STRING MATCHING AND INDEXING WITH SUFFIX DATA STRUCTURES

WONG SWEE SEONG
(*MSc. (School of Computing)*)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2007

Acknowledgments

I like to thank everyone who has been there for me in this quest for knowledge and a journey of self discovery.

I am fortunately blessed with a caring family and am grateful to my parents and sisters for their support. I dedicate my thesis to the memory of my mother for her selflessness and abundant love. To that special someone, my loving and supportive wife Lin Li, thank you for your kindness and believing in me.

To my advisory committee members, Assoc Prof Tan Kian Lee and Assoc Prof Lee Mong Li, thank you for your patience and valuable advice. My sincere appreciation goes to my supervisors Assoc Prof Ken Sung Wing Kin and Prof Wong Lim Soon for their guidance and generosity in sharing their wisdom with me.

Lastly, to all my friends and colleagues at the School of Computing, a big thanks to you. The past years with the school will be fondly remembered.

Contents

List of Figures

List of Tables

Summary

This thesis studies methods for indexing a text so that the occurrences of any given query string in the text can be located efficiently. An occurrence or match may be imprecise, allowing some deviations from the actual query. This gives rise to a family of interesting string matching problems like exact and approximate string matching, and sequence alignment.

Previously, a linear size $O(n)$ word index, where n is the length of the text, is considered manageable given that the index size is relatively small compared to the size of available memory on most desktop computers. As such, we can focus on developing new search algorithms without worrying about the index size. However, a new challenge arises from searching large genome sequences which can easily be billions of characters in length. This leads to the issue of search efficiency on large string index, which is made worst with the ever increasing genome size.

We consider two different computing models to handle the problem. The first is to compress the index so that it is small enough to be stored in the main memory. Another computing model is to make use of secondary disk, where the index resides on the hard disk. Blocks or chunks of the index are fetched into memory upon request. In this case, we are concern with the number of IO accesses to perform string search on the index. In both scenarios, it is essential to have efficient computation algorithms to support various string search. Mixed computing model is also possible with multiple levels of indexing, combining both in-memory and disk-based indices.

We propose several compressed data structures to index string text in $o(n)$ words or $O(n)$ bits. These data structures are suitable for in-memory computation to answer exact, as well as approximate, string matching problems. We study the asymptotic bounds on the query time and show that our indices give the best known solution using different indexing spaces. These proposed indices will be useful to optimize performance for computationally intensive search tasks. However, it is observed that in a pattern search, consecutive accesses of the data structure, can be reading segments of the structure that are very far apart. In fact, the access pattern is very much random. This results in a significant IO cost that slows down the search performance if the index is not able to fit into the memory. Thus, optimizing disk-based solution becomes necessary.

Consequently, we propose a disk-based index representation based on suffix tree called CPS-tree. Current suffix tree developments focus on the construction efficiency and less on the structural design to minimize the IO accesses on the tree. Unfortunately, the few IO efficient suffix tree designs in the literature are very much limited to exact string match alone. As such, we present disk based CPS-tree, and design and engineer

search algorithms on CPS-tree to support various types of string search and tree traversal operations efficiently. Our worst-case IO performance is well bounded in theory. Empirical studies on exact string matching and sequence alignment problems, conducted on a large genome, further demonstrate that our proposed data structure is useful and practical. Through theoretical analysis and experimental investigation, we illustrate the advantages of our suffix tree design.

To summarize, we make our contributions to more efficient string matching and indexing. However, there are still rooms to further improve on the efficiency. It is an unsolved research challenge to come up with a compact string index ($o(n)$ word size) that displays good access locality for string search. This remains as future work to be done.

Chapter 1

Overview

1.1 Introduction

String matching is an important and age-old classical problem. The problem is fundamental to many applications that require processing of some text or sequence data. Very often, it involves finding the occurrences of a pattern string in a given text string. Some of its applications are spell checking in text editor, identity and password validation and checking in system login, and content interpretation in document and programming language parsers. Furthermore, string matching is the very essence of pattern matching languages like Perl and Awk. Over the years, we see more of string matching algorithms being applied to areas like information retrieval, pattern recognition, compiling, data compression, program analysis and security etc. There are also a vast number of research papers, over the past three decades, providing theoretical as well as empirical results to the problem with improved space and time efficiencies.

Exact string matching finds the exact occurrence of any given pattern in the text to be searched. The early works focus on the on-line problem where preprocessing is performed on the pattern string but not the text. Some of the classical works are Knuth, Morris and Pratt (KMP) algorithm [55], and Boyer and Moore (BM) algorithm [12] for string matching. The problem is extended to the approximate string matching where some form of errors are allowed in finding the occurrences. There exists many different variations of the error model but more commonly, we have the followings, as formally defined below.

Consider a text T of length n and a pattern P of length m, both strings over a fixed finite alphabet A.

1. k**-mismatch problem**: Find all approximate occurrences of P in T that have Hamming distances at most k from P. The Hamming distance between two strings is defined to be the minimum number of character replacements to convert one string to the other. The k-**don't-care problem** is a special subproblem where mismatches are allowed only at specfic k positions on the pattern P. The k mismatch positions are indicated on P.

2. k**-difference problem**: Find all approximate occurrences of P in T that have edit distances at most k from P. The edit distance between two strings is defined to be the minimum number of character insertions, deletions, and replacements to

convert one string to the other.

For the on-line version of the problem, the search time depends on the text size, and therefore becomes inefficient in handling large text. New algorithms have been proposed to allow preprocessing of the text, or in other words using indexing, for faster string search. In particular, suffix tree [67, 99] and suffix array [66] are popular data structures to be used for string indexing. More recently, compressed suffix data structures are used in indexing string.

Another class of problem that is closely related to the k-difference problem is the sequence alignment problem. Tools for local alignment in genome sequences like FASTA [82, 83] and BLAST [4, 5], are among the most commonly used tools by biologists today. The problem extends on the k-difference problem by associating different costs to each of the edit operations. Furthermore, in the affine gap cost model, a cost penalty is given to a gap opening, which is defined as a consecutive insertions either on the text or the query but not both at the time. The objective is then to find the alignments between the query and text that minimize the sum up cost.

In this thesis, we focus on a wide range of string matching problems ranging from exact matching, approximate matching (Hamming and Edit distance measures) and sequence alignment problems as well. We study the time and space complexities of various compressed data structures, assumed to be fully residing in memory, and proposed new data structures that are asymptotically smaller and faster to search. Next we extend our work to consider IO-efficiency of specifically suffix tree on secondary disk. A new representation is proposed that is shown empirically to be efficient as well as having nice worst case performance bounds.

1.2 Motivation

One of the driving force for developing string matching techniques stems from the massive availability of biological sequence data that begins in the late 90's. This has created opportunities for researchers to apply their innovative algorithms and techniques to work on real datasets. Our work is also motivated by this uprising trend. In August 2005, it was reported [1] that the collection of DNA and RNA sequence has already reached 100 gigabases. These 100,000,000,000 bases, or "letters" of the genetic code, represent both individual genes and partial and complete genomes of over 165,000 organisms. Submitters to GenBank contribute over 3 million new DNA sequences per month to the database. GenBank (Bethesda, Maryland USA) [2], together with European Molecular Biology Laboratory's European Bioinformatics Institute (EMBL-Bank in Hinxton, UK) [3], and the DNA Data Bank of Japan (Mishima, Japan) [4], form the International Nucleotide Sequence Database Collaboration to share and organize the sequence database.

[1] $http://www.nlm.nih.gov/news/press\_releases/dna\_rna\_100\_gig.html$

[2] $http://www.ncbi.nlm.nih.gov$

[3] $http://www.ebi.ac.uk/embl$

[4] $http://www.ddbj.nig.ac.jp$

Scientists around the world can then have access to the common sequence data, and hopefully through collaborative research on the massive data, scientists can find cures for diseases and improved health in shorter time to benefit all mankind.

The storage size of sequenced genome and annotated biological sequences, is growing in the order of several gigabytes per year. There is a need to collectively organize and manage these sequences to support the data usage requirement of various comparative tools at the application level. Sequences can be indexed so that search is performed more efficiently. There already exists a wide range of computational tools on strings, for searching approximate similarities, and finding consensus, alignments, repeats and motif patterns, etc. Currently, there is a lack of a standard indexing data structure on sequences that can serve the needs of the various tools. Such a generic indexing structure must be robust and flexible. In addition, a management system will be useful to manage processes and allocate the use of system resources. However, traditional relational database systems are inadequate for the task as the sequence data is generally huge and unstructured in nature, without the proper notion of a key.

Another reason to study string matching problem is its wide range of applications. Many algorithmic problems can be mapped into exact or approximate string search problems. This makes analyzing the algorithmic properties important. Furthermore, the problems can be extended to higher dimensional text or multiple patterns search problems where existing algorithms may be borrowed or built upon.

1.3 Research problems and contributions

1.3.1 Exact and approximate string matching

Approximate string matching is an important problem to solve and comparative analysis on sequences often needs to perform close similarity search as part of the process. In some cases, sequence data may contain noise or variations that we would like to tolerate in our search. Given a query string, we would like to find its occurrences in a text by allowing some degree of errors.

We consider two approximate matching problems: the k-mismatch problem and the k-difference problem. Our focus is on constructing compact indices that are of $o(n)$ words or $O(n)$ bits size so that for large text of length n, there is a good chance that the indices can fit into memory for searching. We give some improved data structures for the approximate string search with the best known query time using only less than linear word size indices.

To add on to the above results, we revisit the problem of exact string matching to find more compact indexing structures. It is well known that given a preprocessed text indexed using $O(n)$ words data structure, we can find the exact matches of a query string in time linear to the query length. We push for even more compact data structures (using less than linear words size) that can answer the query in optimal time using bit-compressed query string.

1.3.2 Disk-based string indexing

A text is a string or set of strings. To answer string matching queries over the text, given a query string, the text may be preprocessed and represented in a data structure. This data structure will then provide indexing into the text so that string search and comparison can be performed more efficiently.

Given the query string and text, the traditional approach to string comparison is to scan through the whole text for solution. This is generally fast enough provided that the text is short. There is little to improve upon the query time as no preprocessing of the string is done and the loading of the whole text into memory takes up the main bulk of the processing time. Indexing on the text and index thus allows for only partial access of the text in order to find the solution at the expense of greater storage on disk for the index. Together with efficient search techniques, the query time can be very much improved.

We have considered in-memory indices which may be a favorable alternative to direct scanning for small indices. It may not be suitable for indices larger than the text itself and will be time consuming to load into memory. The exception is when we have a large memory and the indices can be preloaded into the memory to answer batch queries or any incoming queries in a server mode of operation.

Alternatively, we have the indices residing on disk and be fetched into memory as and when needed. The direct choice is to build a hash table for every fixed length-l substrings in the text. Samples of length-l substrings from the query is used to reference the hash table, for fixed length-l matches. Fixed length indexing lacks the flexibility, as the length is fixed, to efficiently handle varied length queries and more importantly, on finding approximate match. Also l has to be short for it to be usable. There are some well studied filtering techniques to overcome these shortcomings like q-grams indexing, which generally performs well in practice. Another popular approach is to use hierarchical level of indices to extend on the length l, where only the top-level indices need to reside in memory, the rest of the indices are fetched from disk into memory when needed. These proposed indexing methods do not have acceptable worst case complexity on query time and I/O disk access for both exact and approximate string matching.

We recommend using suffix tree as a common indexing data structure on string and propose means to improve its IO access efficiency. We can find, using the suffix tree, in time linear to the query length, the locations on the text that match exactly to the query string. One major issue with suffix tree data structures is that it requires a much larger space than the text itself. This comes as a trade-off for faster query time. For example, a text string of n characters needs $4n$ to $20n$ bytes to store the suffix data structure depending on the level of compression and the functionalities to be supported. Recently, there are proposed $O(n)$ bits compressed suffix tree and array implementations that are very space efficient. The problem is that the access pattern on the compressed data structures tends to be highly random and hence it is more suitable if the whole structure can reside in the memory. There are many string related problems that can be efficiently solved using suffix tree [37, 40]. Approximate string matching on suffix data structures is one of them. However, the existing techniques can still be further improved to answer the queries more efficiently. It is still an open problem on how to perform disk-based indexing efficiently for approximate string matching [52]. We address this issue and give a feasible solution.

1.4 Organization of thesis

In chapter two, we introduce some related fundamental concepts in the literature. This is followed by three chapters to showcase our proposed works. In particular, we first focus on in-memory string search and present compact data structures to solve the approximate string matching problem. Next we continue the study with exact string matching problem and proposed several data structures with optimal search time and using less than linear indexing space. Last but not least, we divert our attention to disk-based string indexing using suffix tree. We propose a new suffix tree representation to handle various string matching queries and tree traversal operations efficiently. Finally, in the concluding chapter, we discuss on the future research direction.

1.5 Statement

The preliminary work described in chapter 3 on approximate string matching was first presented in the 16th Annual International Symposium on Algorithms and Computation 2005 [61]. An extended version of the paper was later submitted and accepted for publication in the Algorithmica journal [62]. Another 2 results extended from this initial work, were presented in the 17th Annual Symposium on Combinatorial Pattern Matching 2006 [18] and the 16th Annual European Symposium on Algorithms 2006 [17] respectively. The suffix tree representation proposed in chapter 5 was presented in the 23rd International Conference on Data Engineering 2007 [102].

Chapter 2

Background

2.1 Introduction

The basic data structures used for string indexing are mainly suffix tree [20, 30, 45, 69], suffix array [9, 68, 74] and q-grams [16, 46, 79, 86]. Suffix data structures benefit from linear search time in matching a given pattern string to a text. This is at the expense of larger index size. It goes by matching the query to characters on the edges along a path from the root that ends at some node, and all the leaves in the subtree rooted at the node will contain the locations of exact match in the text. On the other hand, q-grams is another popular index used that stores the locations of every or selected length-q substrings in the text. It is basically a filtering technique that works well in eliminating segments of the text that have no possible match with the query string. The indices takes up much smaller space when compared to suffix tree. There are two main setbacks with the q-grams approach. Firstly, the length q has to be fixed; and hence it lacks the flexibility to cater to all-purposed demands. Secondly, the worst-case running time is less well bounded when compared to suffix data structures, though it has shown to perform reasonably well in practice on real biological sequences.

Inverted file [89] is a common text index used on linguistic text that is constructed from a fixed set of naturally delimited words. We do not consider inverted file as a choice for indexing string in general for the reason that biological sequences is highly unstructured and will not benefit from the indexing. There is an adaptation of inverted file to index biological sequences called CAFE [101] that employs some filtering techniques to reduce the space and time complexities for heuristic search.

There has been on-going developments of fast in-memory and on-disk construction [26, 33, 34, 40, 44, 45, 95, 98] of suffix data structures, and also in more compact but functional suffix representations such as compressed suffix tree and arrays [31, 38, 72, 85, 87, 88]. These advancements have made suffix data structure an attractive choice for indexing strings.

An overview on the various full-text indices in external memory can be found in the paper by K$\ddot{a}$rkk$\ddot{a}$inen and Rao [52]. The reader can refer to the paper by Navarro *et al.* [75], for a survey on various indexing techniques for approximate string matching. There is a recent interest in string matching on compressed text directly without first decompressing the text [6, 27, 51, 77, 78]. The main gain is in reducing the I/O burden of bringing the text into memory and keeping the memory usage low while scanning for matching patterns.

The sections to follow describe the basic data structure of suffix tree and suffix array as well as the compressed forms, and also introduce some string search applications performed on the suffix data structures. These data structures will be refered frequently in the later chapters.

2.2 Suffix tree and suffix array

A trie is a rooted directed tree that stores a set of strings. Each and every leaf node represents a string stored by the trie. It is assumed that no string is a proper prefix of another. For example, “abbc” is a proper prefix of “abbcbbd”, while “abbd” is not. Every edge in the tree is labeled with a single character such that the concatenating of the *edge labels* in order, from the root to a leaf node, corresponds to the suffix string represented by the leaf node. A compact trie is a trie with every node, that has only single outgoing edge, merged into its parent node and the characters on edges are concatenated to form a string (see Figure 2.1). While a Patricia trie [70] is like a compact trie except that every edge label contains only the first character with the length of the original edge label stored in the node that follows. Every internal node in the compact trie has at least 2 child nodes. The *path label* of a node is the concatenated edge labels from the root to that node and the *character depth* of a node is its path label length.

The suffix tree (ST) [67, 99] of a text T, is a compact trie of the set of suffixes of T, as

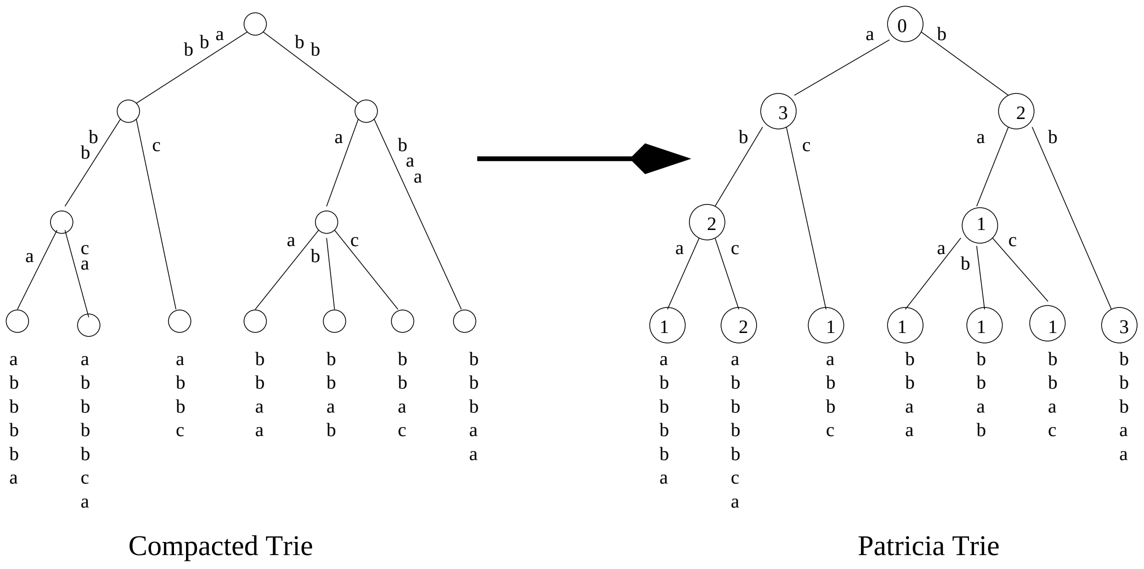


Figure 2.1: Patrica trie for a set of strings $= \{abbbba, abbbbca, abbc, bbaa, bbab, bbac, bbbaa\}$.

shown in Figure 2.2. The text T is usually concatenated by a special symbol \$ that is not found in T. Figure 2.2 gives a suffix tree with the suffixes appearing in lexicographically sorted order from left to right in the tree. In comparison, the suffix array [66] of T is a sorted array containing the starting text position of the suffixes of T. The Patricia trie of a set of suffixes of T is denoted as PAT tree [36].

We often use suffix tree to mean a PAT tree representation. PAT tree and suffix array (SA) are $O(n)$ word data structures where n is the text length with suffix array being a more compact representation.

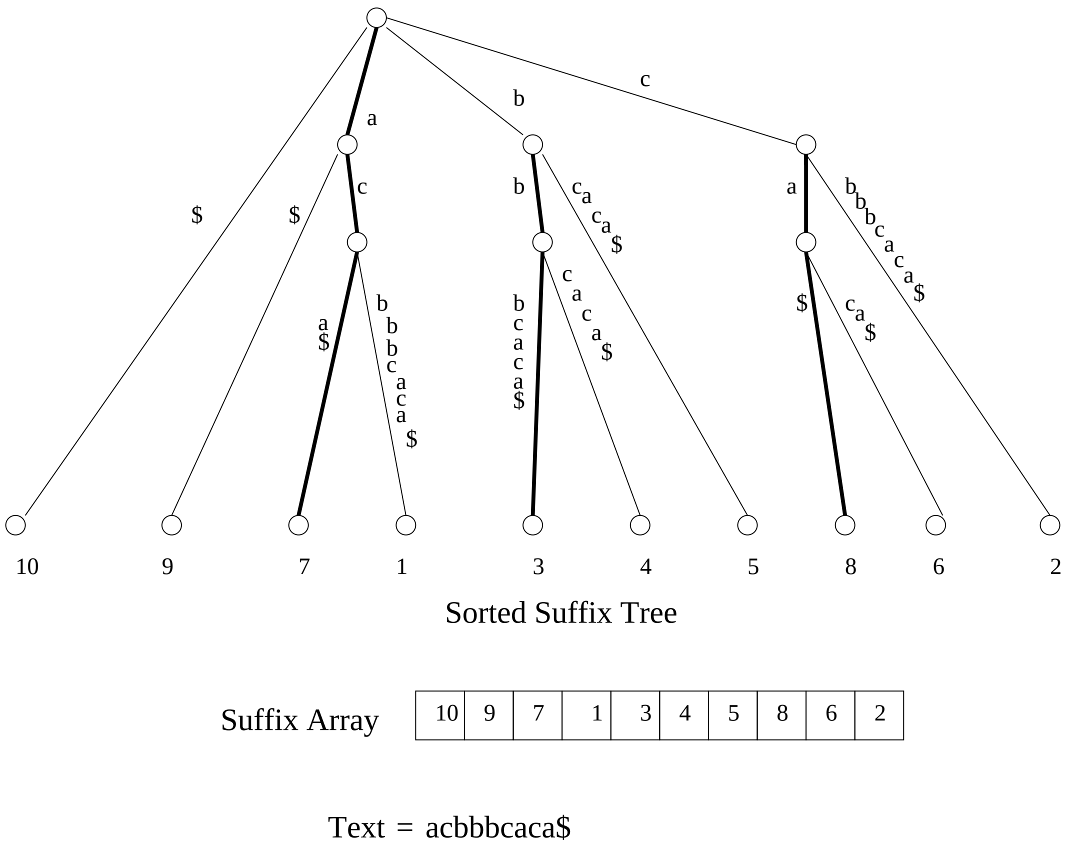


Figure 2.2: Suffix tree and suffix array.

2.3 Compressed suffix data structures

Although suffix array (SA) is compact compared to suffix tree (also PAT tree), it can still be large. An SA built on a large text of size in billion of characters (for example the human genome), will not be able to fit fully in the main memory of most computers. As such a compressed suffix array (CSA) [38, 39] becomes an attractive alternative representation. A CSA stores the array for Ψ function defined as $\Psi[i] = SA^{-1}[SA[i]+1]$ for text $T[1..n]$, $i \in [1..n]$ where $SA[i]$ is the text position found at i-th entry of suffix array.

