}$;
  - $incompat := 0$;
  - **for** $j \leftarrow 1$ **to** $n - i - 1$ **do**
    - $R'_j = \{v \in R_{j+1} | \exists$ common $d$-neighborhood for $T' \cup \{v\}\}$;
    - **if** $|R'_j| == 0$ **then**
      - **if** $incompat \geq qTolerance$ **then**
        - `continue` outerLoop;
      - $incompat++$;
    - $minAdd := \min_{v \in R'_j}$ `Cd`$(T' \cup \{v\}) -$ `Cd`$(T')$;
    - $aliveLmers := |s_{i+j+1}| - |R'_j|$;
    - $sortKey[j] := (minAdd, -aliveLmers)$
  - sort $R'$ decreasingly by $sortKey$;
  - `QGenerateTuples` $(qTolerance - incompat, T', k, R')$;
- **if** $qTolerance > 0$ **then**
  - `QGenerateTuples` $(qTolerance - 1, T, k, R \setminus R_1)$;

and $Q = \lfloor \frac{qn}{100} \rfloor$. Problem $P_{i,j}$ explores the $d$-neighborhood of the $j$-th $\ell$-mer in string $s_i$ and searches for $\ell$-mers $M$ such that there are $Q-1$ instances of $M$ in strings $s_{i+1}, \ldots, s_n$. Notice that $Q$ is fixed, therefore subproblems $P_{i,j}$ get progressively easier as $i$ increases.

### 1.2.6 Speedup Techniques

#### Speed up Hamming Distance calculation by packing $\ell$-mers

By packing $\ell$-mers in advance we can speed up Hamming distance operations. For example, we can pack 8 DNA characters in a 16 bit integer. To compute the Hamming distance between two $l$-mers we first perform an exclusive or of their packed representations. Equal characters produce groups of zero bits, different characters produce non-zero groups of bits. For every possible 16 bit integer $i$ we precompute the number of non-zero groups of bits in $i$ and store it in a table. Therefore, one table look up provides the Hamming distance for 8 DNA characters. The same technique applies to any alphabet $\Sigma$ besides DNA.

For an alphabet $\Sigma$ let $b = \lceil \log |\Sigma| \rceil$ be the number of bits required to encode one character. Then, one compressed $\ell$-mer requires $\ell * b$ bits of storage. However, due to the overlapping nature of the $\ell$-mers in our input strings, we can employ the following trick. In a 16 bit integer we can pack $p = \lfloor 16/b \rfloor$ characters. For every $\ell$-mer we only store the bit representation of its first $p$ characters. The bit representation of the next $p$ characters is the same as the bit representation of the first $p$ characters of the $l$-mer $p$ positions to the right of the current one. Therefore, the table of compressed $l$-mers requires constant memory per $\ell$-mer, for a total of $O(n(m-l+1))$ words of memory.

#### Preprocess Hamming distances for all pairs of input $\ell$-mers

The filtering step tests many times if two $l$-mers have a distance of no more than $2d$. Thus, for every pair of $l$-mers we preprocess this boolean information, provided the required storage memory is not too high.

#### Find motifs for a subset of the strings

We also use the speedup technique described in [76]: compute the motifs for $n' < n$ of the input strings, then test each motif to see where it appears in the remaining $n - n'$ strings.

**Cache locality**

We can update $R$ in an efficient manner as follows. Every row in the updated matrix $R'$ is a subset of the corresponding row in the current matrix $R$, because some elements will be filtered out. Therefore, we can store $R'$ in the same memory locations as $R$. To do this, in each row, we move the elements belonging to $R'$ at the beginning of the row and keep track of how many elements belong to $R'$. To go from $R'$ back to $R$, we just have to restore the row sizes to their previous values. The row elements will be the same even if they have been permuted within the row. The same process can be repeated at every step of the recursion, therefore the whole "stack" of $R$ matrices is stored in a single matrix. This reduces the memory requirement and improves cache locality. The cache locality is improved because at every step of the recursion, in each row, we access a subset of the elements we accessed in the previous step, and those elements are in contiguous locations of memory.

### 1.2.7 Memory and Runtime

Since we store all matrices $R$ in the space of a single matrix they only require $O(n(m-l+1))$ words of memory. To this we add $O(n^2)$ words to store row sizes for the at most $n$ matrices which share the same space. The bits of information for $l$-mer pairs that have Hamming distance no more than $2d$ require $O((n(m-l+1))^2/w)$ words, where $w$ is the number of bits in a machine word. The table of compressed $l$-mers takes $O(n(m-l+1))$ words. Therefore, the total memory used by the algorithm is $O(n(n+m-l+1)+(n(m-l+1))^2/w)$.

### 1.2.8 Expected Number of Spurious Motifs

It is useful to estimate how many "spurious" motifs (motifs expected by random chance) will be found in a random sample. For that, we make the following observations. The probability that a random $\ell$-mer $u$ is within distance at most $d$ from another $\ell$-mer $v$ is

$$p(\ell, d, \Sigma) = \frac{\sum_{i=0}^{d} \binom{\ell}{i} (|\Sigma|-1)^i}{\Sigma^\ell} \tag{1.1}$$

The probability that an $\ell$-mer is within distance $d$ from any of the $\ell$-mers in a string $S$ of length $m$ is:

$$P(m,\ell,d,\Sigma) = 1 - (1 - p(\ell,d,\Sigma))^{m-\ell+1} \tag{1.2}$$

The probability that an $\ell$-mer is within distance $d$ from at least $q$ out of $n$ strings of length $m$ each is:

$$Q(q,n,m,\ell,d,\Sigma) = \sum_{i=q}^{n} \binom{n}{i} P(m,\ell,d,\Sigma)^i (1 - P(m,\ell,d,\Sigma))^{n-i} \tag{1.3}$$

Therefore, the expected number of motifs for a given qPMS instance is: $|\Sigma|^\ell Q(q,n,m,\ell,\Sigma)$. Based on these formulas, we compute for every $\ell$ the largest value of $d$ such that the number of spurious motifs does not exceed 500. This gives us the challenging instance for any $l$. These values are presented in table 1.4 for DNA and table 1.5 for protein.

## 1.3 Results

As mentioned in the introduction, PMS algorithms are typically tested on datasets generated as follows. 20 strings of length 600 each are generated from the i.i.d. We choose an $\ell$-mer $M$ as a motif and plant modified versions of it in $q\%$ of the $n$ strings. Each planted instance is modified in $d$ random positions. For every $(l,d)$ combination we generate 5 random datasets and report the average runtime over all 5.

Programs were executed on the Hornet cluster at the University of Connecticut, which is a high-end, 104-node, 1408-core High Performance Computing cluster. For our experiments we used Intel Xeon X5650 Westmere cores. Results refer to single core execution, unless specified otherwise.

### 1.3.1 PMS8

In this section we analyze the performance of PMS8 [63]. The speedup obtained by the parallel version over the single core version, for several challenging instances, is presented in figure 1.3. The speedup for $p = 48$ cores is close to $S = 45$ and thus the efficiency is $E = S/p = 94\%$.

The runtime of PMS8 on instances with $l$ up to 50 and $d$ up to 21 is shown in figure 1.4. Instances which are expected to have more than 500 motifs simply by random chance (spurious motifs) are excluded. Instances where $d$ is small relative to $l$ are solved using a single CPU core. For more challenging instances we report the time taken using 48 cores.

Figure 1.3: PMS8: Speedup of the multi-core version over the single core version, for several datasets.

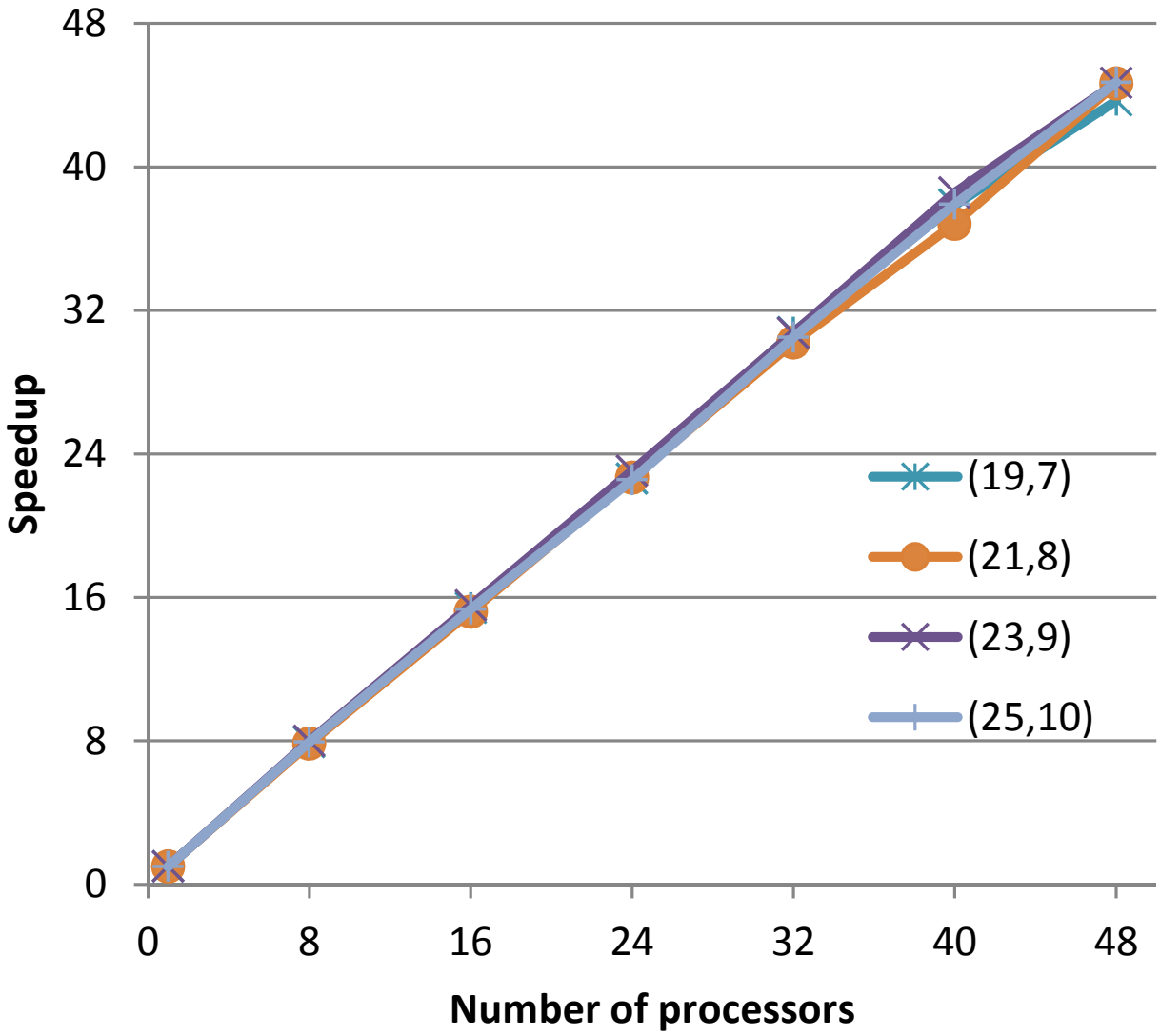
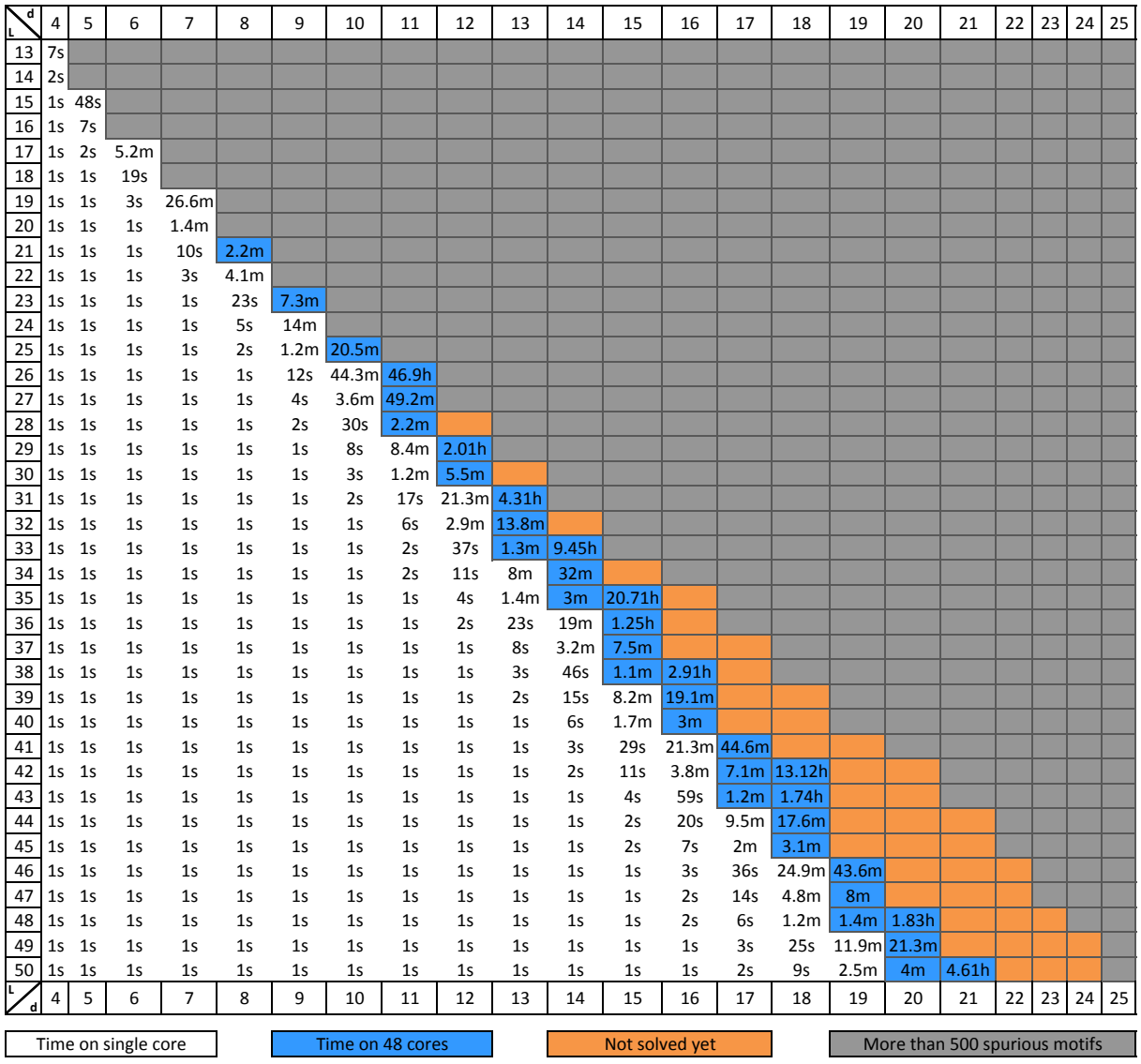


Figure 1.4: PMS8: Runtimes for datasets with $l$ up to 50 and $d$ up to 25.



| l \ d | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 7s | | | | | | | | | | | | | | | | | | | | | |
| 14 | 2s | | | | | | | | | | | | | | | | | | | | | |
| 15 | 1s | 48s | | | | | | | | | | | | | | | | | | | | |
| 16 | 1s | 7s | | | | | | | | | | | | | | | | | | | | |
| 17 | 1s | 2s | 5.2m | | | | | | | | | | | | | | | | | | | |
| 18 | 1s | 1s | 19s | | | | | | | | | | | | | | | | | | | |
| 19 | 1s | 1s | 3s | 26.6m | | | | | | | | | | | | | | | | | | |
| 20 | 1s | 1s | 1s | 1.4m | | | | | | | | | | | | | | | | | | |
| 21 | 1s | 1s | 1s | 10s | 2.2m | | | | | | | | | | | | | | | | | |
| 22 | 1s | 1s | 1s | 3s | 4.1m | | | | | | | | | | | | | | | | | |
| 23 | 1s | 1s | 1s | 1s | 23s | 7.3m | | | | | | | | | | | | | | | | |
| 24 | 1s | 1s | 1s | 1s | 5s | 14m | | | | | | | | | | | | | | | | |
| 25 | 1s | 1s | 1s | 1s | 2s | 1.2m | 20.5m | | | | | | | | | | | | | | | |
| 26 | 1s | 1s | 1s | 1s | 1s | 12s | 44.3m | 46.9h | | | | | | | | | | | | | | |
| 27 | 1s | 1s | 1s | 1s | 1s | 4s | 3.6m | 49.2m | | | | | | | | | | | | | | |
| 28 | 1s | 1s | 1s | 1s | 1s | 2s | 30s | 2.2m | | | | | | | | | | | | | | |
| 29 | 1s | 1s | 1s | 1s | 1s | 1s | 8s | 8.4m | 2.01h | | | | | | | | | | | | | |
| 30 | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 1.2m | 5.5m | | | | | | | | | | | | | |
| 31 | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 17s | 21.3m | 4.31h | | | | | | | | | | | | |
| 32 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 6s | 2.9m | 13.8m | | | | | | | | | | | | |
| 33 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 37s | 1.3m | 9.45h | | | | | | | | | | | |
| 34 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 11s | 8m | 32m | | | | | | | | | | | |
| 35 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 4s | 1.4m | 3m | 20.71h | | | | | | | | | | |
| 36 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 23s | 19m | 1.25h | | | | | | | | | | |
| 37 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 8s | 3.2m | 7.5m | | | | | | | | | | |
| 38 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 46s | 1.1m | 2.91h | | | | | | | | | |
| 39 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 15s | 8.2m | 19.1m | | | | | | | | | |
| 40 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 6s | 1.7m | 3m | | | | | | | | | |
| 41 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 29s | 21.3m | 44.6m | | | | | | | | |
| 42 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 11s | 3.8m | 7.1m | 13.12h | | | | | | | |
| 43 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 4s | 59s | 1.2m | 1.74h | | | | | | | |
| 44 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 20s | 9.5m | 17.6m | | | | | | | |
| 45 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 7s | 2m | 3.1m | | | | | | | |
| 46 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 36s | 24.9m | 43.6m | | | | | | |
| 47 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 14s | 4.8m | 8m | | | | | | |
| 48 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 6s | 1.2m | 1.4m | 1.83h | | | | | |
| 49 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 25s | 11.9m | 21.3m | | | | | |
| 50 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 9s | 2.5m | 4m | 4.61h | | | | |
| d \ l | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |



All runtimes are averages over 5 random datasets. White background signifies single core execution. Blue background signifies execution using 48 cores. Instances in gray have more than 500 spurious motifs. Orange cells indicate unsolved instances. Time is reported in hours (h), minutes (m) and seconds (s).

Table 1.1: Comparison between qPMS7 and PMS8 on challenging instances.

| **Instance** | **qPMS7** | **PMS8**$^{1}$ | **PMS8**$^{16}$ | **PMS8**$^{32}$ | **PMS8**$^{48}$ |
|---|---|---|---|---|---|
| (13,4) | 29s | 7s | 3s | 2s | 2s |
| (15,5) | 2.1m | 48s | 5s | 4s | 3s |
| (17,6) | 10.3m | 5.2m | 22s | 12s | 9s |
| (19,7) | 54.6m | 26.6m | 1.7m | 52s | 37s |
| (21,8) | 4.87h | 1.64h | 6.5m | 3.3m | 2.2m |
| (23,9) | 27.09h | 5.48h | 21.1m | 10.7m | 7.4m |
| (25,10) | - | 15.45h | 1.01h | 30.4m | 20.7m |
| (26,11) | - | - | - | - | 46.9h |

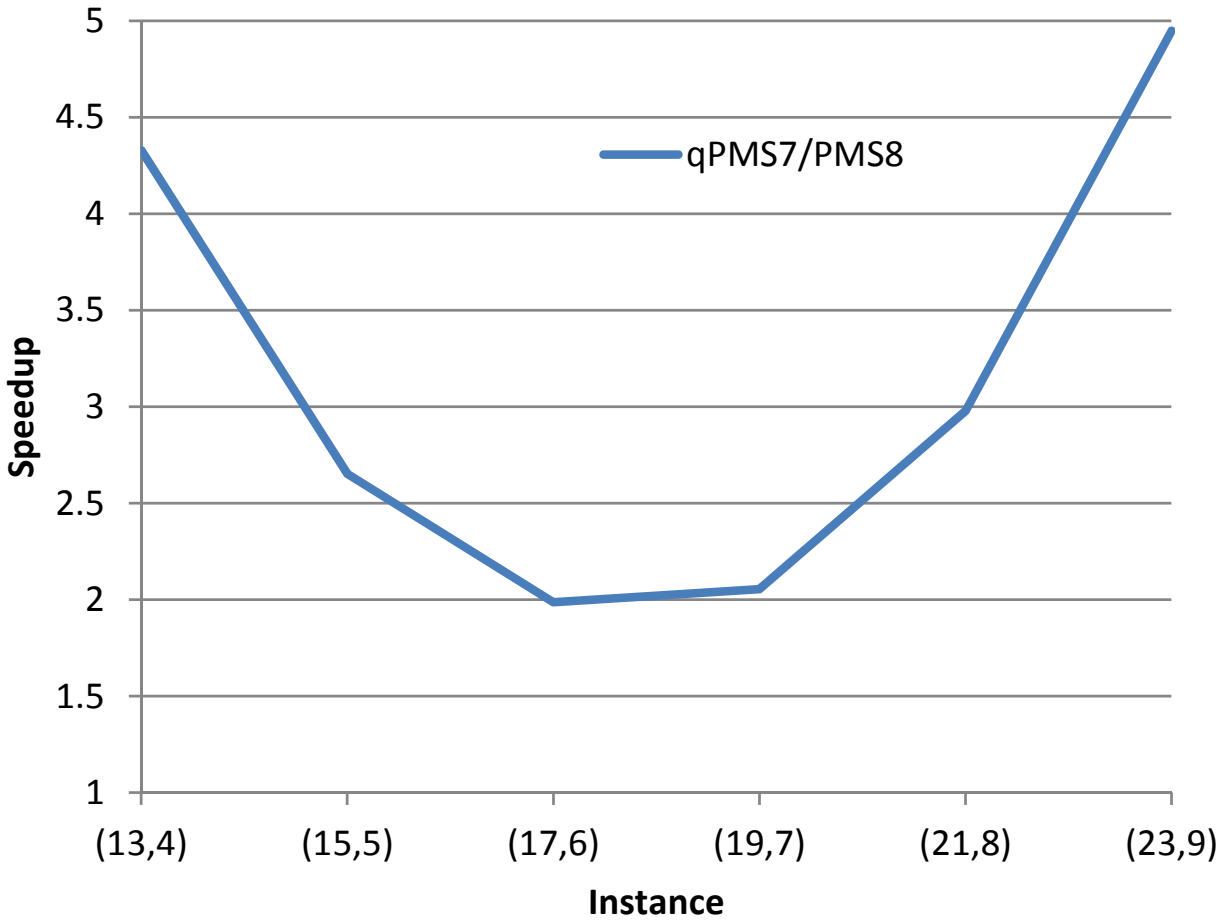
PMS8$^{P}$ means PMS8 used $P$ CPU cores. Both programs have been executed on the same hardware and the same datasets. The times are average runtimes over 5 instances for each dataset.

