Iowa State University
Digital Repository

2007

Suffix trees and suffix arrays in primary and secondary storage

Pang Ko
Iowa State University

Recommended Citation

www.manaraa.com

Suffix trees and suffix arrays in primary and secondary storage

by

Pang Ko

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Srinivas Aluru, Major Professor
David Fernández-Baca
Suraj Kothari
Patrick Schnable
Srikanta Tirthapura

Iowa State University

Ames, Iowa

2007

UMI Number: 3274885

UMI®

DEDICATION

To my parents

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

ABSTRACT

In recent years the volume of string data has increased exponentially, and the speed at which these data is being generated has also increased. Some examples of string data includes biological sequences, internet webpages, and digitalized documents, to name a few. The indexing of biological sequence data is especially challenging due to the lack of natural word and sentence boundaries. Although many algorithms are able to deal with this lack of natural boundaries, they are not able to process the large quantity of data in reasonable time. To speed up the runtime of these algorithms, suffix trees and suffix arrays are routinely used to generate a set of starting positions quickly and/or narrow down the set of possibilities need to be considered.

The first contribution of this dissertation is a linear time algorithm to sort all the suffixes of a string over a large alphabet of integers. The sorted order of suffixes of a string is also called suffix array, a data structure introduced by Manber and Myers that has numerous applications in pattern matching, string processing, and computational biology. Though the suffix tree of a string can be constructed in linear time and the sorted order of suffixes derived from it, a direct algorithm for suffix sorting is of great interest due to the space requirements of suffix trees. Our result is one of the first linear time suffix array construction algorithms, which improve upon the previously known $O(n \log n)$ time direct algorithms for suffix sorting. It can also be used to derive a different linear time construction algorithm for suffix trees. Apart from being simple and applicable for alphabets not necessarily of fixed size, this method of constructing suffix trees is more space efficient.

The second contribution of this dissertation is providing a new suffix tree layout scheme for secondary storage and present construction, substring search, insertion and deletion algorithms

ix

using this layout scheme. For a set of strings of total length n, a pattern p and disk blocks of size B, we provide a substring search algorithm that uses $O(|p|/B + \log_B n)$ disk accesses. We present algorithms for insertion and deletion of all suffixes of a string of length m that take $O(m \log_B(n + m))$ and $O(m \log_B n)$ disk accesses, respectively. Our results demonstrate that suffix trees can be directly used as efficient secondary storage data structures for string and sequence data.

The last contribution of this dissertation is providing a self-adjusting variant of our layout scheme for suffix trees in secondary storage that provides optimal number of disk accesses for a sequence of string or substring queries. This has been an open problem since Sleator and Tarjan presented their splaying technique to create self-adjusting binary search trees in 1985. In addition to resolving this open problem, our scheme provides two additional advantages: 1) The partitions are slowly readjusted, requiring fewer disk accesses than splaying methods, and 2) the initial state of the layout is balanced, making it useful even when the sequence of queries is not highly skewed. Our layout scheme, and its self-adjusting variant are also applicable to PATRICIA trees, and potentially to other data structures.

CHAPTER 1. INTRODUCTION

Suffix tree and suffix array are important data structures in information retrieval, text processing, and computational biology. They are especially suitable for indexing biological sequences where predefined boundaries such as words, phrases, and sentences are absent. Biological sequence databases are increasing exponentially in recent years, both in terms of the number of sequences, and the total size of the database. One of the most popular biological sequence databases, GenBank, contains 71,802,595 sequences and 75,742,041,056 base pairs as of April 2007 and its size has been doubling almost every 18 months [9]. Several applications of suffix trees and suffix arrays in computational biology can be found in [3, 5, 28].

1.1 Suffix Array in Main Memory

Given a string T of length n, the suffix T_i $(1 \leq i \leq n)$ of T is the substring of T starting from position i and ending at the end of T. The suffix array of T is the lexicographically sorted order of all its suffixes, usually denoted as SA. The suffix array is often used in conjunction with another array, called lcp array, containing the lengths of the longest common prefixes between every pair of consecutive suffixes in the suffix array. Figure 1.1 shows the string "MISSISSIPPI\$" and its suffix array and the associated lcp array.

Manber and Myers introduced the suffix array data structure [33] as a space-efficient alternative for suffix trees. Gonnet *et al.*[19] have also independently developed the suffix array, which they refer to as the PAT array. As a lexicographic-depth-first traversal of a suffix tree can be used to produce the sorted list of suffixes, suffix arrays can be constructed in linear time and space using suffix trees. However, this defeats the whole purpose if the goal is to avoid suffix trees. Hence, Manber and Myers presented a direct construction algorithm that

String	M	I	S	S	I	S	S	I	P	P	I	$
Position	1	2	3	4	5	6	7	8	9	10	11	12
Suffix Array	12	11	8	5	2	1	10	9	7	4	6	3
lcp		0	1	1	4	0	0	1	0	2	1	3

Figure 1.1   The string "MISSISSIPPI\$" and its suffix and *lcp* array.

runs in $O(n \log n)$ worst-case time and $O(n)$ expected time. They construct the suffix array by repeated bucketing:

1. Initially all suffixes are bucketed according to their first characters. This is referred to as the partial suffix array SA_1 because it is sorted according to the first characters.

2. Then for each entry j in SA_h, the suffix $T_{SA_h[j]-h}$ is moved to the front of its bucket. All the suffixes in the partial suffix array have been sorted according to the first h characters before the beginning of the step. Then at the end of this step all suffixes will be sorted according to the first $2h$ characters.