Using the SA in Figure 2.2 as an example, we have $\Psi[i] = [0, 1, 8, 10, 6, 7, 9, 2, 3, 5]$. Interestingly, the Ψ array is actually a concatenation of at most $|A|$ number of increasing sequences where $|A|$ is the size of the alphabet from which the text is drawn. This makes Ψ array highly compressible and gives a representation that is $O(n)$ bits depending on the alphabet size. However, this comes as trade-off in terms of computational time to recover the SA value which can be relatively inexpensive if the whole index is fully loaded into memory [43].

Another compressed SA representation is the FM-index [31, 32] using the Burrows-Wheeler compression algorithm [15]. Compressed suffix tree (CST) [88] was proposed to be a compressed representation that supports suffix tree traversal operations efficiently. It is basically a CSA augmented with additional data structures like the balanced parenthesis representation [71] for the tree structure and the LCP (lowest common prefix) query supporting structure [87].

2.4 Application of suffix data structures

There are many string search problems that can be solved using suffix data structures [37, 40, 56]. Beside exact and approximate string matching problems, there are also problems of the longest common substrings between two sequences, palindrome and maximum repeats etc. In computational biology, the applications extend to solving problems in probe design [50], motifs and repeat finding [58, 81] and genome alignments [23, 24, 59]. For local and global sequence alignment problems , we can adopt the commonly used “hit and extend” strategy, by finding only short matches (hits) on the text and then, extend and verify for the rest of the query string. This heuristic strategy helps to reduce the search space tremendously, by finding hits for fixed length substring of the query (suitably short) using the suffix data structures. The choice of “hit” length can be easily varied according to requirements with possibly some allowed errors. Examples are FASTA [82, 83], BLAST family [4, 5, 54, 103] and PatternHunter [64, 65].

Next, we demonstrate how to perform an ordinary depth-first traversal search on a tree (in this case, a suffix tree) for approximate match. Here we consider the problem of k-mismatch or k-difference, given query pattern P and the text T. Recall that the k-mismatch allows for substitution operation only while k-difference allows for additional 2 edit operations, insertion and deletion of characters. The algorithm is shown in Figure 2.3. The routine DFSearch $(cNode, k', i, P')$ takes in 4 parameters, namely, cNode refers to the index of the current node, k' is the number of errors encountered so far, i is the current position on the query string P to match and P' is a copy of the error string encountered. Let $|cNode|$ denotes the length of the path label to the node $cNode$ and $|P|$ denotes the length of pattern string P.

The traversal search for approximate match runs in $O(\min\{n, A^k m^k\}m + kocc)$ time for k-mismatch and an additional factor of 3^k applies for k-difference. occ is the number of occurrences of the approximate match in the text. This example code demonstrates the basic approach to string matching on suffix tree. There are more efficient algorithms

```
**Call** DFSearch $(root, 0, 1, \emptyset)$
**Algorithm** *DFSearch (cNode, k', i, P')*
  If $(i = |P| + 1)$
      Report all text position in the subtree rooted at cNode
  If $(k' > k)$
      return
  /* Query deletion */
  DFSearch $(cNode, k' + 1, i + 1, P')$
  For $c \in A$, and $P'c$ a substring in $T$
      If $(|P'| = |cNode|)$
          sNode = child(cNode, c)
      Else
          sNode = cNode
      /* Query insertion */
      DFSearch $(sNode, k' + 1, i, P'c)$
      If $P[i] \neq c$
          /* Substitution */
          DFSearch $(sNode, k' + 1, i + 1, P'c)$
      Else
          /* Exact match */
          DFSearch $(sNode, k', i + 1, P'c)$
**End**
```

Figure 2.3: Depth first search of the suffix tree for approximate matching.

which incorporated dynamic programming over the suffix tree which will be presented in the later chapters. Dynamic programming [80, 90, 92] is useful in string matching especially in reducing the redundancy in checking for different combinations of editing operations that can be applied. Also it allows for early termination by pruning off sub-trees that have no possible match. For some theoretical results, the reader can refer to works by Ukkonen [97] and Cobbs [21]. Navarro and Baeza-Yates [74] and Hunt *et al.* [45] gave empirical results using the approach.

Chapter 3

Memory-based compressed string index

3.1 Introduction

Consider a text T of length n and a pattern P of length m, both strings over an alphabet A. The approximate string matching problem is to find all approximate occurrences of P in T. Depending on the definition of "error", this problem has two variations: (1) The k-difference problem is to find all occurrences of P in T that have edit distance at most k from P (edit distance is the minimum number of character insertions, deletions and replacements to convert one string to another); and (2) The k-mismatch problem is to find all occurrences of P in T that have Hamming distance at most k from P (Hamming distance is the minimum number of character replacements to convert one string to another). Both k-difference and k-mismatch problems are well-studied and they found applications in many areas including computational biology, text retrieval, multimedia data retrieval, pattern recognition, signal processing, handwriting recognition, etc.

In the past, most of the works focus on the on-line version of the problem, where both the text and the pattern are not known in advance. This version of the problem can be solved by dynamic programming in $O(nm)$ time. Landau and Vishkin [63] gave a solution whose running time depends on k, the number of allowed "errors". They solved the problem in $O(nk)$ time and $O(m)$ space. Amir *et al.* [8] improved upon the result to give an $O(n\sqrt{k \log k})$ time solution. We refer to [73] for a comparison study on various existing techniques.

Recently, people are interested in the off-line approximate matching problem, where we can pre-process the text T and build some indexing data structure so that any pattern query can be answered in a shorter time. Jokinen and Ukkonen [49] were the first to treat the approximate off-line matching problem. Since then, many different approaches have been proposed. Refer to Navarro *et al.* [73] for a brief survey. Some techniques are fast on the average [76, 93, 91, 10, 79, 74]. However, they incur a query time complexity depending on n; i.e., in the worst case, they are inefficient even if the pattern is very short and k is as small as one. The first solution with query time complexity independent of n is proposed by Ukkonen [97]. When $k = 1$ (that is, 1-mismatch or 1-difference problem), Cobbs [21] gave the result of using $O(n \log n)$ bits space and having $O(|A|m^2 + occ)$ query time. Later, Amir *et al.* [7] proposed an $O(n \log^3 n)$-bit indexing data structure with $O(m \log n \log \log n + occ)$ query time. Then, Buchsbaum et al [14] proposed another indexing data structure which uses $O(n \log^2 n)$ bits space so that every query can be solved in $O(m \log \log n + occ)$ time. Cole *et al.* [22] further improved the query time. They gave an $O(n \log^2 n)$-bit data structure so that both the 1-mismatch and the 1-difference problems can be solved in $O(m + \log n \log \log n + occ)$ time, respectively. Recently, motivated by the indexing of long genomic sequences, Trinh *et al.* [96] improves upon the space-efficiency. They proposed two data structures of size $O(n \log n)$ bits and $O(n)$ bits with query time $O(|A|m \log n + occ)$ and $O(|A|m \log^2 n + occ \log n)$, respectively.

Some of the above results can be generalized for $k > 1$. Cobbs's $O(n \log n)$-bit indexing data structure can answer both k-mismatch and k-difference queries in $O(m^{k+1}|A|^k + occ)$ time [21]. Cole *et al.* [22] proposed an $O(n \frac{(c_3 \log n)^k}{k!} \log n)$-bit indexing data structure with query times of $O(\frac{(c_1 \log n)^k \log \log n}{k!} + m + occ)$ and $O(\frac{(c_2 \log n)^k \log \log n}{k!} + m + 3^k \cdot occ)$ for the k-mismatch and k-difference problems, respectively, where c_1, c_2, c_3 are constants with $c_2 > c_1$. Trinh *et al.* [96] gave $O(n \log n)$-bit and $O(n \log |A|)$-bit data structures that can answer a k-mismatch (or a k-difference) query in $O(|A|^k m^k \log n + occ)$ time and $O(|A|^k m^k \log^2 n + occ \log n)$ time, respectively.

All previous data structures for supporting the 1-mismatch (or 1-difference) query either require a space of $\Omega(n \log^2 n)$ bits or $\Omega(m \log n + occ)$ time for fixed alphabet size. It is an open problem whether there exists an $O(n \log n)$ or even $o(n \log n)$-bit data structure so that every 1-mismatch (or 1-difference) query can be answered in $o(m \log n + occ)$ time. In this work, we resolve this open problem in the affirmative by presenting a data structure which uses $O(n\sqrt{\log n} \log |A|)$ bits while every 1-mismatch (or 1-difference) query can be answered in $O(|A| m \log \log n + occ)$ time.

Our improvement is stemmed from the observation that suffix trees allow for faster access of some information when compared with suffix array. So, instead of using suffix array like Trinh *et al.*[96], we use suffix tree as the basic data structure to solve the mismatch (or difference) queries. Furthermore, to reduce space, we apply the results of Rao [85] and Sadakane [88] to reduce the space complexity of the suffix tree from $O(n \log n)$ bits to $O(n\sqrt{\log n} \log |A|)$ bits. Together with a smart use of the y-fast trie [100], we achieve our improvement.

Table 3.1 summarizes the results for 1-mismatch (or 1-difference) problem over a finite alphabet A. Our result can be further extended in two ways. First, we show that the space of the data structure can be reduced to $O(n \log |A|)$ bits if we accept a slow down factor of $\log^{\epsilon} n$ for the query time where $0 < \epsilon \leq 1$. Second, the data structure can be extended to solve the k-mismatch (or the k-difference) problem for $k \geq 1$. Our solution can solve the k-mismatch (or the k-difference) problem in $O(|A|^k m^k (k + \log \log n) + occ)$ or $O(\log^{\epsilon} n \ (|A|^k m^k (k + \log \log n) + occ))$ query time, when the text is encoded using $O(n\sqrt{\log n} \log |A|)$ bits, for $|A| = O(2^{\sqrt{\log n}})$, or using $O(n \log |A|)$ bits.

Reference	***Bit space***	***Query time***
Cobbs [21]	$O(n \log n)$	$O(\|A\|m^2 + occ)$
Buchsbaum *et al.* [14]	$O(n \log^2 n)$	$O(m \log\log n + occ)$
Cole *et al.* [22]	$O(n \log^2 n)$	$O(m + \log n \log\log n + occ)$
Trinh *et al.* [96]	$O(n \log n)$	$O(\|A\|m \log n + occ)$
	$O(n \log \|A\|)$	$O(\|A\|m \log^2 n + occ \log n)$
This work	$O(n\sqrt{\log n} \log \|A\|)$	$O(\|A\|m \log\log n + occ)^*$
	$O(n \log \|A\|)$	$O(\log^{\epsilon} n(\|A\|m \log\log n + occ))$

* assume $|A| = O(2^{\sqrt{\log n}})$.

Table 3.1: Comparison of various results for 1-mismatch (or 1-difference) problem.

3.2 Preliminaries

3.2.1 Edit operations

Let $P = P[1]P[2]...P[m]$ be a string of m characters over a finite alphabet A. A substring of P is denoted by $P[i..j] = P[i]P[i+1]...P[j]$, $1 \le i \le j \le m$. An edit operation applied to a string P is given in the forms of $(a \to \epsilon)$, $(\epsilon \to a)$, and $(a \to b)$ for *deletion*, *insertion* and *substitution* operations respectively, where $a, b \in A$, $a \neq b$ and ϵ is the empty string. The *edit distance* between P and P', is the minimum number of edit operations to convert one string to another. For example, converting string $abbd$ to another string $bbca$ will take at least 3 edit operations. An *edit trace* is defined as a sequence of edit operations that converts a string P to another string P'.

Lemma 1 Given a length-m string P over a fixed alphabet A, there are $O(|A|^k m^k)$ possible edit traces for converting P to some string P' using at most k edit operations.

Proof. The bound on the number of edit traces can be estimated by considering the number of different ways of applying k or less edit operations to the string. There are 2 different groups of operations: The first group is of the form $a \rightarrow b$ and $a \rightarrow \epsilon$ and the second group has the form $\epsilon \rightarrow a$. The first group consists of substitutions and deletions that can be applied to every character in P. Hence the number of possible ways of applying k operations in this group is $\leq \binom{m}{k}(|A|+1)^k$. The second group consists of insertions that can occur at the start or end of string, or in between characters. The number of possibilities in this group is $\leq (m+1)^k|A|^k$.

Summing up for k or less edit operations, we have $\sum_{t=0}^{k}[\binom{m}{t}(|A|+1)^t + (m+1)^{k-t}|A|^{k-t}] = O(m^k|A|^k)$ number of possible edit trace. Refer to Theorem 6 in [97] by Ukkonen for details. $\square$

3.2.2 Suffix array, inverse suffix array and Ψ function

Let $T[0..n] = t_0 t_1 \cdots t_{n-1}$ be a text of length n over an alphabet A, appended with a special symbol $t_n =$'\$' that is not in A and is smaller than any other symbol in A. The j-th suffix of T is defined as $T[j..n] = t_j \cdots t_n$ and is denoted by T_j.

The *suffix array* $SA[0..n]$ of T is an array of integers so that $T_{SA[i]}$ is lexicographically smaller than $T_{SA[j]}$ if and only if $i < j$. Note that $SA[0] = n$. The *inverse suffix* *array* of T is denoted as $SA^{-1}[0..n]$, that is, $SA^{-1}[i]$ equals the number of suffixes which are lexicographically smaller than T_i.

Given a string P, we define $range(T, P)$ or the range of the suffix array of T corresponding to P, to be the largest interval $[st..ed]$ such that P is a prefix of every suffix T_j for $j = SA[st], SA[st+1], \ldots, SA[ed]$.

A concept related to the suffix array is the array $\Psi[0..n]$ [38], which is defined as follows:

$$\Psi[i] = SA^{-1}[SA[i] + 1]$$

and similarly, $\Psi^k[i] = \Psi^{k-1}[\Psi[i]] = SA^{-1}[SA[i] + k]$, for $k > 1$.

Let t_{SA} and t_Ψ be the access time of each entry on SA and Ψ respectively. In this paper, we need a data structure $\mathcal{D}$ which supports, for any i, the following operations.

- reports $SA[i]$ in t_{SA} time,
- reports $SA^{-1}[i]$ in t_{SA} time,
- reports $\Psi[i]$ in t_Ψ time, and
- reports *substring(i,l)* $= T[SA[i]..SA[i] + l - 1]$ in $O(lt_\Psi)$ time for some length l.

Lemmas 2 and 7 give two implementations of the data structure $\mathcal{D}$.

Lemma 2 The data structure $\mathcal{D}$ can be implemented in $O(n \log |A|)$ bits so that $t_{SA} = O(\log^\epsilon n)$ and $t_\Psi = O(1)$, where $0 < \epsilon \leq 1$.

Proof. We refer to Grossi and Vitter's data structure [38] for compressed suffix array (CSA) with the required properties. $\square$

Lemmas 3 to 6 are needed for the second implementation of the data structure $\mathcal{D}$.

Lemma 3 [84, 47] Let S be a subset of m elements drawn from the set $(1, 2, ..., n)$. S can be represented using $m\log(n/m)+O(m)$ bits such that the following rank and select operations can be performed in constant time. A rank operation returns the order of an element $x \in S$, defined as $Rank[x] = |\ \{y < x \mid y \in S\}\ |$. A select operation returns the i-th smallest element in S, where $1 \leq i \leq m$ (i.e. $Select[i] = x$ if $Rank[x] = i$).

Lemma 4 Let $X_1, ..., X_\ell$ be ℓ non-empty subsets of $\{0, ..., n-1\}$ such that $\sum_{j=1}^{\ell} |X_j|$ $= m$ and $m \leq n$. The subsets can be represented using $m\log(n\ell/m) + \ell\log n + O(m)$ bits such that given i and j, the ith smallest element in X_j can be retrieved in constant time.

Proof. This lemma is from Corollary 2 in Rao's paper [85]. First, store the set $X = \{j * n + x \mid x \in X_j\}$ using $m\log(n\ell/m) + O(m)$ bits of space as in Lemma 3. Second, let $c_j = \sum_{t=1}^{j} |X_t|$, for $1 \leq j \leq \ell - 1$. The array c can be represented directly using additional $\ell \log n$ bits. The ith smallest element in X_j is the $(c_{j-1} + i)$th element in X, which can be retrieved in $O(1)$ time. $\square$

Lemma 5 The sequence $\{\Psi^k[i] \mid 0 \leq i \leq n-1\}$ is the concatenation of $|A|^k$ sorted lists.

Proof. This lemma is generalized from Lemma 3 in Rao's paper [85]. □

Lemma 6 Let $X_1, ..., X_\ell$ be ℓ subsets of $\{0, ..., n-1\}$ such that $|X_j| = n/\ell$, $1 \le j \le \ell$. Then $\{\Psi^j[z] \mid z \in X_j\}$ for all j, can be stored in an $O(n\ell \log |A| + |A|^\ell \log n)$-bit data structure such that given i where $X_j[i] = z$, $\Psi^j[z]$ can be accessed in $O(1)$ time.

Proof. For any given j, $\{\Psi^j[z] \mid z \in X_j\}$ contains at most $|A|^j$ sorted lists. Combining Lemmas 4 and 5, $\{\Psi^j[z] \mid z \in X_j\}$ can be represented in $O(|X_j| \log(n|A|^j/|X_j|) + |A|^j \log n + |X_j|) = O((n/\ell) \log(|A|^j \ell) + |A|^j \log n)$ bits. Then, given i where $X_j[i] = z$, we can access $\Psi^j[z]$ in constant time. The space needed to store $\{\Psi^j[z] \mid z \in X_j\}$, for $1 \le j \le \ell$, will then be $O(n \log(|A|^\ell \ell) + \log n \sum_{j=1}^{\ell} |A|^j) = O(n\ell \log |A| + |A|^\ell \log n)$ bits. □

Lemma 7 The data structure $\mathcal{D}$ can be implemented in $O(n\sqrt{\log n} \log |A|)$ bits so that $t_{SA} = O(1)$ and $t_\Psi = O(1)$, for $|A| = O(2^{\sqrt{\log n}})$.

Proof. Building the $O(n \log |A|)$-bit data structure in Lemma 2, the Ψ function can be accessed in $O(1)$ time. Below, we describe $O(n\sqrt{\log n} \log |A|)$-bit data structures so that both SA and SA^{-1} can be computed in $O(1)$ time.

For the access of SA value, recall that Rao [85] gives an implementation of the compressed suffix array that reports $SA[i]$ in $O(1)$ time using $O(n\sqrt{\log n})$ bits for binary text string (refer to Theorem 4 in [85]). For text on a fixed finite alphabet A, Rao's idea can be generalized so that $SA[i]$ can be accessed in constant time using an $O(n\sqrt{\log n} \log |A|)$-bit data structures.

For the access of SA^{-1} value, we need the following data structure. Let $\ell = \sqrt{\log n}$. First, we store $SA^{-1}[x\ell]$ for all $0 \leq x \leq \lfloor n/\ell \rfloor$, which requires $O(n\sqrt{\log n})$ bits. Then, we need a data structure so that $\Psi^j[z]$ can be accessed in $O(1)$ time for any $0 \leq x \leq \lfloor n/\ell \rfloor$ and any $1 \leq j \leq \ell$. By Lemma 6, such data structure can be stored in $O(n\sqrt{\log n}\log|A| + |A|^{\sqrt{\log n}}\log n) = O(n\sqrt{\log n}\log|A|)$ bits (for $|A| = O(2^{\sqrt{\log n}})$).

Now we show how to access $SA^{-1}[i]$ given i in constant time. Let $y = \lfloor i/\ell \rfloor$, $k' = i - y\ell$, and $z' = SA^{-1}[y\ell]$. We claim that $SA^{-1}[i] = \Psi^{k'}[z']$ and $k' \leq \ell$. Then, using the data structures above, $SA^{-1}[i]$ can be computed in $O(1)$ time.

Note that $y\ell \leq i < (y+1)\ell$ and, $k' = i - y\ell \leq \ell$. It is then easy to verify that $\Psi^{k'}[z'] = SA^{-1}[SA[z'] + k']$ and so $SA[\Psi^{k'}[z']] = SA[z'] + k'$. Since $SA[z'] = y\ell$, we have $SA[\Psi^{k'}[z']] = y\ell + k' = i$. Thus, the claim follows. $\square$

3.2.3 Suffix tree

A suffix tree for the text T is an edge-labeled rooted directed tree with exactly $n+1$ leaves numbered 0 to n. Each edge is labeled with a non-empty substring of T such that no two outgoing edges from a node have labels with the same first character. For every node v, its path label $plabel(v)$ is constructed by concatenating the edge labels, in order, from the root to the node. Note that the path label of every leaf i, is a suffix of T that starts at position i.

We assumed that the suffixes of the leaves in the suffix tree are lexicographically ordered so that the collection of leaf nodes from left to right will form the suffix array denoted by $SA[0..n]$. For our approach, we require a suffix tree that support the following operations:

$label(u, v)$: returns the label on the edge joining node u to v in $O(xt_{SA})$ time where x is the length of the edge label of (u, v).

$plen(v)$: returns the length of the path label $plabel(v)$ in $O(t_{SA})$ time .

$leftmost(v)$: returns the SA index of the leftmost leaf in the subtree rooted at node v in $O(1)$ time.

$rightmost(v)$: returns the SA index of the rightmost leaf in the subtree rooted at node v in $O(1)$ time.

$slink(v)$: returns a node u if there is a suffix link from node v to node u in $O(t_\Psi)$ time.

$child(v, c)$: returns a child w of the node v if c is a prefix character to string $label(v, w)$ in $O(|A|t_{SA})$ time.

Lemma 8 A suffix tree with the above properties can be implemented using (1) $O(n\sqrt{\log n}\log|A|)$ bits for $t_{SA} = O(1)$ and $t_\Psi = O(1)$ and assuming $|A| = O(2^{\sqrt{\log n}})$, or (2) $O(n \log |A|)$ bits for $t_{SA} = O(\log^\epsilon n)$ and $t_\Psi = O(1)$.

Proof. We refer to Sadakane's paper [88] on compressed suffix tree (CST) implementation that uses data structure $\mathcal{D}$ and $O(n)$ bits for the balanced parentheses representation of the suffix tree [72]. The space complexities follow from Lemmas 2 and 7. $\square$

The following result on LCP query is also available.

Lemma 9 [88] Given SA indices i and j, the length of the longest common prefix (LCP) between suffixes at positions $SA[i]$ and $SA[j]$, denoted by $|lcp(i,j)|$, can be computed in $O(t_{SA})$ time using additional $O(n)$ bits data structure. The lowest common ancestor (LCA) node between any two nodes in the suffix tree can also be computed in $O(t_{SA})$ time.