A comparison between PMS8 and qPMS7 [30] on challenging instances is shown in table 1.1. qPMS7 is a sequential algorithm. PMS8 was evaluated using up to 48 cores. The speedup of PMS8 single core over qPMS7 is shown in figure 1.5. The speedup is high for small instances because qPMS7 has to load an ILP table. For larger instances the speedup of PMS8 sharply increases. This is expected because qPMS7 always generates neighborhoods for tuples of 3 $l$-mers, which become very large as $l$ and $d$ grow. On the other hand, PMS8 increases the number of $l$-mers in the tuple with the instance size. The peak memory used by qPMS7 for the challenging instances in table 1.1 was 607 MB whereas for PMS8 it was 122 MB. PMS8 was the first algorithm to solve the challenging instance (26,11).

Some results in the literature have also focused on instances other than the challenging ones presented above. A summary of these results and a comparison with PMS8 is presented in table 1.2. These results have been obtained on various types of hardware: single core, multi-core, GPU, grid. In the comparison, we try to match the number of processors whenever possible. The speed difference is of several orders of magnitude in some cases which indicates that the pruning conditions employed by PMS8 significantly reduce the search space compared to other algorithms.

We compared PMS8 with qPMS7 on the real datasets discussed in [88]. We excluded datasets with less than 4 input sequences because these are not very challenging. For each dataset we chose two combinations of $l$ and $d$. These combinations were chosen on a dataset basis because for large values of $d$ the number of reported motifs is excessive and for small values of $d$ the instance is not very challenging. To make qPMS7 behave like PMS8 we set the quorum percent to 100% ($q = n$). The comparison is shown in table 1.3. Note that both algorithms are exact algorithms and therefore the sensitivity and specificity are the same.

Figure 1.5: Speedup of PMS8 single core over qPMS7.




Ratio of runtimes between qPMS7 and PMS8 running on a single core. Both programs have been executed on the same hardware and the same datasets. The times are average runtimes over 5 instances for each dataset.

Table 1.2: Comparison between PMS8 and contemporary results in the literature.

| Previous algorithm | Instance | Time | Cores | PMS8 Time | PMS8 Cores |
|---|---|---|---|---|---|
| Abbas et al. 2012 [1], PHEP_PMSprune | (21,8) | 20.42h | 8 | 6.5m | 1 |
| Yu et al. 2012 [89], PairMotif | (27, 9) | 10h | 1 | 4s | 1 |
| Desaraju and Mukkamala 2011 [28] | (24,6) | 347s | 1 | 1s | 1 |
| | (48,12) | 188s | 1 | 1s | 1 |
| Dasari et al. 2011 [26], mSPELLER / gSPELLER | (21,8) | 3.7h | 16 | 6.5m | 16 |
| | (21,8) | 2.2h | 4 GPUs x 240 cores | 6.5m | 16 |
| Dasari et al. 2010 [25], BitBased | (21,8) | 1.1h | | 6.5m | 16 |
| Dasari and Desh 2010 [24], BitBased | (21,8) | 6.9h | 16 | 6.5m | 16 |
| Sahoo et al. 2011 [80] | (16,4) | 106s | 4 | 1s | 1 |
| Sun et al. 2011 [84], TreeMotif | (40,14) | 6h | 1 | 6s | 1 |
| He et al. 2010 [83], ListMotif | (40,14) | 28,087s | 1 | 6s | 1 |
| Faheem 2010 [32], skip-Brute Force | (15,4) | 2934s | 96 nodes | 1s | 1 |
| Ho et al. 2009 [39], iTriplet | (24,8) | 4h | 1 | 5s | 1 |
| | (38,12) | 1h | 1 | 1s | 1 |
| | (40,12) | 5m | 1 | 1s | 1 |

Time is reported in seconds (s), minutes (m) or hours (h). Note that the hardware is different, though we tried to match the number of processors when possible. Also, the instances are randomly generated using the same algorithm, however the actual instances used by the various papers are most likely different. For PMS8, the times are averages over 5 randomly generated instances.

### 1.3.2 qPMS9

In the previous section we have seen that PMS8 outperformed all of the algorithms we could find in the literature at the time. After the publication of PMS8, the TraverStringRef [86] algorithm came out. Therefore, in this section we only compare PMS8, TraverStringRef and qPMS9. For $q = 100\%$ we compare all three algorithms, for $q = 50\%$ we compare only the algorithms that solve the quorum PMS problem: TraverStringRef and qPMS9.

For $q = 100\%$, we compare the three algorithm on DNA data in table 1.6. A similar comparison on protein data is given in table 1.7.

For $q = 50\%$, we compare TraverStringRef and qPMS9 on DNA data in table 1.8. A similar comparison on protein data is given in 1.9.

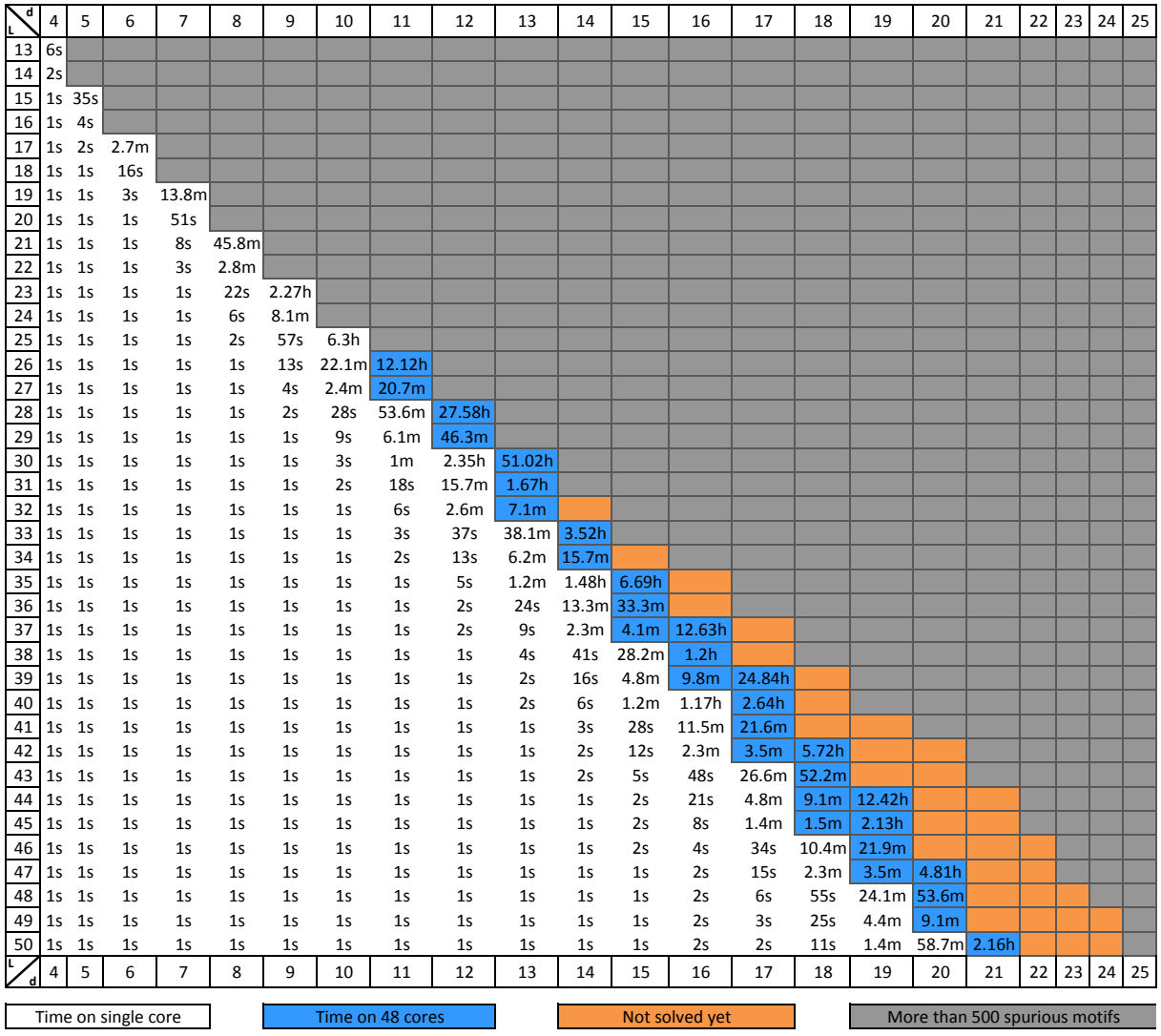
The running time of qPMS9 on DNA datasets for all combinations of $\ell$ and $d$ with $\ell$ up to 50 and $d$ up to 25, with $q = 100\%$, is given in Figure 1.6. The running time of qPMS9 on protein datasets for all combinations of $\ell$ and $d$ with $\ell$ up to 30 and $d$ up to 21, with $q = 100\%$, is given in Figure 1.7.

Table 1.3: Runtime comparison between PMS8 and qPMS7 on real datasets from [88]

| Dataset | n | Total no. bases | $\ell$ | d | PMS8 time | qPMS7 time |
|---|---|---|---|---|---|---|
| dm01r | 4 | 6000 | 21 | 4 | 1 | 55 |
| dm01r | 4 | 6000 | 23 | 5 | 1 | 6 |
| dm04r | 4 | 8000 | 21 | 4 | 1 | 5 |
| dm04r | 4 | 8000 | 23 | 5 | 1 | 5 |
| hm01r | 18 | 36000 | 21 | 6 | 10 | 14 |
| hm01r | 18 | 36000 | 23 | 7 | 25 | 40 |
| hm02r | 9 | 9000 | 21 | 6 | 1 | 11 |
| hm02r | 9 | 9000 | 23 | 7 | 4 | 35 |
| hm03r | 10 | 15000 | 21 | 6 | 3 | 24 |
| hm03r | 10 | 15000 | 23 | 7 | 14 | 146 |
| hm04r | 13 | 26000 | 21 | 6 | 6 | 44 |
| hm04r | 13 | 26000 | 23 | 7 | 15 | 39 |
| hm05r | 3 | 3000 | 21 | 4 | 1 | 6 |
| hm05r | 3 | 3000 | 23 | 5 | 1 | 46 |
| hm08r | 15 | 7500 | 17 | 5 | 1 | 7 |
| hm08r | 15 | 7500 | 17 | 6 | 46 | 251 |
| hm19r | 5 | 2500 | 23 | 5 | 1 | 5 |
| hm19r | 5 | 2500 | 23 | 6 | 1 | 5 |
| hm20r | 35 | 70000 | 21 | 6 | 27 | 32 |
| hm20r | 35 | 70000 | 23 | 7 | 56 | 136 |
| hm26r | 9 | 9000 | 23 | 6 | 1 | 5 |
| hm26r | 9 | 9000 | 23 | 7 | 5 | 46 |
| mus02r | 9 | 9000 | 21 | 6 | 1 | 11 |
| mus02r | 9 | 9000 | 23 | 7 | 2 | 45 |
| mus04r | 7 | 7000 | 21 | 6 | 1 | 15 |
| mus04r | 7 | 7000 | 23 | 7 | 2 | 22 |
| mus05r | 4 | 2000 | 21 | 5 | 1 | 79 |
| mus05r | 4 | 2000 | 23 | 6 | 1 | 5 |
| mus07r | 4 | 6000 | 21 | 5 | 1 | 79 |
| mus07r | 4 | 6000 | 23 | 5 | 1 | 6 |
| mus10r | 13 | 13000 | 21 | 6 | 2 | 56 |
| mus10r | 13 | 13000 | 23 | 7 | 2 | 70 |
| mus11r | 12 | 6000 | 21 | 7 | 8 | 150 |
| mus11r | 12 | 6000 | 23 | 8 | 23 | 938 |
| yst01r | 9 | 9000 | 21 | 6 | 2 | 14 |
| yst01r | 9 | 9000 | 23 | 7 | 8 | 63 |
| yst02r | 4 | 2000 | 21 | 5 | 1 | 5 |
| yst02r | 4 | 2000 | 23 | 6 | 1 | 6 |
| yst03r | 8 | 4000 | 21 | 6 | 1 | 8 |
| yst03r | 8 | 4000 | 23 | 7 | 1 | 19 |
| yst04r | 6 | 6000 | 21 | 4 | 1 | 5 |
| yst04r | 6 | 6000 | 23 | 5 | 1 | 5 |
| yst05r | 3 | 1500 | 21 | 4 | 1 | 5 |
| yst05r | 3 | 1500 | 23 | 5 | 1 | 5 |
| yst06r | 7 | 3500 | 21 | 6 | 1 | 6 |
| yst06r | 7 | 3500 | 23 | 7 | 2 | 12 |
| yst08r | 11 | 11000 | 21 | 5 | 1 | 6 |
| yst08r | 11 | 11000 | 23 | 6 | 1 | 6 |
| yst09r | 16 | 16000 | 21 | 6 | 2 | 17 |
| yst09r | 16 | 16000 | 23 | 7 | 6 | 68 |

For each dataset we tested two combinations of $l$ and $d$. For qPMS7 we set $q = n$. Both algorithms were executed on a single CPU core. Time is reported in seconds, rounded up to the next second.

Figure 1.6: qPMS9 runtimes on DNA datasets for multiple combinations of $\ell$ and $d$ where $q = 100\%$.

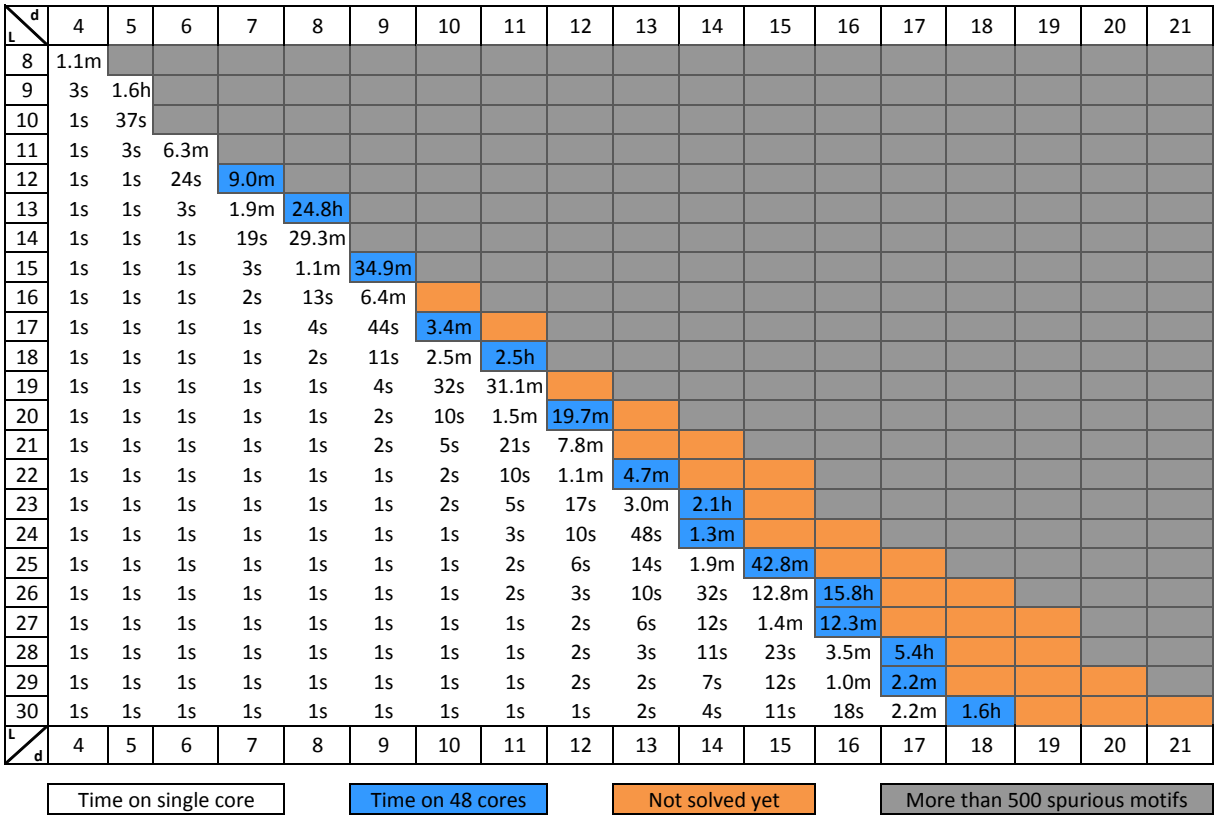


| L \ d | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 6s | | | | | | | | | | | | | | | | | | | | | |
| 14 | 2s | | | | | | | | | | | | | | | | | | | | | |
| 15 | 1s | 35s | | | | | | | | | | | | | | | | | | | | |
| 16 | 1s | 4s | | | | | | | | | | | | | | | | | | | | |
| 17 | 1s | 2s | 2.7m | | | | | | | | | | | | | | | | | | | |
| 18 | 1s | 1s | 16s | | | | | | | | | | | | | | | | | | | |
| 19 | 1s | 1s | 3s | 13.8m | | | | | | | | | | | | | | | | | | |
| 20 | 1s | 1s | 1s | 51s | | | | | | | | | | | | | | | | | | |
| 21 | 1s | 1s | 1s | 8s | 45.8m | | | | | | | | | | | | | | | | | |
| 22 | 1s | 1s | 1s | 3s | 2.8m | | | | | | | | | | | | | | | | | |
| 23 | 1s | 1s | 1s | 1s | 22s | 2.27h | | | | | | | | | | | | | | | | |
| 24 | 1s | 1s | 1s | 1s | 6s | 8.1m | | | | | | | | | | | | | | | | |
| 25 | 1s | 1s | 1s | 1s | 2s | 57s | 6.3h | | | | | | | | | | | | | | | |
| 26 | 1s | 1s | 1s | 1s | 1s | 13s | 22.1m | 12.12h | | | | | | | | | | | | | | |
| 27 | 1s | 1s | 1s | 1s | 1s | 4s | 2.4m | 20.7m | | | | | | | | | | | | | | |
| 28 | 1s | 1s | 1s | 1s | 1s | 2s | 28s | 53.6m | 27.58h | | | | | | | | | | | | | |
| 29 | 1s | 1s | 1s | 1s | 1s | 1s | 9s | 6.1m | 46.3m | | | | | | | | | | | | | |
| 30 | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 1m | 2.35h | 51.02h | | | | | | | | | | | | |
| 31 | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 18s | 15.7m | 1.67h | | | | | | | | | | | | |
| 32 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 6s | 2.6m | 7.1m | | | | | | | | | | | | |
| 33 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 37s | 38.1m | 3.52h | | | | | | | | | | | |
| 34 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 13s | 6.2m | 15.7m | | | | | | | | | | | |
| 35 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 5s | 1.2m | 1.48h | 6.69h | | | | | | | | | | |
| 36 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 24s | 13.3m | 33.3m | | | | | | | | | | |
| 37 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 9s | 2.3m | 4.1m | 12.63h | | | | | | | | | |
| 38 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 4s | 41s | 28.2m | 1.2h | | | | | | | | | |
| 39 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 16s | 4.8m | 9.8m | 24.84h | | | | | | | | |
| 40 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 6s | 1.2m | 1.17h | 2.64h | | | | | | | | |
| 41 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 28s | 11.5m | 21.6m | | | | | | | | |
| 42 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 12s | 2.3m | 3.5m | 5.72h | | | | | | | |
| 43 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 5s | 48s | 26.6m | 52.2m | | | | | | | |
| 44 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 21s | 4.8m | 9.1m | 12.42h | | | | | | |
| 45 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 8s | 1.4m | 1.5m | 2.13h | | | | | | |
| 46 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 4s | 34s | 10.4m | 21.9m | | | | | | |
| 47 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 15s | 2.3m | 3.5m | 4.81h | | | | | |
| 48 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 6s | 55s | 24.1m | 53.6m | | | | | |
| 49 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 3s | 25s | 4.4m | 9.1m | | | | | |
| 50 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 2s | 11s | 1.4m | 58.7m | 2.16h | | | | |
| L \ d | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |



The runtimes are averages over 5 random datasets. The times are given in hours (h) minutes (m) or seconds (s). Grey cells indicate instances that are expected to have more than 500 motifs by random chance (spurious motifs). Blue cells indicate that the program used 48 cores whereas white cells indicate single core execution. Instances in orange could not be solved efficiently.

Figure 1.7: qPMS9 runtimes on protein datasets for multiple combinations of $\ell$ and $d$ where $q = 100\%$.



| L \ d | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 1.1m | | | | | | | | | | | | | | | | | |
| 9 | 3s | 1.6h | | | | | | | | | | | | | | | | |
| 10 | 1s | 37s | | | | | | | | | | | | | | | | |
| 11 | 1s | 3s | 6.3m | | | | | | | | | | | | | | | |
| 12 | 1s | 1s | 24s | 9.0m | | | | | | | | | | | | | | |
| 13 | 1s | 1s | 3s | 1.9m | 24.8h | | | | | | | | | | | | | |
| 14 | 1s | 1s | 1s | 19s | 29.3m | | | | | | | | | | | | | |
| 15 | 1s | 1s | 1s | 3s | 1.1m | 34.9m | | | | | | | | | | | | |
| 16 | 1s | 1s | 1s | 2s | 13s | 6.4m | | | | | | | | | | | | |
| 17 | 1s | 1s | 1s | 1s | 4s | 44s | 3.4m | | | | | | | | | | | |
| 18 | 1s | 1s | 1s | 1s | 2s | 11s | 2.5m | 2.5h | | | | | | | | | | |
| 19 | 1s | 1s | 1s | 1s | 1s | 4s | 32s | 31.1m | | | | | | | | | | |
| 20 | 1s | 1s | 1s | 1s | 1s | 2s | 10s | 1.5m | 19.7m | | | | | | | | | |
| 21 | 1s | 1s | 1s | 1s | 1s | 2s | 5s | 21s | 7.8m | | | | | | | | | |
| 22 | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 10s | 1.1m | 4.7m | | | | | | | | |
| 23 | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 5s | 17s | 3.0m | 2.1h | | | | | | | |
| 24 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 3s | 10s | 48s | 1.3m | | | | | | | |
| 25 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 6s | 14s | 1.9m | 42.8m | | | | | | |
| 26 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 3s | 10s | 32s | 12.8m | 15.8h | | | | | |
| 27 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 6s | 12s | 1.4m | 12.3m | | | | | |
| 28 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 3s | 11s | 23s | 3.5m | 5.4h | | | | |
| 29 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 2s | 7s | 12s | 1.0m | 2.2m | | | | |
| 30 | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 2s | 4s | 11s | 18s | 2.2m | 1.6h | | | |
| L \ d | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |



The runtimes are averages over 5 random datasets. The times are given in hours (h) minutes (m) or seconds (s). Grey cells indicate instances that are expected to have more than 500 motifs by random chance (spurious motifs). Blue cells indicate that the program used 48 cores whereas white cells indicate single core execution. Instances in orange could not be solved efficiently.

Table 1.4: Maximum value of $d$ such that the expected number of spurious motifs in random datasets does not exceed 500, for $\ell$ up to 50 and $q$ between 50% and 100%, on DNA data.

| | max d | | |
|---|---|---|---|
| L | $q = 50\%$ | $q = 75\%$ | $q = 100\%$ |
| 13 | 3 | 3 | 4 |
| 14 | 3 | 4 | 4 |
| 15 | 4 | 4 | 5 |
| 16 | 4 | 5 | 5 |
| 17 | 4 | 5 | 6 |
| 18 | 5 | 6 | 6 |
| 19 | 5 | 6 | 7 |
| 20 | 6 | 7 | 7 |
| 21 | 6 | 7 | 8 |
| 22 | 7 | 8 | 8 |
| 23 | 7 | 8 | 9 |
| 24 | 8 | 9 | 9 |
| 25 | 8 | 9 | 10 |
| 26 | 9 | 10 | 11 |
| 27 | 9 | 10 | 11 |
| 28 | 10 | 11 | 12 |
| 29 | 10 | 11 | 12 |
| 30 | 11 | 12 | 13 |
| 31 | 11 | 12 | 13 |
| 32 | 12 | 13 | 14 |
| 33 | 12 | 13 | 14 |
| 34 | 13 | 14 | 15 |
| 35 | 13 | 15 | 16 |
| 36 | 14 | 15 | 16 |
| 37 | 14 | 16 | 17 |
| 38 | 15 | 16 | 17 |
| 39 | 15 | 17 | 18 |
| 40 | 16 | 17 | 18 |
| 41 | 16 | 18 | 19 |
| 42 | 17 | 18 | 20 |
| 43 | 17 | 19 | 20 |
| 44 | 18 | 19 | 21 |
| 45 | 18 | 20 | 21 |
| 46 | 19 | 21 | 22 |
| 47 | 19 | 21 | 22 |
| 48 | 20 | 22 | 23 |
| 49 | 20 | 22 | 24 |
| 50 | 21 | 23 | 24 |

Table 1.5: Maximum value of $d$ such that the expected number of spurious motifs in random datasets does not exceed 500, for $\ell$ up to 30 and $q$ between 50% and 100%, on protein data.

| | max d | | |
|---|---|---|---|
| L | $q = 50\%$ | $q = 75\%$ | $q = 100\%$ |
| 9 | 4 | 4 | 5 |
| 10 | 4 | 5 | 5 |
| 11 | 5 | 6 | 6 |
| 12 | 6 | 6 | 7 |
| 13 | 6 | 7 | 8 |
| 14 | 7 | 8 | 8 |
| 15 | 8 | 9 | 9 |
| 16 | 9 | 9 | 10 |
| 17 | 9 | 10 | 11 |
| 18 | 10 | 11 | 11 |
| 19 | 11 | 12 | 12 |
| 20 | 11 | 12 | 13 |
| 21 | 12 | 13 | 14 |
| 22 | 13 | 14 | 15 |
| 23 | 14 | 15 | 15 |
| 24 | 14 | 15 | 16 |
| 25 | 15 | 16 | 17 |
| 26 | 16 | 17 | 18 |
| 27 | 16 | 18 | 19 |
| 28 | 17 | 18 | 19 |
| 29 | 18 | 19 | 20 |
| 30 | 19 | 20 | 21 |

Table 1.6: PMS runtimes for DNA data when $q = 100\%$.

| $(\ell, d)$ | TraverStringRef | PMS8 | qPMS9 |
|---|---|---|---|
| (13,4) | 14s | 7s | **6s** |
| (15,5) | 55s | 48s | **34s** |
| (17,6) | 3.5m | 5.2m | **2.7m** |
| (19,7) | 14.5m | 26.6m | **13.4m** |
| (21,8) | 59.8m | 1.64h | **45.4m** |
| (23,9) | 4.08h | 5.48h | **2.26h** |
| (25,10) | 17.55h | 15.45h | **6.3h** |

The time is given in hours (h), minutes (m) or seconds (s), averaged over 5 datasets.

Table 1.7: PMS runtimes for protein data when $q = 100\%$.

| $(\ell, d)$ | TraverStringRef | PMS8 | qPMS9 |
|---|---|---|---|
| (10,5) | 2.6m | 42s | **37s** |
| (11,6) | 1.67h | 11m | **6.1m** |
| (13,7) | 58.2m | 2.6m | **19s** |
| (14,8) | TL | 1.03h | **29.6m** |
| (15,8) | 28.5m | 1.2m | **1.1m** |
| (17,9) | 16.6m | 45s | **43s** |
| (19,10) | 5.9m | **32s** | **32s** |
| (19,11) | TL | 1.23h | **30.1m** |
| (22,12) | 3.73h | 1.2m | **1.1m** |
| (24,13) | 1.84h | 48s | **47s** |
| (26,14) | 30.7m | **31s** | 32s |
| (26,15) | TL | 1.19h | **12.5m** |

The time is given in hours (h), minutes (m) or seconds (s), averaged over 5 datasets. TL means that the program runs for more than 24h.

Table 1.8: PMS runtimes for DNA data when $q = 50\%$.

| Instance | TraverStringRef | qPMS9 |
|---|---|---|
| (20,6) | 3m | **1.5m** |
| (22,7) | 12.9m | **6.3m** |
| (23,7) | 2.6m | **48s** |
| (24,8) | 56m | **26.3m** |
| (25,8) | 9.9m | **3.1m** |
| (26,9) | 4.31h | **1.55h** |
| (27,9) | 39.9m | **10.6m** |
| (28,10) | 20.86h | **5.15h** |
| (29,10) | 2.89h | **34.5m** |

The time is given in hours (h), minutes (m) or seconds (s), averaged over 5 datasets.

Table 1.9: PMS runtimes for protein data when $q = 50\%$.

| Instance | TraverStringRef | qPMS9 |
|---|---|---|
| (9,4) | 11.3m | **3.7m** |
| (11,5) | 14m | **4.1m** |
| (12,6) | 6.22h | **57.5m** |
| (13,6) | 17.4m | **4.9m** |
| (14,7) | 5.09h | **41.3m** |
| (15,8) | TL | **4.62h** |
| (17,9) | TL | **1.79h** |
| (18,9) | 2.71h | **33.1m** |
| (20,10) | 2.33h | **33.3m** |
| (21,11) | TL | **50.9m** |

The time is given in hours (h), minutes (m) or seconds (s), averaged over 5 datasets. TL means that the program runs for more than 24h.

## 1.4 Discussion

We have presented PMS8 and its successor qPMS9, which are efficient algorithms for (Quorum) Planted Motif Search. PMS8 makes use of novel pruning techniques for generating motif candidates. qPMS9 adds a new procedure for exploring the search space and adds support for the quorum version of PMS. qPMS9 is the first algorithm to solve the challenging DNA instances $(28, 12)$ and $(30, 13)$. qPMS9 can also efficiently solve instances with larger $\ell$ and $d$ such as $(50, 21)$ for DNA data or $(30, 18)$ for protein data.

# Chapter 2

# Suffix Array Construction

## 2.1 Introduction

The suffix array is a data structure that finds numerous applications in string processing problems for both linguistic texts and biological data. It has been introduced in [58] as a memory efficient alternative to suffix trees. The suffix array of a string $T$ is an array $A$, ($|T| = |A| = n$) which gives the lexicographic order of all the suffixes of $T$. Thus, $A[i]$ is the starting position of the lexicographically $i$-th smallest suffix of $T$.

The original suffix array construction algorithm [58] runs in $O(n \log n)$ time. It is based on a technique called *prefix doubling:* assume that the suffixes are grouped into buckets such that suffixes in the same bucket share the same prefix of length $k$. Let $b_i$ be the bucket number for suffix $i$. Let $q_i = (b_i, b_{i+k})$. Sort the suffixes with respect to $q_i$ using radix sort. As a result, the suffixes become sorted by their first $2k$ characters. Update the bucket numbers and repeat the process until all the suffixes are in buckets of size 1. This process takes no more than $\log n$ rounds. The idea of sorting suffixes in one bucket based on the bucket information of nearby suffixes is called *induced copying.* It appears in some form or another in many of the algorithms for suffix array construction.

Numerous papers have been written on suffix arrays. A survey on some of these algorithms can be found in [73]. The authors of [73] categorize suffix array construction algorithms (SACA) into five based on the main techniques employed: 1) Prefix Doubling (examples include [58] - run time = $O(n \log n)$; [56] - run time = $O(n \log n)$); 2) Recursive (examples include [48] - run time = $O(n \log \log n)$); 3) Induced Copying (examples include [8] - run time = $O(n\sqrt{\log n})$); 4) Hybrid (examples include [42] and [51] - run time = $O(n^2 \log n)$); and 5) Suffix Tree (examples include [52] - run time = $O(n \log \sigma)$ where $\sigma$ is the size

of the alphabet).

In 2003, three independent groups [51, 45, 49] found the first linear time suffix array construction algorithms which do not require building a suffix tree beforehand. For example, in [51] the suffixes are classified as either $L$ or $S$. Suffix $i$ is an $L$ suffix if it is lexicographically larger than suffix $i+1$, otherwise it is an $S$ suffix. Assume that the number of $L$ suffixes is less than $n/2$, if not, do this for $S$ suffixes. Create a new string where the segments of text in between $L$ suffixes are renamed to single characters. The new text has length no more than $n/2$ and we recursively find its suffix array. This suffix array gives the order of the $L$ suffixes in the original string. This order is used to induce the order of the remaining suffixes.

Another linear time algorithm, called *skew*, is given in [45]. It first sorts those suffixes $i$ with $i \bmod 3 \neq 0$ using a recursive procedure. The order of these suffixes is then used to infer the order of the suffixes with $i \bmod 3 = 0$. Once these two groups are determined we can compare one suffix from the first group with one from the second group in constant time. The last step is to merge the two sorted groups, in linear time.

Several other SACAs have been proposed in the literature in recent years (e.g., [67, 81]). Some of the algorithms with superlinear worst case run times perform better in practice than the linear ones. One of the currently best performing algorithms in practice is the $BPR$ algorithm of [81] which has an asymptotic worst-case run time of $O(n^2)$. $BPR$ first sorts all the suffixes up to a certain depth, then focuses on one bucket at a time and repeatedly refines it into sub-buckets.

In this chapter we present an elegant algorithm for suffix array construction. This algorithm takes linear time with high probability. Here the probability is on the space of all possible inputs. Our algorithm is one of the simplest algorithms known for constructing suffix arrays. It opens up a new dimension in suffix array construction, i.e., the development of algorithms with provable expected run times. This dimension has not been explored before. We prove a lemma on the $\ell$-mers of a random string which might find independent applications. Our algorithm is also nicely parallelizable. We offer parallel implementations of our algorithm on various parallel models of computing.

We also present another algorithm for suffix array construction that utilizes the above algorithm. This algorithm, called RadixSA, is based on bucket sorting and has a worst case run time of $O(n \log n)$. It employs an idea which, to the best of our knowledge, has not been directly exploited until now. RadixSA selects the order in which buckets are processed based on a heuristic such that, downstream, they impact as many other buckets as possible. This idea may find independent application as a standalone speedup technique for other SACAs based on bucket sorting. RadixSA also employs a generalization of Seward's

copy method [82] (initially described in [12]) to detect and handle repeats of any length (section 2.2.5). We compare RadixSA with other algorithms on various datasets.

## 2.2 Methods

### 2.2.1 A Useful Lemma

Let $\Sigma$ be an alphabet of interest and let $S = s_1 s_2 \dots s_n \in \Sigma^*$. Consider the case when $S$ is generated randomly, i.e., each $s_i$ is picked uniformly randomly from $\Sigma$ ($1 \leq i \leq n$). Let $L$ be the set of all $\ell$-mers of $S$. Note that $|L| = n - \ell + 1$. What can we say about the independence of these $\ell$-mers? In several papers analyses have been done assuming that these $\ell$-mers are independent (see e.g., [10]). These authors point out that this assumption may not be true but these analyses have proven to be useful in practice. In this Section we prove the following Lemma on these $\ell$-mers.

**Lemma 4.** *Let $L$ be the set of all $\ell$-mers of a random string generated from an alphabet $\Sigma$. Then, the $\ell$-mers in $L$ are pairwise independent. These $\ell$-mers need not be $k$-way independent for $k \geq 3$.*

*Proof.* Let $A$ and $B$ be any two $\ell$-mers in $L$. If $x$ and $y$ are non-overlapping, clearly, $Prob[A = B] = (1/\sigma)^\ell$, where $\sigma = |\Sigma|$. Thus, consider the case when $x$ and $y$ are overlapping.

Let $P_i = s_i s_{i+1} \dots s_{i+\ell-1}$, for $1 \leq i \leq (n-\ell+1)$. Let $A = P_i$ and $B = P_j$ with $i < j$ and $j \leq (i+\ell-1)$. Also let $j = i + k$ where $1 \leq k \leq (\ell - 1)$.

Consider the special case when $k$ divides $\ell$. If $A = B$, then it should be the case that $s_i = s_{i+k} = s_{i+2k} = \dots = s_{i+\ell}$; $s_{i+1} = s_{i+k+1} = s_{i+2k+1} = \dots = s_{i+\ell+1}$; $\dots$; and $s_{i+k-1} = s_{i+2k-1} = s_{i+3k-1} = \dots = s_{i+\ell+k-1}$. In other words, we have $k$ series of equalities. Each series is of length $(\ell/k) + 1$. The probability of all of these equalities is $\left(\frac{1}{\sigma}\right)^{\ell/k} \left(\frac{1}{\sigma}\right)^{\ell/k} \cdots \left(\frac{1}{\sigma}\right)^{\ell/k} = \left(\frac{1}{\sigma}\right)^{\ell}$.

As an example, let $S = abcdefghi$, $\ell = 4$, $k = 2$, $A = P_1$, and $B = P_3$. In this case, the following equalities should hold: $a = c = e$ and $b = d = f$. The probability of all of these equalities is $(1/\sigma)^2(1/\sigma)^2 = (1/\sigma)^4 = (1/\sigma)^\ell$.

Now consider the general case (where $k$ may not divide $\ell$). Let $\ell = qk + r$ for some integers $q$ and $r$ where $r < k$. If $A = B$, the following equalities will hold: $s_i = s_{i+k} = s_{i+2k} = \dots = s_{i+\lfloor(\ell+k-1)/k\rfloor k}$; $s_{i+1} = s_{i+1+k} = s_{i+1+2k} = \dots = s_{i+1+\lfloor(\ell+k-2)/k\rfloor k}$; $\dots$; and $s_{i+k-1} = s_{i+k-1+k} = s_{i+k-1+2k} = \dots = s_{i+k-1+\lfloor(\ell/k)\rfloor k}$.

Here again we have $k$ series of equalities. The number of elements in the $q$th series is $1 + \left\lfloor \frac{\ell+k-q}{k} \right\rfloor$,

for $1 \leq q \leq k$. The probability of all of these equalities is $(1/\sigma)^x$ where $x = \sum_{q=1}^{k} \left\lfloor \frac{\ell+k-q}{k} \right\rfloor$.

$$x = \left\lfloor \frac{(q+1)k + r - 1}{k} \right\rfloor + \left\lfloor \frac{(q+1)k + r - 2}{k} \right\rfloor + \cdots + \left\lfloor \frac{(q+1)k}{k} \right\rfloor$$

$$+ \left\lfloor \frac{(q+1)k - 1}{k} \right\rfloor + \left\lfloor \frac{(q+1)k - 2}{k} \right\rfloor + \cdots + \left\lfloor \frac{(q+1)k - (k-r)}{k} \right\rfloor$$

$$= (q+1)r + (k-r)q = kq + r = \ell.$$

The fact that the $\ell$-mers of $L$ may not be $k$-way independent for $k \geq 3$ is easy to see. For example, let $S = abcdefgh$, $\ell = 3$, $A = P_1$, $B = P_3$, and $C = P_4$. What is $Prob.[A = B = C]$? If $A = B = C$, then it should be the case that $a = c, b = d = a, b = c = e$, and $c = f$. In other words, $a = b = c = d = e = f$. The probability of this happening is $(1/\sigma)^5 \neq (1/\sigma)^6$. □

**Note:** To the best of our knowledge, the above lemma cannot be found in the existing literature. In [85] a lemma is proven on the expected depth of insertion of a suffix tree. If anything, this only very remotely resembles our lemma but is not directly related. In addition the lemma in [85] is proven only in the limit (when $n$ tends to $\infty$).

### 2.2.2 Our Basic Algorithm

Let $S = s_1 s_2 \cdots s_n$ be the given input string. Assume that $S$ is a string randomly generated from an alphabet $\Sigma$. In particular, each $s_i$ is assumed to have been picked uniformly randomly from $\Sigma$ (for $1 \leq i \leq n$). For all the algorithms presented in this chapter, no assumption is made on the size of $\Sigma$. In particular, it could be anything. For example, it could be $O(1)$, $O(n^c)$ (for any constant $c$), or larger.

The problem is to produce an array $A[1:n]$ where $A[i]$ is the starting position of the $i$th smallest suffix of $S$, for $1 \leq i \leq n$. The basic idea behind our algorithm is to sort the suffixes only with respect to their prefixes of length $O(\log n)$ (bits). The claim is that this amount of sorting is enough to order the suffixes with *high probability*. By high probability we mean a probability of $\geq (1 - n^{-\alpha})$ where $\alpha$ is the probability parameter (typically assumed to be a constant $\geq 1$). The probability space under concern is the space of all possible inputs.

Let $S_i$ stand for the suffix that starts at position $i$, for $1 \leq i \leq n$. In other words, $S_i = s_i s_{i+1} \cdots s_n$. Let $P_i = s_i s_{i+1} \cdots s_{i+\ell-1}$, for $i \leq (n - \ell)$. When $i > (n - \ell)$, let $P_i = S_i$. The value of $\ell$ will be decided

in the analysis. A pseudocode of our basic algorithm follows.

**Algorithm 4:** SA1

Sort $P_1, P_2, \ldots, P_n$ using radix sort;

The above sorting partitions the $P_i$'s into buckets where equal $\ell$-mers are in the same bucket;

Let these buckets be $B_1, B_2, \ldots, B_m$ where $m \leq n$;

**for** $i := 1$ **to** $m$ **do**

  **if** $|B_i| > 1$ **then**

    sort the suffixes corresponding to the $\ell$-mers in $B_i$ using any relevant algorithm;

**Lemma 5.** *Algorithm SA1 has a run time of $O(n)$ with high probability.*

*Proof.* Consider a specific $P_i$ and let $B$ be the bucket that $P_i$ belongs to after the radix sorting step in Algorithm SA1. How many other $P_j$'s will there be in $B$? Using Lemma 4, $Prob.[P_i = P_j] = (1/\sigma)^\ell$. This means that $Prob.[\exists j : i \neq j \& P_i = P_j] \leq n(1/\sigma)^\ell$. As a result, $Prob.[\exists j : |B_j| > 1] \leq n^2(1/\sigma)^\ell$. If $\ell \geq ((\alpha + 2)\log_\sigma n)$, then, $n^2(1/\sigma)^\ell \leq n^{-\alpha}$.

In other words, if $\ell \geq ((\alpha + 2)\log_\sigma n)$, then each bucket will be of size 1 with high probability. Also, the radix sort will take $O(n)$ time. Note that we only need to sort $O(\log n)$ bits of each $P_i$ $(1 \leq i \leq n)$ and this sorting can be done in $O(n)$ time (see e.g., [40]). □

**Observation 1.** We could have a variant of the algorithm where if any of the buckets is of size greater than 1, we abort this algorithm and use another algorithm. A pseudocode follows.

**Algorithm 5:** SA2

1. Sort $P_1, P_2, \ldots, P_n$ using radix sort;

   The above sorting partitions the $P_i$'s into buckets where equal $\ell$-mers are in the same bucket;

   Let these buckets be $B_1, B_2, \ldots, B_m$ where $m \leq n$;
2. **if** $|B_i| = 1$ **for each** $i$,$1 \leq i \leq m$ **then**
3.  output the suffix array and quit;

   **else**
4.  use another algorithm (let it be **Algorithm SA**) to find and output the suffix array;

**Observation 2.** Algorithm SA1 as well as Algorithm SA2 run in $O(n)$ time on at least $(1 - n^{-\alpha})$ fraction of all possible inputs. Also, if the run time of Algorithm SA is $t(n)$, then the expected run time of Algorithm SA2 is $(1 - n^{-\alpha})O(n) + n^{-\alpha}(O(n) + t(n))$. For example, if Algorithm SA is the skew algorithm [45], then the expected run time of Algorithm SA2 is $O(n)$ (the underlying constant will be smaller than the constant in the run time of skew).

**Observation 3.** In general, if $T(n)$ is the run time of Algorithm SA2 lines 1 through 3 and if $t(n)$ is the run time of Algorithm SA, then the expected run time of Algorithm SA2 is $(1-n^{-\alpha})T(n)+n^{-\alpha}(T(n)+t(n))$.

**The case of non-uniform probabilities.** In the above algorithm and analysis we have assumed that each character in $S$ is picked uniformly randomly from $\Sigma$. Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$. Now we consider the possibility that for any $s_i \in S$, $Prob.[s_i = a_j] = p_j$, $1 \leq i \leq n; 1 \leq j \leq \sigma$. For any two $\ell$-mers $A$ and $B$ of $S$ we can show that $Prob.[A = B] = \left(\sum_{j=1}^{\sigma} p_j^2\right)^{\ell}$. In this case, we can employ Algorithms SA1 and SA2 with $\ell \geq (\alpha+2)\log_{1/P} n$, where $P = \sum_{j=1}^{\sigma} p_j^2$.

**Observation 4.** Both SA1 and SA2 can work with any alphabet size. If the size of the alphabet is $O(1)$, then each $P_i$ will consist of $O(\log n)$ characters from $\Sigma$. If $|\Sigma| = \Theta(n^c)$ for some constant $c$, then $P_i$ will consist of $O(1)$ characters from $\Sigma$. If $|\Sigma| = \omega(n^c)$ for any constant $c$, then each $P_i$ will consist of a prefix (of length $O(\log n)$ bits) of a character in $\Sigma$.

### 2.2.3 Parallel Versions

In this Section we explore the possibility of implementing SA1 and SA2 on various models of parallel computing.

#### Parallel Disks Model

In a Parallel Disks Model (PDM), there is a (sequential or parallel) computer whose core memory is of size $M$. The computer has $D$ parallel disks. In one parallel I/O, a block of size $B$ from each of the $D$ disks can be fetched into the core memory. The challenge is to devise algorithms for this model that perform the least number of I/O operations. This model has been proposed to alleviate the I/O bottleneck that is common for single disk machines especially when the dataset is large. In the analysis of PDM algorithms the focus is on the number of parallel I/Os and typically the local computation times are not considered. A lower bound on the number of parallel I/Os needed to sort $N$ elements on a PDM is $\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)}$. Numerous asymptotically optimal parallel algorithms have been devised for sorting on the PDM. For practical values of $N, M, D$, and $B$, the lower bound basically means a constant number of passes through the data. Therefore, it is imperative to design algorithms wherein the underlying constants in the number of I/Os is small. A number of algorithms for different values of $N, M, D$, and $B$ that take a small number of passes have been proposed in [74].