3.2.4 Other data structures

Given a suffix tree ST built from the text T, and a query pattern P of length m, we define the following terminologies and data structures:

Definition 1 Given a node x in ST, let x_{le} and x_{ri} denote indices of SA corresponding to the leftmost and rightmost leaf nodes in the subtree spanned by x.

Based on the above definition, for any node x in ST, we have $[x_{le}..x_{ri}] = range(T, plabel(x))$.

Definition 2 Arrays $F_{st}[1..m]$ and $F_{ed}[1..m]$ are such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$ for $1 \leq i \leq m$. We also define $F_{st}[j] = 0$ and $F_{ed}[j] = n$ for $j > m$.

Lemma 10 $F_{st}[1..m]$ and $F_{ed}[1..m]$ can be constructed in $O(mt_{\Psi} + m|A|t_{SA})$ time.

Proof. This can be done using the suffix links in ST in $O(mt_{\Psi})$ time, given that

$range(T, P[1..m])$ can be obtained by traversing the suffix tree in $O(m|A|t_{SA})$ time.

□

Furthermore, the following two lemmas are needed to support exact pattern search over a subtree in the suffix tree.

Lemma 11 Given a pattern P, let x be a node such that $[x_{le}..x_{ri}] = range(T, P)$. For any position i in T, P is a prefix of $T[i..n]$ if and only if $x_{le} \leq SA^{-1}[i] \leq x_{ri}$.

Proof. If P is a prefix of $T[i..n]$ then by the definition of suffix tree, i corresponds to a leaf node such that P is the prefix to its path label. It follows that x must be on the path from the root to the leaf node. Hence the corresponding SA index of the leaf node, which is $SA^{-1}[i]$ must fall within the range of x_{le} and x_{ri}. On the other hand, if $x_{le} \leq SA^{-1}[i] \leq x_{ri}$, then the leaf i is in the subtree rooted at x and so $T[i..n]$ must have P as its prefix. □

Lemma 12 Given a pattern P, let x be a node such that $[x_{le}..x_{ri}] = range(T, P)$. Then, $SA^{-1}[SA[x_{le}] + |P|] < SA^{-1}[SA[x_{le} + 1] + |P|] < \ldots < SA^{-1}[SA[x_{ri}] + |P|]$.

Proof. Let i and j be any two SA indices such that $x_{le} \leq i < j \leq x_{ri}$, then $T[SA[i]..SA[i] +|P| - 1] = T[SA[j]..SA[j] + |P| - 1] = P$. Since $i < j$, then $T[SA[i] + |P|..n] < T[SA[j] + |P|..n]$ must be true. Now assume that $SA^{-1}[SA[i] + |P|] \geq SA^{-1}[SA[j] + |P|]$, which implies $T[SA[i] + |P|..n] \geq T[SA[j] + |P|..n]$. Hence a contradiction. □

3.2.5 Heavy path decomposition

We introduce a standard technique to partition $O(n)$ nodes of a tree into $O(\log n)$ levels. Similar schemes have been used for tree structure compression to give a depth of $O(\log n)$ [14, 22]. The heavy path decomposition scheme is as such: Given a suffix tree ST, we assign a level to every node in ST. The root is assigned $level\ 1$. If a node v has $level\ i$, we assign $level\ i$ to the single child node of v, that has the largest subtree (in terms of number of nodes) among all the other child nodes of v. The other child nodes of v are assigned $level\ i+1$. Edges joining 2 nodes with the same $level$ are denoted as *core edges* and the rest of edges that join nodes at $level\ i$ to nodes at $level\ i+1$ are denoted as *side edges*. An internal node will have exactly one outgoing *core edge* and the rest of the outgoing edges are *side edges*. We also denote a node with an incoming core edge as a *core node* and otherwise, a *side node*. The root, is by default, a *side node*. The followings are also observed:

Lemma 13 There are $O(\log n)$ levels.

Proof. If a subtree rooted at a node v having $level\ i$ contains l leaves, then the number of leaves of the subtree rooted at a child node u of v having $level\ i+1$ is at most $l/2$. As the suffix tree has n leaves, there are at most $O(\log n)$ levels. □

Then we can easily get the following corollary:

Corollary 14 There are $O(\log n)$ side edges on the path from the root to any node in the suffix tree.

Lemma 15 Consider any two distinct side edges e_1 and e_2 with end nodes v_1 and v_2 respectively. If both v_1 and v_2 have $level\ i$, then the two subtrees rooted at v_1 and v_2 are disjoint. In other words, the subtrees rooted at any two distinct side nodes of the same level are disjoint.

Proof. Suppose by contradiction that the two subtrees are not disjoint, then either v_1 is an ancestor of v_2 or vice versa. Suppose v_1 is an ancestor of v_2. As v_1 and v_2 are on the same level, edges running from v_1 to v_2 are all core edges, including e_2, this contradicts with the assumption that e_2 is a side edge. $\square$

Lemma 16 Given any side node v, that begins a core path (where all edges in the path are core edges), we can find the leaf node u that terminates the core path in $O(1)$ time using additional $O(n)$ bits data structure.

Proof. Let u_i be the i-th node in the suffix tree according to the preorder traversal on the nodes, and $t = O(n)$ be the number of nodes in the suffix tree. Let $X[1..t]$ be an array first initialized as blanks, and $X[i] =$'(' if u_i is an internal side node, else $X[i] =$')' if u_i is a core leaf node. It can be checked easily that the parentheses in $X[1..t]$ is balanced as every core path finishes at a leaf. More importantly, for every pair of parentheses (i, j) in X, u_i and u_j form the start and the end of some core path. Figure 3.1 gives an example of the representation. The parentheses array X can be encoded in binary representation using $O(n)$ bits based on the *Rank and Select* data structure (from Lemma 3). By the data structure for balanced parentheses (see Theorem 1 in [71]), the position of the close parenthesis that matches the given open parenthesis, can be found in constant time using additional $o(n)$ auxiliary bits. Hence, the lemma follows. $\square$

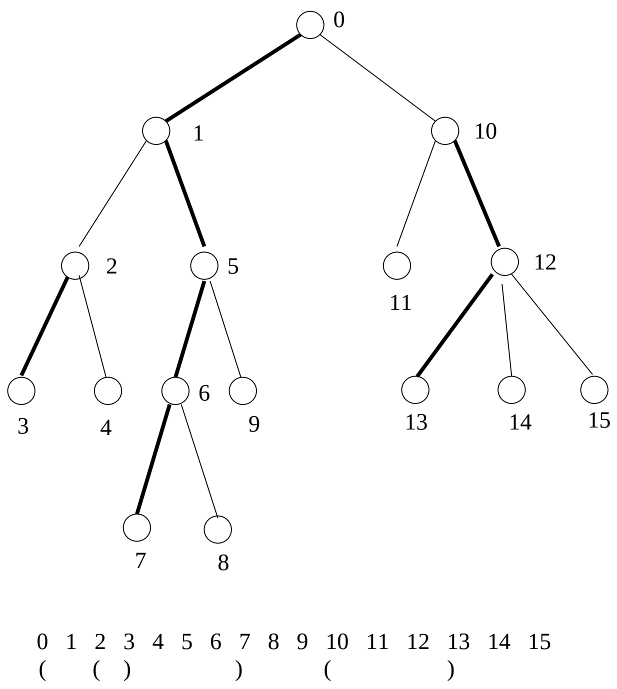


Figure 3.1: Balanced parentheses representation of core paths (thickened lines) in a suffix tree.

3.3 Approximate string matching problem

3.3.1 The data structure for 1-approximate matching

Our 1-approximate matching data structure is basically the suffix tree ST of the text T (see Section 3.2.3), together with two other data structures. First, for every side node v (see Section 3.2.5), let u be the parent node of v, we maintain a set $\Gamma_v = \{SA^{-1}[SA[i] + plen(u) + 1] \mid i \equiv 1 (\text{mod } \log^2 n) \text{ and } v_{le} \leq i \leq v_{ri}\}$.

Second, for every core leaf node u (whose SA index is k), let v be the start of the corresponding core path, we maintain 2 lists of SA indices, $H_u^l = \{i \mid i \equiv 1(\text{mod } \log^2 n), i \leq k \text{ and } |lcp(k,i)| \geq plen(v)\}$ and $H_u^r = \{i \mid i \equiv 1(\text{mod } \log^2 n),$
$i > k \text{ and } |lcp(k,i)| \geq plen(v)\}$. The values in H_u^l and H_u^r are ordered by increasing longest common prefix length $|lcp(k,i)|$.

Lemma 17 We can store Γ_v for all side nodes v using $O(n)$ bit space. In addition, for any range $[x..y]$, we can find a value i, such that $\{i \in \Gamma_v \mid x \leq i \leq y\}$ using $O(\log \log n)$ time.

Proof. By Lemma 15, all subtrees rooted at different side nodes, of the same level-ℓ, are disjoint. Hence, the total size of Γ_v for all level-ℓ side nodes v is at most $n/\log^2 n$. Since there are $O(\log n)$ levels, (by Lemma 13), the total size of Γ_v for all side nodes v is $O(n/\log n)$. We store Γ_v, for every side node v, using the *y-fast trie* [100] data structure. The size of the data structure is $O(|\Gamma_v| \log n)$ bits and it allows efficient range query in

$O(\log\log n)$ time. Since the total size of all Γ_v is $O(n/\log n)$, the lemma follows. □

Lemma 18 We can store H^l_u and H^r_u for all core leaf nodes u (whose SA index is k) using $O(n)$ bit space. In addition, for any range $[x..y]$, we can report the values $i^l \in H^l_u$ and $i^r \in H^r_u$ such that $x \le lcp(i^l, k) \le y$ and $x \le lcp(i^r, k) \le y$ using $O(\log\log n)$ time.

Proof. There are at most $n/\log^2 n$ leaf nodes whose corresponding SA index, $i \equiv 1(\text{mod } \log^2 n)$. Also, each leaf node is reachable from $O(\log n)$ side nodes v whereby each of the side node v is the beginning of some core path (by Corollary 14). It then follows that each leaf node must be included in $O(\log n)$ different H lists. Hence, the total size of H^l_u and H^r_u for all core paths is $O(n/\log n)$. Each H^l_u and H^r_u is stored using the *y-fast trie* [100] data structure. The size of the data structures is $O((|H^l_u| + |H^r_u|)\log n)$ bits. Since the total size of all H^l_u and H^r_u is $O(n/\log n)$, the lemma follows. □

By Lemmas 8, 17 and 18, we have the following:

Lemma 19 The 1-approximate matching data structure can be stored in $O(n\log|A|)$ and $O(n\sqrt{\log n}\log|A|)$ bits for $t_{SA} = O(\log^\epsilon n)$ and $t_{SA} = O(1)$ respectively. In both cases, $t_\psi = O(1)$.

Below is the key lemma for our algorithm.

Lemma 20 Consider (1) a node u in ST such that $P_1 = plabel(u)$, (2) a character c, and (3) another string P_2 with $[st..ed] = range(T, P_2)$. Let $v = child(u, c)$. Then, all occurrences of P_1cP_2 can be computed in $O(t_{SA}(\log\log n + occ))$ time where occ is the total number of occurrences of P_1cP_2 in T.

Proof. Note that $[v_{le}..v_{ri}] = range(T, P_1c)$. If $P = P_1cP_2$ occurs in T, as P_1c is a prefix of P, P must occur at position $SA[i]$ for some $v_{le} \le i \le v_{ri}$. By Lemma 11, we can verify if P occurs at position $SA[i]$ by checking if $st \le SA^{-1}[SA[i] + |P_1| + 1] \le ed$. Hence, the occurrence of P can be found by performing the above checking for all $i \in [v_{le}..v_{ri}]$. Moreover, by Lemma 12, $SA^{-1}[SA[i]+|P_1|+1]$ is increasing for $i \in [v_{le}..v_{ri}]$. Note that, for any i, $SA^{-1}[SA[i]+|P_1|+1]$ can be retrieved in $O(t_{SA})$ time. Hence, one occurrence of P can be found in $O(t_{SA}\log(v_{ri} - v_{le}))$ time by binary search.

If v is a side node, recall that we associate a set Γ_v to it, where $\Gamma_v = \{SA^{-1}[SA[i] + |P_1| + 1] \mid i \equiv 1(\text{mod } \log^2 n)$ and $i \in [v_{le}..v_{ri}]\}$. There are 3 cases.

- Case 1: Γ_v is empty. This means that the number of leaves in the subtree of v ($v_{ri} - v_{le} + 1$) is $< \log^2 n$. Using the method we just discussed, one occurrence of P can be found in $O(t_{SA}\log(v_{ri} - v_{le})) = O(t_{SA}\log\log n)$ time.

- Case 2: Γ_v is non-empty and, by Lemma 17, we find some i such that $st \le SA^{-1}[SA[i] + |P_1| + 1] \le ed$. Since any range query of ***y-fast trie*** takes $O(\log\log n)$ time, the second case follows.

- Case 3: Γ_v is non-empty and, by Lemma 17, we cannot find any i such that $st \le SA^{-1}[SA[i]+|P_1|+1] \le ed$. In this case, using $O(\log\log n)$ time, we apply ***y-fast*** *trie* to find a and b such that $SA^{-1}[SA[a] + |P_1| + 1] \in \Gamma_v$ is just smaller than st and $SA^{-1}[SA[b] + |P_1| + 1] \in \Gamma_v$ is just bigger than ed. Note that $b - a \leq \log^2 n$. Then, using the method described at the beginning of the proof, we can find one occurrence of P in $O(t_{SA} \log(b - a)) = O(t_{SA} \log \log n)$ time.

If v is a core node, let CP be the core path containing v. Since the side node lies on the path from the root node to v and would have been uncovered from traversing the suffix tree to obtain P_1, here we assume that the side node that begins the core path CP is known. We obtain the terminating leaf node x (whose SA index is k) of CP by Lemma 16 using $O(1)$ time. Next, we search for the node $r \in CP$ whose path label is of length $|P_1| + q + 1$ where $q = |lcp(SA^{-1}[SA[k] + |P_1| + 1], st)|$. By Lemma 9, q is computed in $O(t_{SA})$ time. There are 3 cases.

- Case 1: H_x is empty. This means that the number of leaves hanging from the core path is $< \log^2 n$. We can find one occurrence of P in $O(t_{SA} \log(v_{ri} - v_{le})) = O(t_{SA} \log \log n)$ time.

- Case 2: $q \geq |P_2|$. This means that leaf node x corresponds to a suffix with P as its prefix. We have recovered one occurrence of P.

- Case 3: $q < |P_2|$. First, we would like to find $j^l \in H_x^l$ and $j^r \in H_x^r$ such that $|lcp(j^l - \log^2 n, k)| \leq |P_1| + q + 1 \leq |lcp(j^l, k)|$ and $|lcp(j^r, k)| \leq |P_1| + q + 1 \leq |lcp(j^r + \log^2 n, k)|$ respectively. This can be computed in $O(\log \log n)$ time given Lemma 18. Next, using binary search, we locate i^l and i^r within the range of $j^l - \log^2 n ... j^l$ and $j^r ... j^r + \log^2 n$ such that $|lcp(i^l, k)| = |P_1| + q + 1$ and $|lcp(i^r, k)| = |P_1| + q + 1$ respectively. If both j^l and j^r are not found, the binary search is performed for $k - \log^2 n \leq i^l \leq k + \log^2 n$. The binary search takes $O(t_{SA} \log \log n)$ time. Given i^l or i^r whichever one is found, we can recover node r by performing a LCA on the leaf node at i^l or i^r with x in $O(t_{SA})$ time. If P_2 is not completely matched after node r, we can continue to search the outgoing side edges from node r as described above (case where v is a side node) using additional $O(t_{SA} \log \log n)$ time. Overall, the time taken is still $O(t_{SA} \log \log n)$.

Once we confirm that P occurs in position $SA[i]$, the remaining occurrences of P could be found by performing, for entries i' to the left and to the right of i, the above checking (that is, $st \leq SA^{-1}[SA[i'] + |P_1| + 1] \leq ed$), until we reach the boundary of $[v_{le}..v_{ri}]$ or a false case occurs. The time required is $O(t_{SA}(occ + 2))$. □

Here, we define the procedure $TreeSearch(u, c, [st..ed])$ to be the routine which finds all the occurrences of P_1cP_2 where $P_1 = plabel(u)$ and $[st..ed] = range(T, P_2)$. By Lemma 20, this procedure runs in $O(t_{SA}(\log \log n + occ))$ time.

3.3.2 The 1-approximate matching algorithm

The algorithm traverses the suffix tree from the root to find the pattern P character by character. Then, for every position i, it introduces an “error” at that position and checks for occurrences by calling $TreeSearch$. The details of the algorithm is stated in Figure 3.2.

Lemma 21 Given the indexing data structure in Section 3.3.1, we can locate all 1-mismatch (or 1-difference) occurrences of a length-m pattern P in T, using $O(t_\Psi m + t_{SA}(|A|m \log\log n + occ))$ time.

Proof. By Lemma 10, Step 1 takes $O(mt_\Psi + m|A|t_{SA})$ time. Step 2 takes $O(1)$ time.

When we traverse down the suffix tree to a node u (with $plabel(u) = P[1..i-1]$), we will execute Steps 3(a-c) for the node u. By Lemma 20, Steps 3(a-c) in total takes $O(t_{SA}(|A|m \log\log n + occ))$ time.

For Step 3(d), it takes $O(t_{SA}|A|m)$ time. The lemma follows. $\square$

By Lemmas 19 and 21, we get the following 2 theorems:

Theorem 22 Given an $O(n\sqrt{\log n}\log|A|)$-bit indexing data structure, the 1-mismatch or 1-difference problem can locate all 1-approximate occurrences of a length-m pattern P in T, using $O(|A|m \log\log n + occ)$ time, for $|A| = O(2^{\sqrt{\log n}})$.

Theorem 23 Given an $O(n \log |A|)$-bit indexing data structure, the 1-mismatch or 1-difference problem can locate all 1-approximate occurrences of a length-m pattern P in T, using $O(\log^\epsilon n(|A|m \log\log n + occ))$ time, where $0 < \epsilon \leq 1$.

```
**Algorithm** *1-approximate match*
1. Construct $F_{st}[1..m]$ and $F_{ed}[1..m]$ such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$.
2. $u =$ root node, $i = 1$.
3. Repeat
   /* Note: we maintain the invariant that $plabel(u) = P[1..i-1]$. */
   (a) Deletion at $i$ (find occurrences of $P[1..i-1]P[i+1..m]$)
       If $P[i] \neq P[i+1]$
           report the occurrences found by
           $TreeSearch(u, P[i+1], [F_{st}[i+2]..F_{ed}[i+2]])$.
   (b) Substitution at $i$ (find occurrences of $P[1..i-1]cP[i+1..m]$ for
   all $c \in A - \{P[i]\}$)
       For $c \in A - \{P[i]\}$,
           report the occurrences found by
           $TreeSearch(u, c, [F_{st}[i+1]..F_{ed}[i+1]])$.
   (c) Insertion at $i$ (find occurrences of $P[1..i-1]cP[i..m]$ for all $c \in A - \{P[i]\}$)
       For $c \in A - \{P[i]\}$,
           report the occurrences found by $TreeSearch(u, c, [F_{st}[i]..F_{ed}[i]])$.
   (d) No insertion, deletion, and substitution at $i$
       Let $v = child(u, P[i])$, $E = label(u, v)$.
       If $P[i..i+|E|-1] = E$
           $u = v$, $i = i + |E|$
       Else
           Find the smallest $j > i$ such that $P[j] \neq E[j-i+1]$.
           Report all the occurrences of $P$ so that the error is at $j$.
           Terminate and return.
```

Figure 3.2: Algorithm for 1-mismatch and 1-difference.

3.3.3 The k-approximate matching problem with $k \geq 1$

Extending the data structure to address the k-mismatch or the k-difference problem requires the result from dynamic programming for string correction. Given 2 strings P and P' of length m and n, we can use standard dynamic programming approach to find the edit distance between any prefix of P and P' in $O(mn)$ [90], by filling a table E of size $(m+1) \times (n+1)$. Entry $E(i,j)$ stores the edit distance between $P[1..i]$ and $P'[1..j]$. For $1 \leq i \leq m$ and $1 \leq j \leq n$, the table $E(i,j)$ is evaluated as follows:

$$\begin{aligned}
E(0,0) &= 0; \\
E(0,j) &= E(0,j-1) + c(\epsilon, P'[j]), \qquad 1 \leq j \leq n; \\
E(i,0) &= E(i-1,0) + c(P[i], \epsilon), \qquad 1 \leq i \leq m;
\end{aligned}$$

$$E(i,j) = \min \begin{cases} E(i-1,j) + c(P[i],\epsilon) \\ E(i-1,j-1) + c(P[i],P'[j]) \\ E(i,j-1) + c(\epsilon, P'[j]) \end{cases}$$

Note that $c(P[i],\epsilon)$ and $c(\epsilon, P'[j])$ are the edit costs for deletion and insertion. Their values equal 1. $c(P[i],P'[j])$ is the edit cost for substitution. We have $c(P[i],P'[j]) = 0$ if $P[i] = P'[j]$; and 1 otherwise. Moreover, we need to find only those prefixes (or actually the shortest prefix) of P' that is at most k edit distance from P. A match is read from the entries in the last row of table E that is $\leq k$. We proceed by filling the table column by column from left to right, up to the $\min\{m+k, n\}$th columns. There are at most $2k+1$ row entries to be filled in each column as entries in $E(i,j)$ where $i > j+k$ or $i < j-k$ will have edit cost $> k$. An example is shown in Figure 3.3 for $k = 2$. Hence, finding the prefixes of P' that is at most k edit distance from P, takes $O((m+k) \times (2k+1)) = O(mk)$ time with $k \leq m$. We state the result in the following lemma.

		C	A	T	A	G	T	T	C	A	C	G	G
	0	1	2	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
A	1	1	1	2	▨	▨	▨	▨	▨	▨	▨	▨	▨
A	2	2	1	2	2	▨	▨	▨	▨	▨	▨	▨	▨
T	▨	3	2	1	2	3	▨	▨	▨	▨	▨	▨	▨
G	▨	▨	3	2	2	2	3	▨	▨	▨	▨	▨	▨
T	▨	▨	▨	3	3	3	2	3	▨	▨	▨	▨	▨
T	▨	▨	▨	▨	4	4	3	2	3	▨	▨	▨	▨
C	▨	▨	▨	▨	▨	5	4	3	2	3	▨	▨	▨
A	▨	▨	▨	▨	▨	▨	5	4	3	**2**	3	▨	▨

The shaded entries will have edit distance > 2 and hence need not be filled.

Figure 3.3: Edit distance table between 2 strings P = "AATGTTCA" and P' = "CATAGTTCACGG" with $k = 2$.

Lemma 24 Given 2 strings P and P' of length m and n respectively, we can find the prefixes of P' that is at most k edit distance from P in $O(mk)$ time.

Our solution for the k-approximate matching tries to apply the dynamic programming on various paths of the suffix tree. We need the following 2 lemmas.

Lemma 25 Given a length-m string P, there are $O(|A|^k m^k)$ different path labels P' in the suffix tree such that the edit distance between P and P' is at most k, where $|A|$ is the fixed alphabet size.

Proof. The number of possible P' in the suffix tree is bounded by the number of edit traces that can be applied to the string P. Hence, by Lemma 1, the lemma follows. $\square$

Lemma 26 After preprocessing the text T of length n and obtaining an $O(n\sqrt{\log n}\log|A|)$ or $O(n\log|A|)$ bits data structure, the k-mismatch or k-difference problem can locate all approximate occurrences of a length-m pattern P in T, using $O(|A|^k m^{k+1}(k + t_{SA}) + t_{SA}occ)$ time where $|A|$ is the alphabet size and occ is the number of approximate occurrences of P in T.

Proof. The result can be achieved by applying dynamic programming over the suffix tree (see Lemma 8) by computing the shortest prefixes on all paths starting from the root that has edit distance at most k from P. This is performed through a preorder traversal on the suffix tree. We maintain P' to be the path label in the suffix tree during the traversal. First, traverse down the leftmost path, and as we advance to a new character at position j on path label, P', we add and compute for new column $E(*, j)$ in the edit distance table E between strings $P[1..m]$ and $P'[1..j]$. A new column can be computed in $O(k)$ time (see Lemma 24), while accessing $P'[j]$, a character on some edge label, takes $O(t_{SA})$ time. The path terminates at position j when the edit distance in the new column, $E(*, j)$, is $> k$, or when $E(m, j)$ is filled. If $E(m, j) \leq k$, we can then report the occurrences within the SA range of the subtree rooted at current position j, using the $leftmost$ and $rightmost$ operations in $O(t_{SA}occ)$ time. This is followed by backtracking to the next path (based on the preordering of the nodes) and erasing the last few columns added to E, which equals to the number of characters backtracked. Since the number of paths to traverse is bounded (refer to Lemma 25), we conclude that the search time is $O(|A|^k m^k \times m(k + t_{SA}) + t_{SA}occ) = O(|A|^k m^{k+1}(k + t_{SA}) + t_{SA}occ)$.

It is worth mentioning here that similar results can be obtained by applying backward search on CSA [44] that gives $O(|A|^k m^{k+1}(k + \log\log|A|) + occ \log^\epsilon n)$ search time, using $O(n \log |A|)$ bits space. □

Theorem 27 After preprocessing the text T of length n and obtaining an $O(n\sqrt{\log n}\log|A|)$ (assume $|A| = O(2^{\sqrt{\log n}})$) or $O(n \log |A|)$ bits data structure, the k-mismatch or k-difference problem can locate all approximate occurrences of a length-m pattern P in T, using $O(|A|^k m^k (k+\log\log n)+occ)$ or $O(\log^\epsilon n(|A|^k m^k(k+\log\log n)+occ))$ time respectively, where $0 < \epsilon \leq 1$, $|A|$ is the alphabet size and occ is the number of approximate occurrences of P in T.

Proof. The result is achieved in 2 steps. First, by Lemma 26, we solve for the $(k-1)$-mismatch or $(k-1)$-difference problem in $O(|A|^{k-1}m^k(k + t_{SA}) + t_{SA}occ_{<k})$ where $occ_{<k}$ is the number of approximate occurrences of P in T with edit distance $< k$. What remains is to find those occurrences of P with edit distance exactly k in the text. In step 2, for those paths terminating with edit distance $k-1$, we can further solve for the kth error using the previous results for 1-mismatch or 1-difference (refer to Lemma 21). Since there are $O(|A|^{k-1}m^{k-1})$ paths with edit distance $k-1$ (see Lemma 25), we can find the occurrences of P with exactly k-mismatch or k-difference using $O(|A|^{k-1}m^{k-1} \times (t_{\Psi}m + t_{SA}|A|m \log\log n) + t_{SA}occ_k)$ time where occ_k is the number of approximate occurrences of P in T with edit distance k.

Combining steps 1 and 2, we have the solution in $O(|A|^{k-1}m^k(k + t_{SA} + t_{\Psi} + t_{SA}|A|\log\log n) + t_{SA}occ)$ time. By Lemmas 8 and 19, the theorem follows.

$\square$

3.3.4 The k-don't-cares problem

A restrictive form of the k mismatch problem is the k don't cares or wildcards problem where mismatches are allowed only at specific k positions on the pattern (as presented with the pattern). We give the following results using the data structures described in Section 3.3.1.

Lemma 28 After preprocessing the text T of length n and obtaining an $O(n\sqrt{\log n}\log|A|)$ (assume $|A| = O(2^{\sqrt{\log n}})$) or $O(n\log|A|)$ bits data structure, the 1-don't-care problem can locate all approximate occurrences of a length-m pattern P in T, using $O(|A|(m+\log\log n)+occ)$ or $O(\log^{\epsilon} n(|A|(m+\log\log n)+occ))$ time respectively, where $0 < \epsilon \leq 1$, $|A|$ is the alphabet size and occ is the number of approximate occurrences of P in T.

Proof. The proof follows from Lemma 21, where the algorithm outlined in Figure 3.2 is simplified with Steps 3(a-c) replaced by a single substitution only at the single position i where the don't care is allowed. Now Step 3c will take $O(t_{SA}|A|\log\log n + occ)$ time. This gives a total of $O(t_\Psi m + t_{SA}|A|(m + \log\log n) + occ))$ time. (Refer to the proof of Lemma 21 for details.)

By Lemma 19, the lemma follows. □

Also, we can extend the search for $k > 1$ don't cares.

Lemma 29 After preprocessing the text T of length n and obtaining an $O(n\sqrt{\log n}\log|A|)$ (assume $|A| = O(2^{\sqrt{\log n}})$) or $O(n\log|A|)$ bits data structure, the k-don't-care problem can locate all approximate occurrences of a length-m pattern P in T, using $O(|A|^k(m + \log\log n) + occ)$ or $O(\log^\epsilon n(|A|^k(m + |A|\log\log n) + occ))$ time respectively, where $0 < \epsilon \leq 1$, $|A|$ is the alphabet size and occ is the number of approximate occurrences of P in T.

Proof. For k don't cares, we might have to search up to $|A|^k$ different paths down the suffix tree. Traversing and matching the characters on a single path of length m takes $O(t_{SA}|A|m)$ time and reporting the occurrences takes $(t_{SA}occ)$. This gives a total time of $O(t_{SA}(|A|^{k+1}m + occ))$ to report the location of all k don't cares patterns.

However, we can further reduce the time by searching for the k-th don't care using the result in Lemma 28. We can now search for k don't cares in $O(t_\Psi m + t_{SA}|A|^k(m + |A|\log\log n) + occ)$ time. By Lemma 19, the lemma follows. □

3.4 Summary

Approximate string matching is about finding a given string pattern in a text by allowing some degree of errors. In this work we present a space-efficient data structure to solve the 1-mismatch and 1-difference problems. Given a text T of length n over an alphabet A, we can preprocess T and give an $O(n\sqrt{\log n}\log|A|)$-bit space data structure so that, for any query pattern P of length m, we can find all 1-mismatch (or 1-difference) occurrences of P in $O(|A|m\log\log n\ +occ)$ time, where occ is the number of occurrences. This is the fastest known query time given that the space of the data structure is $o(n\log^2 n)$ bits.

The space of our data structure can be further reduced to $O(n\log|A|)$ if we can afford a slow down factor of $\log^\epsilon n$, for $0 < \epsilon \leq 1$. Furthermore, our solution can be generalized to solve the k-mismatch (and the k-difference) problem in $O(|A|^k m^k(k+\log\log n)+occ)$ and $O(\log^\epsilon n(|A|^k m^k(k+\log\log n)+occ))$ query time using an $O(n\sqrt{\log n}\log|A|)$-bit and an $O(n\log|A|)$-bit indexing data structures, respectively. We assume that the alphabet size $|A|$ is bounded by $O(2^{\sqrt{\log n}})$ for the $O(n\sqrt{\log n}\log|A|)$-bit space data structure.

The above results were first presented in [61] and the extended version has been accepted for journal publication [62]. More recently, H.L. Chan *et al.* [18] incorporated some results presented here to give an improvement using fixed index space in answering approximate pattern matching. The results are $O(m+(c\log n)^{k(k+1)}\log\log n+occ)$ and

$O(m + \log^{\epsilon} n((c \log n)^{k(k+2)} \log \log n + occ))$ time using $O(n \log n)$ and $O(n \log |A|)$ bit space respectively for some positive constants c and ϵ. Also in another recent work, H.L. Chan *et al.* [17] gave results using $O(n \log n)$-bit index to report approximate pattern matching in $O(m + \log n \log \log n + occ)$ and $O(m \log n \log \log n + occ)$ time for $k = 1$ and $k = 2$ respectively.

Chapter 4

Optimal exact match index

4.1 Introduction

Given a text $T[1..n]$, characters from a finite alphabet A, and with necessary preprocessing and building an indexing structure, we locate the exact matches of any given pattern $P[1..m]$ in T. This defines the exact string matching problem. In this work, we are interested in finding the optimal query time for exact string matching problem using $o(n \log n)$-bits data structures to index the text. In other words, we build highly compressed indices to answer exact match query efficiently.

It is well known that suffix tree data structure uses $O(n \log n)$ bits so that exact string matching can be answered in $O(m + occ)$ time where occ is the number of occurrences of the query pattern found in the text. Compressed suffix array (CSA)[44] reduces the space to $O(n \log |A|)$ bits in query time $O(m \log \log |A| + occ \log^{\epsilon} n)$, for $\epsilon > 0$, based on backward search. Grossi and Vitter [38, 39] gave an index on compressed suffix array (CSA) for exact match in $O(m/\log_{|A|} n + \log^{1+\epsilon} n(\log|A| + \log\log n) + occ)$ time using $O(n\log|A|)$ bits for $m = \Omega(\log^{1+\epsilon} n)$ or $occ = \Omega(n^{\epsilon})$. Their result is optimal in $O(m/\log_{|A|} n + occ)$ query time for $m = \Omega(\log^{2+\epsilon} n \log_{|A|}\log n)$. The Lempel-Ziv (LZ) index by K$\ddot{a}$rkk$\ddot{a}$inen and Sutinen [53] was cited where occupying $O(n)$ bits, any pattern of length $m \leq \epsilon\log_{|A|} n$, where $0 < \epsilon \leq 1$, can be found in $O(m + occ)$ time (actually $O(1 + occ)$ time with suitable table lookup).

Here, we present an algorithm that finds and enumerates all the occurrences in $O(m/\log_{|A|} n + \log^{1+\epsilon} n \log_{|A|}\log n + occ)$ time for $m = \Omega(\log_{|A|} n \log^{\epsilon} n)$, $\epsilon > 0$. We give the optimal search time of $O(m/\log_{|A|} n + occ)$ for $m = \Omega(\log^2_{|A|} n \log^{\epsilon} n \log\log n)$. This improves the previously reported result especially for large alphabet size using $O(n\log|A|)$ bits by a factor of $O(\log|A|)$. Next, to handle query of any given length m, we use $O(n\log^{\epsilon} n\log|A|)$ bit data structures to answer exact match query in $O(m/\log_{|A|} n + \log n\log\log n + occ)$ time. Finally, we give $O(n\sqrt{\log n}\log|A|)$ bit data structures with $O(m/\log_{|A|} n + \log^{\epsilon}_{|A|} n + occ)$ or $O(m + occ)$ query time. The result in $O(m + occ)$ time is optimal if we take into consideration the $O(m)$ preprocessing time to encode a pattern string of length m into $m/\log_{|A|} n$ words. This gives the first linear $O(m + occ)$ time using only $o(n\log n)$ bits space for exact string matching over a fixed finite alphabet. Table 4.1 summarizes the results for exact string matching over a finite alphabet.

Reference	***Bit space***	***Query time***
Suffix tree	$O(n \log n)$	$O(m + occ)$
CSA* [44]	$O(n \log \vert A \vert)$	$O(m \log\log \vert A \vert + occ \log^\epsilon n)$
Grossi et al. [39]	$O(n \log \vert A \vert)$	$O(m / \log_{\vert A \vert} n + occ)$ for $m = \Omega(\log^{2+\epsilon} n \log_{\vert A \vert} \log n)$
This work	$O(n \log \vert A \vert)$	$O(m / \log_{\vert A \vert} n + occ)$ for $m = \Omega(\log^2_{\vert A \vert} n \log^\epsilon n \log\log n)$
	$O(n \log^\epsilon n \log \vert A \vert)$	$O(m / \log_{\vert A \vert} n + \log n \log\log n + occ)$
	$O(n \sqrt{\log n} \log \vert A \vert)$	$O((m / \log_{\vert A \vert} n + \log^\epsilon_{\vert A \vert} n + occ)$ or $O(m + occ)$

* - backward search on CSA, and $\epsilon > 0$

Table 4.1: Comparison of various results for exact string matching problem.

4.2 The approach

4.2.1 Basic concept

We first consider a compact trie, ST_w, for suffixes of T at positions or split-points $w, 2w, \ldots, \lfloor n/w \rfloor w$, using $O(n/w \log n)$ bits. w is a constant value to be determined later. Let T_i denotes the suffix $T[i..n]$. The compact trie, ST_w, consists of suffixes T_{iw}, $1 \leq i \leq \lfloor n/w \rfloor$. Any matching pattern of length $\geq w$ in T will cover at least one split-point. We have another compact trie, $\overline{ST_w}$, for reverse prefixes of T at positions $w-1, 2w-1, \ldots, \lfloor n/w \rfloor w - 1$, using $O(n/w \log n)$ bits. The compact trie, $\overline{ST_w}$, consists of $\lfloor n/w \rfloor$ substrings $T[iw-1..1]$, $1 \leq i \leq \lfloor n/w \rfloor$. The compact tries are sorted in lexicographical order, with leaf nodes storing the text positions iw and $iw-1$, $1 \leq i \leq \lfloor n/w \rfloor$, for ST_w and $\overline{ST_w}$ respectively. This gives a search strategy to find exact match given the query pattern $P[1..m]$. We say that P occurs at position i on the text T if there exists an integer j where $(j-1)w < i \leq jw$, such that $P[jw-i+1..m]$ is a prefix to T_{jw} in ST_w, and $P[jw-i..1]$ is a prefix to $T[jw-1..1]$ in $\overline{ST_w}$.

We generalize the search for pattern P in text T, by first splitting P into 2 parts, head and tail, $P[j-1..1]$ and $P[j..m]$, for $j = 1$ to w. Next for each j, we search for $P[j-1..1]$ in $\overline{ST_w}$ and $P[j..m]$ in ST_w respectively. Let the largest leaf ranges with common prefixes matching the respective head and tail patterns, in $\overline{ST_w}$ and ST_w for a given j, be $[x^l_j..x^r_j]$ and $[y^l_j..y^r_j]$. The leaf indices are enumerated in left to right order. Also let $\overline{ST_w}[x]$ denotes the text position stored in the leaf node x in $\overline{ST_w}$, similarly for $ST_w[y]$. The final step is to return those positions where $x \in [x^l_j..x^r_j]$ and $y \in [y^l_j..y^r_j]$, such that $ST_w[y] = \overline{ST_w}[x]+1$, with the matching occurrence located at position $ST_w[y] - j + 1$ in T.

4.2.2 Data structures

We now describe some data structures that can help to speed up the search. Given the leaf ranges $[x^l_j..x^r_j]$ and $[y^l_j..y^r_j]$, we identify the "correct" matching leaf nodes, by transforming the problem into a two-dimensional orthogonal range search. Below is a known result:

Lemma 30 [3] Let S be a set of points in $[1..n] \times [1..n]$, where $|S| = n$. Given x_1, x_2, y_1, y_2, we can find $L = \{(x, y) \in S \mid x_1 \leq x \leq x_2 \text{ and } y_1 \leq y \leq y_2\}$, in $O(\log \log n + |L|)$ query time using $O(n \log^{1+\epsilon} n)$ bit space, $\epsilon > 0$.

We have S defined as the leaf index range of $\overline{ST_w} \times$ the leaf index range of ST_w . A point in S, (x, y), corresponds to the occurrence of $ST_w[y] = \overline{ST_w}[x] + 1$. In our case, there are only $\lfloor n/w \rfloor$ points.

The following data structures are needed for our results:

Lemma 31 [39] Compressed suffix array (CSA) using $O(n \log |A|)$ bit space, reports SA (suffix array) and Ψ entries in $t_{SA} = O(\log^{\epsilon}_{|A|} n)$, where $0 < \epsilon \leq 1$, and $t_{\Psi} = O(1)$ time respectively. (The function Ψ is defined as such $\Psi[i] = SA^{-1}[SA[i] + 1]$).)

Lemma 32 [85] Compressed suffix array (CSA) can be implemented using $O(n\sqrt{\log n} \log |A|)$ bit space, to report SA (suffix array) and Ψ entries in constant time.

Proof. This is generalized from Rao's paper for binary text string. □

Lemma 33 [87, 88] Given SA indices i and j and an implementation of the CSA, the length of the longest common prefix (LCP) between suffixes at positions $SA[i]$ and $SA[j]$, denoted by $|lcp(i, j)|$, can be computed in $O(t_{SA})$ time using additional $O(n)$ bits data structure. The SA range corresponding to the lowest common ancestor (LCA) node between any two suffixes can be computed in constant time.

Lemma 34 Given a Patricia trie storing s strings of length at least $\log_{|A|} n$, each over the alphabet A, we can search for a pattern of length m in $O(m/\log_{|A|} n + \log^{\epsilon}_{|A|} n)$ time. The Patricia trie uses $O(s \log n)$ bit space.

Proof. Refer to Lemma 9 in Grossi and Vitter's paper [39]. □

Lemma 35 [84] Let S be a set of m elements drawn from $[1..n]$. S can be represented using $m \log(n/m) + O(m + \log\log n)$ bits such that the following rank and select operations can be performed in constant time. A rank operation returns the order of an element $x \in S$, defined as $Rank[x] = |\{y < x \mid y \in S\}|$. A select operation returns the i-th smallest element in S, where $1 \leq i \leq m$ (i.e. $Select[i] = x$ if $Rank[x] = i$).

4.2.3 Using $O(n \log |A|)$ bit data structures

We first use $SA[1..n]$ to denote the compressed SA in the CSA build for the text T. Let PT be a compact trie, more specifically a Patricia trie as defined in Lemma 34 for suffix strings $T[i..m]$, such that $i = SA[j]$ and $j \bmod w = 0$. Suppose the pattern $P[1..m]$ exists in the PT which corresponds to suffixes $SA[iw]$ for $i \in [a..b]$. The range $[a..b]$ is mapped to its SA range $[aw..bw]$ in the CSA. Next the index range $[aw..bw]$ is extended to its left and right for $[a'..b']$ which correspond to the LCA to the suffixes in the range $[aw..bw]$.

There is the special case where $P[1..m]$ is not found in PT, as only $P[1..i]$, $i < m$, has been matched. We can determine the leaf range $[a..a+1]$ so that $P[1..m]$ if found will exist between leaf index a and $a + 1$. We search within the SA range $[aw..bw]$, $(b = a + 1)$, of the CSA for $[a'..b']$ with prefix matching $P[1..m]$. This is performed using binary search in $O(m/\log_{|A|} n + t_{SA} \log w)$ by packing $\log_{|A|} n$ characters into a single word for constant time comparison, and applying the LCP result. This is similar to binary search over suffix array with the use of LCP.

Now that we have found the SA range for $P[1..m]$, using the Ψ function in CSA, we locate the rest of the SA ranges for $P[i..m]$, $1 \leq i \leq w$. For each SA range, we find within the range, the ranks of the first and last entries whose text position mod $w = 0$. The ranks are in fact the leaf range in ST_w (see Section 4.2.1) with prefix matching some $P[i..m]$. Let the leaf range in ST_w with strings starting with $P[i..m]$ be $[y_i^l..y_i^r]$. The following Lemma gives the details:

Lemma 36 Using $O(n/w \log w)$ bit data structures, and given the SA range $[i..j]$, $1 \leq i \leq j \leq n$, where w is a constant, we can find the smallest i' and the largest j' such that $i', j' \in [i..j]$, and $i', j' \in D$, in $O(1)$ time. We define the list $D = \{i \in [1..n] \mid SA[i] \text{mod } w = 0\}$. We can also determine the rank of $i = |\{j \leq i \mid j \in D\}|$, in $O(1)$ time.

Proof. Imagine a bit array of size n of 0's and 1's where the 1's mark the positions of D in the SA. There are n/w 1's in the bit array of length n and the space to store using the rank and select data structure (see Lemma 35) is $O(n/w \log(n/(n/w))) = O(n/w \log w)$. Given a position i, we can count the number of 1's before and at i in

$O(1)$ time. Similarly, given the rank of some 1's in the array, we can find its position in $O(1)$ time. □

To reiterate, for each SA range corresponding to $P[i..m]$, $1 < i \leq w$, using Lemma 36, we find the leaf range $[y_i^l..y_i^r]$ in ST_w in constant time. Next we construct the compact trie $\overline{ST_w}$ as described earlier and find $P[i-1..1]$ for $1 < i \leq w$, to yield the leaf range $[x_i^l..x_i^r]$.

Given the leaf ranges $[y_i^l..y_i^r]$ and $[x_i^l..x_i^r]$, we search using the range query data structure in $O(\log\log n + occ_i)$ time (see Lemma 30). This is performed w times, for $1 \leq i \leq w$, to collect all the occurrences. Notice that ST_w is not actually constructed.

This leads to the following theorem:

Theorem 37 Given $O(n \log |A|)$ bit data structures, we find all the exact match locations of length-m pattern P, in text T of size n, drawn from alphabet A, in $O(m/\log_{|A|} n + \log^{1+\epsilon} n \log_{|A|} \log n + occ)$ time, where occ is the number of occurrences of P in T and $\epsilon > 0$. We assume that $m = \Omega(\log_{|A|} n \log^{\epsilon} n)$.