One of the algorithms given in [74] is for sorting integers. In particular it is shown that we can sort $N$ random integers in the range $[1, R]$ (for any $R$) in $(1+\nu)\frac{\log(N/M)}{\log(M/B)}+1$ passes through the data, where

$\nu$ is a constant $< 1$. This bound holds with probability $\geq (1 - N^{-\alpha})$, this probability being computed in the space of all possible inputs.

We can adapt the algorithm of [74] for constructing suffix arrays as follows. We assume that the word length of the machine is $O(\log n)$. This is a standard assumption made in the algorithms literature. Note that if the length of the input string is $n$, then we need a word length of at least $\log n$ to address the suffixes. To begin with, the input is stored in the $D$ disks striped uniformly. We generate all the $\ell$-mers of $S$ in one pass through the input. Note that each $\ell$-mer occupies one word of the machine. The generated $\ell$-mers are stored back into the disks. Followed by this, these $\ell$-mers are sorted using the algorithm of [74]. At the end of this sorting, we have $m$ buckets where each bucket has equal $\ell$-mers. As was shown before, each bucket is of size 1 with high probability.

We get the following:

**Theorem 2.** *We can construct the suffix array for a random string of length $n$ in $(1 + \nu)\frac{\log(n/M)}{\log(M/B)} + 2$ passes through the data, where $\nu$ is a constant $< 1$. This bound holds for $\geq (1 - n^{-\alpha})$ fraction of all possible inputs.* □

**The Mesh and the Hypercube**

Optimal algorithms exist for sorting on interconnection networks such as the mesh (see e.g., [87] and [43]), the hypercube (see e.g., [78]), etc. We can use these in conjunction with Algorithms SA1 and SA2 to develop suffix array construction algorithms for these models. Here again we can construct all the $\ell$-mers of the input string. Assume that we have an interconnection network with $n$ nodes and each node stores one of the characters in the input string. In particular node $i$ stores $s_i$, for $1 \leq i \leq n$. Depending on the network, a relevant indexing scheme has to be used. For instance, on the mesh we can use a snake-like row-major indexing. Node $i$ communicates with nodes $i+1, i+2, \ldots, i+\ell-1$ to get $s_{i+1}, s_{i+2}, \ldots, s_{i+\ell-1}$. The communication time needed is $O(\log n)$. Once the node $i$ has these characters it forms $P_i$. Once the nodes have generated the $\ell$-mers, the rest of the algorithm is similar to the Algorithm SA1 or SA2. As a result, we get the following:

**Theorem 3.** *There exists a randomized algorithm for constructing the suffix array for a random string of length $n$ in $O(\log n)$ time on a $n$-node hypercube with high probability. The run time of [78]'s algorithm is $O(\log n)$ with high probability, the probability being computed in the space of all possible outcomes for the coin flips made. Also, the same can be done in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh with high probability.* □

**Observation:** Please note that on a $n$-node hypercube, sorting $n$ elements will need $\Omega(\log n)$ time even if these elements are bits, since the diameter of the hypercube is $\Omega(\log n)$. For the same reason, sorting $n$ elements on a $\sqrt{n} \times \sqrt{n}$ mesh will need $\Omega(\sqrt{n})$ time since $2(\sqrt{n} - 1)$ is the diameter.

**PRAM Algorithms**

In [45] several PRAM algorithms are given. One such algorithm is for the EREW PRAM that has a run time of $O(\log^2 n)$, the work done being $O(n \log n)$. We can implement Algorithm SA2 on the EREW PRAM so that it has an expected run time of $O(\log n)$, the expected work done being $O(n \log n)$. Details follow. Assume that we have $n$ processors. 1) Form all possible $\ell$-mers. Each $\ell$-mer occupies one word; 2) Sort these $\ell$-mers using the parallel merge sort algorithm of [22]; 3) Using a prefix computation check if there is at least one bucket of size $> 1$; 4) Broadcast the result to all the processors using a prefix computation; 5) If there is at least one bucket of size more than one, use the parallel algorithm of [45].

Steps 1 through 4 of the above algorithm take $O(\log n)$ time each. Step 5 takes $O(\log^2 n)$ time. From Observation 3, the expected run time of this algorithm is $(1-n^{-\alpha})O(\log n)+n^{-\alpha}(O(\log n)+O(\log^2 n)) = O(\log n)$. Also, the expected work done by the algorithm is $(1 - n^{-\alpha})O(n \log n) + n^{-\alpha}(O(n \log n) + O(n \log^2 n)) = O(n \log n)$.

### 2.2.4 Practical Implementation

In algorithm SA2, if some of the buckets are of size greater than 1, we employ further processing to complete the sorting. This gives the RadixSA algorithm. The pseudocode for RadixSA is the following:

**Algorithm 6:** RadixSA

```
1. radixSort all suffixes with respect to their first d characters;
2. let b[i] = bucket of suffix i ;
3. for i := n down to 1 do
4.     if b[i].size > 1 then
5.         if detectPeriods(b[i]) then
6.             handlePeriods(b[i]);
           else
7.             radixSort all suffixes j ∈ b[i] with respect to b[j + d];
```

A bucket is called *singleton* if it contains only one suffix, otherwise it is called *non-singleton*. A *singleton suffix* is the only suffix in a singleton bucket. A singleton suffix has its final position in the

suffix array already determined.

We number the buckets such that two suffixes in different buckets can be compared simply by comparing their bucket numbers. The **for** loop traverses the suffixes from the last to the first position in the text. This order ensures that after each step, suffix $i$ will be found in a singleton bucket. This is easy to prove by induction. Thus, at the end of the loop, all the buckets will be singletons. If each bucket is of size $O(1)$ before the **for** loop is entered, then it is easy to see that the algorithm runs in $O(n)$ time.

Second, even if the buckets are not of constant size (before the **for** loop is entered) the algorithm is still linear if every suffix takes part in no more than a constant number of radix sort operations. For that to happen, we want to give priority to buckets which can influence as many downstream buckets as possible. Intuitively, say a pattern $P$ appears several times in the input. We want to first sort the buckets which contain the suffixes that start at the tails of the pattern instances. Once these suffixes are placed in different buckets we progress towards the buckets containing the heads of the pattern instances. This way, every suffix is placed in a singleton bucket at a constant cost per suffix. Our traversal order gives a good approximation of this behavior in practice, as we show in the results section.

Table 2.1 shows an example of how the algorithm works. Each column illustrates the state of the suffix array after sorting one of the buckets. The order in which buckets are chosen to be sorted follows the pseudocode of RadixSA. The initial radix sort has depth 1 for illustration purpose. The last column contains the fully sorted suffix array.

However, the algorithm as is described above has a worst case runtime of $O(n\sqrt{n})$ (proof omitted). We can improve the runtime to $O(n \log n)$ as follows. If, during the for loop, a bucket contains suffixes which have been accessed more than a constant $C$ number of times, we skip that bucket. This ensures that the for loop takes linear time. If at the end of the loop there have been any buckets skipped, we do another pass of the for loop. After each pass, every remaining non-singleton bucket has a sorting depth at least $C+1$ times greater than in the previous round (easy to prove by induction). Thus, no more than a logarithmic number of passes will be needed and so the algorithm has worst case runtime $O(n \log n)$.

### 2.2.5 Periodic Regions

In lines 5 and 6 of the RadixSA pseudocode we detect periodic regions of the input as follows: if suffixes $i, i-p, i-2p, \ldots$ appear in the same bucket $b$, and bucket $b$ is currently sorted by $d \geq p$ characters, then we have found a periodic region of the input, where the period is $p$. If suffix $i$ is less than suffix $i+p$, then suffix $i-p$ is less than $i$, $i-2p$ is less than $i-p$, and so on. The case where $i$ is greater than $i+p$ is analogous. Periods of any length are eventually detected because the depth of sorting in each

Table 2.1: Example of RadixSA suffix array construction steps.

| Initial buckets | Sort b[a] | Sort b[ca] | Sort b[dayca] | Sort b[cdayca] |
|---|---|---|---|---|
| a | a | a | a | a |
| ayca | axcdayca | axcdayca | axcdayca | axcdayca |
| axcdayca | ayca | ayca | ayca | ayca |
| ca | ca | ca | ca | ca |
| cdayca | cdayca | cdayca | cdayca | cdaxcdayca |
| cdaxcdayca | cdaxcdayca | cdaxcdayca | cdaxcdayca | cdayca |
| dayca | dayca | dayca | daxcdayca | daxcdayca |
| daxcdayca | daxcdayca | daxcdayca | dayca | dayca |
| xcdayca | xcdayca | xcdayca | xcdayca | xcdayca |
| yca | yca | yca | yca | yca |

Example of suffix array construction steps for string 'cdaxcdayca'. b[*suffix*] stands for the bucket of *suffix*. Underlines show the depth of sorting in a bucket at a given time. The initial radix sort has depth 1 for illustration purpose.

bucket increases after each sort operation. This method can be viewed as a generalization of Seward's copy method [82] where a portion of text of size $p$ is treated as a single character.

### 2.2.6 Implementation Details

Radix sorting is a central operation in RadixSA. We tried several implementations, both with Least Significant Digit (LSD) and Most Significant Digit (MSD) first order. The best of our implementations was a cache-optimized LSD radix sort. The cache optimization is the following. In a regular LSD radix sort, for every digit we do two passes through the data: one to compute bucket sizes, one to assign items to buckets. We can save one pass through the data per digit if in the bucket assignment pass we also compute bucket sizes for the next round [53]. We took this idea one step forward and we computed bucket counts for all rounds before doing any assignment. Since in our program we only sort numbers of at most 64 bits, we have a constant number of bucket size arrays to store in memory. To sort small buckets we employ an iterative merge sort.

To further improve cache performance, in the bucket array we store not only the bucket start position but also a few bits indicating the length of the bucket. Since the bucket start requires $\lceil \log n \rceil$ bits, we use the remaining bits, up to the machine word size, to store the bucket length. This prevents a lot of cache misses when small buckets are the majority. For longer buckets, we store the lengths in a separate array which also stores bucket depths.

The total additional memory used by the algorithm, besides input and output, is $5n + o(n)$ bytes: $4n$ for the bucket array, $n$ bytes for bucket depths and lengths, and a temporary buffer for radix sort.

## 2.3 Experimental Results

One of the fastest SACAs, in practice, is the Bucket Pointer Refinement (BPR) algorithm [81]. Version 0.9 of BPR has been compared [81] with several other algorithms: deep shallow [59], cache and copy by Seward [82], qsufsort [57], difference-cover [11], divide and conquer by Kim et al. [48], and skew [45]. BPR 0.9 has been shown to outperform these algorithms on most inputs [81]. Version 2.0 of BPR further improves over version 0.9. We compare RadixSA with both versions of BPR.

Furthermore, a large set of SACAs are collected in the jSuffixArrays library [68] under a unified interface. This library contains Java implementations of: DivSufSort [60] , QsufSort [57], SAIS [67], skew [45] and DeepShallow [59]. We include them in the comparison with the note that these Java algorithms may incur a performance penalty compared to their C counterparts.

We tested all algorithms on an Intel core $i3$ machine with $4GB$ of RAM, Ubuntu 11.10 Operating System, Sun Java 1.6.0_26 virtual machine and gcc 4.6.1. The Java Virtual Machine was allowed to use up to $3.5GB$ of memory. As inputs, we used the datasets of [81] which include DNA data, protein data, English alphabet data, general ASCII alphabet data and artificially created strings such as periodic and Fibonacci strings[1].

For every dataset, we executed each algorithm 10 times. The average run times are reported in table 2.2 where the best run times are shown in bold. Furthermore, we counted the number of times RadixSA accesses each suffix. The access counts are shown in figure 2.1. For almost all datasets, the number of times each suffix is accessed is a small constant. For the Fibonacci string the number of accesses is roughly logarithmic in the length of the input.

## 2.4 Discussion and Conclusions

In this chapter we have presented an elegant algorithm for the construction of suffix arrays. This algorithm is one of the simplest algorithms known for suffix arrays construction and runs in $O(n)$ time on a large fraction of all possible inputs. It is also nicely parallelizable. We have shown how our algorithm can be implemented on various parallel models of computing.

We have also given an extension of this algorithm, called RadixSA, which has a worst case runtime of $O(n \log n)$ and proved to be efficient in practice. RadixSA uses a heuristic to select the order in which buckets are processed so as to reduce the number of operations performed. RadixSA performed a linear

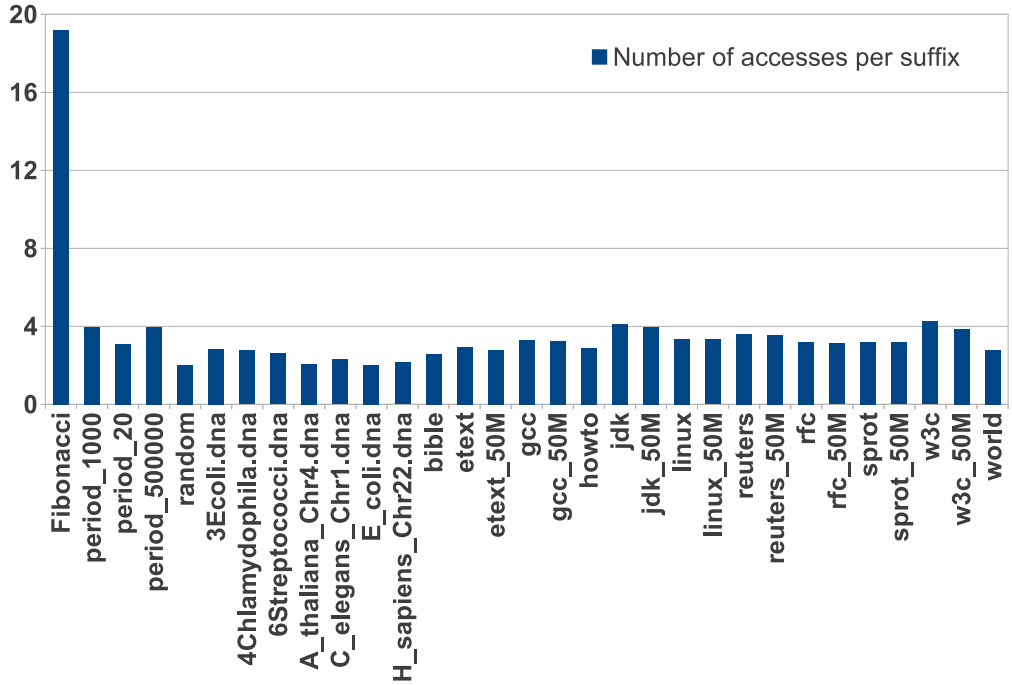
[1]Fibonacci strings are similar to Fibonacci numbers, but addition is replaced with concatenation ($F_0 = b, F_1 = a, F_i$ is a concatenation of $F_{i-1}$ and $F_{i-2}$).

Table 2.2: Comparison of suffix array construction algorithms run times.

| Dataset | | | Run time | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | Length | $\|\Sigma\|$ | RadixSA | BPR2 | BPR.9 | DivSuf Sort | QSuf Sort | SAIS | skew | Deep Shallow |
| Fibonacci | 20000000 | 2 | 7.88 | 12.48 | 14.05 | 6.81 | 26.44 | **5.50** | 14.53 | 369.48 |
| period_1000 | 20000000 | 26 | **2.12** | 3.52 | 5.71 | 3.15 | 20.42 | 6.59 | 23.27 | TL |
| period_20 | 20000000 | 17 | **1.44** | 1.95 | 43.39 | 1.83 | 11.05 | 2.83 | 7.15 | TL |
| period_500000 | 20000000 | 26 | **2.78** | 4.60 | 6.31 | 4.74 | 23.32 | 8.56 | 25.68 | 2844.37 |
| random | 20000000 | 26 | **2.25** | 3.34 | 4.87 | 6.35 | 5.02 | 11.75 | 22.05 | 5.69 |
| 3Ecoli.dna | 14776363 | 5 | **2.23** | 2.67 | 3.43 | 4.00 | 13.85 | 6.14 | 19.62 | 433.54 |
| 4Chlamydophila.dna | 4856123 | 6 | **0.61** | 0.67 | 0.90 | 1.71 | 3.24 | 1.93 | 5.24 | 4.80 |
| 6Streptococci.dna | 11635882 | 5 | **1.63** | 1.79 | 2.38 | 2.88 | 7.08 | 4.98 | 14.88 | 4.26 |
| A_thaliana_Chr4.dna | 12061490 | 7 | **1.27** | 1.74 | 2.40 | 3.02 | 5.13 | 5.37 | 15.71 | 3.52 |
| C_elegans_Chr1.dna | 14188020 | 5 | **1.61** | 1.95 | 2.65 | 3.21 | 6.91 | 5.69 | 17.18 | 6.92 |
| E_coli.dna | 4638690 | 4 | **0.41** | 0.51 | 0.58 | 1.36 | 1.72 | 1.96 | 5.04 | 1.37 |
| H_sapiens_Chr22.dna | 34553758 | 5 | **4.40** | 5.66 | 8.21 | 7.76 | 15.31 | 15.59 | 49.70 | 10.98 |
| bible | 4047391 | 63 | 0.51 | **0.48** | 0.80 | 1.24 | 1.38 | 1.56 | 4.64 | 1.08 |
| etext | 105277339 | 146 | **19.40** | 23.09 | 43.46 | 26.56 | 62.63 | 54.70 | ML | 119.96 |
| etext_50M | 50000000 | 120 | **8.13** | 9.74 | 17.26 | 11.94 | 26.40 | 24.46 | 88.57 | 79.07 |
| gcc | 86630400 | 150 | **13.84** | 15.58 | 24.50 | 15.84 | 46.20 | 33.62 | 135.12 | 80.78 |
| gcc_50M | 50000000 | 121 | **7.21** | 9.56 | 13.26 | 8.31 | 28.43 | 17.73 | 68.65 | 264.90 |
| howto | 39422104 | 197 | **5.96** | 6.35 | 10.26 | 8.41 | 17.64 | 16.67 | 64.73 | 16.33 |
| jdk | 69728898 | 113 | **12.07** | 12.54 | 26.86 | 12.74 | 39.92 | 24.66 | 102.76 | 58.22 |
| jdk_50M | 50000000 | 110 | **8.32** | **8.30** | 17.05 | 8.91 | 26.30 | 17.58 | 71.31 | 36.98 |
| linux | 116254720 | 256 | **19.27** | **19.34** | 29.67 | 21.17 | 61.99 | 44.47 | ML | 58.71 |
| linux_50M | 50000000 | 256 | **7.62** | **7.60** | 10.50 | 8.84 | 27.54 | 18.18 | 76.10 | 31.92 |
| reuters | 114711150 | 93 | **19.76** | 25.08 | 60.72 | 25.07 | 74.78 | 49.17 | ML | 87.57 |
| reuters_50M | 50000000 | 91 | **7.84** | 9.53 | 20.41 | 10.24 | 26.94 | 20.29 | 77.25 | 33.68 |
| rfc | 116421900 | 120 | **21.18** | 22.08 | 42.75 | 22.55 | 66.28 | 47.99 | ML | 42.14 |
| rfc_50M | 50000000 | 110 | **8.23** | 8.39 | 14.85 | 9.24 | 24.80 | 19.61 | 76.64 | 16.63 |
| sprot | 109617186 | 66 | **18.48** | 22.79 | 47.07 | 25.52 | 69.58 | 50.40 | ML | 48.69 |
| sprot_50M | 50000000 | 66 | **7.57** | 9.10 | 16.81 | 10.88 | 28.07 | 21.69 | 78.47 | 20.03 |
| w3c | 104201578 | 256 | **18.82** | **18.78** | 35.94 | 20.01 | 74.09 | 38.29 | ML | 1964.80 |
| w3c_50M | 50000000 | 255 | **7.93** | 8.33 | 17.67 | 8.73 | 25.95 | 17.26 | 71.42 | 36.59 |
| world | 2473399 | 94 | 0.30 | **0.27** | 0.42 | 0.91 | 0.86 | 0.91 | 2.35 | 0.78 |

Comparison of suffix array construction algorithms run times on datasets from [81] on a 64-bit Intel CORE i3 machine with 4GB of RAM, Ubuntu 11.10 Operating System, Sun Java 1.6.0_26 and gcc 4.6.1. Run times are in seconds, averaged over 10 runs. Bold font indicates the best time. ML means out of memory, TL means more than 1 hour.

Figure 2.1: Average number of times RadixSA accesses each suffix, for datasets from [81].




number of operations on all but one of the inputs tested. The heuristic could find application as an independent speedup technique for other algorithms which use bucket sorting and induced copying. For example, BPR could use it to determine the order in which it chooses buckets to be refined. A possible research direction is to improve RadixSA's heuristic. Buckets can be processed based on a topological sorting of their dependency graph. Such a graph has at most $n/2$ nodes, one for each non singleton bucket, and at most $n/2$ edges. Thus, it has the potential for a lightweight implementation.

An interesting open problem is to devise a randomized algorithm that has a similar performance.

# Chapter 3

# Pattern Matching With Mismatches

## 3.1 Introduction

The problem of string matching has been studied extensively due to its wide range of applications from Internet searches to computational biology. String matching can be defined as follows. Given a text $T = t_1 t_2 \cdots t_n$ and a pattern $P = p_1 p_2 \cdots p_m$, with letters from an alphabet $\Sigma$, find all the occurrences of the pattern in the text. This problem can be solved in $O(n+m)$ time by using well known algorithms (e.g., KMP [50]). A variation of this problem is to search for multiple patterns at the same time. An algorithm for this version is given in [3]. The problem has been generalized to use trees instead of sequences or to use sets of characters instead of single characters (see [23]).

A more general formulation allows "don't care" or "wild card" characters in the text and the pattern. A wild card matches any character. An algorithm for pattern matching with wild cards is given in [34] and has a runtime of $O(n \log |\Sigma| \log m)$. The algorithm maps each character in $\Sigma$ to a binary code of length $\log |\Sigma|$. Then, a constant number of convolution operations are used to check for mismatches between the pattern and any position in the text. For the same problem, a randomized algorithm that runs in $O(n \log n)$ time with high probability is given in [41]. A slightly faster randomized $O(n \log m)$ algorithm is given in [44]. A simple deterministic $O(n \log m)$ time algorithm based on convolutions is given in [14].

A more challenging formulation of the problem is pattern matching with mismatches. This formulation appears in two versions: a) for every alignment of the pattern in the text, find the distance between the pattern and the text, or b) identify only those alignments where the distance between the pattern and the text is less than a given threshold. The distance metric can be the Hamming distance, edit distance,

$L_1$ metric, and so on. A survey of string matching with mismatches is given in [61]. A description of practical on-line string searching algorithms can be found in [62].

The algorithms in this chapter are using the Hamming distance. The Hamming distance between two strings $A$ and $B$, of equal length, is defined as the number of positions where the two strings differ and is denoted by $Hd(A, B)$. We are interested in the following two problems, with and without wild cards.