Proof. The data structure PT occupies $O(n/w \log n)$ bits and the search for $P[1..m]$ takes $O(m/\log_{|A|} n + \log_{|A|}^{\epsilon} n)$ time (by Lemma 34). The SA range on the CSA for pattern $P[1..m]$ is found using further $O(n + |CSA|)$ bits for lcp query (by Lemma 33) and takes $O(m/\log_{|A|} n + t_{SA} \log w)$ time. The CSA takes $|CSA|$ bits and accessing the Ψ function takes a total of $O(wt_{\Psi})$ time to recover the rest of $P[i..m]$, $1 < i \leq w$. Mapping the SA ranges to ST_w leaf ranges takes $O(w)$ time as well using additional

$O(n/w \log w)$ bits (by Lemma 36). $\overline{ST_w}$ occupies $O(n/w \log n)$ bits and searching $\overline{ST_w}$ takes $O(w/\log_{|A|} n + \log_{|A|}^{\epsilon} n)$ time if we implement $\overline{ST_w}$ similar to PT. The final step of two-dimensional range query takes $O(n/w \log^{1+\epsilon} n)$ bits and $O(w \log\log n + occ)$ time.

In total, we need $O(n/w \log^{1+\epsilon} n + |CSA| + n)$ bits to answer the exact match query in $O(m/\log_{|A|} n + t_{SA} \log w + t_{\Psi} w + \log_{|A|}^{\epsilon} n + w \log\log n + occ)$ time. The theorem follows if we set $w = \log_{|A|} n \log^{\epsilon} n$ and uses the CSA given in Lemma 31 where $|CSA| = O(n \log |A|)$, $t_{\Psi} = O(1)$ and $t_{SA} = O(\log^{\epsilon} n)$. $\square$

The theorem can be extended to give the following result.

Corollary 38 Exact match can be found in $O(m/\log_{|A|} n + occ)$ optimal time using $O(n \log |A|)$ bit data structures for $m = \Omega(\log_{|A|}^{2} n \log^{\epsilon} n \log\log n)$.

4.2.4 Using $O(n \log^{\epsilon} n \log |A|)$ bit data structures

We now give a general solution for query of any length m.

Theorem 39 Given $O(n \log^{\epsilon} n \log |A|)$ bit data structures, we find all the exact match locations of length-m pattern P, in text T of size n, drawn from alphabet A, in $O(m/\log_{|A|} n + \log n \log\log n + occ)$ time, where occ is the number of occurrences of P in T and $\epsilon > 0$.

Proof. By setting $w = \epsilon \log_{|A|} n$, $0 < \epsilon \leq 1$, as given in the proof for Theorem 37, we have the query time in $O(m/\log_{|A|} n + \log_{|A|} n \log\log n + \log^{\epsilon} n \log\log_{|A|} n + occ)$

$= O(m/\log_{|A|} n + \log n \log\log n + occ)$ for $m \geq \epsilon \log_{|A|} n$ using $O(n \log^{\epsilon} n \log |A|)$ bit space. For $m < \epsilon \log_{|A|} n$, we use the LZ-index by K$\ddot{a}$rkk$\ddot{a}$inen and Sutinen in $O(1+occ)$ time using $O(n)$ bits. $\square$

Corollary 40 Exact match can be found in $O(m/\log_{|A|} n + occ)$ optimal time using $O(n \log^{\epsilon} n \log |A|)$ bit data structures for $m = \Omega(\log^2_{|A|} n \log\log n + \log^{1+\epsilon} n \log_{|A|} \log_{|A|} n)$.

4.2.5 Using $O(n\sqrt{\log n}\log |A|)$ bit data structures

We replace the CSA used in previous results with Rao's implementation in $O(n\sqrt{\log n}\log |A|)$ bits so that SA can be accessed directly in constant time i.e. $t_{SA} = O(1)$. First we build compact trie PT for suffix strings $T[i..m]$, such that $i = SA[j]$ and $j \bmod w = 0$. We search for pattern $P[1..m]$ in PT to obtain the leaf range $[a..b]$ and then find the SA range in CSA with common prefix matching $P[1..m]$. The details has been discussed in Section 4.2.3. Now using Rao's CSA, we can then recover the matching position entries in the SA range, in $O(occ)$ time.

We give the following result.

Theorem 41 Given $O(n\sqrt{\log n}\log |A|)$ bit data structures, we find all the exact match locations of length-m pattern P, in text T of size n, drawn from alphabet A, in $O(m/\log_{|A|} n + \log^{\epsilon}_{|A|} n + occ)$ time, where occ is the number of occurrences of P in T and $\epsilon > 0$.

Proof. The data structure PT occupies $O(n/w \log n)$ bits and the search for $P[1..m]$ takes $O(m/\log_{|A|} n + \log_{|A|}^{\epsilon} n)$ time (by Lemma 34). The SA range on the CSA for pattern $P[1..m]$ is found using further $O(n + |CSA|)$ bits for lcp query (by Lemma 33) and takes $O(m/\log_{|A|} n + t_{SA} \log w)$ time.

In total, using Rao's CSA (see Lemma 32) in $O(n\sqrt{\log n} \log |A|)$ bits with $t_{SA} = O(1)$, we need $O(n/w \log n + n\sqrt{\log n} \log |A| + n)$ bits to answer the exact match query in $O(m/\log_{|A|} n + \log_{|A|}^{\epsilon} n + \log w + occ)$ time. The theorem follows if we set $w = \sqrt{\log_{|A|} n}$, for $m \geq \sqrt{\log_{|A|} n}$. For $m < \sqrt{\log_{|A|} n}$, we refer to the LZ-index. $\square$

We can further deduce the followings.

Corollary 42 Exact match can be found in $O(m/\log_{|A|} n + occ)$ optimal time using $O(n\sqrt{\log n} \log |A|)$ bit data structures for $m = \Omega(\log_{|A|}^{1+\epsilon} n)$.

Corollary 43 Exact match can be found in $O(m + occ)$ time using $O(n\sqrt{\log n} \log |A|)$ bit data structures.

4.3 Summary

We have studied the exact string matching problem and provided tighter bounds on the optimal solution in terms of space and query time trade-offs. First we show that using $O(n \log |A|)$ bits data structure, the optimal query time of $O(m/\log_{|A|} n + occ)$ can be achieved for $m = \Omega(\log_{|A|}^2 n \log^{\epsilon} n \log \log n)$, where $\epsilon > 0$. This extends the range of m

answerable by a factor of $\log |A|$ from previously known result. Next using $o(n \log n)$ bits data structure, for fixed finite alphabet and any pattern of length m, we answer the exact string matching problem with an $\log n \log \log n$ term added to the optimal query time. Also we show that $(m + occ)$ query time is achievable using $o(n \log n)$ bits data structure.

Chapter 5

Disk-based suffix tree index

5.1 Introduction

Suffix tree is an important data structure for indexing text string since it can answer pattern search query efficiently independent of the text string size. There exists many practical applications that rely on suffix tree, especially for processing biological sequence data [40, 42, 50, 58, 59, 81]. As various genome sequencing projects are ongoing and more genome sequences are made known, the application of suffix tree on biological research is expected to increase.

Since genome size is in the order of gigabytes, maintaining suffix trees becomes an important issue. There are two immediate problems. The first problem is on constructing suffix tree efficiently. Many suffix tree construction algorithms have been proposed over the years [13, 19, 34, 44, 45, 67, 95, 98, 99]. We are now able to construct a suffix tree (or a suffix array) for the human genome of 3 billion characters within 30 hours on a desktop machine with 4GB RAM [60, 95]. Hence, the problem on suffix tree construction is largely solved in practice.

The second problem is on accessing the suffix tree. As the genome database gets bigger, maintaining suffix tree in memory is no longer feasible. We need to have a disk-based representation of suffix tree which allows for efficient access. We have seen a number of disk-based representations of suffix tree [20, 29, 45, 69, 95] in the literature. However, these disk-based suffix trees either fail to support all the general suffix tree operations well or have high IO disk access for certain operations.

This work focuses on having a practical suffix tree implementation on disk that supports various suffix tree operations efficiently. We propose a Compact Partitioned Suffix tree representation (CPS-tree) for disk-based access. Our CPS-tree achieves good IO bound and time complexity, and is shown to be efficient on real datasets as well. We study ways to localize information in the tree so that further traversal down the tree is minimized. This is achieved by propagating the suffix position in selected leaf nodes up the tree to be stored locally. Also we add "shortcuts" into the tree so that some intermediate nodes (or more correctly, pages containing the nodes) can be skipped when traversing the tree. This guarantees matching any substring in the tree with $O(\log n)$ index pages. Table 5.1 gives a list of notations used throughout this chapter for easy reference. We study and compare various tree partitioning methods to divide the tree into logical blocks and the method that works best with our CPS-tree. Index buffers are also created to identify suitable buffer replacement policy that generates fewer page faults on the CPS-tree index. We build CPS-tree index on the human genome and study the IO and computational performances. Results show that CPS-tree can support exact match and local alignment queries efficiently for large genome.

Notation	Description
n	Index size
N	Length of text string to be indexed
b	Logical block size (in bytes)
B	Memory page size (in bytes)
m	Query string length
occ	Number of matching occurrences of the query on the text
$\|A\|$	Alphabet size + 1
ℓ	Edge label length
H	Suffix tree depth

Table 5.1: Description of notations used.

Suffix tree finds many applications in pattern searching on genome sequences (such examples are Mummer[59] and Weeder[81]). These applications are memory based and hence only handle genomes that are small. For a large genome that needs to reside on disk, the disk IO efficiency becomes an important issue. As such we first study the disk IO efficiency of our proposed suffix tree to answer exact match query and to handle tree traversal operations. In Table 5.2, we present the worst case disk access performance of

Suffix structure	Exact match query	Exact match count query	Edge label access	Child node access
SB-tree[29]	$\log_B n + \frac{m+occ}{B}$	$\log_B n + \frac{m}{B}$	$\log_B n + \frac{\ell}{B}$	$\log_B n + \frac{\ell}{B}$
CPT [20]	$\frac{H}{\sqrt{B}} + \log_B n$ $+\frac{m+occ}{B}$	$\frac{H}{\sqrt{B}} + \log_B n$ $+\frac{m+occ}{B}$	$\frac{H}{\sqrt{B}} + \log_B n$ $+\frac{\ell}{B}$	$\log \lvert A \rvert$
WOTD-tree[34, 95]	$\min\{m, H\}$ $+\frac{m+occ}{B}$	$\min\{m, H\}$ $+\frac{m+occ}{B}$	$\frac{\ell}{B}$	1
CPS-tree	$\min\{m, \log n\}$ $+\frac{m+occ}{B}$	$\min\{m, \log n\}$ $+\frac{m}{B}$	$\frac{\ell}{B}$	1
suffix array[66]	$m \log n + \frac{occ}{B}$	$m \log n$	$\log n + \frac{\ell}{B}$	$\log n + \frac{\ell}{B}$

The WOTD-tree is generated using the TDD construction algorithm[95].
The SB-tree does not maintain the original suffix tree structure, so we derived the worst case complexity for the node and edge label access to recover the original suffix tree information.
Note that H, the depth of the suffix tree, is bounded by $O(n)$.

Table 5.2: Worst case big-O IO bounds for operations on various proposed suffix data structures.

our CPS-tree and compare with some proposed suffix structures in the literature. Finding exact match on most suffix tree structures is generally IO bounded by m. For SB-tree, it runs in $O(\log_B n + (m + occ)/B)$. CPS-tree is second to SB-tree, with an IO bound of $O(\min\{m, \log n\} + (m + occ)/B)$ for exact match query, at most an $O(\log B)$ factor behind. For CPS-tree, counting the number of exact match is handled easily without going through all the matching occurrences.

Next, structures like SB-tree, CPT and suffix array are not designed for basic tree traversal operations like child node and edge label access (refer to Table 5.2). CPS-tree and WOTD-tree support these operations with IO access bound independent of n. Tree traversal operations are essential to handle complex queries and to support various search techniques over suffix tree (an example is the local alignment search).

As for disk space usage, we need to balance between maintaining a small index (to the extend of trimming off any extra bits), and keeping fast access by replicating information to be stored in the tree so that access is kept in near proximity. Experiments show that for a DNA sequence with N characters, we need 7N to 9N bytes to store our bit-packed suffix tree on disk in practice. This is assuming that every position on the text is to be indexed. For example, the CPS-tree index built on the human genome with 3 billion characters is 27GB in size. Our scheme is comparable to the space efficient suffix tree representations [20, 29, 30, 57] that work on bit-level packing.

Alternatively, we can enhance the CPS-tree index with the suffix array [66] on disk using additional $4N$ bytes, so that the occurrences which correspond to an index range in the array can be retrieved sequentially from disk directly. This will increase the total disk usage for CPS-tree to around 11N to 13N bytes. Moreover, its size is still small when compare with most suffix tree implementations which use 17N to 65N bytes [45, 57] with more compact version in 12.5N bytes [34, 69, 95].

In brief, we have made improvements to CPS-tree giving the following results: (1) Fast searching and traversal of the suffix tree in terms of IO paging and computational time; (2) fast enumeration of the occurrences; and lastly, (3) compact the suffix tree using bit-packing representation and other space optimization.

We show in our experiments that the CPS-tree is space-efficient, and performing exact match query on it generates very few page faults (even for human genome). We also show that approximate match query and local alignment search (using the affine gap cost model) can be handled efficiently using the CPS-tree.

The rest of this chapter is organized as follows. We first introduce various suffix data structures and other related works available in the literature. Next, we describe in depth, the structure of our CPS-tree. This is followed by the experimental results on disk-based and memory-based query search to show that CPS-tree performs well in practice. We conclude with discussion on our ongoing work and research direction.

5.2 Related work

The issue of IO efficiency in suffix tree for exact string matching has been addressed considerably in the literature with the following 2 main contributions, Compact Pat Tree (CPT)[20] and String B-Tree (SB-tree)[29]. [1] CPT is a partitioned PAT tree [36, 70] which is essentially a suffix tree with space highly optimized. Every leaf node in the tree denotes a suffix of the text with the path to the leaf from the root labeled with the suffix

[1]For in-memory representation of suffix tree, we have compressed suffix tree (CST) [88], FM-index (FMI)[31] and compressed suffix array (CSA)[38] representations, which are more compact in size ($\leq N$ bytes) but display poor access locality [68] and require more computation. They are therefore better suited for memory-based computational model.

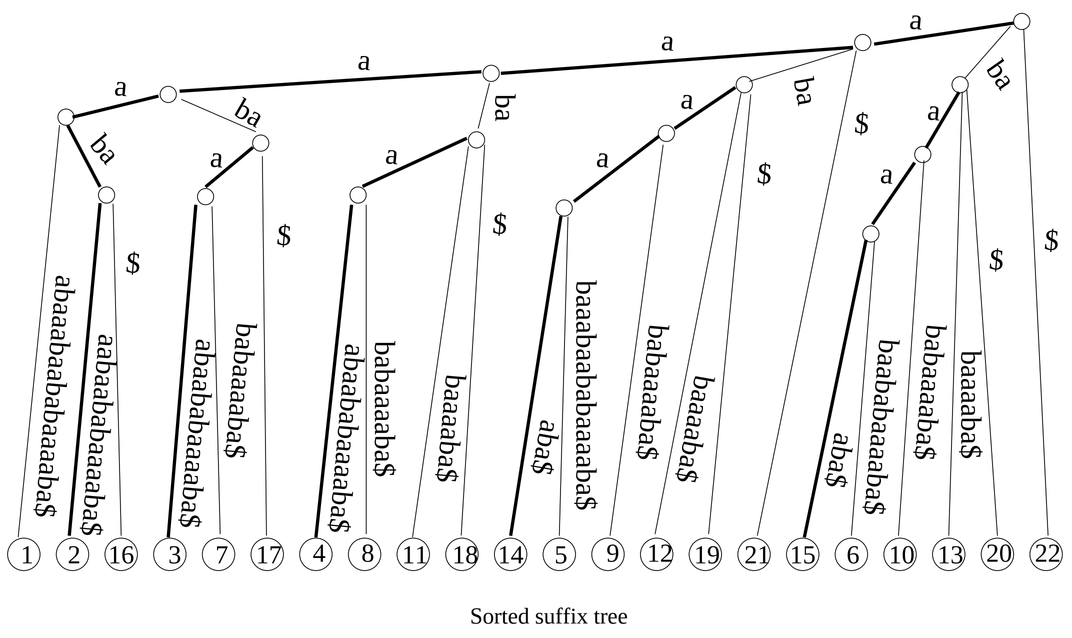
string. Though it is small in size, finding the occurrences of a pattern takes time linear to the height of the suffix tree which can be very inefficient both computationally and in terms of IO cost. Our work improves on the CPT's scheme so that we do not need to traverse the full path to a leaf node to answer exact match query. This greatly enhances the search efficiency.

SB-tree, on the other hand, applies the structure of B-tree over string to give a well-balanced tree structure that promises $O(\log_B N)$ worst case IO access to traverse the tree from the root to a leaf node. As SB-tree does not explicitly preserve the suffix tree structure, it is not obvious how SB-tree can be extended to handle complex query like approximate search efficiently [28]. This limits the usefulness of SB-tree in practice.

More recently, Hunt *et. al.* [45] gave a disk-based suffix tree for DNA and protein sequences. It is shown that dynamic programming over their suffix tree can efficiently solve the local alignment problem. Their main setback is that the suffix tree is large, requiring 21N to 65N bytes depending on implementation. Meek *et. al.* followed up with OASIS [69]. They gave a dynamic programming A*-search driven algorithm over suffix tree for exhaustive local alignment search on protein sequences that surpasses the performance of Smith-Waterman algorithm [92]. Previously, Giegerich *et. al.* [34] have proposed the WOTD suffix tree representation. Tata *et. al.* [95] gave an improved top-down disk-based suffix tree construction algorithm named TDD, based on WOTD suffix tree representation by Giegerich *et. al.* [34], that can scale up efficiently for large text sequence while using a fixed memory space. Halachev *et. al.* [41] further suggested storing the nodes in the tree by depth-first order that is slightly more efficient for enumerating the occurrences in exact match search. It was shown that the top levels in the suffix tree can be further compressed using arrays for lookup [48]. However, none of the above results addresses the issue of IO efficiency in handling pattern matching and for the suffix tree traversals.

Bedathur *et. al.* [11] proposed a partitioning method for suffix tree, catering to both suffix link and tree node traversals. In searching for the maximal common substrings, they showed that their approach is more IO efficient. However, their proposed method penalizes search that does not requires suffix link, for example in solving exact and approximate pattern matching.

Another promising direction is to simulate suffix tree using the suffix array (SA)[66]. One weakness of suffix array is that there is an additional cost of $O(\log n)$ time factor to simulate each operation on a suffix tree. Also access to the suffix array does not display a regular pattern, as a result, the disk IO cost to search for a pattern of length m on a disk-based suffix array can be as high as $O(m \log n)$. Enhanced Suffix Array [1], essentially consisting of the suffix array and additional tables, was proposed to execute with same time complexity as suffix tree except that it is more space efficient. However, it did not address the IO issue of suffix array. There are approaches that replicate selected suffix array entries in the memory to improve on the IO access performance [9] in practice. There are several works [16, 46, 94] that apply the filtering strategy with q-gram (or alike) indexing. The basic approach consists of these steps: neighborhood generation, index lookup and followed by site verification. There is no tight bound on IO performance as it is dependent on the size of the candidate list generated. Also when searching for local alignment, there is a chance that it may not find all alignments. In any case, CPS-tree is a more versatile indexing structure which supports both pattern matching and suffix tree traversal efficiently.



Suffix array	1	2	16	3	7	17	4	8	11	18	14	5	9	12	19	21	15	6	10	13	20	22
SA index	1				5		7			10					15					20		22

Figure 5.1: Suffix tree and suffix array built on the text = “aaaaabaaabaababaaaaba$”.

5.3 Structures and algorithms

This section describes the CPS-tree structure and its application in exact pattern search. The reader can refer to Section 2.2 for description of the structures of suffix tree and suffix array. We adopt the convention of letting the terminating symbol ‘$\$$’, appended to the text T, to be larger than any character found in the text. The suffixes in the suffix tree are in lexicographically sorted order from left to right. We begin with a short revision on how string matching is performed on a suffix tree.

The existence of any given query string in T can be found in time linear to the query length, m, using a suffix tree. Given a length-m query string, we traverse the tree from the root down a path matching the query with characters on the edges until no further matching is possible. There is at most m edges or nodes to visit and naturally, the query string exists in T if the whole query can be matched. The occurrences of the query on T can be retrieved by visiting the leaf nodes in the subtree under the node where the matching completes. For example, by traversing the suffix tree in Figure 5.1, the occurrences of “abaa” can be found in one subtree which contains the text positions ‘14’, ‘5’, and ‘9’. Note that the positions in T where a query string occurs, are stored consecutively in the suffix array (SA). For example, in Figure 5.1, the SA entries ranging from index 7 to 10, store the positions in T where the query string “aab” occurs.

5.3.1 CPS-tree representation

Our CPS-tree representation is illustrated in Figure 5.2 based on the same text string used in Figure 5.1. CPS-tree is basically a modified PAT tree [36, 70] where an edge stores the first character of the edge label and its length instead of the actual edge label. We first partition the suffix tree into many small trees, to be addressed as “local” trees, in a top-down fashion so that each local tree fits into a logical block of fixed size (the bounding boxes in Figure 5.2). The end node in a local tree is either a leaf node (terminating circular node) or an external node (rectangular node with an outgoing dashed edge) pointing to the descending local tree in another logical block. Each local tree, rooted at node v, is constructed by first including the node v and its children, then we recursively include the node with the heaviest subtree (most number of leaf nodes), among all nodes at the local tree boundary, and its children. The process repeats until the local tree is full (that is, too big to fit into a logical block). This partitioning method is referred as the greedy approach which is fairly intuitive. It was shown in the paper by Alstrup *et. al.* [2] that the average block access of the greedy approach is bounded by a factor of $O(\log B)$ (and $\Omega(\frac{\log B}{\log \log B})$) more than the optimal layout.

There are several tree partitioning methods in the literature [25, 35]. In the paper by Diwan *et. al.* [25], bottom-up, tree partitioning methods were proposed, that find the optimal layout minimizing either the worst (maximum) or average block access when traversing from the root to any leaf in the tree. This however gives a partitioned tree with possibly many under-filled internal blocks or pages [25] which is undesirable. Another common approach is to build the partitions naively by grouping the nodes in the breadth-first order [11]. This is expected to give good average performance in general. We implemented the breadth-first order partitioning and the greedy approach and, through the experiments (see Table 5.4), showed that greedy approach achieves fewer expected page faults for CPS-tree in practice.