**1. Pattern matching with mismatches:** Given a text $T = t_1 t_2 \dots t_n$, and a pattern $P = p_1 p_2 \dots p_m$, output $Hd(P, t_i t_{i+1} \dots t_{i+m-1})$, for every $i$, $1 \leq i \leq n - m + 1$.

**2. Pattern matching with $k$ mismatches (or the $k$-mismatches problem):** Take the same input as above, plus an integer $k$. Output all $i$, $1 \leq i \leq n-m+1$, for which $Hd(P, t_i t_{i+1}, \dots t_{i+m-1}) \leq k$.

### 3.1.1 Pattern Matching with Mismatches

For pattern matching with mismatches, a naive algorithm computes the Hamming distance for every alignment of the pattern in the text, in time $O(nm)$. A faster algorithm, in the absence of wild cards, is Abrahamson's algorithm [2] that runs in $O(n\sqrt{m \log m})$ time. Abrahamson's algorithm can be extended to solve pattern matching with mismatches *and* wild cards, as we prove in section 3.2.2. The new algorithm runs in $O(n\sqrt{g \log m})$ time, where $g$ is the number of non-wild card positions in the pattern. This gives a simpler and faster alternative to an algorithm proposed in [5].

In the literature, we also find algorithms that approximate the number of mismatches for every alignment. For example, an approximate algorithm for pattern matching with mismatches, in the absence of wild cards, that runs in $O(rn \log m)$ time, where $r$ is the number of iterations of the algorithm, is given in [6]. Every distance reported has a variance bounded by $(m - c_i)/r^2$ where $c_i$ is the exact number of matches for alignment $i$.

Furthermore, a randomized algorithm that approximates the Hamming distance for every alignment within an $\epsilon$ factor and runs in $O(n \log^c m/\epsilon^2)$ time, in the absence of wild cards, is given in [46]. Here $c$ is a small constant. We extend this algorithm to pattern matching with mismatches *and* wild cards, in section 3.2.5. The new algorithm approximates the Hamming distance for every alignment within an $\epsilon$ factor in time $O(n \log^2 m/\epsilon^2)$ with high probability.

Recent work has also addressed the online version of pattern matching, where the text is received in a streaming model, one character at a time, and it cannot be stored in its entirety (see e.g., [16], [70], [71]). Another version of this problem matches the pattern against multiple input streams (see e.g., [15]). Another interesting problem is to sample a representative set of mismatches for every alignment (see e.g., [17]).

### 3.1.2 Pattern Matching with $k$ Mismatches

For the $k$-mismatches problem, without wild cards, two algorithms that run in $O(nk)$ time are presented in [55] and [37]. A faster algorithm, that runs in $O(n\sqrt{k \log k})$ time, is given in [5]. This algorithm combines the two main techniques known in the literature for pattern matching with mismatches: filtering and convolutions. We give a significantly simpler algorithm in section 3.2.3, having the same worst case run time. The new algorithm will never perform more operations than the one in [5] during marking and convolution.

An intermediate problem is to check if the Hamming distance is less or equal to $k$ for a subset of the aligned positions. This problem can be solved with the Kangaroo method proposed in [37] at a cost of $O(k)$ time per alignment, using $O(n + m)$ additional memory. We show how to achieve the same run time per alignment using only $O(m)$ additional memory, in section 3.2.3.

Further, we look at the version of $k$-mismatches where wild cards are allowed in the text and the pattern. For this problem, two randomized algorithms are presented in [15]. The first one runs in $O(nk \log n \log m)$ time and the second one in $O\left(n \log m(k + \log n \log \log n)\right)$ time. Both are Monte Carlo algorithms, i.e., they output the correct answer with high probability. The same paper also gives a deterministic algorithm with a run time of $O(nk^2 \log^3 m)$. Also, a deterministic $O(nk \log^2 m(\log^2 k + \log \log m))$ time algorithm is given in [18]. We present a Las Vegas algorithm (that always outputs the correct answer), in section 15, which runs in time $O(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log \log n)$ with high probability. Deterministically, pattern matching with $k$ mismatches and wild cards can be solved in time $O(nk^2 \log^2 m)$ as shown in [21].

If we allow for wild cards in the pattern but not the text, an $O(nm^{1/3}k^{1/3}log^{2/3}m)$ time algorithm is given in [20]. We improve this runtime as follows. Given a pattern $P$, with wild cards, a maximal length substring of $P$ that has no wild cards is called an "island". We denote the number of islands in $P$ as $q$. In chapter 3.2.4 we give two algorithms for pattern matching with $k$ mismatches where there are wild cards in the pattern. The first one runs in $O(n\sqrt{(q+k)\log m})$ time. The second one runs in time $O(n\sqrt[3]{qk \log^2 m} + n\sqrt{k \log m})$ where $q$ is the number of islands in $P$. By combining the two, we show that pattern matching with $k$ mismatches and wild cards in the pattern can be solved in $O(n\sqrt{k \log m} + n \min\{\sqrt[3]{qk \log^2 m}, \sqrt{q \log m}\})$ time. If the number of islands is $O(k)$ our runtime becomes $O(n\sqrt{k \log m})$, which essentially matches the best known runtime for pattern matching with $k$ mismatches without wild cards ($O(n\sqrt{k \log k})$). If $q = o(m)$, our algorithm outperforms the $O(n\sqrt[3]{mk \log^2 m})$ run time of [19]. Therefore, our algorithm is a step towards bridging the gap between the versions with and

without wild cards cares for pattern matching with $k$ mismatches. In other words, with the previous algorithm, even with a few wild cards in the pattern, the runtime could be much larger than the runtime if there were no wild cards. In the new algorithm, if the number of wild cards is relatively small then the runtime is the same as in the version without wild cards.

### 3.1.3 Our Results

The contributions of this chapter can be summarized as follows.

**For pattern matching with mismatches:**

- An algorithm for pattern matching with mismatches and wild cards that runs in $O(n\sqrt{g \log m})$ time, where $g$ is the number of non-wild card positions in the pattern. See section 3.2.2.
- A randomized algorithm that approximates the Hamming distance for every alignment, when wild cards are present, within an $\epsilon$ factor in time $O(n \log^2 m/\epsilon^2)$ with high probability. See section 3.2.5.

**For pattern matching with $k$ mismatches:**

- An algorithm that tests if the Hamming distance is less than $k$ for a subset of the alignments, without wild cards, at a cost of $O(k)$ time per alignment, using only $O(m)$ additional memory. This achieves the same runtime per alignment as the Kangaroo method, but with less memory. See section 3.2.3.
- An algorithm for pattern matching with $k$ mismatches, without wild cards, that runs in $O(n\sqrt{k \log k})$ time. This algorithm is simpler and has a better expected run time than the one in [5]. See section 3.2.3.
- An algorithm for pattern matching with $k$ mismatches with wild cards in the pattern, that runs in $O(n\sqrt{k \log m} + n \min\{\sqrt[3]{qk \log^2 m}, \sqrt{q \log m}\})$ time, where $q$ is the number of non-wild card islands in the pattern. If $q = o(m)$, our algorithm outperforms the $O(n\sqrt[3]{mk \log^2 m})$ run time of [19]. See section 3.2.4.
- A Las Vegas algorithm for the $k$-mismatches problem with wild cards that runs in time $O(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log \log n)$ with high probability. See section 15.

These algorithms are also included in our papers [64] and [66]. The rest of the chapter is organized as follows. First we introduce some notations and definitions. Then we describe the exact, deterministic algorithms for pattern matching with mismatches and for $k$-mismatches. Then we present the randomized

and approximate algorithms: first the algorithm for approximate counting of mismatches in the presence of wild cards, then the Las Vegas algorithm for $k$-mismatches with wild cards. Finally, we present an empirical run time comparison of the deterministic algorithms, and conclusions.

## 3.2 Materials and Methods

In terms of notation, $T_{i..j}$ is the substring of $T$ between $i$ and $j$ and $T_i$ stands for $T_{i..i+m-1}$. Furthermore, the value at position $i$ in array $X$ is denoted by $X[i]$.

### 3.2.1 Background

In this section we review a number of well known techniques used in the literature for pattern pattern matching with $k$ mismatches (e.g., see [5]), namely: convolution, marking, filtering and the Kangaroo method.

#### Convolution

Given two arrays $T = t_1 t_2 \dots t_n$ and $P = p_1 p_2 \dots p_m$ (with $m \leq n$), the convolution of $T$ and $P$ is a sequence $C = c_1, c_2, \dots, c_{n-m+1}$ where $c_i = \sum_{j=1}^{m} t_{i+j-1} p_j$, for $1 \leq i \leq (n - m + 1)$.

The convolution can be applied to pattern matching with mismatches, as follows. Given a string $S$ and a character $\alpha$ define string $S^\alpha$ as $S^\alpha[i] = 1$ if $S[i] = \alpha$ and 0 otherwise. Let $C^\alpha = \mathit{convolution}(T^\alpha, P^\alpha)$. Then $C^\alpha[i]$ gives the number of matches between $P$ and $T_i$ where the matching character is $\alpha$. Therefore, one convolution gives us the number of matches contributed by a single character to each of the alignments. Then $\sum_{\alpha \in \Sigma} C^\alpha[i]$ is the total number of matches between $P$ and $T_i$.

One convolution can be computed in $O(n \log m)$ time by using the Fast Fourier Transform. If the convolutions are applied on binary inputs, as is often the case in pattern matching applications, some speedup techniques are presented in [35].

#### Marking

Marking is an algorithm that counts the number of matches of every alignment. Specifically, the algorithm scans the text one character at a time and "marks" all the alignments that would produce a match between the current character in the text and the corresponding character in the pattern.

The marking algorithm is generally used only on a subset of the pattern. That is, given a set $A$ of positions in $P$ the marking algorithm counts matches between the text and the subset of $P$ given by $A$.

The pseudocode of the marking algorithm is given in Algorithm 7.

**Algorithm 7:** Mark($T, P, A$)

**input** : Text $T$, pattern $P$ and a set $A$ of positions in $P$
**output**: An array $M$ where $M[i]$ gives the number of matches between $T_i$ and $P$, on the subset of positions of $P$ given by $A$

```
for i ← 1 to n do M[i] = 0;
for i ← 1 to n do
    for j ∈ A s.t. P[j] = T[i] do
        if i − j + 1 > 0 then M[i − j + 1]++ ;
return M;
```

### Filtering

The marking algorithm in the previous section is generally followed by a filtering method. Filtering is based on the following principle. If we restrict our pattern to only $2k$ positions, any alignment that has no more than $k$ mismatches, must have at least $k$ matches among the $2k$ positions. To count matches among the $2k$ positions selected, for every alignment, we use the marking algorithm. If the total number of marks generated is $B$ then there can be no more than $B/k$ positions that have at least $k$ marks (i.e., matches). The alignments that have at least $k$ marks are further inspected using the Kangaroo method.

### The Kangaroo method

The Kangaroo method allows us to check if the number of mismatches for a particular alignment is no more than $k$, in $O(k)$ time. The Kangaroo method constructs a generalized suffix tree of $T\#P$ where $\#$ means concatenation. This suffix tree can be enhanced to answer Lowest Common Ancestor (LCA) queries in $O(1)$ time [4]. LCA queries give us the longest common prefix between any portion of the text and any portion of the pattern, essentially telling us where the first mismatch appears. Specifically, to count mismatches between $P$ and $T_i$, first perform an LCA query to find the position of the first mismatch between $P$ and $T_i$. Let this position be $j$. Then, perform another LCA to find the first mismatch between $P_{j+1..m}$ and $T_{i+j+1..i+m-1}$, which gives the second mismatch of alignment $i$. Continue to "jump" from one mismatch to the next, until the end of the pattern is reached or we have found more than $k$ mismatches. Therefore, after $O(k)$ LCA queries we will either find all the mismatches or determine that there are more than $k$ of them. The Kangaroo pseudocode is given in Algorithm 8.

**Algorithm 8:** Kangaroo$(P, T_i, k)$

**input** : A pattern $P$, an alignment $T_i$ and an integer $k$
**output**: **true** if the pattern matches the alignment with no more than $k$ mismatches, **false** otherwise

```
j = 0;
d = 0;
while d ≤ k do
    j = j + LCA(T_{i+j}, P_j) + 1;
    if j > m then
        return true;
    d = d + 1;
return false;
```

**Probability Bounds**

In the context of randomized algorithms, by high probability we mean a probability greater or equal to $(1 - n^{-\epsilon})$ where $n$ is the input size and $\epsilon$ is a probability parameter usually assumed to be a constant greater than 0. The run time of a Las Vegas algorithm is said to be $\widetilde{O}(f(n))$ if the run time is no more than $c\epsilon f(n)$ with probability greater or equal to $(1 - n^{-\epsilon})$ for all $n \geq n_0$, where $c$ and $n_0$ are some constants, and for any constant $\epsilon \geq 1$.

In the analysis of our algorithms, we will employ the following Chernoff bounds.

**Chernoff Bounds** [13]. These bounds can be used to closely approximate the tail ends of a binomial distribution.

A Bernoulli trial has two outcomes namely *success* and *failure*, the probability of success being $p$. A binomial distribution with parameters $n$ and $p$, denoted as $B(n, p)$, is the number of successes in $n$ independent Bernoulli trials.

Let $X$ be a binomial random variable whose distribution is $B(n, p)$. If $m$ is any integer $> np$, then the following are true:

$$Prob.[X > m] \ \leq \ \left(\frac{np}{m}\right)^m \ e^{m-np}; \tag{3.1}$$

$$Prob.[X > (1+\delta)np] \ \leq \ e^{-\delta^2 np/3}; \text{ and} \tag{3.2}$$

$$Prob.[X < (1-\delta)np] \ \leq \ e^{-\delta^2 np/2} \tag{3.3}$$

for any $0 < \delta < 1$.

### 3.2.2 Exact Algorithms for Pattern Matching with Mismatches

For pattern matching with mismatches, without wild cards, the following $O(n\sqrt{m \log m})$ time algorithm was given by Abrahamson [2]. Let $A$ be a set of the most frequent characters in the pattern. 1) Using convolutions, count how many matches each character in $A$ contributes to every alignment. 2) Using marking, count how many matches each character in $\Sigma - A$ contributes to every alignment. 3) Add the two numbers to find for every alignment, the number of matches between the pattern and the text. The convolutions take $O(|A|n \log m)$ time. A character in $\Sigma - A$ cannot appear more than $m/|A|$ times in the pattern, otherwise, each character in $A$ has a frequency greater than $m/|A|$, which is not possible. Thus, the run time for marking is $O(nm/|A|)$. If we equate the two run times we find the optimal $|A| = \sqrt{m/\log m}$ which gives a total run time of $O(n\sqrt{m \log m})$.

**An Example.** Consider the case of $T$ = 2 3 1 1 4 1 2 3 4 4 2 1 1 3 2 and $P$ = 1 2 3 4. Since each character in the pattern occurs an equal number of times, we can pick $A$ arbitrarily. Let $A = \{1, 2\}$. In step 1, convolution is used to count the number of matches contributed by each character in $A$. We obtain an array $M_1[1:12]$ such that $M_1[i]$ is the number of matches contributed by characters in $A$ to the alignment of $P$ with $T_i$, for $1 \leq i \leq 12$. In this example, $M_1 = [0, 0, 1, 1, 0, 2, 0, 0, 0, 1, 0, 1]$. In step 2 we compute, using marking, the number of matches contributed by the characters 3 and 4 to each alignment between $T$ and $P$. We get another array $M_2[1:12]$ such that $M_2[i]$ is the number of matches contributed by 3 and 4 to the alignment between $T_i$ and $P$, for $1 \leq i \leq 12$. Specific to this example, $M_2 = [0, 1, 0, 0, 0, 2, 1, 0, 0, 0, 0, 1]$. In step 3, we add $M_1$ and $M_2$ to get the number of matches between $T_i$ and $P$, for $1 \leq i \leq 12$. In this example, this sum yields: $[0, 1, 1, 1, 0, 4, 1, 0, 0, 1, 0, 2]$.

#### Pattern matching with mismatches and wild cards in $O(n\sqrt{g \log g})$

For pattern matching with mismatches and wild cards, a fairly complex algorithm is given in [5]. The run time of this algorithm is $O(n\sqrt{g} \log m)$ where $g$ is the number of non-wild card positions in the pattern. The problem can also be solved through a simple modification of Abrahamson's algorithm, in time $O(n\sqrt{m \log m})$, as pointed out in [15]. We now prove the following result:

**Theorem 4.** *Pattern matching with mismatches and wild cards can be solved in $O(n\sqrt{g \log m})$ time, where $g$ is the number of non-wild card positions in the pattern.*

*Proof.* Ignoring the wild cards for now, let $A$ be the set of the most frequent characters in the pattern. As above, count matches contributed by characters in $A$ and $\Sigma - A$ using convolution and marking, respectively. By a similar reasoning as above, the characters used in the marking phase will not appear more

than $g/|A|$ times in the pattern. If we equate the run times for the two phases we obtain $O(n\sqrt{g \log m})$ time. We are now left to count how many matches are contributed by the wild cards. For a string $S$ and a character $\alpha$, define $S^{\neg\alpha}$ as $S^{\neg\alpha}[i] = 1 - S^{\alpha}[i]$. Let $w$ be the wild card character. Compute $C = convolution(T^{\neg w}, P^{\neg w})$. Then, for every alignment $i$, the number of positions that have a wild card either in the text or the pattern or both, is $m - C[i]$. Add $m - C[i]$ to the previously computed counts and output. The total run time is $O(n\sqrt{g \log m})$. □

### 3.2.3 Exact Algorithms for Pattern Matching with $k$ Mismatches

For the $k$-mismatches problem, without wild cards, an $O(k(m \log m + n))$ time algorithm that requires $O(k(m + n))$ additional space is presented in [55]. Another algorithm, that takes $O(m \log m + kn)$ time and uses only $O(m)$ additional space is presented in [37]. We define the following problem which is of interest in the discussion.

#### The Subset $k$-mismatches problem

**Problem 3. Subset $k$-mismatches:** *Given a text $T$ of length $n$, a pattern $P$ of length $m$, a set of positions $S = \{i | 1 \leq i \leq n - m + 1\}$ and an integer $k$, output the positions $i \in S$ for which $Hd(P, T_i) \leq k$.*

The Subset $k$-mismatches problem becomes the regular $k$-mismatches problem if $|S| = n - m + 1$. Thus, it can be solved by the $O(nk)$ algorithms mentioned above. However, if $|S| << n$ then the $O(nk)$ algorithms are too costly. A better alternative is to use the Kangaroo method proposed in [5] and described in section 3.2.1. The Kangaroo method can verify if $Hd(P, T_i) \leq k$ in $O(k)$ time for any $i$. The Kangaroo method can process $|S|$ positions in $O(n + m + |S|k)$ time and it uses $O(n + m)$ additional memory for the LCA enhanced suffix tree. The memory requirement can be improved as follows:

**Theorem 5. Subset $k$-mismatches** *can be solved in $O(n + m + |S|k)$ time using only $O(m)$ additional memory.*

*Proof.* The algorithm is the following. Build an LCA-enhanced suffix tree of the pattern. Scan the text from left to right. 1) Find the longest unscanned region of the text that can be found somewhere in the pattern, say starting at position $i$ of the pattern. Call this region of the text $R$. Therefore, $R$ is identical to $P_{i..i+|R|-1}$. 2) For every alignment in $S$ that overlaps $R$, count the number of mismatches between $R$ and the alignment, within the overlap region. To do this, consider an alignment in $S$ which overlaps $R$ such that the beginning of $R$ aligns with the $j$-th character in the pattern. We want to count the number of mismatches between R and $P_{j..j+|R|-1}$. However, since $R$ is identical to $P_{i..i+|R|-1}$, we can

simply compare $P_{i..i+|R|-1}$ and $P_{j..j+|R|-1}$. This comparison can be done efficiently by jumping from one mismatch to the next, like in the Kangaroo method. Repeat from step 1 until the entire text has been scanned. Every time we process an alignment, in step 2, we either discover at least one additional mismatch or we reach the end of the alignment. This is true because, otherwise, the alignment would match the text for more than $|R|$ characters, which is not possible, from the way we defined $R$. Every alignment for which we have found more than $k$ mismatches is excluded from further consideration to ensure $O(k)$ time per alignment. It takes $O(m)$ time to build the LCA enhanced suffix tree of the pattern and $O(n)$ additional time to scan the text from left to right. Thus, the total run time is $O(n+m+|S|k)$ with $O(m)$ additional memory. The pseudocode is given in Algorithm 9. □

**Algorithm 9:** Subset $k$-mismatches$(S, T, P, k)$

**input** : $S$ - set of positions in the text; $T_{1..n}$ - text; $P_{1..m}$ -pattern; k - max number of mismatches;
**output**: $M$ - the positions in $S$ for which the pattern matches the text with at most $k$ mismatches;
**begin**
  Assume we have a suffix tree/array of the pattern;
  **for** $a \in S$ **do** $M[a] = 0$;
  $i = 1$ ;
  **while** $i \leq n$ **do**
    Find the largest $l$ such that $T_{i..i+l-1}$ describes a path in the suffix tree;
    This means that $T_{i..i+l-1} = P_{j..j+l-1}$ for some $j$;
    **for** $a \in S$ **where** $a \leq i < a + m$ **do**
      $M[a] =$ `countMismatches` $(M[a], i - a + 1, j, l)$ ;
      **if** $t_{i+l} \neq p_{j+l}$ **then** $M[a] = M[a] + 1$;
      **if** $M[a] > k$ **then** $S = S - \{a\}$;
    $i = i + l + 1$ ;
  **return** $\{a \in S | M[a] \leq k\}$
**function** `countMismatches`$(c, s_1, s_2, l)$
  **input** : $c$ - current number of mismatches; $s_1, s_2$ - starting positions of two suffixes of the pattern; $l$ - a maximum length;
  **output**: compare the two suffixes on their first $l$ positions and add to $c$ the number of mismatches found; if $c$ exceeds $k$, return $k + 1$, otherwise return the updated $c$ ;
  **begin**
    **while** $l > 0$ **and** $c \leq k$ **do**
      $d =$ `lcp`$(s_1, s_2)$; // longest common prefix
      **if** $d \geq l$ **then return** $c$;
      $c = c + 1$;
      $d = d + 1$;
      $s_1 = s_1 + d$;
      $s_2 = s_2 + d$;
      $l = l - d$;
  **return** $c$

**An $O(n\sqrt{k \log k})$ Time Algorithm for $k$-Mismatches without Wild Cards**

For the $k$-mismatches problem, without wild cards, a fairly complex $O(n\sqrt{k \log k})$ time algorithm is given in [5]. The algorithm classifies the inputs into several cases. For each case it applies a combination of marking followed by a filtering step, the Kangaroo method, or convolutions. The goal is to not exceed $O(n\sqrt{k \log k})$ time in any of the cases. We now present an algorithm with only two cases which has the same worst case run time. The new algorithm can be thought of as a generalization of the algorithm in [5] as we will discuss later. This generalization not only greatly simplifies the algorithm but it also reduces the expected run time. This happens because we use information about the frequency of the characters in the text and try to minimize the work done by convolutions and marking.

We will now give the intuition for this algorithm. For any character $\alpha \in \Sigma$, let $f_\alpha$ be its frequency in the pattern, and $F_\alpha$ be its frequency in the text. Note that in the marking algorithm, a specific character $\alpha$ will contribute to the runtime a cost of $F_\alpha \times f_\alpha$. On the other hand, in the case of convolution, a character $\alpha$ costs us one convolution, regardless of how frequent $\alpha$ is in the text or the pattern. Therefore, we want to use infrequent characters for marking and frequent characters for convolution. The balancing of the two will give us the desired runtime.

A position $j$ in the pattern where $p_j = \alpha$ is called an *instance* of $\alpha$. Consider every instance of character $\alpha$ as an object of size 1 and cost $F_\alpha$. We want to fill a knapsack of size $2k$ at a minimum cost and without exceeding a given budget $B$. The $2k$ instances will allow us to filter some of the alignments with more than $k$ mismatches, as it will become clear later. This problem can be optimally solved by a greedy approach where we include in the knapsack all the instances of the least expensive character, then all the instances of the second least expensive character and so on, until we have $2k$ items or we have exceeded $B$. The last character considered may have only a subset of its instances included, but for ease of explanation assume that there are no such characters.

**Note:** Even though the above is described as a Knapsack problem, the particular formulation can be optimally solved in linear time. This formulation should not be confused with other formulations of the Knapsack problem that are NP-Complete.

Case 1) Assume we can fill the knapsack at a cost $C \leq B$. We apply the marking algorithm for the characters whose instances are included in the knapsack. It is easy to see that the marking takes time $O(C)$ and creates $C$ marks. For alignment $i$, if the pattern and the text match for all the $2k$ positions in the knapsack, we will obtain exactly $2k$ marks at position $i$. Conversely, any position which has less than $k$ marks must have more than $k$ mismatches, so we can filter it out. Therefore, there will be at most

$C/k$ positions with $k$ marks or more. For such positions we run Subset $k$-mismatches to confirm which of them have less than $k$ mismatches. The total runtime of the algorithm in this case is $O(C)$.

Case 2) If we cannot fill the knapsack within the given budget $B$ we do the following: for the characters we could fit in the knapsack, we use the marking algorithm to count the number of matches they contribute to each alignment. For characters not in the knapsack, we use convolutions to count the number of matches they contribute to each alignment. We add the two counts and get the exact number of matches for every alignment.

Note that at least one of the instances in the knapsack has a cost larger than $B/(2k)$ (if all the instances in the knapsack had a cost less or equal to $B/(2k)$ then we would have at least $2k$ instances in the knapsack). Also note that all the instances not in the knapsack have a cost at least as high as any instance in the knapsack, because we greedily fill the knapsack starting with the least costly instances. This means that every character not in the knapsack appears in the text at least $B/(2k)$ times. This means that the number of characters not in the knapsack does not exceed $n/(B/(2k))$. Therefore the total cost of convolutions is $O(nk/B \log m)$. Since the cost of marking was $O(B)$ we can see that the best value of $B$ is the one that equalizes the two costs. This gives $B = O(n\sqrt{k \log m})$. Therefore, the algorithm takes $O(n\sqrt{k \log m})$ time. If $k < m^{1/3}$ we can employ a different algorithm that solves the problem in linear time, as in [5]. For larger $k$, $O(\log m) = O(\log k)$ so the run time becomes $O(n\sqrt{k \log k})$. We call this algorithm Knapsack $k$-mismatches. The pseudocode is given in algorithm 10. The following theorem results.

**Theorem 6. Knapsack $k$-mismatches** *has worst case run time* $O(n\sqrt{k \log k})$.□

We can think of the algorithm in [5] as a special case of our algorithm where, instead of trying to minimize the cost of the $2k$ items in the knapsack, we just try to find $2k$ items for which the cost is less than $O(n\sqrt{k \log m})$. As a result, it is easy to verify the following:

**Theorem 7. Knapsack $k$-mismatches** *spends at most as much time as the algorithm in [5] to do convolutions and marking.*

*Proof.* **Observation:** In all the cases presented below, Knapsack $k$-mismatches can have a run time as low as $O(n)$, for example if there exists one character $\alpha$ with $f_\alpha = O(k)$ and $F_\alpha = O(n/k)$.

**Case 1:** $|\Sigma| \geq 2k$. The algorithm in [5] chooses $2k$ instances of distinct characters to perform marking. Therefore, for every position of the text at most one mark is created. If the number of marks is $M$, then the cost of the marking phase is $O(n + M)$. The number of remaining positions after filtering is no more than $M/k$ and thus the algorithm takes $O(n+M)$ time. Our algorithm puts in the knapsack 2k instances,

**Algorithm 10:** Knapsack $k$-mismatches$(T, P, k)$

**input** : $T_{1..n}$ - text; $P_{1..m}$ -pattern; k - max number of mismatches;
**output**: $S$ - set of positions in the text where the pattern matches with at most $k$ mismatches;
**begin**
    Compute $F_i$ and $f_i$ for every $i \in \Sigma$ ;
    Sort $\Sigma$ with respect to $F_i$ ;
    $s = 0$;
    $c = 0$;
    $i = 1$;
    $B = n\sqrt{k \log k}$ ;
    **while** $s < 2k$ **and** $c < B$ **do**
        $t = \min(f_i, 2k - s)$ ;
        $s = s + t$ ;
        $c = c + t \times F_i$ ;
        $i = i + 1$ ;
    $\Gamma = \Sigma[1..i]$ ;
    $M = \texttt{Mark}(T, n, \Gamma)$; // $M$ counts matches
    **if** $s = 2k$ **then**
        $S = \{i | M[i] \geq k\}$ ;
        **return** `Subset` $k$-`mismatches` $(S, T, P, k)$;
    **else**
        **for** $\alpha \in \Sigma - \Gamma$ **do**
            $C = \texttt{convolution}(T^\alpha, P^\alpha)$ ;
            **for** $i \leftarrow 1$ **to** $n$ **do**
                $M[i] = M[i] + C[i]$ ;
        $S = \{i | M[i] \geq m - k\}$;
        **return** $S$;

of not necessarily different characters, such that the number of marks $B$ is minimized! Therefore $B \leq M$ and the total runtime is $O(n + B)$.

**Case 2:** $|\Sigma| < 2\sqrt{k}$. The algorithm in [5] performs one convolution per character to count the total number of matches for every alignment, for a run time of $\Omega(|\Sigma| n \log m)$. In the worst case, Knapsack $k$-mismatches cannot fill the knapsack at a cost $B < |\Sigma| n \log m$ so it defaults to the same run time. However, in the best case, the knapsack can be filled at a cost $B$ as low as $O(n)$ depending on the frequency of the characters in the pattern and the text. In this case the runtime will be $O(n)$.

**Case 3:** $2\sqrt{k} \leq |\Sigma| \leq 2k$. A symbol that appears in the pattern at least $2\sqrt{k}$ times is called frequent.

**Case 3.1:** There are at least $\sqrt{k}$ frequent symbols. The algorithm in [5] chooses $2\sqrt{k}$ instances of $\sqrt{k}$ frequent symbols to do marking and filtering at a cost $M \leq 2n\sqrt{k}$. Since Knapsack $k$-mismatches will minimize the marking time $B$ we have $B \leq M$ so the run time is the same as for [5] only in the worst case.

**Case 3.2:** There are $A < \sqrt{k}$ frequent symbols. The algorithm in [5] first performs one convolution for each frequent character for a run time of $O(An \log m)$. Two cases remain:

**Case 3.2.1:** All the instances of the non-frequent symbols number less than $2k$ positions. The algorithm in [5] replaces all instances of frequent characters with wild cards and applies a $O(n\sqrt{g} \log m)$ algorithm to count mismatches, where $g$ is the number of non-wild card positions. Since $g < 2k$ the run time for this stage is $O(n\sqrt{k} \log m)$ and the total run time is $O(An \log m + n\sqrt{k} \log m)$. Knapsack $k$-mismatches can always include in the knapsack all the instances of non-frequent symbols since their total cost is no more than $O(n\sqrt{k})$ and in the worst case do convolutions for the remaining characters. The total run time is $O(An \log m + n\sqrt{k})$. Of course, depending on the frequency of the characters in the pattern and text, Knapsack $k$-mismatch may not have to do any convolutions.

**Case 3.2.2:** All the instances of the non-frequent symbols number at least $2k$ positions. The algorithm in [5] chooses $2k$ instances of infrequent characters to do marking. Since each character has frequency less than $2\sqrt{k}$, the time for marking is $M < 2n\sqrt{k}$ and there are no more than $M/k$ positions left after filtering. Knapsack $k$-mismatches chooses characters in order to minimize the time $B$ for marking, so again $B \leq M$. □

### 3.2.4 Algorithms for $k$-Mismatches with Wild Cards in the Pattern

In this section we consider pattern matching with $k$ mismatches where there could be some wild cards in the pattern. Given a pattern $P$, with wild cards, a maximal length substring of $P$ that has no wild cards is called an "island". We will denote the number of islands in $P$ as $q$.

For this problem, an algorithm that runs in $O(nm^{1/3}k^{1/3}log^{2/3}m)$ time is given in [20]. We improve this runtime as follows. We give two algorithms for pattern matching with $k$ mismatches where there are wild cards in the pattern. The first one runs in $O(n\sqrt{(q+k)\log m})$ time. The second one runs in time $O(n\sqrt[3]{qk\log^2 m} + n\sqrt{k\log m})$ where $q$ is the number of islands in $P$. By combining the two, we show that pattern matching with $k$ mismatches and wild cards in the pattern can be solved in $O(n\sqrt{k\log m} + n\min\{\sqrt[3]{qk\log^2 m}, \sqrt{q\log m}\})$ time.

If the number of islands is $O(k)$ our runtime becomes $O(n\sqrt{k\log m})$, which essentially matches the best known runtime for pattern matching with $k$ mismatches without wild cards ($O(n\sqrt{k\log k})$). If $q = o(m)$, our algorithm outperforms the $O(n\sqrt[3]{mk\log^2 m})$ run time of [19].

Both algorithms in this section have the same basic structure. The difference is in how fast we can verify whether the distance between $P$ and a given alignment is no more than $k$. In other words, the difference is in how fast we can answer the single alignment verification question:

**Question 1.** *Given $i$, is the distance between $P$ and $T_i$ no more than $k$?*

In the first algorithm, we can answer this question in $O(q+k)$ time. In the second algorithm, we can answer this question in $O(\sqrt[3]{k^2q^2\log m} + k)$ time.

The general structure of both the algorithms is given in Algorithm 11 and is essentially a slight generalization of the Knapsack $k$-mismatches algorithm of section 3.2.3.

**Algorithm 11:** $K$-Mismatches with Wild Cards

```
Let F_a be the number of occurrences of character a in T for all a ∈ Σ;
Let Cost(A) = Σ_{i∈A} F_{P[i]};
Let A be a set of positions in P such that |A| ≤ 2k and Cost(A) ≤ B;
M = Mark(T, P, A);
if |A| == 2k then
    R = {};
    for i = 1 to n do
        if M_i ≥ k and DistNoMoreThanK(T_i, P, k) then
            R = R ∪ {i};
else
    for a ∈ Σ s.t. a ≠ P[i], ∀i ∈ A do
        M' = Convolution(T, P, a);
        M+ = M';
    R = {i ∈ [1..n] | M_i ≥ m − k};
return R;
```

**Algorithm and analysis:** For each position $i$ in $P$ such that $P[i] = a$, we assign a cost $F_a$ where $F_a$ is the number of occurrences of $a$ in $T$. The algorithm starts by choosing up to $2k$ positions from the

pattern such that the total cost does not exceed a "budget" $B$. The positions are chosen by a simple greedy strategy. Sort all the characters by their cost $F_a$. Start choosing positions equal to the "cheapest" character, then choose positions equal to the next cheapest character, and so on until we have chosen $2k$ positions or we have exceeded the budget $B$.

**Case 1:** If we can find $2k$ positions that cost no more than $B$, then we call the marking algorithm with those $2k$ positions. Any position in $T$ that receives less than $k$ marks, has more than $k$ mismatches, so we now focus on positions in $T$ that have at least $k$ marks. If the total number of marks is $B$, then there will be no more than $B/k$ positions that have at least $k$ marks. We verify each of these positions to see if they have more than $k$ mismatches. Let the time for a single verification be $O(V)$. Then, the runtime is $O(BV/k)$.

**Case 2:** If we cannot find $2k$ positions that cost no more than $B$, then we compute marking for the positions that we did choose before we ran out of budget. Then, for each of the characters that we did not choose, we compute one convolution to count how many matches they contribute to each alignment. It is easy to see that each of the characters not chosen for marking must have $F_a > B/(2k)$. Therefore, the total number of such characters is no more than $n/(B/(2k))$. Therefore, the runtime of the convolution stage is $O(nk/B * n \log m)$. The runtime of the marking stage is $O(B)$, therefore the total runtime is $O(B + nk/B * n \log m)$.

If we make the runtime of the two cases equal, we can find the optimal value of $B$.

$$BV/k = B + n^2k/B \log m \Rightarrow B = nk\sqrt{\frac{\log m}{V}}$$

This gives an asymptotic runtime of $O(BV/k) = O(n\sqrt{V \log m})$. Therefore, the runtime of the algorithm depends on $V$, which is the time it takes to verify whether a given single alignment has no more than $k$ mismatches.

#### Single alignment distance in $O(q + k)$ time

We can answer the single alignment question in $O(q + k)$ time where $q$ is the number of *islands* in the pattern as shown in Algorithm 12. The algorithm uses Kangaroo jumps [55] to go to the next mismatch within an island in $O(1)$ time. If there is no mismatch left in the island, the algorithm goes to the next island also in $O(1)$ time. Therefore, the runtime is $O(q + k)$. With $V = O(q + k)$, Algorithm 11 does pattern matching with $k$ mismatches in $O(n\sqrt{(q + k) \log m})$ time.

**Algorithm 12:** KangarooDistNoMoreThanK($T_i, P, k$)

```
d = 0;
j = 0;
while d ≤ k and j < q do
    r = no. of mismatches between island j and corresponding region of T_i (use Kangaroo jumps);
    d+ = r;
    j+ = 1;
return d ≤ k
```

**Single alignment distance in $O(\sqrt[3]{k^2q^2 \log m} + k)$ time**

This idea is based on splitting the pattern into sections. We know that no more than $k$ sections can have mismatches. The remaining sections have to match exactly. Consider exact pattern matching with wild cards. We can check where a pattern matches the text exactly by using a constant number of convolutions. This is true because we can compute the values $C_i = \Sigma_{j=0}^{m-1}(T_{i+j} - P_j)^2 T_{i+j} P_j$ using a constant number of convolutions (see [14]). If $C_i = 0$ then the pattern matches the text at position $i$.

Using this result, we will split the pattern into $S$ sections. In each section we include $q/S$ islands. For each of the $S$ sections, we use a constant number of convolutions to check where the section matches the text. If $P$ has no more than $k$ mismatches at a particular alignment, then at least $S - k$ sections have to match exactly. Each of the at most $k$ sections that do not match exactly are verified using Kangaroo jumps as seen earlier. One section takes at most $O(q/S + k')$ time, where $k'$ is the number of mismatches discovered in that section. Over all the sections, the $k'$ terms add up to no more than $k$, therefore the entire alignment can be verified in time $O(S + k + kq/S)$.

If we make $V = O(S + k + kq/S)$ in Algorithm 11, then its runtime becomes $O(n\sqrt{V \log m}) = O(n\sqrt{(S + k + kq/S) \log m})$. The preprocessing time for the $S$ sections is $O(Sn \log m)$. The optimal value of $S$ is such that the preprocessing equals the main runtime:

$$
\begin{aligned}
& n\sqrt{(S + k + kq/S) \log m} = Sn \log m \\
\Rightarrow & S + k + kq/S = S^2 \log m \\
\Rightarrow & S^2/\log m + kS/\log m + kq/\log m = S^3 \\
\Rightarrow & S \approx O(\sqrt[3]{kq/\log m})
\end{aligned}
$$

This makes $V = O(S + k + kq/S) = O(k + \sqrt[3]{k^2q^2 \log m})$. This gives a runtime for pattern matching with $k$ mismatches of:

$$O(nS\log m + n\sqrt{V\log m}) = O\left(n\sqrt[3]{kq\log^2 m} + n\sqrt{(k+\sqrt[3]{k^2q^2\log m})\log m}\right)$$
$$= O\left(n\sqrt[3]{kq\log^2 m} + n\sqrt{k\log m}\right)$$

**Combined result**

If $q < k^2$ then we can use the algorithm of section 3.2.4, which runs in $O(n\sqrt{(q+k)\log m})$ time. Otherwise, if $q > k^2$, we use the algorithm of section 3.2.4, which runs in $O(n\sqrt[3]{qk\log^2 m} + n\sqrt{k\log m})$ time. Thus we have the following:

**Theorem 8.** *Pattern matching with k mismatches, with wild card symbols in the pattern, can be solved in* $O\left(n\sqrt{k\log m} + n\min\{\sqrt{q\log m}, \sqrt[3]{qk\log^2 m}\}\right)$ *time.*

### 3.2.5 Approximate Counting of Mismatches

The algorithm of [46] takes as input a text $T = t_1t_2\ldots t_n$ and a pattern $P = p_1p_2\ldots p_m$ and approximately counts the Hamming distance between $T_i$ and $P$ for every $1 \le i \le (n-m+1)$. In particular, if the Hamming distance between $T_i$ and $P$ is $H_i$ for some $i$, then the algorithm outputs $h_i$ where $H_i \le h_i \le (1+\epsilon)H_i$ for any $\epsilon > 0$ with high probability (i.e., a probability of $\ge (1-m^{-\alpha})$). The run time of the algorithm is $O(n\log^2 m/\epsilon^2)$. In this section we show how to extend this algorithm to the case where there could be wild cards in the text and/or the pattern.

Let $\Sigma$ be the alphabet under concern and let $\sigma = |\Sigma|$. The algorithm runs in phases and in each phase we randomly map the elements of $\Sigma$ to $\{1,2\}$. A wild card is mapped to a zero. Under this mapping we transform $T$ and $P$ to $T'$ and $P'$, respectively. We then compute a vector $C$ where $C[i] = \sum_{j=1}^{m}(t'_{i+j-1} - p'_j)^2 t'_{i+j-1}p'_j$. This can be done using $O(1)$ convolution operations (as in Section 3.2.6; see also [15]). A series of $r$ such phases (for some relevant value of $r$) is done at the end of which we produce estimates on the Hamming distances. The intuition is that if a character $x$ in $T'$ is aligned with a character $y$ in $P'$, then across all the $r$ phases, the expected contribution to $C$ from these characters is $r$ if $x \ne y$ (assuming that $x$ and $y$ are non-wild cards). If $x = y$ or if one or both of $x$ and $y$ are a wild card, the contribution to $C$ is zero.

**Analysis:** Let $x$ be a character in $T$ and let $y$ be a character in $P$. Clearly, if $x = y$ or if one or both of $x$ and $y$ are a wild card, the contribution of $x$ and $y$ to any $C_\ell[i]$ is zero. If $x$ and $y$ are non-wild

**Algorithm 13:** Approximate Counting of Mismatches

1. **for** $i \leftarrow 1$ **to** $(n-m+1)$ **do** $C[i] = 0$;
2. **for** $\ell \leftarrow 1$ **to** $r$ **do**
   - Let $Q$ be a random mapping of $\Sigma$ to $\{1, 2\}$.
   - In particular, each element of $\Sigma$ is mapped to 1 or 2 randomly with equal probability.
   - Each wild card is mapped to a zero.
   - Obtain two strings $T'$ and $P'$ where $t'_i = Q(t_i)$ for $1 \leq i \leq n$ and $p'_j = Q(p_j)$ for $1 \leq j \leq m$;
   - Compute a vector $C_\ell$ where $C_\ell[i] = \sum_{j=1}^{m} (t'_{i+j-1} - p'_j)^2 \; t'_{i+j-1} p'_j$ for $1 \leq i \leq (n-m+1)$;
   - **for** $i \leftarrow 1$ **to** $(n-m+1)$ **do** $C[i] = C[i] + C_\ell[i]$ ;
3. **for** $i \leftarrow 1$ **to** $(n-m+1)$ **do**
   - Output $h_i = \frac{C[i]}{r}$;
   - Here $h_i$ is an estimate on the Hamming distance $H_i$ between $T_i$ and $P$.

cards and if $x \neq y$ then the expected contribution of these to any $C_\ell[i]$ is 1. Across all the $r$ phases, the expected contribution of $x$ and $y$ to any $C_\ell[i]$ is $r$. For a given $x$ and $y$, we can think of each phase as a Bernoulli trial with equal probabilities for success and failure. A success refers to the possibility of $Q(x) \neq Q(y)$. The expected number of successes in $r$ phases is $\frac{r}{2}$. Using Chernoff bounds (Equation 3.2), this contribution is no more than $(1+\epsilon)r$ with probability $\geq 1 - \exp(-\epsilon^2 r/6)$. Probability that this statement holds for every pair $(x, y)$ is $\geq 1 - m^2 \exp(-\epsilon^2 r/6)$. This probability will be $\geq 1 - m^{-\alpha}/2$ if $r \geq \frac{6(\alpha+3)\log_e m}{\epsilon^2}$. Similarly, we can show that for any pair of non-wild card characters, the contribution of them to any $C_\ell[i]$ is no less than $(1-\epsilon)r$ with probability $\geq 1 - m^{-\alpha}/2$ if $r \geq \frac{4(\alpha+3)\log_e m}{\epsilon^2}$.

Put together, for any pair $(x, y)$ of non-wild cards, the contribution of $x$ and $y$ to any $C_\ell[i]$ is in the interval $(1 \pm \epsilon)r$ with probability $\geq (1 - m^{-\alpha})$ if $r \geq \frac{6(\alpha+3)\log_e m}{\epsilon^2}$. Let $H_i$ be the Hamming distance between $T_i$ and $P$ for some $i$ $(1 \leq i \leq (n-m+1))$. Then, the estimate $h_i$ on $H_i$ will be in the interval $(1 \pm \epsilon)H_i$ with probability $\geq (1 - m^{-\alpha})$. As a result, we get the following Theorem.

**Theorem 9.** *Given a text $T$ and a pattern $P$, we can estimate the Hamming distance between $T_i$ and $P$, for every $i$, $1 \leq i \leq (n-m+1)$, in $O(n \log^2 m/\epsilon^2)$ time. If $H_i$ is the Hamming distance between $T_i$ and $P$, then the above algorithm outputs an estimate that is in the interval $(1 \pm \epsilon)H_i$ with high probability.*

**Observation 1.** In the above algorithm we can ensure that $h_i \geq H_i$ and $h_i \leq (1+\epsilon)H_i$ with high probability by changing the estimate computed in step 3 of **Algorithm 13** to $\frac{C[i]}{(1-\epsilon)r}$.