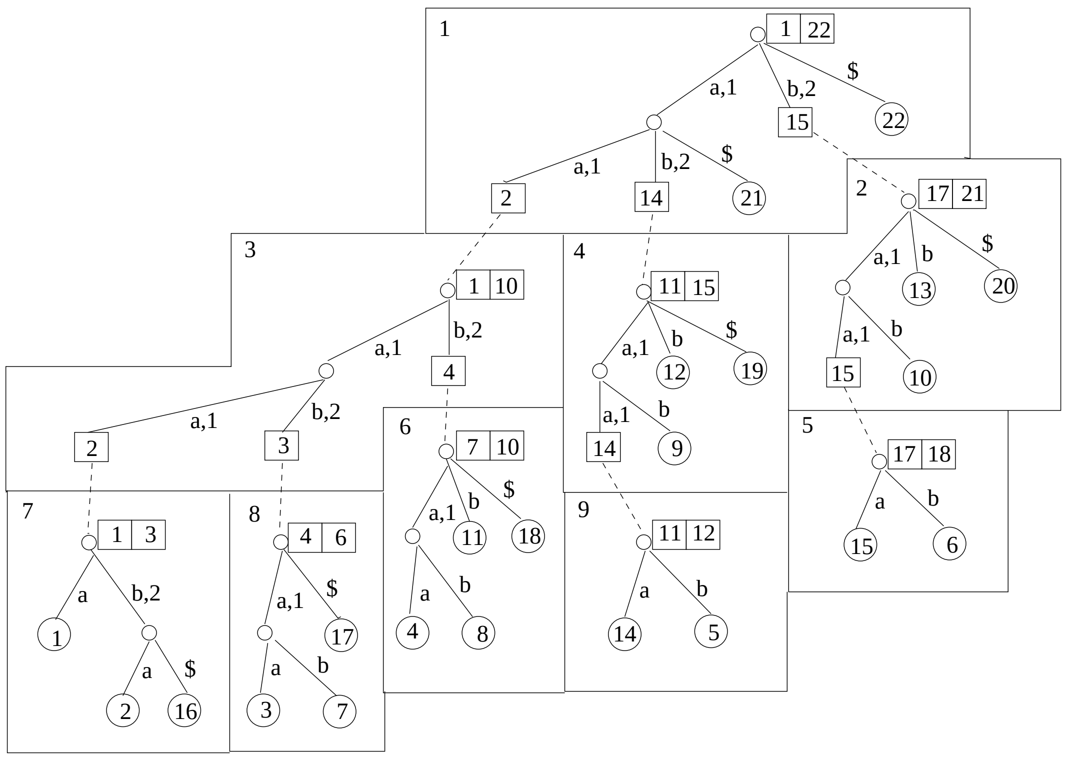


Figure 5.2: CPS-tree representation for text = “aaaaabaaabaababaaaaba$”.

To compact the local tree, we do not store the edge label explicitly. We store the first character on the edge together with its label length for non-leaf edges. For leaf edges, we only store the first character of the edge. The label length of a leaf edge can be computed if we know the character depth of the parent node (of the leaf node) and the text position stored in the leaf node. For each external node u in a local tree, we store the replicate of the text position of some leaf node in the subtree rooted at u (consider the whole suffix tree). In our implementation, we select the leaf node which is reachable from u through the heavy path where a heavy path is a path of heavy edges and a heavy edge of a node u is the edge (u, v) such that v is the child of u which has the largest subtree (with the most number of leaves) when compared to v's siblings. Figure 5.1 shows the heavy paths in the suffix tree, with thickened edges. The text positions replicated in the external nodes, help to localize access and to improve IO efficiency of pattern search. Also, at the root v of each local tree, we store the SA range of the subtree rooted at v (see Figure 5.2 for an example). This information comes useful to access the external SA on disk when we need to enumerate the occurrences in a search.

To facilitate searching of nodes further down the tree, we maintain extra link, denoted as “forward link”, at the block-level (in addition to the CPS-tree structure presented in Figure 5.2). We can then access any node from the root, by traversing through, in the worst case $O(\log n)$ logical blocks. This property is useful for applications that demand worst case guarantee in query time. We will elaborate further in Section 5.3.3.

The top few levels of nodes in the suffix tree are most frequently visited in answering queries. As such, CPS-tree is written to disk in a top-down order. The order to be written is illustrated in Figure 5.2 as the block label on the top left corner of each block.

Memory buffer is implemented to handle access to the suffix tree where the memory buffer can be initialized very quickly through sequential read of the first few pages of the suffix tree from disk. Using an optimized bit-packing scheme to encode the individual tree structure, CPS-tree can further achieve good space utilization and IO efficiency in answering string matching query.

5.3.2 Space optimization

Each local tree is packed using bit representation. For example, a DNA character is encoded using 3 bits given that the DNA alphabets is of size 4 plus the terminating symbol '\$'. Each node stores its outgoing edges in an edge array. An edge can be selected by performing binary search over the edge array in $\log |A|$ time at most. The non-leaf edge label length is generally short, as such, we use 8 bits to store the length and 32 bits if the length is longer than 255. From our experiments, we find that there are only a handful of non-leaf edges with label length longer than 255. A similar design was used in CPT for better compression.

We pad each local tree with extra bits (< 8 bytes) so that the size of each block is a multiple of 8 (in bytes). With the blocks written consecutively into the index file, we can record the starting location of each block using fewer bits.

The blocks are written sequentially into the index file on disk, ignoring the physical boundaries that divide the file equally into sectors on disk. A physical page read from the index file will fetch 1 or more blocks at a time, assuming that the physical page is larger than the block size. It is also possible for a logical block to reside across the boundary of 2 consecutive pages. A logical block may be under-filled and hence the blocks may differ in size.

5.3.3 Forward link

The logical blocks in the CPS-tree is arranged in a tree. This section describes the concept of forward link which allows us to access any logical block within $O(\log n)$ block accesses.

We need some definitions. For any logical block i, the leaf count of the logical block i, denoted as $|i|$, refers to the number of leaf nodes in the subtree spanned by the first node of the logical block i. As an example, the leaf count of the logical block 3 in Figure 5.2 is 10. For every logical block i, (i, j) is called a heavy link if $|j|$ is the biggest among all child blocks j of block i. The chain spanned by the heavy links is called the heavy chain of logical blocks. For example, logical blocks 1, 3, and 6 form a heavy chain in Figure 5.2.

Note that for any child block j of block i, if (i, j) is not a heavy link, $|j| \leq |i|/2$. Hence, when we search downward to access a logical block, we need to access at most $\log n$ non-heavy link. However, we may need to go through $O(n)$ heavy links to reach a logical block. To speed-up, we introduce forward link which skips some of the logical blocks.

Consider a particular chain, let blocks i and j be the first and the last blocks respectively. We define forward links for every block in the chain from i to j as follows. First, a forward link is introduced from block i to block r where block r is a descendant of block i and an ancestor of block j such that i is the deepest block with $|r| \geq (|i|+|j|)/2$. Then, the chain is partitioned into two chains: The first chain is from the child of i to the parent of r and the second chain is from r to j. Lastly, we recursively define the forward links for the two chains. An illustration of the forward links (arc arrows) is given in Figure 5.3. The above procedure ensures we can find any block in a chain within $2\log n$ block accesses.

Now, imagine that we start at the first block i containing the root node of the suffix tree, in the process to find the exact match of a given query string. The match count of block i could be as large as n. If the matching reaches an external node v in block i, and v points to the child block j, then there are 3 possibilities to continue the search down the tree. Case 1 is that (i, j) is not a heavy link, so we can continue the search in child block j whose leaf count will be reduced by at least half, and so the possible match count to be returned, is reduced to $n/2$ in block j. Otherwise, we have (i, j) being a heavy link, and let r be the block pointed to by the forward link in the current block i. Case 2 is when we can fully match the path label to the first node in the forward block r with the query string and so the search continues in block r. Since a forward link reduced the leaf range by half each time, we will have the possible match count reduced to $n/2$ in block r as well. Next, we have the final case 3 where the path label to r does not match with the query string. In this case, we will continue the search in child block j, knowing that we have already eliminated the possibility of visiting forward block r and beyond as in case 2. Hence the possible match count in block j is reduced by the leaf count of forward block r, which is still $n/2$. This process is repeated as we visit a new block with the possible match count reduces by half each time until the match count is 1 or when the query string is fully match or when mismatch occurs with no match to be found. In this way, we can find the exact match of any query string in the suffix tree with $O(\log n)$ logical block accesses.

5.3.4 Exact string matching

Exact string matching on CPS-tree is performed by repeating the search process on each local tree visited as we traverse down the suffix tree. From the root of the CPS-tree, we traverse through the nodes matching the first character on the edges while skipping the in-between characters. At any one time, if no match is possible after searching through the outgoing edges of a node, we conclude that no exact match exists in T. Otherwise, from the last matching node, we will proceed further down to a leaf or an external node within the same local tree, containing the text position, `Spos`. With `Spos`, we can then retrieve the substring from the text T to verify on the matching of the skipped characters.

We illustrate exact search with query string “aaa” on the suffix tree shown in Figure

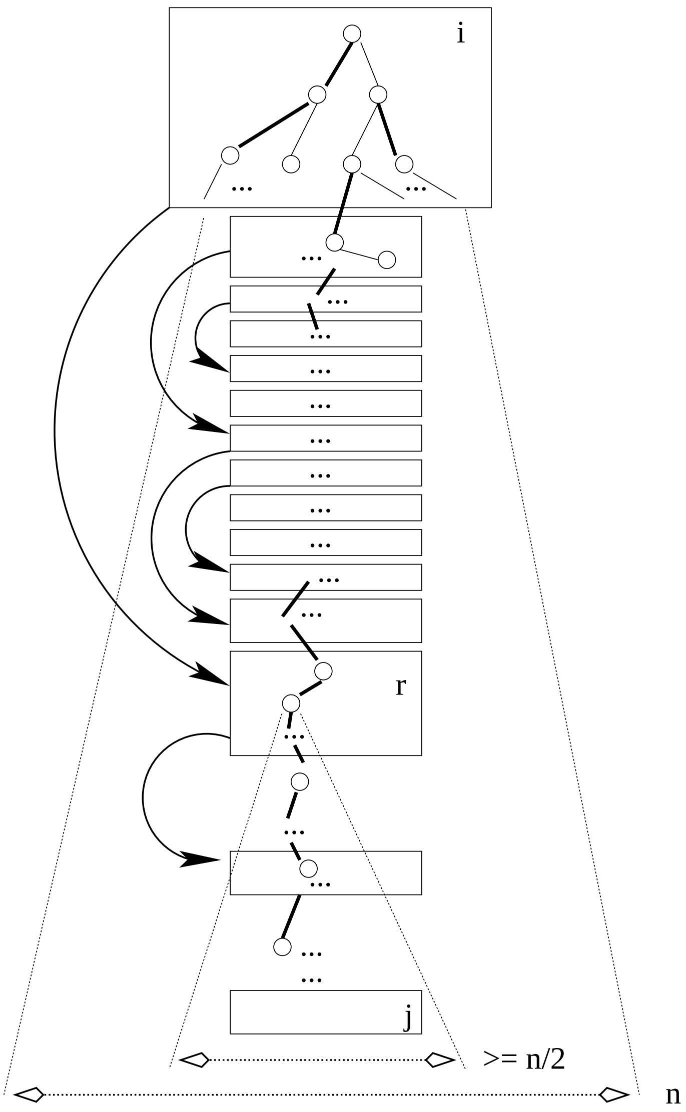


Figure 5.3: Forward links illustration.

5.2. Starting with block 1, we match the first character on the leftmost outgoing edge of the root, labeled “a” and then again another character “a” on the next leftmost edge, ending at the external node in block 1 with text position 2 stored. Since both edges

```
CPS_Search($i, P, c$)
  let $v$ = the first node in $i$ & $d = c$
  while (not done)
    binary search on $v$'s edge-array for edge $e$ with first character == $P[d+1]$
    let node $u$ be the target of $e$ & $Spos$ be the text position in $u$ if exists
    CASE 1: [$e$ is not found]    return no match
    CASE 2: [$u$ is an internal node]
      $d$ += label length of $e$
      if ($d < m$)    $v$ = u
      else
        read text position $Spos$ from an end node below $u$
        $c$ += strC($T$[Spos+$c..N$],$P[c+1..d]$)
        if ($c < m$)    return no match
        else    return all matches in the subtree under $u$
    CASE 3: [$u$ is a leaf node]
      $c$ += strC($T$[Spos+$c..N$],$P[c+1..m]$)
      if ($c < m$)    return no match
      else    return position Spos as a match
    CASE 4: [$u$ is an external node]
      let $j$ be the local tree referenced by $u$
      let fw_i be the local tree (block) pointed to by the forward link in $i$
      let fw_dep be the character depth of the first node in local tree fw_i
      $d$ += label length of $e$
      $c$ += strC($T$[Spos+$c..N$],$P[c+1..m]$)
      if ($d \geq m$)
        if ($c < m$)    return no match
        else    return all matches in the subtree under $u$
      else-if ($c < d$)    return no match
      else-if ((($i, j$) is a heavy link) & ($c \geq$ fw_dep))
        /* using forward link */    return CPS_Search(fw_i,$P$,fw_dep)
      else    return CPS_Search($j, P, d$)
```

Figure 5.4: Exact string matching on CPS-tree.

encountered are each of length 1, there is no need to verify for skipped characters. Otherwise, we would need to retrieve the text starting at position 2 as stored in the external node to verify against the query string. We proceed to the next block 3 pointed by the external node and match the next character “a”, ending at an internal node (last match node) after completely match the query string “aaa”. Now we need to enumerate all the occurring positions on the text. To do so, we obtain the SA range from the local trees in the next level, so that the occurring positions can be read directly from the SA. We obtain the left SA index by traversing the leftmost path down the last match node, to find block 7 with the left SA index 1. Similarly, we obtain the right SA index by traversing the rightmost path to reach block 8 with the right SA index 6. The matching positions can then be read from the SA in entries 1 to 6.

If SA is not available, we can still recover the text positions by traversing the whole subtree rooted at the last match node to retrieve the text positions stored in the leaf nodes. However, this process is much more time consuming and IO expensive, especially for large number of occurrences, which we would like to avoid.

The procedure *CPS_Search($i, P[1..m], c$)*, given in Figure 5.4, performs the exact string matching on the CPS-tree. It takes in 3 arguments: (1) i, the local tree to begin the search, (2) $P[1..m]$, the length-m query string and (3) c, the number of characters matched so far. The search procedure returns the enumeration of occurrences of P in T. We have incorporated the use of forward link into the procedure. Exact string search on string query $P[1..m]$, is invoked by calling *CPS_Search($lTree, P[1..m], 0$)* where

$lTree$ is the local tree containing the root of the suffix tree. We also define a supporting procedure *strC(s, q)* which returns the longest matching prefix length between strings s and q.

5.3.5 Tree construction

CPS-tree is constructed in 3 steps as given in Figure 5.5. First, we obtain the SA from text using existing construction package [60] available. Second, we construct the CPS-tree in a top-down order, by searching the SA as depicted in Figure 5.6 as procedure *CPS_Build*(i, j, r, d, h). We have i and j as the SA index range to search with, d is the character depth to the current node in the suffix tree, h is the current node height, and r is the reference to the parent node. The procedure is invoked with the call *CPS_Build*$(1, n+1, null, 0, 1)$. In the final step, we traverse the entire constructed CPS-tree to update the text positions in the external nodes. The procedure is *CPS_Update*(i, r, x) as shown in Figure 5.7 where i is the current node, r is the SA range size under the current node returned, and x is the text position being returned. *CPS_Update*$(root, 0, 0)$ is invoked, a recursive procedure that performs basically a depth-first traversal of the whole CPS-tree.

The whole construction takes approximately twice the time to construct the WOTD-tree using the TDD package [95]. We are less concern with the construction time as it is a one time effort. We could, in the future, speed up the construction process by building

```
CPS-tree construction
1. Build SA from text $T$, using existing construction package.
2. Build CPS-tree from the SA: invoke CPS_Build$(1, n+1, null, 0, 1)$
3. Update the text positions in CPS-tree: invoke CPS_Update$(root, 0, 0)$
where $root$ is the root node to whole suffix tree
```

Figure 5.5: CPS-tree construction process.

the CPS-tree directly from text instead of the SA.

5.3.6 Buffer management

We implemented 2 simple buffer replacement policies, the least recently loaded (LRL) and the most recently loaded (MRL). The LRL policy replaces the oldest loaded page with the newly loaded page whenever a page fault occurs. The top few levels of the suffix tree are most frequently accessed and hence we can make the first few pages of the index persistently reside in the memory. This is the motivation for the MRL policy where the next page being fetched will replace the second last page loaded (i.e. the last page loaded besides the current loaded page). We will demonstrate the IO efficiency of the two buffer replacement polices on the index and show that MRL is recommended for CPS-tree index (see Section 5.5.2). The LRL policy is used on the text buffer as there is no clear access pattern for the text string.

```
/* maintain two global lists L and K */
CPS_Build($i, j, r, d, h$)
    Allocate and create a new block $I$.
    Store the block overhead fields $i, j$ as the left and right $SA$ index respectively in $I$.
    Add a new node $p$ as the root into $I$.
    while ($I$ is not full)
        for each character $c$ in the alphabet $A$
            Binary search on $SA$ over the index range $i..j$ for the leftmost and rightmost
            index $x$ and $y$ s.t. $T[SA[x] + d + 1] = T[SA[y] + d + 1] = c$.
            if ($(x, y)$ is found)
                Add a child node $q$ to $p$ with first character on the edge label as $c$.
                if ($x == y$)
                    Set node $q$ as a leaf node with suffix position $SA[x]$.
                else
                    Set node $q$ as a internal node.
                    The edge label length to $q$, $t$, is the longest common prefix
                    length of text starting at $SA[x] + d + 1$ and $SA[y] + d + 1$.
                    Add $(x, y, q, d + t, h + 1)$ to list $K$.
        if ($I$ is not full)
            Extract entry $(x', y', q', d', h')$ from $K$ s.t. $(y' - x')$ is the largest range in $K$.
            Set $(i, j, p, d, h)$ = $(x', y', q', d', h')$.
    For every entries $(x', y', q', d', h')$ in $K$, convert internal node $q'$ to an external node in $I$.
    Transfer all entries in $K$ to $L$.
    Write out block $I$.
    Update external node $r$ to point to block $I$, if $r \neq null$.
    if ($L$ is not empty)
        Extract entry $(x', y', q', d', h')$ from $L$ s.t. $h'$ is the smallest in $L$.
        Invoke $CPS\_Build(x', y', q', d', h')$.
```

Figure 5.6: CPS-tree building from SA.

```
CPS_Update($i, r, x$)
    Set $r = 0$.
    for each child $j$ of node $i$
        if ($j$ is a leaf node)
            Set $r' = 1$ and $x'$ = the suffix position stored in the leaf node.
        else if ($j$ is an internal node)
            Invoke CPS_Update($j, r', x'$).
        else if ($j$ is an external node)
            Let $k$ be the root node in the next block pointed to by $j$.
            Invoke CPS_Update($k, r', x'$).
            Set the text position of $j$ to $x'$.
        if ($r' > r$)
            Set $r = r'$ and $x = x'$.
```

Figure 5.7: CPS-tree updating of text positions.

5.4 Bit representation and analysis

5.4.1 Search time and IO access analysis

Given a node and the next character to match, it takes $O(\log |A|)$ time to perform binary search on the edge array of the node to find the edge with its first character that match. Accessing the label on a selected outgoing edge of a node takes $O(l + b)$ time where l is the label length, given the character depth of the node. $O(b)$ is the time to traverse down any path to an ending node in the local tree to retrieve the text position (in the leaf or external node). The traversal takes $O(b)$ time as the local tree size is bound by $O(b)$.

Reading length-l substring from the text requires $O(l/B)$ disk access.

To search for a length-m query string in the suffix tree, it takes $O(m \log |A|)$ time. It takes another $O(b)$ time to further obtain the left and right bounds of the matching SA range. In total, it takes $O(m \log |A| + b + occ)$ time for exact string matching on CPS-tree where occ is the number of occurrences of the query string.

For disk access, we analyze the disk access on text and on index separately. The disk access on text is bounded by $O(\log n + m/B)$. We always read from the text position found on some path that is the heaviest. This allows us to assert that there is $O(\log n)$ read off positions from the text. $O(m/B)$ is the IO bound in retrieving the matching query substrings from the text. Note that without using the forward links, the IO disk bound still holds.

The disk access on the index is bounded by $O(\log n)$ as we ensure that the subtree in the next block retrieved will have leaf nodes at least halved using the forward link. Reporting the occurrences from the reading the SA takes $O(occ/B)$ time. Hence the disk IO bound for exact string matching is $O(\log n + (m + occ)/B)$.

5.4.2 Bit-packing scheme

Each node in the CPS-tree stores information about its outgoing edges. An outgoing edge of a node can be one of the following: a leaf edge, a local edge or an external edge. Leaf edge, as the name suggests, points to a leaf, and a local edge points to the next node

Node:

`|child|`	`leaf_B`	`long_B`	`edge-array ...`

leaf edge (in the edge-array):

`char`	`S_pos`

local edge (in the edge-array):

`char`	`skip_len`	0	`next`

external edge (in the edge-array):

`char`	`skip_len`	1	`next`

External edge extension:

`S_pos`	`Bidx`

(a)

Block overhead - SA bound indices, forward link:

`SA_Lidx`		`SA_Ridx`
`fw_edge`	`fw_dep`	`fw_Bidx`

(b)

Field size (bits)	Fields
$\log \lvert A \rvert$	`|child|, char`
$\log(8b) = \log b + 3$	`next, fw_edge`
$\log N$	`S_pos, Bidx, SA_Lidx,`
	`SA_Ridx, fw_dep, fw_Bidx`
$\leq \lvert A \rvert$	`leaf_B, long_B`
8 or $\log N$	`skip_len`

(c)

Figure 5.8: (a) Bit-packing representation of the nodes in a local tree, (b) block overhead fields in a block and (c) the bit size of the respective fields used in the encoding.

in the same local tree. An external edge is the connecting edge to the first node in the next local tree. We denote the first node in a local tree as the *head node*.

Figure 5.8 gives the bit-packing representation of a node in the CPS-tree. In CPS-tree, a node consists of 2 parts, the aggregated child information, followed by the `edge-array`, an array of its outgoing edges. The first part contains (1) `|child|`, the number of child nodes, (2) `leaf_B`, a bit array to mark the leaf edges in the `edge-array` and (3) `long_B`, a bit array to mark those edges in the `edge-arrray`, whose label length requires > 8 bits to represent (irrelevant to leaf edges). In the `edge-array`, the edges are stored in increasing order, based on the first character of the edge label.

For a leaf edge, we store the first character of the edge label, `char`, and `Spos`, the suffix position on the text corresponding to the leaf node. Notice that we do not store the label length of the leaf edge which can be computed given the text length and the character depth of the node.

Local and external edges share similar representation, we have (1) `char`, the first character of the edge label, followed by (2) `skip_len`, which is the edge label length - 1, (3) a single bit to identify the edge as external, and (4) `next`, the bit offset from the start of the block where the next node (for local edge) or the extension to the external edge, is located in the block. We use 8 bits for `skip_len` as most of the edge labels are very short and can be more compactly encoded using only 1 byte (this idea is borrowed from CPT). We store the label length in `skip_len` for length up to 256. If the label length is > 256 (indicated by marking the corresponding position in the `long_B` field), we use a longer field of $\log N$ bits instead.

For an external edge, we store additional information in the edge extension (access through the `next` field). The edge extension contains `Spos`, a selected suffix position on the text in a descending leaf node and `Bidx`, the index of the block containing the next local tree. The block index is the 8-bytes offset (byte offset $/8$) from the start of the index file and is stored using $\log N$ bits.

`Spos` (available in a leaf or an external edge), provides the localized information needed to retrieve from the text, the label of any edges in the local tree. Take for example, a node v with an outgoing edge e in the local tree whose character depth, d, is known. To retrieve the edge label of e, we need to traverse through e to an ending node in the local tree to recover the suffix position, `Spos`. The edge label of e can then be read from the text starting at position `Spos` $+d$.

Now we need to explain how the `Spos` in an external edge is obtained. Every heavy path is terminated by a leaf node with an assigned `Spos`. Imagine that for every leaf node on the heavy paths, we propagate the `Spos` value backwards to all the nodes and edges on the respective heavy paths only. So if an external edge is on a heavy path, it stores the propagated `Spos`. Otherwise, the external edge can obtain the `Spos` value from the head node it points to.

`Spos` provides localized information for the local tree so that edge label on an outgoing edge of a node on the path to the external edge can be obtained from the text at position, `Spos` $+$ the character depth to the node. `Spos` is propagated from a selected leaf node in the subtree spanned by the head node such that the path from the head node to the leaf node forms (or is part of) a heavy path.

As the heavy path can be interpreted as the most frequently traveled path down a subtree (as it contains the most leaf nodes), it makes sense to select the most common string suffix for comparison. This increases the access locality on the text as consecutive text segments will more likely be retrieved and compared with as we traverse down the path. Of course, this holds under the assumption that every suffix strings in the text is equally likely to match with any given query string.

We store additional overhead fields before the local tree structure in each logical block. First is the SA index corresponding to the leftmost and rightmost leaf nodes in the suffix tree reachable from the head node. We address them as `SA_Lidx` and `SA_Ridx` respectively, and collectively as the SA bound indices. These information give direct access to the SA stored on disk if available, to retrieve all the suffix positions in the leaf nodes under the current subtree. This improves in enumerating all the positions especially if the the subtree is large as we can skip traversing through the subtree to visit all the leaf nodes.

Next we store in the block overhead, the forward link information consisting of 3 fields: (1) `fw_edge`, the bit offset to the external edge in the block that leads to the forward block, (2) `fw_dep`, the character depth of the head node in the forward block and (3) `fw_Bidx`, the forward block index.

5.4.3 Disk space usage analysis

We refer to the node representation scheme in Figure 5.8 for our analysis of the disk space usage. Each leaf accounts for $\log N + \log |A| + 2$ bits to store the `Spos`, `char`, and 1 bit for each entry in the `leaf_B` and `long_B` fields of its parent node. Each internal node accounts for $8 + \log |A| + 2 + \log |A| + (\log b + 3) + 1 = 14 + 2 \log |A| + \log b$ bits for `skip_len`, `|child|`, 1 bit for each entry in `leaf_B` and `long_B` fields of its parent node, `char`, `next` and 1 external edge bit indicator respectively. Each logical block also uses $\log N + \log N + 2 \log N + 2 \log N + (\log b + 3) = 6 \log N + \log b + 3$ bits to store `Spos` and `Bidx` for external edge, and the rest are for the block overhead fields consisting of, `SA_Lidx` and `SA_Ridx`, `fw_dep`, `fw_Bidx` and `fw_edge`.

Since there are n leaf nodes in the suffix tree and the number of internal node is also bounded by n (though it is much less than n in practice especially with large branching factor), the total bit space required is at most $(\log N + 3 \log |A| + \log b + 16)n + (6 \log N + \log b + 3)c + v$ where c is the number of logical blocks. The term v accounts for the extra variable bit space needed for padding some logical blocks with extra bits at the end and for label length on the edges that require more than 8 bits to store. If we assume that a position on the text can be addressed using 4 bytes word, that is $\log N = 32$ bits, and given that $\log b = 13$ since $b = 8K$ bytes, we have an upper bound in CPS-tree size of $7.625n + 0.375n \log |A| + 25.625c + v$ bytes.

5.5 Performance studies

We consider the query of reporting on the exact match locations in the text sequence. CPS-tree is compared with WOTD-tree and SA for both on-disk and in-memory settings. We study the IO performance of index search and reporting on exact match locations. More complex queries like approximate matches and local alignment search are shown feasible on the CPS-tree. The WOTD-tree (write-only top-down construction algorithm) [34] is constructed in a top-down approach using the TDD package [95] available. We also perform string searching over the suffix array using binary search technique.

5.5.1 Experimental settings

The datasets used are the fruit fly genome of 118.3 million bases (http://www.fruitfly.org/sequence, Release 4) and the E. coli K12 genome of 4.6 million bases (http://www.ncbi.nih.gov, GI: 49175990). These are DNA sequences consisting of characters ‘A’, ‘C’, ‘G’ and ‘T’. The data and index are buffered separately. Table 5.3 gives the index size on the fruit fly dataset used for CPS-tree and WOTD-tree.

The buffers are first initialized fully with the first few blocks read from the text sequence and index files respectively. Initialization of the buffers can be performed very quickly with sequential reads from the files. We ignore the physical organization of the files on disk and every page read from file is assumed to take a constant time to perform. If a block to access is not in the buffer, a page fault occurs and a new page of

Index	fruit fly, 118.3M (Mbytes)	
CPS-tree–	829.77	$\approx 7.0N$
CPS-tree	849.53	$\approx 7.2N$
WOTD-tree–	1089.97	$\approx 9.2N$
WOTD-tree	1474.61	$\approx 12.5N$

CPS-tree– : CPS-tree without in-built forward links.

WOTD-tree– : WOTD-tree excluding the 2 bit arrays.

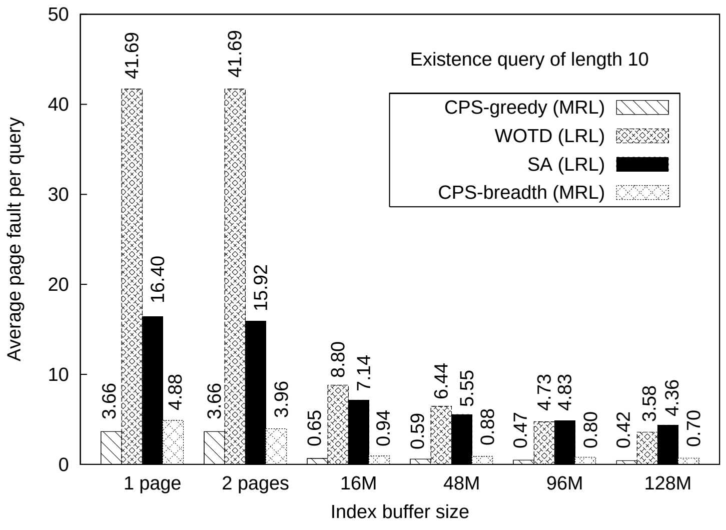
Table 5.3: Index tree structure file size.

$8K$ bytes containing the required block is fetched into the buffer.

Queries are generated from random positions on the genome itself so that it is guaranteed to return a match in the indexing structure. This allows us to compare the performance of various indexing structures over the same matching query length. Queries are generated for length 10, 100, 1000 and 10000. The average performance is measured from running consecutively, 1000 different random queries of the same length.

The experiments are carried out on an Intel P4 2.4GHz machine with 512KB cache and 1GB of RAM, running Linux, with codes written in C++ using gcc v4.1.1 with level 3 optimization flag. We implemented the search algorithms for CPS-tree, WOTD-tree and SA, so that they all share the same access routines to the buffers.

The WOTD-tree consists of 3 data structures: (1) An integer array representing the tree structure with sibling nodes stored consecutively using 2 integers for a branching node and an integer for leaf node, (2) bit array to identify the leaf node and (3) another bit array to identify the last sibling node. To access the edge label would require accessing the text sequence.



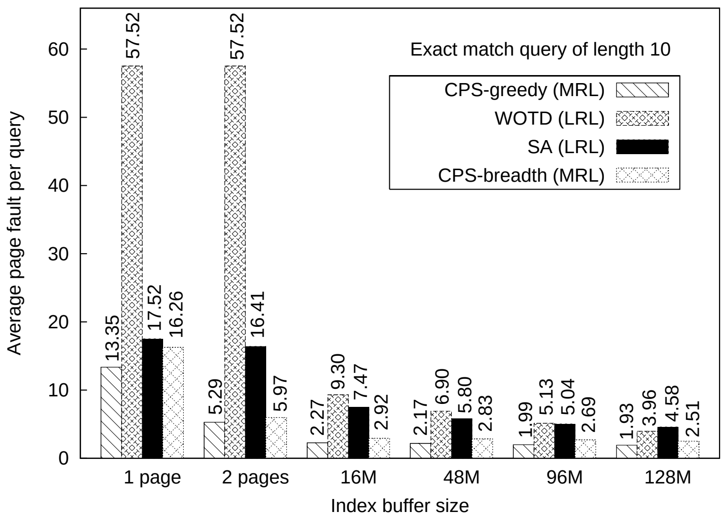


Figure 5.9: Result 1 - Average page fault on index buffer for fruit fly genome.

Existence query											
Query length	Index	1 page	2 pages	LRL buffer replacement				MRL buffer replacement			
				16M	48M	96M	128M	16M	48M	96M	128M
10	CPS	3.66	3.66	1.18	1.11	0.96	0.90	**0.65**	**0.59**	**0.47**	**0.42**
	CPS^{*b}	4.88	3.96	1.43	1.38	1.28	1.18	0.94	0.88	0.80	0.70
	WOTD	41.69	41.69	8.80	6.44	4.73	3.58	17.05	14.86	11.77	6.87
	SA	16.40	15.92	7.14	5.55	4.83	4.36	15.49	14.50	12.36	11.00
100	CPS	4.24	4.24	1.79	1.72	1.63	1.58	**1.22**	**1.16**	**1.08**	**1.02**
	CPS^{*b}	5.41	4.54	2.04	1.97	1.86	1.78	1.52	1.45	1.34	1.27
	WOTD	42.28	42.28	9.09	6.68	4.88	3.70	17.13	15.19	11.32	6.71
	SA	16.76	16.11	7.32	5.73	5.08	4.59	15.80	14.94	13.02	11.52
1K	CPS	4.27	4.26	1.79	1.73	1.62	1.58	**1.25**	**1.18**	**1.07**	**1.03**
	CPS^{*b}	5.49	4.58	2.08	2.02	1.89	1.80	1.57	1.50	1.38	1.29
	WOTD	42.91	42.91	8.99	6.45	4.89	3.69	17.52	15.14	11.82	7.16
	SA	16.74	16.08	7.28	5.61	4.94	4.49	15.56	14.51	12.57	11.29
10K	CPS	4.27	4.27	1.85	1.77	1.67	1.62	**1.25**	**1.18**	**1.08**	**1.04**
	CPS^{*b}	5.49	4.54	2.06	1.96	1.88	1.79	1.52	1.43	1.34	1.26
	WOTD	42.50	42.50	9.06	6.72	4.94	3.68	17.10	15.15	11.48	6.79
	SA	16.78	16.09	7.38	5.74	5.03	4.53	15.72	14.71	12.67	11.29

Exact match query											
Query length	Index	1 page	2 pages	LRL buffer replacement				MRL buffer replacement			
				16M	48M	96M	128M	16M	48M	96M	128M
10	CPS	13.35	5.29	2.73	2.66	2.53	2.47	**2.27**	**2.17**	**1.99**	**1.93**
	CPS^{*b}	16.26	5.97	3.00	2.92	2.81	2.70	2.92	2.83	2.69	2.51
	WOTD	57.52	57.52	9.30	6.90	5.13	3.96	20.23	17.82	14.38	9.43
	SA	17.52	16.41	7.47	5.80	5.04	4.58	15.93	14.90	12.71	11.32
100	CPS	4.29	4.26	1.80	1.74	1.65	1.59	**1.24**	**1.18**	**1.10**	**1.04**
	CPS^{*b}	5.44	4.55	2.05	1.98	1.87	1.79	1.52	1.43	1.35	1.27
	WOTD	42.28	42.28	9.09	6.68	4.88	3.70	17.13	15.19	11.32	6.71
	SA	16.77	16.11	7.32	5.73	5.08	4.59	15.80	14.94	13.02	11.53
1K	CPS	4.27	4.27	1.80	1.73	1.62	1.58	**1.25**	**1.18**	**1.07**	**1.03**
	CPS^{*b}	5.49	4.58	2.08	2.02	1.89	1.80	1.57	1.50	1.38	1.29
	WOTD	42.92	42.92	8.99	6.45	4.89	3.69	17.53	15.15	11.82	7.16
	SA	16.76	16.08	7.28	5.61	4.94	4.49	15.56	14.51	12.57	11.29
10K	CPS	4.27	4.27	1.85	1.77	1.67	1.62	**1.25**	**1.18**	**1.08**	**1.04**
	CPS^{*b}	5.49	4.54	2.06	1.96	1.88	1.79	1.52	1.43	1.34	1.26
	WOTD	42.50	42.50	9.06	6.72	4.94	3.68	17.10	15.15	11.48	6.79
	SA	16.78	16.09	7.38	5.74	5.03	4.53	15.72	14.71	12.67	11.29

CPS^{*b} - CPS-tree with breadth first partition. The default uses greedy partition.

Table 5.4: Average page fault on index buffer using different buffer replacement policies for fruit fly genome.

5.5.2 Performance results

Result 1 - *IO on index buffer:* First, we examine the IO efficiency in traversing CPS-tree index structure. We use the fruit fly genome in this comparison with the main portion of the index structure residing on disk. The size of the index buffer ranges from 1-2 pages, to 128MB. For existence match query, the page faults are generated from traversing the indices alone without reporting on the occurrences. While the exact match query finds the query pattern in the index and further enumerates the occurrences and hence incurring more IO cost. We report on queries of length 10 to 10000 and compare the two buffer replacement polices, LRL and MRL respectively for the index. For CPS-tree, we also look into the IO efficiency of our greedy approach of tree partitioning versus the breadth first approach.

The results are tabulated in Table 5.4. We find the buffer replacement policy that works well with each of the indexing structures, CPS-tree, WOTD-tree and SA indices, and present the comparison in Figure 5.9. The figures show the average page fault for existence and exact match queries of length 10. CPS-tree has better average performance using MRL buffer replacement policy than LRL, while WOTD-tree and SA work better with LRL policy. The two figures show the difference where exact match query gets more page faults than existence query for the same indexing structure. This is contributed by the enumeration of the occurrences for exact match query. The difference quickly disappears as the query length increases to 100 and beyond as the number of occurrences for query length of 100 and more is near to 1. The average occurrences, occ, per query found are 388.58, 1.64, 1.20 and 1.00 for query length 10, 100, 1000 and 10000 respectively.

We report the following observations based on results from Table 5.4: (1) CPS-tree consistently outperforms the other indices on different index buffer size and for query of length 10 to 10000. It can be seen that CPS-tree displays very good access locality, generating very few page fault per query. (2) Our greedy tree partitioning gives fewer IO than the breadth first approach for CPS-tree. Our finding shows that a careful organization of the nodes into blocks does significantly improve search performance. (3) Query length of 100 and more have very similar IO performance as very rarely can you find 2 or more positions with matching length ≥ 100 on the genome. Query of length 10 generates more page faults than those of length 100, mainly from reporting on the occurrences.

To conclude, CPS-tree generates at most 1-2 page faults per query on the index which is much lesser compared to WOTD-tree and SA. CPS-tree also performs more consistently with different index buffer size and policy. We have considered using bigger index buffer for WOTD-tree in our comparison as WOTD-tree consists of 3 data structures that need to be buffered separately. It may seem bias at one glance to compare a bit-packed CPS-tree against WOTD-tree that is word based. Note that it is not straight forward to modify and pack WOTD-tree using bit representation as some engineering and design issues need to be addressed. However, from what is observed, CPS-tree with 16M bytes of index buffer is more IO efficient than WOTD-tree with 128M bytes of index buffer on the fruit fly genome.

From here onwards, we use MRL index buffer replacement policy for CPS-tree and LRL for WOTD-tree and SA in our experiments to compare their best performances.

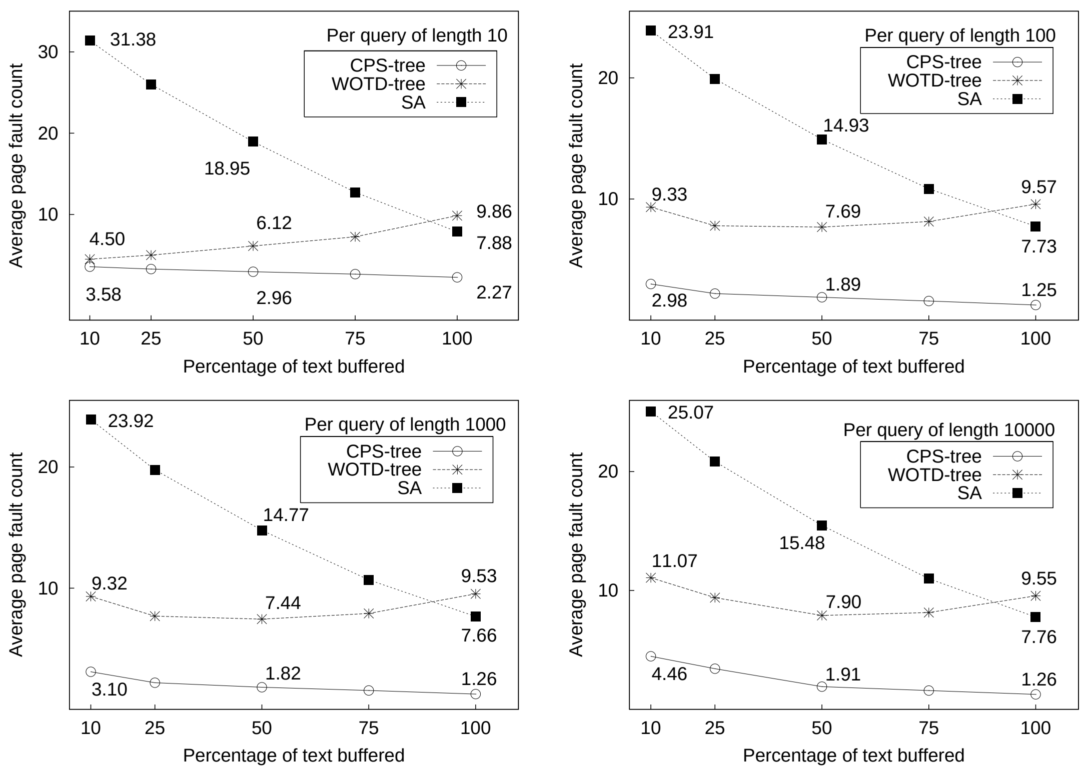


Figure 5.10: Result 2 - Average page fault on text and index buffers for fruit fly genome to answer exact match query (total 128MB).

Result 2 *- IO on combined buffers:* Here, we look at the buffer size allocation between the text and the index to answer exact match query. Given a total of 128MB for buffering, we varied the text buffer size as a percentage of the text size, and use the remaining space available to buffer the index. Text buffer uses the LRL replacement policy and the initial portion of the text is first read into the buffer. Results in Figure 5.10 show that CPS-tree has the best IO performance, generating significantly less page faults when compared to the other 2 indices. Also CPS-tree and SA work best with full text buffering (in memory) while WOTD-tree gives mixed results depending on the query length.

We observed that CPS-tree works well with small index buffer. Increasing the size of the index buffer does not result in as many page fault reduction as increasing the size of the text buffer. As such, to optimize performance with limited memory space, we should allocate a smaller buffer space to the index while the rest of the memory space is used to buffer the text. For example, on the fruit fly genome, with 128 MB of memory space for buffering, we can answer exact match query with an average of < 3 page faults per query.

In the running of the queries, we find that CPS-tree, WOTD-tree and SA, all took from a few to tenths of milliseconds, on the average to answer a query. It is noted that CPS-tree is generally 2 to 3 times faster than WOTD-tree and SA, with SA being the slowest. Effort is taken to flush the system cache between each run by executing some unrelated memory intensive routines so as to minimize the memory effect on the timing.

Result 3 *- computational time analysis:* We study the computational time needed to perform search on the indices. The index and text are both fully loaded into the memory and the results are shown in Table 5.5. This is performed on the E. coli genome (4.6M).

When compared to WOTD-tree, CPS-tree is much faster, showing that CPS-tree has a better representation scheme for suffix tree. Despite the fact that CPS-tree is bit-packed and would incur some computational cost in extracting the fields for processing, it is still faster than WOTD-tree with fields of word size. CPS-tree is equally fast when compared to SA for short queries with CPS-tree gaining faster performance as the query gets longer.

Query	Per query	Query time (μsec) per query		
length	occ count	CPS-tree	WOTD-tree	SA
10	9.960	21	52	22
100	1.078	16	39	18
1K	1.003	25	50	36
5K	1.000	51	71	52

Table 5.5: Result 3 - In-memory (exact match) query timing on E. coli genome.

Query	k = 1 (per query)		k = 2 (per query)	
length	occ count	paging	occ count	paging
10	7369	58	79346	684
100	1.68	37	1.72	460

Table 5.6: Result 4 - k-mismatch query on fruit fly genome.

***Result 4** - inexact match search:* CPS-tree is capable of handling more complex query. We run k-mismatch query on CPS-tree and the results are shown in Table 5.6.

K-mismatch query finds all occurrences on the text that has Hamming distance $\leq k$ from the query string. The search strategy used is to first find the exact match string in the CPS-tree. Next, we backtrack along the traveled path and “erase” the matched characters on the path as we do so. At each node on the path (starting from the deepest node), we branch to compare the remaining characters on the query with every other paths, incurring one mismatch (first character of the branch edge with the query) at the time. This recursive process is then extended to k mismatches. The search is intuitive and is a simplification of the general dynamic programming approach for string comparison which caters to edit distance measure [21, 97]. The number of occurrences increases sharply for large k, especially for short queries. Using a total of 200MB for buffering, we find that the running time is around 0.1 to 0.2 sec per query for $k = 1$ and 0.4 to 2 sec when extended to 2 mismatches. For $k = 1$, there is a total of 31 and 3001 substituted query patterns being searched, for query length 10 and 100, and that gives an average page fault of 1.87 and 0.01 per substituted query pattern respectively. These numbers are much lower than searching the individual pattern directly as there is saving in the page access through the search approach. For long query of length 100, not many substituted pattern can find a match and hence resulting in early termination of the search and faster running time.