**Observation 2.** As in [46], with $O\left(\frac{m^2 \log m}{\epsilon^2}\right)$ pre-processing we can ensure that **Algorithm 13** never errs (i.e., the error bounds on the estimates will always hold).

### 3.2.6 A Las Vegas Algorithm for $k$-Mismatches

#### The 1-Mismatch Problem

**Problem Definition:** For this problem also, the input are two strings $T$ and $P$ with $|T| = n, |P| = m$, $m \leq n$, and possible wild cards in $T$ and $P$. Let $T_i$ stand for the substring $t_i t_{i+1} \ldots t_{i+m-1}$, for any $i$, with $1 \leq i \leq (n-m+1)$. The problem is to check if the Hamming distance between $T_i$ and $P$ is exactly 1, for $1 \leq i \leq (n-m+1)$. The following Lemma is shown in [15].

**Lemma 6.** *The 1-mismatch problem can be solved in $O(n \log m)$ time using a constant number of convolution operations.*

**The Algorithm:** Assume that each wild card in the pattern as well as the text is replaced with a zero. Also, assume that the characters in the text as well as the pattern are integers in the range $[1 : |\Sigma|]$ where $\Sigma$ is the alphabet under concern. Let $e_{i,j}$ stand for the "error term" introduced by the character $t_{i+j-1}$ in $T_i$ and the character $p_j$ in $P$ and its value is $(t_{i+j-1} - p_j)^2 t_{i+j-1} p_j$. Also, let $E_i = \sum_{j=1}^{m} e_{i,j}$. There are four steps in the algorithm:

1. Compute $E_i$ for $1 \leq i \leq (n-m+1)$. Note that $E_i$ will be zero if $T_i$ and $P$ match (assuming that a wild card can be matched with any character). $E_i = \sum_{j=1}^{m} (t_{i+j-1} - p_j)^2 t_{i+j-1} p_j = \sum_{j=1}^{m} t_{i+j-1}^3 p_j + \sum_{j=1}^{m} t_{i+j-1} p_j^3 - 2 \sum_{j=1}^{m} t_{i+j-1}^2 p_j^2$. Thus this step can be completed with three convolution operations.

2. Compute $E'_i$ for $1 \leq i \leq (n-m+1)$, where $E'_i = \sum_{j=1}^{m} (i+j-1)(t_{i+j-1} - p_j)^2 p_j t_{i+j-1}$ (for $1 \leq i \leq (n-m+1)$). Like step 1, this step can also be completed with three convolution operations.

3. Let $B_i = E'_i / E_i$ if $E_i \neq 0$, for $1 \leq i \leq (n-m+1)$. Note that if the Hamming distance between $T_i$ and $P$ is exactly one, then $B_i$ will give the position in the text where this mismatch occurs.

4. If for any $i$ $(1 \leq i \leq (n-m+1))$, $E_i \neq 0$ and if $(t_{B_i} - p_{B_i - i + 1})^2 t_{B_i} p_{B_i - i + 1} = E_i$ then we conclude that the Hamming distance between $T_i$ and $P$ is exactly one.

**Note:** If the Hamming distance between $T_i$ and $P$ is exactly 1 (for any $i$), then the above algorithm will not only detect it but also identify the position where there is a mismatch. Specifically, it will identify the integer $j$ such that $t_{i+j-1} \neq p_j$.

**An Example.** Consider the case where $\Sigma = \{1, 2, 3, 4, 5, 6\}$, $T = 5\ 6\ 4\ 6\ 2\ *\ 3\ 3\ 4\ 5\ 1\ *\ 1\ 2\ 5\ 5\ 5\ 6\ 4\ 3$, and $P = 2\ 5\ 6\ 3$. Here $*$ represents the wild card.

In step 1 we compute $E_i$, for $1 \leq i \leq 17$. For example, $E_1 = (5-2)^2 \times 5 \times 2 + (6-5)^2 \times 6 \times 5 + (4-6)^2 \times 4 \times 6 + (6-3)^2 \times 6 \times 3 = 378$; $E_2 = (6-2)^2 \times 6 \times 2 + (4-5)^2 \times 4 \times 5 + (6-6)^2 \times 6 \times 6 + (2-3)^2 \times 2 \times 3 = 218$; $E_3 = 254$; $E_5 = (2-2)^2 \times 2 \times 2 + 0 + (6-3)^2 \times 6 \times 3 + (3-3)^2 \times 3 \times 3 = 162$. Note that since $t_5$ is a wild card, it matches with any character in the pattern. Also, $E_9 = 182$.

In step 2 we compute $E'_i$, for $1 \leq i \leq 17$. For instance, $E'_1 = 1 \times (5-2)^2 \times 5 \times 2 + 2 \times (6-5)^2 \times 6 \times 5 + 3(4-6)^2 \times 4 \times 6 + 4 \times (6-3)^2 \times 6 \times 3 = 1410$; $E'_5 = 5 \times (2-2)^2 \times 2 \times 2 + 0 + 7 \times (6-3)^2 \times 6 \times 3 + 8 \times (3-3)^2 \times 3 \times 3 = 1134$.

In step 3, the value of $B_i = E'_i/E_i$ is computed for $1 \leq i \leq 17$. For example, $B_1 = E'_1/E_1 = 1410/378 \approx 3.73$; $B_5 = E'_5/E_5 = 1134/162 = 7$.

In step 4, we identify all the positions in the text corresponding to a single mismatch. For instance, we note that $E_5 \neq 0$ and $(t_7 - p_3)^2 \times t_7 \times p_3 = E_5$. As a result, position 5 in the text corresponds to 1-mismatch.

**The Randomized Algorithms of [15]**

Two different randomized algorithms are presented in [15] for solving the $k$-mismatches problem. Both are Monte Carlo algorithms. In particular, they output the correct answers with high probability. The run times of these algorithms are $O(nk \log m \log n)$ and $O(n \log m(k + \log n \log \log n))$, respectively. In this section we provide a summary of these algorithms.

The first algorithm has $O(k \log n)$ sampling phases and in each phase a 1-mismatch problem is solved. Each phase of sampling works as follows. We choose $m/k$ positions of the pattern uniformly at random. The pattern $P$ is replaced by a string $P'$ where $|P'| = m$, the characters in $P'$ in the randomly chosen positions are the same as those in the corresponding positions of $P$, and the rest of the characters in $P'$ are set to wild cards. The 1-mismatch algorithm of Lemma 6 is run on $T$ and $P'$. In each phase of random sampling, for each $i$, we get to know if the Hamming distance between $T_i$ and $P'$ is exactly 1 and, if so, identify the $j$ such that $t_{i+j-1} \neq p'_j$.

As an example, consider the case when the Hamming distance between $T_i$ and $P$ is $k$ (for some $i$). Then, in each phase of sampling we would expect to identify exactly one of the positions (i.e., $j$) where $T_i$ and $P$ differ (i.e., $t_{i+j-1} \neq p_j$). As a result, in an expected $k$ phases of sampling we will be able to identify all the $k$ positions in which $T_i$ and $P$ differ. It can be shown that if we make $O(k \log n)$ sampling phases, then we can identify all the $k$ mismatches with high probability [15]. It is possible that the same $j$ might be identified in multiple phases. However we can easily keep track of this information to identify the unique $j$ values found in all of the phases.

Let the number of mismatches between $T_i$ and $P$ be $q_i$ (for $1 \leq i \leq (n - m + 1)$. If $q_i \leq k$, the

algorithm of [15] will compute $q_i$ exactly. If $q_i > k$, then the algorithm will report that the number of mismatches is $> k$ (without estimating $q_i$) and this answer will be correct with high probability. The algorithm starts off by first computing $E_i$ values for every $T_i$. A list $L(i)$ of all the mismatches found for $T_i$ is kept, for every $i$. Whenever a mismatch is found between $T_i$ and $P$ (say in position $(i+j-1)$ of the text), the value of $E_i$ is reduced by $e_{i,j}$. If at any point in the algorithm $E_i$ becomes zero for any $i$ it means that we have found all the $q_i$ mismatches between $T_i$ and $P$ and $L(i)$ will have the positions in the text where these mismatches occur. Note that if the Hamming distance between $T_i$ and $P$ is much larger than $k$ (for example close or equal to $m$), then the probability that in a random sample we isolate a single mismatch is very low. Therefore, if the number of sample phases is only $O(k \log n)$, the algorithm can only be Monte Carlo. Even if $q_i$ is less or equal to $k$, there is a small probability that we may not be able to find all the $q_i$ mismatches. Call this algorithm **Algorithm 14**. If for each $i$, we either get all the $q_i$ mismatches (and hence the corresponding $E_i$ is zero) or we have found more than $k$ mismatches between $T_i$ and $P$ then we can be sure that we have found all the correct answers (and the algorithm will become Las Vegas).

**An Example.** Consider the example of $\Sigma = \{1,2,3,4,5,6\}$, $T = 5\ 6\ 4\ 6\ 2\ *\ 3\ 3\ 4\ 5\ 1\ *\ 1\ 2\ 5\ 5\ 5\ 6\ 4\ 3$, and $P = 2\ 5\ 6\ 3$. Here $*$ represents the wild card.

As has been computed before, $E_5 = 162$ and $E_9 = 182$. Let $k = 2$. In each phase we choose 2 random positions of the pattern.

In the first phase let the two positions chosen be 2 and 3. In this case $P' = *\ 5\ 6\ *$. We run the 1-mismatch algorithm with $T$ and $P'$. At the end of this phase we realize that $t_3 \neq p_2; t_5 \neq p_2; t_7 \neq p_3; t_{11} \neq p_2; t_{11} \neq p_3; t_{13} \neq p_3; t_{16} \neq p_3$; and $t_{17} \neq p_3$. The corresponding $E_i$ values will be decremented by $e_{i,j}$ values. Specifically, if $t_i \neq p_j$ then $E_{i-j+1}$ is decremented by $e_{i,j}$. For example, since $t_7 \neq p_3$, we decrement $E_5$ by $e_{7,3} = (6-3)^2 \times 6 \times 3 = 162$. $E_5$ becomes zero and hence $T_5$ is output as a correct answer. Likewise since $t_{11} \neq p_3$, we decrement $E_9$ by $e_{11,3} = (1-6)^2 \times 1 \times 6 = 150$. Now $E_9$ becomes 32.

In the second phase let the two positions chosen be 1 and 2. In this case $P' = 2\ 5\ *\ *$. At the end of this phase we learn that $t_7 \neq p_2; t_9 \neq p_1; t_{11} \neq p_1; t_{13} \neq p_2; t_{15} \neq p_1; t_{16} \neq p_1$. Here again relevant $E_i$ values are decremented. For instance, since $t_9 \neq p_1$, $E_9$ is decremented by $e_{9,1} = (4-2)^2 \times 4 \times 2 = 32$. The value of $E_9$ now becomes zero and hence $T_9$ is output as a correct answer; and so on.

If the distance between $T_i$ and $P$ (for some $i$) is $\leq k$, then out of all the phases attempted, there is a high probability that all of these mismatches between $T_i$ and $P$ will be identified.

The authors of [15] also present an improved algorithm whose run time is $O(n \log m(k + \log n \log\log n))$. The main idea is the observation that if $q_i = k$ for any $i$, then in $O(k \log n)$ sampling steps we can identify

$\geq k/2$ mismatches. There are several iterations where in each iteration $O(k + \log n)$ sampling phases are done. At the end of each iteration the value of $k$ is changed to $k/2$. Let this algorithm be called **Algorithm 15**.

### A Las Vegas Algorithm

In this section we present a Las Vegas algorithm for the $k$-mismatches problem when there are wild cards in the text and/or the pattern. This algorithm runs in time $\widetilde{O}(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log \log n)$. This algorithm is based on the algorithm of [15]. When the algorithm terminates, for each $i$ $(1 \leq i \leq (n - m + 1))$, either we would have identified all the $q_i$ mismatches between $T_i$ and $P$ or we would have identified more than $k$ mismatches between $T_i$ and $P$.

**Algorithm 14** will be used for every $i$ for which $q_i \leq 2k$. For every $i$ for which $q_i > 2k$ we use the following strategy. Let $2^\ell k < q_i \leq 2^{\ell+1}k$ (where $1 \leq \ell \leq \log\left(\lfloor \frac{m}{2k} \rfloor\right)$). Let $w = \log\left(\lfloor \frac{m}{2k} \rfloor\right)$. There will be $w$ phases in the algorithm and in each phase we perform $O(k)$ sampling steps. Each sampling step in phase $\ell$ involves choosing $\frac{m}{2^{\ell+1}k}$ positions of the pattern uniformly at random (for $1 \leq \ell \leq w$). As we show below, if for any $i$, $q_i$ is in the interval $[2^\ell, 2^{\ell+1}]$, then at least $k$ mismatches between $T_i$ and $P$ will be found in phase $\ell$ with high probability. A pseudocode for the algorithm is given in **Algorithm 16**.

**Algorithm 16:** Las Vegas Algorithm for $k$ Mismatches

```
while true do
1.    Run Algorithm 14 or Algorithm 15;
2.    for ℓ ← 1 to w do
          for r ← 1 to ck (c being a constant) do
              Uniformly randomly choose m/2^{ℓ+1}k positions of the pattern;
              Generate a string P' such that |P'| = |P| and P' has the same characters as P in these
              randomly chosen positions and zero everywhere else;
              Run the 1-mismatch algorithm on T and P';
              As a result, if there is a single mismatch between T_i and P' then add the position of
              mismatch to L(i) and reduce the value of E_i by the right amount, for
              1 ≤ i ≤ (n − m + 1);
3.    if either E_i = 0 or |L(i)| > k for every i, 1 ≤ i ≤ (n − m + 1) then quit;
```

**Theorem 10. Algorithm 16** *runs in time* $\widetilde{O}(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log \log n)$ *if* **Algorithm 15** *is used in step 1. It runs in time* $\widetilde{O}(nk \log m \log n + nk \log^2 m + n \log^2 m \log n)$ *if step 1 uses* **Algorithm 14**.

*Proof.* As shown in [15], the run time of **Algorithm 14** is $O(nk \log m \log n)$ and that of **Algorithm 15**

is $O(n \log m(k + \log n \log\log n))$. The analysis will be done with respect to an arbitrary $T_i$. In particular, we will show that after the specified amount of time, with high probability, we will either know $q_i$ or realize that $q_i > k$. It will then follow that the same statement holds for every $T_i$ (for $1 \leq i \leq (n-m+1)$.

Consider phase $\ell$ of step 2 (for an arbitrary $1 \leq \ell \leq w$). Let $2^\ell k < q_i \leq 2^{\ell+1}k$ for some $i$. Using the fact that $\binom{a}{b} \approx \left(\frac{ae}{b}\right)^b$, the probability of isolating one of the mismatches in one run of the sampling step is:

$$\frac{\binom{m-q_i}{m/(2^{\ell+1}k)-1} q_i}{\binom{m}{m/(2^{\ell+1}k)}} \geq \frac{\binom{m-2^{\ell+1}k}{m/(2^{\ell+1}k)-1} 2^\ell k}{\binom{m}{m/(2^{\ell+1}k)}} \geq \frac{1}{2e}$$

As a result, using Chernoff bounds (Equation 3.3 with $\delta = 1/2$, for example), it follows that if $13ke$ sampling steps are made in phase $\ell$, then at least $6k$ of these steps will result in the isolation of single mismatches (not all of them need be distinct) with high probability (assuming that $k = \Omega(\log n)$). Moreover, we can see that at least $1.1k$ of these mismatches will be distinct. This is because the probability that $\leq 1.1k$ of these are distinct is $\leq \binom{q_i}{1.1k} / \left(\frac{1.1k}{q_i}\right)^{6k} \leq 2^{-2.64k}$ using the fact that $q_i \geq 2k$. This probability will be very low when $k = \Omega(\log n)$.

In the above analysis we have assumed that $k = \Omega(\log n)$. If this is not the case, in any phase of step 2, we can do $c\alpha \log n$ sampling steps, for some suitable constant $c$. In this case also we can perform an analysis similar to that of the above case using Chernoff bounds. Specifically, we can show that with high probability we will be able to identify all the mismatches between $T_i$ and $P$. As a result, each phase of step 2 takes $O(n \log m(k + \log n))$ time. We have $O(\log m)$ phases. Thus the run time of step 2 is $O(n \log^2 m(k + \log n))$. Also, the probability that the condition in step 3 holds is very high.

Therefore, the run time of the entire algorithm is $\widetilde{O}(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log\log n)$ if **Algorithm 15** is used in step 1 or $\widetilde{O}(nk \log m \log n + nk \log^2 m + n \log^2 m \log n)$ if **Algorithm 14** is used in step 1. □

## 3.3 Results

The above algorithms are based on symbol comparison, arithmetic operations, or a combination of both. Therefore, it is interesting to see how these algorithms compare in practice.

In this section we compare deterministic algorithms for pattern matching. Some of these algorithms solve the pattern matching with mismatches problem, others solve the $k$-mismatches problem. For the

sake of comparison, we treated all of them as algorithms for the $k$-mismatches problem, which is a special case of the pattern matching with mismatches problem.

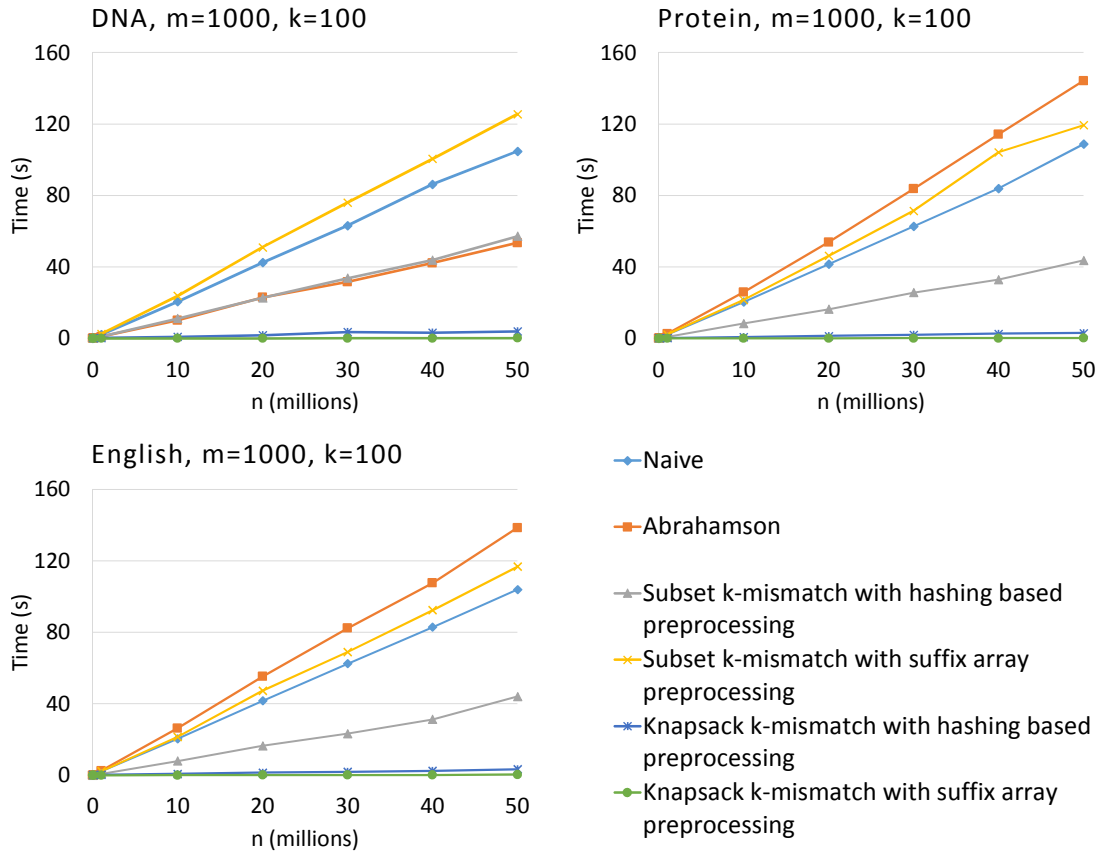
We implemented the following algorithms: the naive $O(nm)$ time algorithm, Abrahamson's algorithm [2], Subset $k$-mismatches (section 3.2.3) and Knapsack $k$-mismatches (section 3.2.3). For Subset $k$-mismatches, we simulate the suffix tree and LCA extensions by a suffix array with an LCP (Longest Common Prefix [47]) table and data structures to perform RMQ queries (Range Minimum Queries [9]) on it. This adds a $O(\log n)$ factor to preprocessing. For searching in the suffix array we use a simple forward traversal with cost of $O(\log n)$ per character. The traversal uses binary search to find the interval of suffixes that start with the first character of the pattern. Then, another binary search is performed to find the suffixes that start with the first two characters of the pattern, and so on. However, more efficient implementations are possible (e.g., [33]). For Subset $k$-mismatches, we also tried a simple $O(m^2)$ time pre-processing using dynamic programming to precompute LCPs and hashing to quickly determine whether a portion of the text is present in the pattern. This method takes more preprocessing time, but it does not have the $O(\log n)$ factor when searching. Knapsack $k$-mismatches uses Subset $k$-mismatches as a subroutine, so we have two versions of it also.

We tested the algorithms on protein, DNA and English inputs generated randomly. We randomly selected a substring of length $m$ from the text and used it as pattern. The algorithms were tested on an Intel Core i7 machine with 8GB of RAM, Linux Mint 17.1 Operating System and gcc 4.8.2. All convolutions were performed using the fftw [36] library version 3.3.3. We used the suffix array algorithm RadixSA of [77].

Figure 3.1 shows run times for varying the length of the text $n$. All algorithms scale linearly with the length of the text. Figure 3.2 shows run times for varying the length of the pattern $m$. Abrahamson's algorithm is expensive because, for alphabet sizes smaller than $\sqrt{m/\log m}$, it computes one convolution for every character in the alphabet. The convolutions proved to be expensive in practice, so Abrahamson's algorithm was competitive only for DNA data where the alphabet is small. Figure 3.3 shows runtimes for varying the maximum number of mismatches $k$ allowed. The naive algorithm and Abrahamson's algorithm do not depend on $k$, therefore their runtime is constant. Subset $k$-mismatch, with its $O(nk)$ runtime, is competitive for relatively small $k$. Knapsack $k$-mismatch, on the other hand, scaled very well with $k$. Figure 3.4 shows runtimes for varying the alphabet from 4 (DNA) to 20 (protein) to 26 (English). As expected, Abrahamson's algorithm is the most sensitive to the alphabet size.