5.5.3 CPS-tree on human genome

We constructed CPS-tree for the human genome of 3.08 billion characters (concatenating the 24 chromosomes in the human reference assembly and substituting all character ‘N’ with randomly picked ‘A’, ‘C’, ‘G’ or ‘T’). The human genome is packed into 770MB of space (2 bits per character). The setup consists of a desktop computer running Linux with Intel Core 2 Duo 2.66GHz CPU, $4GB$ of RAM and a single SATA 500GB hard disk. We conduct similar investigation on the IO performance of the CPS-tree index (27 GB size) to see if similar performance can be observed as compared to the smaller index for fruit fly genome.

***Result 5** - IO on index buffer:* Table 5.7 gives the average page fault on the index buffer (using MRL replacement policy) of different sizes. Query of length 10 generates significantly more page fault due to enumeration of the large number of occurrences (an average of 13761 matches per query as shown in Table 5.8). On the whole, for human genome, it takes around 4 disk access to search the index. Also it is observed that increasing the index buffer size from 16M to 2GB ($\leq$ 7% of the full index size) has minimal impact on reducing the number of page fault. The same behaviour is observed on smaller index for the fruit fly genome. We increase the memory page size from the default setting of $8k$ to $16k$ and $64k$ to investigate the amount of IO reduction using a larger page swap. As shown in the table, the number of page faults is reduced to around 3 per query. However, we find that each page loading becomes more computationally expensive and the gain is not realized in practice.

Result 6 *- IO on combined buffers:* We examine the IO performance on a total buffer size of 1GB for both text and index (as shown in Table 5.8). It is without surprise that the best performance goes to buffering the full text in memory. The performance is consistent with that reported for the fruit fly genome.

Query length	1 page	2 pages	16M	512M	1GB	2GB
8k page size (default)						
10	14.71	13.71	13.71	13.49	13.39	13.25
100	5.23	4.23	4.23	4.18	4.10	3.99
1K	5.18	4.19	4.19	4.14	4.07	3.96
10K	5.18	4.18	4.18	4.13	4.07	3.91
16k page size						
10	13.52	12.52	12.52	12.44	12.35	12.23
100	4.52	3.52	3.52	3.48	3.42	3.32
1K	4.53	3.53	3.53	3.49	3.44	3.34
10K	4.51	3.51	3.51	3.46	3.41	3.28
64k page size						
10	12.53	11.53	11.53	11.47	11.41	11.33
100	3.83	2.83	2.83	2.79	2.75	2.62
1K	3.80	2.81	2.81	2.78	2.74	2.67
10K	3.79	2.79	2.79	2.75	2.71	2.61

Table 5.7: Result 5 - Average page fault on index buffer for Human Genome to answer exact match query.

Query	Per query	Percentage of text buffered				
length	occ count	10%	25%	50%	75%	100%
10	13761.80	15.11	14.82	14.39	13.93	13.54
100	3.41	7.02	6.54	5.79	4.97	4.21
1K	1.00	6.99	6.51	5.80	5.00	4.16
10K	1.00	7.27	6.73	5.88	5.01	4.16

Table 5.8: Result 6 - Average page fault on text and index buffers for Human Genome to answer exact match query (total 1GB).

Query length	50	100	1K	5K	10K
Query time (sec)	22	45	435	1382	1993
Match count	1094	1901	258449	83351	37656
Filtered match count	781	1332	143404	38277	17643

Table 5.9: Result 7 - Local alignment search on the Human Genome.

***Result 7** - Local alignment search:* BLAST [4, 5, 103] and FASTA [82, 83] are some popular heuristic search tools to find the local alignment between 2 biological sequences. The well-known Smith-Waterman dynamic programming algorithm [92] is able to exhaustively locate all the alignments however the approach is computationally intensive. This limits its usage. There are works that effectively adapt the dynamic programming technique over suffix tree [21, 45, 69, 97]. We will show that CPS-tree is capable of supporting local alignment search. Our experiment differs from the previous reported studies on suffix tree in that we handle the affine gap model (rather than basic edit distance measure) on DNA sequence. The affine gap model is more realistic for biological sequences but it is more complex to compute where there are three dynamic programming matrices to be filled.

We compute matrices $B[i, j]$, $T[i, j]$ and $Q[i, j]$ for ith row and jth column with the query pattern P along the y-axis and the suffix string S on the x-axis. Matrix $T[i, j]$ finds the optimal score to align $P[1..i]$ and $S[1..j]$, ending with the last character $S[j]$ of the suffix string matches a gap space appended to the end of $P[1..i]$. Matrix $Q[i.j]$ is similar to $T[i, j]$, finding the optimal score with last character $P[i]$ of the query matches a gap space appended to the end of $S[1..j]$. Score matrix $B[i, j]$, gives the optimal score to align $P[1..i]$ with $S[1..j]$, from the best score out of $T[i, j]$, $Q[i, j]$ and a possible substitution (can be a match as well) of $P[i]$ with $S[j]$. The matrices are updated, in column-wise order (increasing i first then j), as follows:

$$
\begin{aligned}
B[i,j] &= \max \begin{cases} T[i,j], \\ Q[i,j], \\ B[i-1,j-1] + d(x_i, y_i) \end{cases} \\
T[i,j] &= \max \begin{cases} B[i-1,j] - o - e, \\ T[i-1,j] - e \end{cases} \\
Q[i,j] &= \max \begin{cases} B[i,j-1] - o - e, \\ Q[i,j-1] - e \end{cases} \\
Initialization &: \\
B[0,0] &= B[i,0] = 0 \\
T[i,0] &= T[0,j] = -\infty \\
Q[i,0] &= Q[0,j] = -\infty \\
B[0,j] &= -\infty
\end{aligned}
$$

We have o as the gap open cost, e is the gap extension cost and the substitution cost $d(x_i, y_i)$ to compare the ith character of P with the jth character of S. $d(x_i, y_i) = m$ if $x_i == y_i$ else $d(x_i, y_i) = s$ (m is the match score and s is the substitution penalty).

We prune the entries once the score is ≤ 0 and use the score setting with $o = 5$, $e = 2$, $s = -3$ and $m = 1$. Only alignments with score at and above the threshold (15 for query length ≤ 1000 and 25 otherwise) are reported. There are 50 query patterns of length 50, 100, 1000, 5000 and 10000 each, randomly generated from the fruit fly genome. The human genome is fully loaded into memory for the search. We report the locations on the human genome where the alignments occur with score that met the threshold. The average search time per query is given in Table 5.9. We also report on the average number of matches per query and the average count after filtering out overlaps on the genome. As can be seen, there are many alignments returned from the search for query length 1000, as such, we increased the threshold from 15 to 25 for even longer queries, to keep the number of alignments returned manageable. For any 2 alignments that overlap fully (one is enclosed within the other) on the genome, the alignment with the lesser score is removed during filtering. It is noted that the number of alignments varied widely from 0 to 384K per query for long queries. Hence the average match count per query, as given in the table, is not consistent with the query length.

The goal of our investigation is to show that CPS-tree is efficient to handle practical queries like local alignment search in finding all alignments. We conclude that our approach is a lot faster than Smith-Waterman algorithm. CPS-tree is still unable to outperform BLAST especially for long queries. On short queries of short length (≤ 100), the performance on CPS-tree and BLAST are comparable. We might argue that the quality of the results returned are different as exhaustive search is performed on CPS-tree to account for all alignments while BLAST is a heuristic approach.

5.6 Discussion

There are rooms for further optimization and investigation in our current work. The tree structure used in CPS-tree may be further compressed using techniques like the balanced parenthesis representation [71]. Alternatively, for better space usage, we can limit the maximum query length (which should be much shorter than the indexed sequence length), so that those logical blocks whose character depth is larger than the maximum query length can be pruned off, resulting in a smaller index. It is also possible to reduce the index size by sampling the text positions. Reduction in index size may often result in heavier computational cost as a trade-off. However, we hope that smaller index means bigger part of the index can reside in memory at any one time, and reduces the number of index pages to be fetched from disk. There is gain if the reduction in disk IO time is much more than the increased time in computation. In general, we would like to explore ways to further reduce the index size without significant impact on the computational time.

Next, we have so far started to explore basic alignment search using CPS-tree for DNA sequences. We would like to consider protein sequences and other variations of sequence comparison supported by BLAST package. We hope that there are more advantages of using suffix tree approach for more complex substitution cost matrix like for protein where the substitution cost for a character x with another character y is dependent on the (x, y) pair. In such scenario, the heuristic approach of filtering used in BLAST to reduce the candidate matches will be less effective or computationally more costly.

5.7 Summary

Suffix tree is an important data structure for indexing a long sequence (like a genome sequence) or a concatenation of sequences. It finds many applications in practice, especially in the domain of bioinformatics. Suffix tree allows for efficient pattern search with time independent of the indexed sequence length. However, the performance of disk-based suffix tree is a concern as it may be slowed down significantly due to poor access locality amounting to high disk IO cost.

The focus of this work is to design an IO-efficient suffix tree representation on disk. We show that representing suffix tree using CPS-tree has several advantages. First, our representation gives tight upper IO bounds on various tree traversal and search operations. For example to recover a matching substring in the CPS-tree takes $O(\log n)$ page accesses on the tree where n is the length of the indexed sequence. Second, our representation and storage scheme improves access locality and reduces the number of page fault, resulting in efficient pattern matching and efficient tree traversal operations. Third, by bit packing, our index remains compact. Experimental results show that CPS-tree outperforms other index structures resulting in significantly fewer page faults using a small memory space for buffering. When fully loaded into the main memory, CPS-tree is still efficient. We build CPS-tree on the human genome of 3 billion characters, and further show that CPS-tree is scalable to handle large genome and to answer queries like exact match and local alignment search. To our knowledge, this is the first reported IO performance of suffix tree indexing at this genome scale. We are also the first to report the performance of local alignment search using the affine gap cost model on suffix tree built on the human genome. Hence, we expect CPS-tree to be a good disk-based representation of suffix tree, with potential use in practical applications. The preliminary results are presented in [102].

Chapter 6

Conclusion

Suffix data structures are popularly used to index string datasets especially in the area of computational biology. In this thesis, we study suffix data structures in two computing models, in memory as well as disk based processing. We propose a number of efficient data structures to tackle string search problems ranging from exact and approximate matching to sequence alignment.

First in the in-memory setting, we give compressed data structures using $o(n)$ words or $O(n)$ bits to index the text and present fastest known search time for exact and approximate string search. Specifically, we claim that given a text T of length n and a length-m query string P over an alphabet A, we can build an $O(n\sqrt{\log n}\log|A|)$-bit space data structure to answer 1-mismatch (or 1-difference) query in $O(|A|m\log\log n + occ)$ time, where occ is the number of occurrences. The space of our data structure can be further reduced to $O(n\log|A|)$ bits with a slow down factor of $\log^{\epsilon} n$, for $0 < \epsilon \leq 1$.

Extending to k-mismatch (and k-difference) problem, we can solve the problem in $O(|A|^k m^k(k+\log\log n)+occ)$ and $O(\log^\epsilon n(|A|^k m^k(k+\log\log n)+occ))$ query time using an $O(n\sqrt{\log n}\log|A|)$-bit (assume $|A| = O(2^{\sqrt{\log n}})$) and an $O(n\log|A|)$-bit indexing data structures, respectively. The k-don't-care problem, a special case of k-mismatch problem, can be solved in $O(|A|^k(m+\log\log n)+occ)$ or $O(\log^\epsilon n(|A|^k(m+|A|\log\log n)+occ))$ time, using $O(n\sqrt{\log n}\log|A|)$ (assume $|A| = O(2^{\sqrt{\log n}})$) or $O(n\log|A|)$ bits data structure respectively.

We also work on the exact string matching problem using $O(n\log|A|)$ bits data structure. The optimal query time of $O(m/\log_{|A|} n + occ)$ can be achieved for $m = \Omega(\log^2_{|A|} n \log^\epsilon n \log\log n)$, where $\epsilon > 0$. Relaxing the index size to $o(n\log n)$ bits data structure, for fixed finite alphabet and any pattern of length m, we answer the exact string matching problem in $O(m/\log_{|A|} n + \log n\log\log n + occ)$ time. Next we show that $(m + occ)$ query time is achievable using $o(n\log n)$ bits data structure.

Working on suffix tree on disk, we present CPS-tree, an IO-efficient suffix tree representation. We give both experimental results as well as analysis on the IO performances for various tree traversal and search operations to justify that our suffix tree is the choice to be used in disk based setting. We also introduce a mechanism denoted as "forward links" into the tree structure to reduce the number of page access to $O(\log n)$ pages in matching any given string (where n is the length of the text indexed). To illustrate that CPS-tree has practical applications, we construct CPS-tree for the human genome and perform local alignment search over CPS-tree. To our knowledge, this is the first reported IO performance study, and local alignment performance using affine gap cost model, for a suffix tree at this genome scale.

6.1 Future directions

We have performed detailed study on algorithms to search string indices comprising of compressed suffix data structures efficiently. However, we have yet to address the issue of randomness in access pattern on compressed suffix data structures. This limits its deployment to machines with large enough main memory to hold the indices. On the other hand, explicit suffix tree representation is too large to fit into main memory. It remains an open research problem to find a hybrid or new data structure that exhibits good access locality with index size close to that of compressed suffix data structures.

Other techniques like sampling the text to reduce the size of the suffix tree (not all text position are being indexed) deserves further investigation. The practical trade-offs point of reduced index size and increased computation can only be determined with more empirical studies.

Bibliography

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] S. Alstrup, M. A. Bender, E. D. Demaine, M. Farach-Colton, T. Rauhe, and M. Thorup. Efficient tree layout in a multilevel memory hierarchy. The revised version of the published paper. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 165–173, 2002.

[3] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[4] S. F. Altschul, W. Gish, W. Miller, E. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[5] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and PSI-blast: A new generation of protein database

search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[6] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[7] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *Journal of Algorithms*, 37(2):309–325, 2000.

[8] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 794–803, 2000.

[9] R. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Journal of Information Systems*, 21(6):497–514, 1996.

[10] R. A. Baeza-Yates and G. Navarro. A practical index for text retrieval allowing errors. In *CLEI*, pages 273–282, 1997.

[11] S. J. Bedathur and J. R. Haritsa. Search-optimized suffix-tree storage for biological applications. In *Proceedings of the International Conference on High Performance Computing*, pages 29–39, 2005.

[12] R. S. Boyer and S. J. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[13] A. L. Brown. Constructing chromosome scale suffix trees. In *Proceedings of the 2nd Conference on Asia-Pacific Bioinformatics*, pages 105–112, 2004.

[14] A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In *Proceedings of the 8th Annual European Symposium on Algorithms*, pages 120–131, 2000.

[15] M. Burrows and D. Wheeler. *A Block Sorting Lossless Data Compression Algorithm*. Technical Report 124, Digital Equipment Corporation, 1994.

[16] X. Cao, S. C. Li, and A. Tung. Indexing DNA sequences using q-grams. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications*, pages 4–16, 2005.

[17] H. L. Chan, T. W. Lam, W. K. Sung, S. M. Yiu, and S. S. Wong. Compressed indexes for approximate string matching. In *Proceedings of the European Symposium on Algorithms*, pages 208–219, 2006.

[18] H. L. Chan, T. W. Lam, W. K. Sung, S. M. Yiu, and S. S. Wong. A linear size index for approximate string matching. In *Proceedings of the Symposium on Combinatorial Pattern Matching*, pages 49–59, 2006.

[19] C. F. Cheung, J. X. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):90–105, 2005.

[20] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[21] A. L. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 41–54, July 1995.

[22] R. Cole, L-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 91–100, 2004.

[23] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole geome. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[24] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[25] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings of the International Conference on Very Large Data Bases*, pages 343–353, 1996.

[26] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 390–398, 1997.

[27] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[28] P. Ferragina. String search in external memory: Data structures and algorithms. In Srinivas Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 35, pages 35.1–35.49. Chapman & Hall/CRC, 1 edition, 2005.

[29] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382, 1996.

[30] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[31] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[32] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 369–378, 2001.

[33] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[34] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proceedings of the 3rd Workshop on Algorithm Engineering*, pages 30–42, 1999.

[35] J. Gil and A. Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.

[36] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 5: New Indices for Text: PAT Trees and PAT Arrays, pages 66–82. Prentice-Hall, 1992.

[37] R. Grossi and G. Italiano. Suffix trees and their applications in string algorithms. In *Proceedings of the 1st South American Workshop on String Processing*, pages 57–76, 1993.

[38] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of ACM Symposium on Theory of Computing*, pages 397–406, 2000.

[39] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, accepted.

[40] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.

[41] M. Halachev, N. Shiri, and A. Thamildurai. Exact match search in sequence data using suffix trees. In *Proceedings of the International Conference on Information and Knowledge Management,* pages 123–130, 2005.

[42] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics,* 18(Suppl. 1):S312–S320, 2002.

[43] W. K. Hon, T. W. Lam, W. K. Sung, W. L. Tse, C. K. Wong, and S. M. Yiu. Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments.,* pages 31–38, 2004.

[44] W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of IEEE Symposium on Foundations of Computer Science,* pages 251–260, 2003.

[45] E. Hunt, M. P. Atkinson, and R. W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11:256–271, 2002.

[46] H. Hyyrö and G. Navarro. A practical index for genome searching. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval*, pages 241–349, 2003.

[47] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of Symposium on Foundations of Computer Science,* pages 549–554, 1989.

[48] R. Japp. The top-compressed suffix tree: A disk-resident index for large sequences. In *Bioinformatics Workshop, 21st Annual British national Conference on Databases*, 2004.

[49] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science*, pages 240–248, September 1991.

[50] L. Kaderali and A. Schliep. Selecting signature oligonucleotides to identify organisms using DNA arrays. *Bioinformatics*, 18:1340–1349, 2002.

[51] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 195–209, 2000.

[52] J. Kärkkäinen and S. S. Rao. Full-text indexes in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierachies: Advanced Lecttures*, volume 2625 of *LNCS*, chapter 7, pages 149–170. Springer-Verlag Berlin Heidelberg, 2003.

[53] J. Kärkkäinen and E. Sutinen. Ziv-Lempel index for q-grams. In *Proceedings of the 4th Annual European Symposium on Algorithms*, pages 378–391, 1996.

[54] W. J. Kent. BLAT: The BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[55] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[56] P. Ko and S. Aluru. Suffix tree applications in computational biology. In Srinivas Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 6, pages 6.1–6.27. Chapman & Hall/CRC, 1 edition, 2005.

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

[58] S. Kurtz and C.Schleiermacher. REPuter: Fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15:426–427, 1999.

[59] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12), 2004. http://mummer.sourceforge.net.

[60] T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proceedings of the International Computing and Combinatics Conference*, pages 401–410, 2002.

[61] T. W. Lam, W. K. Sung, and S. S. Wong. Improved approximate string matching using compressed suffix data structures. In *Proceedings of the Annual International Symposium on Algorithms and Computation*, pages 339–348, 2005.

[62] T. W. Lam, W. K. Sung, and S. S. Wong. Improved approximate string matching using compressed suffix data structures. *Algorithmica*, accepted.

[63] G. M. Landau and U. Vishkin. Fasl parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[64] M. Li, B. Ma, and D. Kisman. PatternHunterII: Highly sensitive and fast homology search. In *Proceedings of the 14th International Conference on Genome Informatics*, pages 164–175, 2003.

[65] B. Ma, J. Tromp, and M. Li. PatternHunter: Faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, March 2002.

[66] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[67] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[68] V. M$\ddot{a}$kinen, G. Navarro, and K. Sadakane. Advantages of backward searching - efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proceedings of the Annual International Symposium on Algorithms and Computation*, pages 681–692, 2004.

[69] C. Meek, J. M. Patel, and S. Kasetty. OASIS: An online and accurate technique for local-alignment searches on biological sequences. In *Proceedings of the International Conference on Very Large Data Bases*, pages 910–921, 2003.

[70] D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.

[71] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[72] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[73] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, March 2001.

[74] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[75] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[76] G. Navarro and R. A. Baeza-Yates. A new indexing method for approximate string matching. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, pages 163–185, 1999.

[77] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proceedings of the 11th Data Compression Conference*, pages 459–468, 2001.

[78] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, pages 14–36, 1999.

[79] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 350–365, 2000.

[80] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[81] G. Pavesi, G. Mauri, and G. Pesole. An algorithm for finding signals of unknown length in DNA sequences. *Bioinformatics*, 17(Suppl. 1):S207–S214, 2001.

[82] W. R. Pearson. Flexible sequence similarity searching with the FASTA3 program package. *Methods in Molecular Biology*, 132:185–219, 2000.

[83] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of National Academy of Sciences USA*, 85:2444–2448, 1988.

[84] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k–ary trees and multisets. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[85] S. S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82:307–311, 2002.

[86] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all ϵ-matches over a given length. In *Proceedings of the 9th Annual International Conference on Research in Computational Molecular Biology,* pages 189–203, 2005.

[87] K. Sadakane. Succinct representation of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms,* pages 225–232, 2002.

[88] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems,* accepted.

[89] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[90] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms,* 1:359–373, 1980.

[91] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 257–271. Carleton University Press, 1996.

[92] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[93] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 50–63, 1996.

[94] Z. Tan, X. Cao, B. C. Ooi, and A. K. H. Tung. The ed-tree: An index for large DNA sequence databases. In *International Conference on Scientific and Statistical Database Management*, pages 151–160, 2003.

[95] S. Tata, R. A. Hankins, and J. M. Patel. Practical suffix tree construction. In *Proceedings of the International Conference on Very Large Data Bases*, pages 36–47, 2004. http://www.eecs.umich.edu/tdd/index.html.

[96] H. N. D. Trinh, W. K. Hon, T. W. Lam, and W. K. Sung. Approximate string matching using compressed suffix arrays. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching*, pages 434–444, 2004.

[97] E. Ukkonen. Approximate string-matching over suffix trees. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, pages 228–242, 1993.

[98] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[99] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[100] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17:81–84, August 1983.

[101] H. E. Williams and J. Zobel. Indexing and retrieval for genomic database. *Proceedings of IEEE Transactions on Knowledge and Data Engineering*, 14:63–78, 2002.

[102] S. S. Wong, W. K. Sung, and L. S. Wong. CPS-tree: A compact partitioned suffix tree for disk-based indexing on large genome sequences. In *Proceedings of the International Conference on Data Engineering*, pages 1350–1354, 2007.

[103] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1):203–214, 2000.