Overall, the naive algorithm performed well in practice most likely due to its simplicity and cache locality. Abrahamson's algorithm was competitive only for small alphabet size or for large $k$. Subset

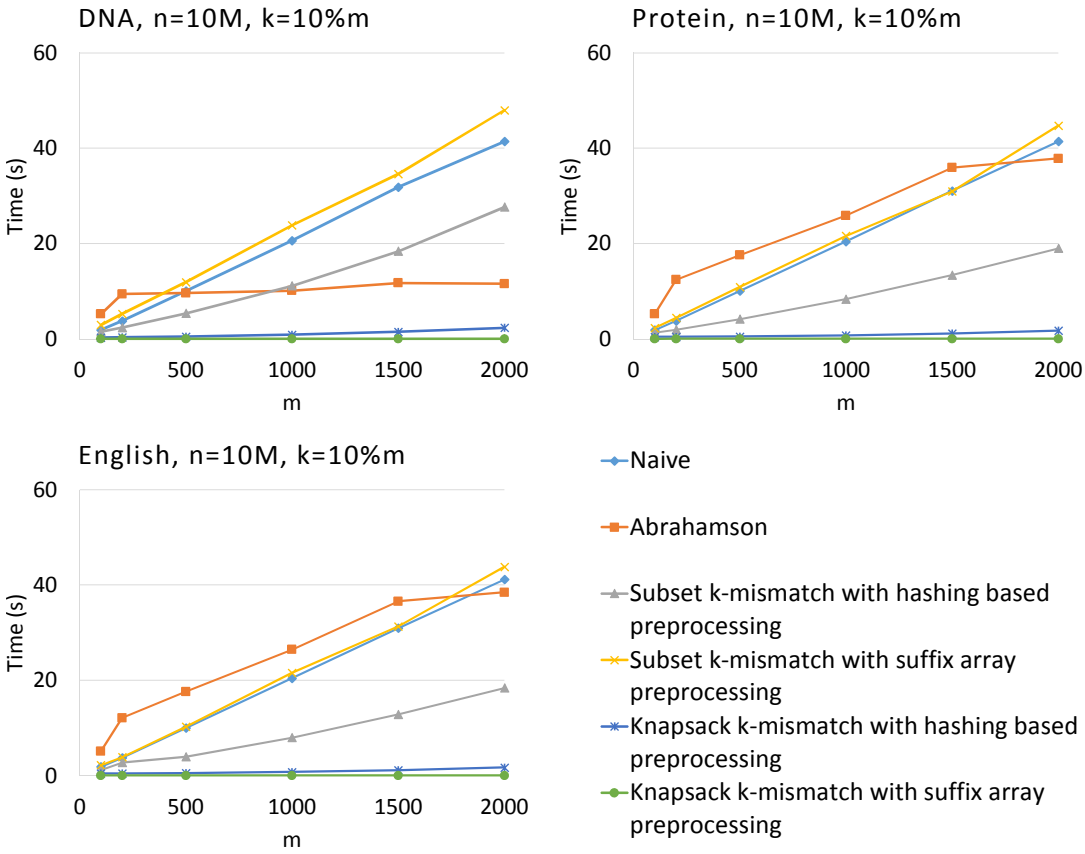
Figure 3.1: Run times for pattern matching when the length of the text varies.




Run times for pattern matching on DNA, protein and English alphabet data, when the length of the text ($n$) varies. The length of the pattern is $m = 1000$. The maximum number of mismatches allowed is $k = 100$. Our algorithms are Subset k-mismatch with hashing based preprocessing (section 3.2.3), Subset k-mismatch with suffix array preprocessing (section 3.2.3), Knapsack k-mismatch with hashing based preprocessing (section 3.2.3), and Knapsack k-mismatch with suffix array preprocessing (section 3.2.3).

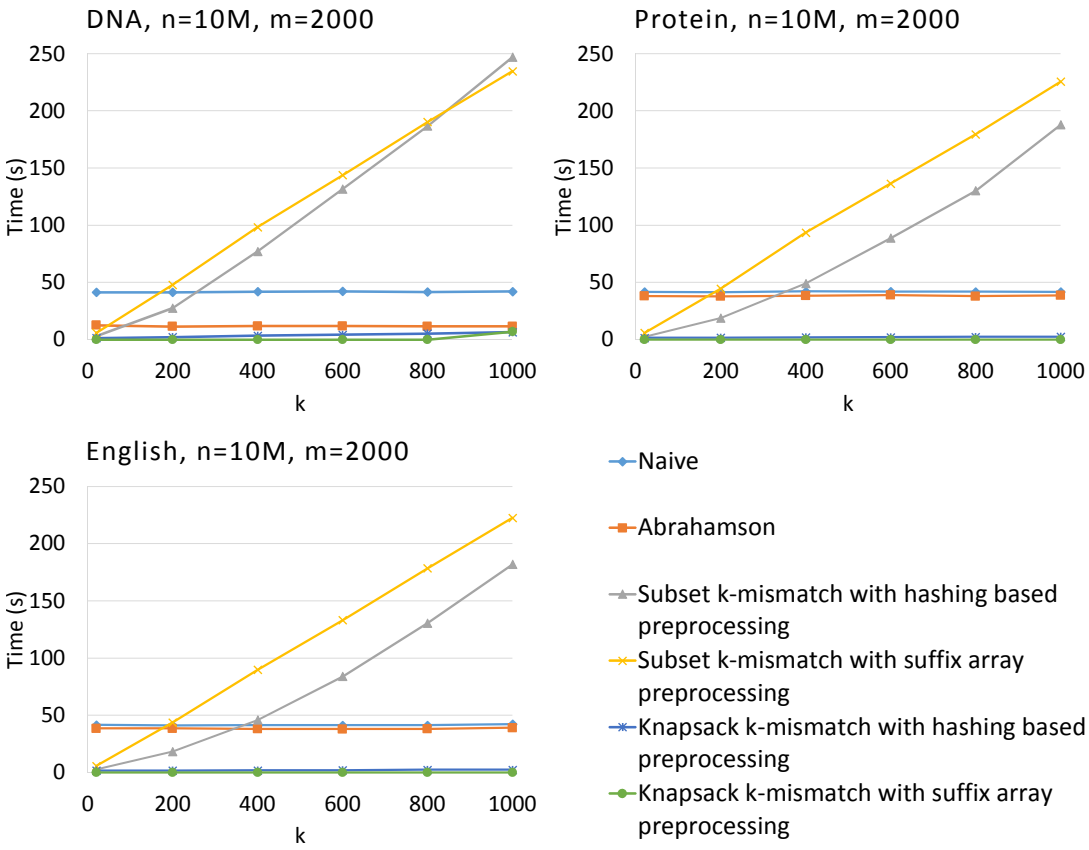
$k$-mismatches performed well for relatively small $k$. In most cases, the suffix array version was slower than the hashing based one with $O(m^2)$ time pre-processing because of the added $O(\log n)$ factor when searching in the suffix array. It would be interesting to investigate how the algorithms compare with a more efficient implementation of the suffix array. Knapsack $k$-mismatches was the fastest among the algorithms compared because, in most cases, the knapsack could be filled with less than the given "budget" and thus the algorithm did not have to perform any convolution operations.

Figure 3.2: Run times for pattern matching when the length of the pattern varies.



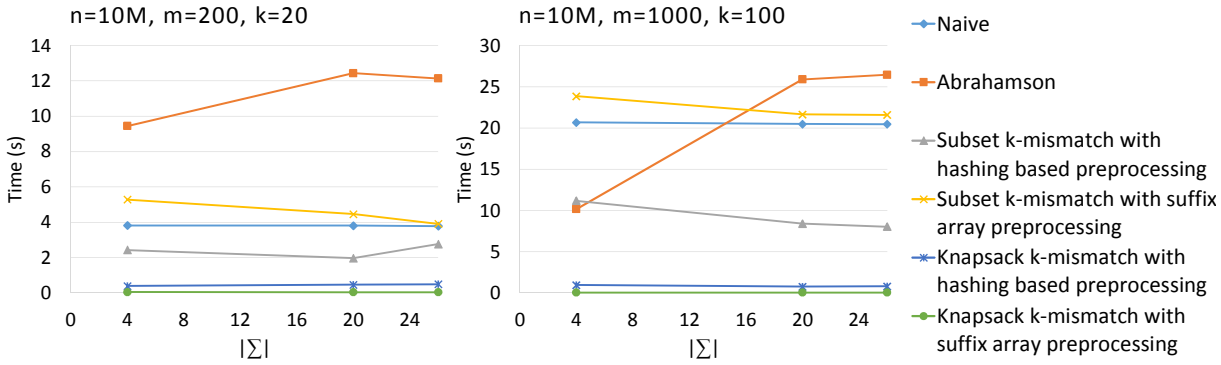

Run times for pattern matching on DNA, protein and English alphabet data, when the length of the pattern ($m$) varies. The length of the text is $n = 10$ millions. The maximum number of mismatches allowed is $k = 10\%$ of the pattern length. Our algorithms are Subset k-mismatch with hashing based preprocessing (section 3.2.3), Subset k-mismatch with suffix array preprocessing (section 3.2.3), Knapsack k-mismatch with hashing based preprocessing (section 3.2.3), and Knapsack k-mismatch with suffix array preprocessing (section 3.2.3).

Figure 3.3: Run times for pattern matching when the maximum number of mismatches allowed varies.




Run times for pattern matching on DNA, protein and English alphabet data, when the maximum number of mismatches allowed ($k$) varies. The length of the text is $n = 10$ millions. The length of the pattern is $m = 2000$. Our algorithms are Subset k-mismatch with hashing based preprocessing (section 3.2.3), Subset k-mismatch with suffix array preprocessing (section 3.2.3), Knapsack k-mismatch with hashing based preprocessing (section 3.2.3), and Knapsack k-mismatch with suffix array preprocessing (section 3.2.3).

Figure 3.4: Run times for pattern matching when the size of the alphabet varies.




Run times for pattern matching when the size of the alphabet varies from 4 (DNA) to 20 (protein) to 26 (English). The length of the text is $n = 10$ millions. The length of the pattern is $m = 200$ in the first graph and $m = 1000$ in the second. The maximum number of mismatches allowed is $k = 20$ in the first graph and $k = 100$ in the second. Our algorithms are Subset k-mismatch with hashing based preprocessing (section 3.2.3), Subset k-mismatch with suffix array preprocessing (section 3.2.3), Knapsack k-mismatch with hashing based preprocessing (section 3.2.3), and Knapsack k-mismatch with suffix array preprocessing (section 3.2.3).

## 3.4 Conclusions

We have introduced several deterministic and randomized, exact and approximate algorithms for pattern matching with mismatches and the $k$-mismatches problems, with or without wild cards. These algorithms improve the run time, simplify, or extend previous algorithms wild cards. We have also implemented some of the deterministic algorithms. An empirical comparison of these algorithms showed that the algorithms based on character comparison outperform those based on convolutions.

# Bibliography


[1] Mostafa Abbas, Mohamed Abouelhoda, and Hazem Bahig. A hybrid method for the exact planted (l, d) motif finding problem and its parallelization. *BMC Bioinformatics*, 13(Suppl 17):S10, 2012.

[2] Karl Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[3] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[4] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. On finding lowest common ancestors in trees. *SIAM Journal on computing*, 5(1):115–132, 1976.

[5] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.

[6] M. J. Atallah, F. Chyzak, and P. Dumas. A randomized algorithm for approximate string matching. *Algorithmica*, 29(3):468–486, 2001.

[7] S. Bandyopadhyay, S. Sahni, and S. Rajasekaran. Pms6: A fast algorithm for motif discovery. In *IEEE 2nd International Conference on Computational Advances in Bio and Medical Sciences, ICCABS 2012, Las Vegas, NV, USA, February 23-25, 2012*, pages 1–6. IEEE, 2012.

[8] D. Baron and Y. Bresler. Antisequential suffix sorting for bwt-based data compression. *IEEE Transactions on Computers*, 54(4):385–397, April 2005.

[9] Michael Bender and Martn Farach-Colton. The lca problem revisited. In Gaston Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer Berlin / Heidelberg, 2000.

[10] Jeremy Buhler and Martin Tompa. Finding motifs using random projections. *J. Comp. Biol.*, 9(2):225–242, 2002.

[11] Stefan Burkhardt and Juha Krkkinen. Fast lightweight suffix array construction and checking. In Ricardo Baeza-Yates, Edgar Chvez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer Berlin / Heidelberg, 2003.

[12] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[13] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 2:241–256, 1952.

[14] Peter Clifford and Raphal Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53 – 54, 2007.

[15] R. Clifford, K. Efremenko, E. Porat, and A. Rothschild. k-mismatch with don't cares. *Algorithms–ESA 2007*, pages 151–162, 2007.

[16] Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. In *Combinatorial Pattern Matching*, pages 143–151. Springer-Verlag, 2008.

[17] Raphaël Clifford, Klim Efremenko, Benny Porat, Ely Porat, and Amir Rothschild. Mismatch sampling. *Inf. Comput.*, 214:112–118, May 2012.

[18] Raphael Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. From coding theory to efficient pattern matching. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 778–784, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[19] Raphaël Clifford and Ely Porat. A filtering algorithm for k-mismatch with dont cares. In *String Processing and Information Retrieval*, pages 130–136. Springer, 2007.

[20] Raphaël Clifford and Ely Porat. A filtering algorithm for k-mismatch with don't cares. *Information Processing Letters*, 110(22):1021–1025, 2010.

[21] Raphal Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. Pattern matching with don't cares and few errors. *Journal of Computer and System Sciences*, 76(2):115 – 124, 2010.

[22] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[23] Richard Cole, Ramesh Hariharan, and Piotr Indyk. Tree pattern matching and subset matching in deterministic o(n log3 n)-time. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 245–254, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[24] Naga Shailaja Dasari, Ranjan Desh, and Zubair M. An efficient multicore implementation of planted motif problem. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010, June 28 - July 2, 2010, Caen, France*, pages 9 –15. IEEE, 28 2010-july 2 2010.

[25] N.S. Dasari, R. Desh, and M. Zubair. Solving planted motif problem on gpu. In *International Workshop on GPUs and Scientific Applications, GPUScA 2010, Vienna, Austria, September 11, 2010*. Department of Scientific Computing, University of Vienna, TR-10-3, 2010.

[26] N.S. Dasari, D. Ranjan, and M. Zubair. High performance implementation of planted motif problem using suffix trees. In *2011 International Conference on High Performance Computing & Simulation, HPCS 2012, Istanbul, Turkey, July 4-8, 2011*, pages 200 –206. IEEE, july 2011.

[27] J. Davila, S. Balla, and S. Rajasekaran. Fast and practical algorithms for planted $(l, d)$ motif search. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(4):544–552, 2007.

[28] S. Desaraju and R. Mukkamala. Multiprocessor implementation of modeling method for planted motif problem. In *2011 World Congress on Information and Communication Technologies (WICT), Dec 11-14, 2011, Mumbai, India*, pages 524–529. IEEE, 2011.

[29] H. Dinh, S. Rajasekaran, and V. Kundeti. Pms5: an efficient exact algorithm for the $(l, d)$-motif finding problem. *BMC bioinformatics*, 12(1):410, 2011.

[30] Hieu Dinh, Sanguthevar Rajasekaran, and Jaime Davila. qpms7: A fast algorithm for finding $(l, d)$-motifs in dna and protein sequences. *PLoS ONE*, 7(7):e41425, 07 2012.

[31] Eleazar Eskin and Pavel A Pevzner. Finding composite regulatory patterns in dna sequences. *Bioinformatics*, 18(suppl 1):354–363, 2002.

[32] H. M. Faheem. Accelerating motif finding problem using grid computing with enhanced brute force. In *Proceedings of the 12th international conference on Advanced communication technology*, ICACT'10, pages 197–202, Piscataway, NJ, USA, 2010. IEEE Press.

[33] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[34] Michael J Fischer and Michael S Paterson. String-matching and other products. Technical Report MAC-TM-41, Massachusetts Institute of Technology Cambridge Project MAC, Cambridge, MA, USA, 1974.

[35] Kimmo Fredriksson and Szymon Grabowski. Combinatorial algorithms. chapter Fast Convolutions and Their Applications in Approximate String Matching, pages 254–265. Springer-Verlag, Berlin, Heidelberg, 2009.

[36] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[37] Z Galil and R Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, March 1986.

[38] Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Exact solutions for closest string and related problems. In Peter Eades and Tadao Takaoka, editors, *Algorithms and Computation*, volume 2223 of *Lecture Notes in Computer Science*, pages 441–453. Springer Berlin Heidelberg, 2001.

[39] E.S. Ho, C.D. Jakubowski, S.I. Gunderson, et al. itriplet, a rule-based nucleic acid sequence motif finder. *Algorithms for Molecular Biology*, 4(1):14, 2009.

[40] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Silicon Press, 2008.

[41] Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *In Proceedings of the 39th Symposium on Foundations of Computer Science*, pages 166–173, 1998.

[42] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of the sixth Symposium on String Processing and Information Retrieval. IEEE Computer Society, Cancun, Mexico*, pages 81–88, 1999.

[43] C. Kaklamanis, D. Krizanc, L. Narayanan, and T. Tsantilas. Randomized sorting and selection on mesh-connected processor arrays (preliminary version). In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 17–28, New York, NY, USA, 1991. ACM.

[44] Adam Kalai. Efficient pattern-matching with don't cares. In *In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 655–656, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[45] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *ICALP*, pages 943–955, 2003.

[46] Howard Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53 – 60, 1993.

[47] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. pages 181–192. Springer-Verlag, 2001.

[48] D.K. Kim, J. Jo, and H. Park. A fast algorithm for constructing suffix arrays for fixed size alphabets. In *Proceedings of the 3rd Workshop on Experimental and Efficient Algorithms (WEA 2004), C. C. Ribeiro and S. L. Martins, Eds. Springer-Verlag, Berlin*, pages 301–314, 2004.

[49] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *CPM*, pages 186–199, 2003.

[50] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[51] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *CPM*, pages 200–210, 2003.

[52] S. Kurtz. Reducing the space requirement of suffix trees. *Software, Practice and Experience*, 29(13):1149–1171, 1999.

[53] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 370–379, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[54] J.K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland*, pages 633–642. Society for Industrial and Applied Mathematics, ACM/SIAM, 1999.

[55] Gad M. Landau and Uzi Vishkin. Efficient string matching in the presence of errors. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 126 –136. IEEE, oct. 1985.

[56] J.N. Larsson and K. Sadakane. Faster suffix sorting. In *Tech. Rep. LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)/1-20/(1999)], Department of Computer Science, Lund University, Sweden*, 1999.

[57] N.J. Larsson and K. Sadakane. Faster suffix sorting. *Theor. Comput. Sci.*, 387(3):258–272, 2007.

[58] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[59] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[60] Y. Mori. Short description of improved two-stage suffix sorting algorithm `http://homepage3.nifty.com/wpage/software/itssort.txt`, 2005.

[61] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[62] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[63] Marius Nicolae and Sanguthevar Rajasekaran. Efficient sequential and parallel algorithms for planted motif search. *BMC bioinformatics*, 15(1):34, 2014.

[64] Marius Nicolae and Sanguthevar Rajasekaran. On string matching with mismatches. *Algorithms*, 8(2):248–270, 2015.

[65] Marius Nicolae and Sanguthevar Rajasekaran. qpms9: An efficient algorithm for quorum planted motif search. *Scientific reports*, 5, 2015.

[66] Marius Nicolae and Sanguthevar Rajasekaran. On pattern matching with k mismatches and few don't cares. *CoRR*, abs/1602.00621, 2016.

[67] G. Nong, S. Zhang, and W.H. Chan. Linear suffix array construction by almost pure induced-sorting. *Data Compression Conference*, 0:193–202, 2009.

[68] S. Osiski and D. Weiss. jsuffixarrays: Suffix arrays for java, `http://labs.carrotsearch.com/jsuffixarrays.html`, 2002-2011.

[69] P.A. Pevzner, S.H. Sze, et al. Combinatorial approaches to finding subtle signals in dna sequences. In *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology, August 19-23, 2000, La Jolla / San Diego, CA, USA*, volume 8, pages 269–278. AAAI, 2000.

[70] B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS '09. 50th Annual IEEE Symposium on*, pages 315–323, Oct 2009.

[71] Ely Porat and Ohad Lipsky. Improved sketching of hamming distance with error correcting. In *Combinatorial Pattern Matching*, pages 173–182. Springer, 2007.

[72] Alkes Price, Sriram Ramabhadran, and Pavel A Pevzner. Finding subtle motifs by branching from sample strings. *Bioinformatics*, 19(suppl 2):149–155, 2003.

[73] S.J. Puglisi, W.F. Smyth, and A.H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), July 2007.

[74] S. Rajasekaran and S. Sen. Optimal and practical algorithms for sorting on the pdm. *IEEE Trans. Computers*, 57(4):547–561, 2008.

[75] Sanguthevar Rajasekaran, Sudha Balla, and C-H Huang. Exact algorithms for planted motif problems. *J. Comp. Biol.*, 12(8):1117–1128, 2005.

[76] Sanguthevar Rajasekaran and Hieu Dinh. A speedup technique for (l, d)-motif finding algorithms. *BMC Research Notes*, 4(1):54, 2011.

[77] Sanguthevar Rajasekaran and Marius Nicolae. An elegant algorithm for the construction of suffix arrays. *Journal of Discrete Algorithms*, 27:21–28, 2014.

[78] J.H. Reif and L.G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, 1987.

[79] I. Roy and S. Aluru. Finding motifs in biological sequences using the micron automata processor. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS 14, Washington, DC, USA*, pages 415–424. IEEE, May 2014.

[80] B. Sahoo, R. Sourav, R. Ranjan, and S. Padhy. Parallel implementation of exact algorithm for planted motif search problem using smp cluster. *European Journal of Scientific Research*, 64(4):484–496, 2011.

[81] K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Softw: Pract. Exper.*, 37(3):309–329, 2007.

[82] J. Seward. On the performance of bwt sorting algorithms. In *Proceedings of the Conference on Data Compression*, DCC '00, pages 173–, Washington, DC, USA, 2000. IEEE Computer Society.

[83] He Quan Sun, M.Y.H. Low, Wen Jing Hsu, and J.C. Rajapakse. Listmotif: A time and memory efficient algorithm for weak motif discovery. In *2010 International Conference on Intelligent Systems and Knowledge Engineering (ISKE), 15-16 November 2010, Hangzhou, China*, pages 254 –260. IEEE, nov. 2010.

[84] H.Q. Sun, M.Y.H. Low, W.J. Hsu, C.W. Tan, and J.C. Rajapakse. Tree-structured algorithm for long weak motif discovery. *Bioinformatics*, 27(19):2641–2647, 2011.

[85] W. Szpankowski. *Average Case Analysis of Algorithms on Sequences*. John Wiley & Sons, Inc., 2001.

[86] S. Tanaka. Improved exact enumerative algorithms for the planted (l, d)-motif search problem. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 11(2):361–374, March 2014.

[87] C.D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Commun. ACM*, 20(4):263–271, 1977.

[88] Martin Tompa, Nan Li, Timothy L Bailey, George M Church, Bart De Moor, Eleazar Eskin, Alexander V Favorov, Martin C Frith, Yutao Fu, W James Kent, et al. Assessing computational tools for the discovery of transcription factor binding sites. *Nature biotechnology*, 23(1):137–144, 2005.

[89] Qiang Yu, Hongwei Huo, Yipu Zhang, and Hongzhi Guo. Pairmotif: A new pattern-driven algorithm for planted $(l, d)$ dna motif search. *PLoS ONE*, 7(10):e48442, 10 2012.