3. Step 2 is repeated at most $\log n$ times, or until all the suffixes are in their own buckets.

Although Manber and Myers also presented a *lcp* array construction algorithm in their paper, it requires $O(n \log n)$ additional time. The main idea is: if two adjacent suffixes are in the same bucket after the suffix array is sorted according to the first h characters, then h is stored as their temporary *lcp*. If two adjacent suffixes, say T_i and T_j, are split into two different buckets after sorting according to the first h characters but not the first $\frac{1}{2}h$ characters, then their correct *lcp* can be calculated by adding $\frac{1}{2}h$ to the *lcp* of $T_{i+\frac{1}{2}h}$ and $T_{j+\frac{1}{2}h}$.

Kasai *et al.*[25] presented a linear time algorithm for constructing the *lcp* array directly from the suffix array. Their algorithm uses the fact that if suffix T_i is next to suffix T_j in the suffix array and shares a *lcp* of length ℓ, then suffix T_{i+1} will share a *lcp* of length $\ell - 1$ with suffix T_{j+1}. If suffix T_{i+1} is next to suffix T_k in the suffix array where $k \neq j + 1$, then the *lcp* between T_{i+1} and T_k is at least $\ell - 1$.

1. An array R is constructed such that if suffix T_i is in position j in the suffix array, then $R[i] = j$. This allows the algorithm to locate any suffix in constant time.

2. Suppose the lcp between T_i and its adjacent suffix in the suffix array is ℓ. Then use $R[i+1]$ to find the position of T_{i+1} in the suffix array, and start comparing T_{i+1} with its adjacent suffix in the suffix array from the $(\ell-1)$ th character.

3. Repeat Step 2 until the entire lcp array is constructed.

It clear that only $O(n)$ comparisons are needed because the lcp of the last suffix T_n is at most 1. One of the major reasons for the construction of lcp array is to speed up the problem of finding a pattern P in a string T of length n. To search for a pattern P in the string T, a binary search is performed using P and the suffix array of T. Unlike binary searches performed on integer arrays, each element in the suffix array corresponds to a suffix of the string T, therefore, $O(n)$ character comparisons are needed to decide whether P or a suffix is lexicographically greater. By using the lcp array, the number of character comparisons needed during the search can be reduced. Suppose P was compared to suffix T_i and has a lcp of ℓ, and P is going to be compared with T_j. The lcp array can be used to find the lcp between T_i and T_j, say k. If $k < \ell$ then we only need to compare the $(k+1)$ th characters of P and T_j; If $k > \ell$ then only the $(\ell+1)$-th characters of P and T_j are compared; Otherwise the comparison starts from $(\ell+1)$ th character. This algorithm takes $O(|P| + \log n)$ time, without any restriction on $|\Sigma|$ – the size of alphabet that T and P are drawn from.

However, the classic problem of finding a pattern P in a string T of length n can be solved in $O(|P|)$ time for fixed size $|\Sigma|$ using a suffix tree of T. Recently, Abouelhoda $et\ al.$ [2, 4] have improved the search time using suffix arrays to $O(|P|)$ time by using additional linear time preprocessing, thus making the suffix array based algorithm superior in terms of space requirement. In fact, many problems involving top-down or bottom-up traversal of suffix trees can now be solved with the same asymptotic run-time bounds using suffix arrays [1, 2, 4]. Such problems include many queries used in computational biology applications including finding exact matches, maximal repeats, tandem repeats, maximal unique matches and finding all

shortest unique substrings. For example, the whole genome alignment tool MUMmer [13] uses the computation of maximal unique matches.

While considerable advances are made in designing optimal algorithms for queries using suffix arrays and for computing auxiliary information that is required along with suffix arrays, the complexity of direct construction algorithms for suffix arrays remained $O(n \log n)$ so far. Several alternative algorithms for suffix array construction have been developed, each improving the previous best algorithm by an additional constant factor [23, 32]. The algorithm by Itoh and Tanaka is especially interesting. They divided all the characters of the string into two types:

- Type A characters are the characters of string T that are lexicographically greater than the next character, i.e. $T[i]$ is a type A character if $T[i] > T[i+1]$.

- Otherwise the character is said to be of type B.

The algorithm then sorts all the suffixes starting with a type B character; it also sorts all the suffixes begins with a type A character by their first characters. Finally the sorted order of all suffixes are obtained by using the order of all suffixes that starts with a type B character. Since 2003, many linear time direct suffix array construction algorithms were produced [21, 24, 26, 27, 29]. All of them divide the string into substrings, rank the substrings, and use the rankings to from a new string, then recursively process this new string.

Of all the linear time construction methods, Kärkkäinen and Sanders method is the easiest to understand.

1. Given a string T of length n, a three-letter substring (triplet) starting at i is constructed for each position i .

2. All the resulting triplets are sorted lexicographically.

3. Let i be the starting position of a triplet, then $i = k \mod 3$, $0 \leq k \leq 2$. A string T^k is formed by concatenating all the triplets whose starting positions are equal to $k \mod 3$ in the same order they appear in the original string.

4. A new string T' is constructed by concatenating T^1 and T^2, and is recursively sorted.

5. The suffix array of T' contains the lexicographical order of all suffixes T_i of T such that $i = 1 \bmod 3$ and $i = 2 \bmod 3$.

6. The sorted order of all suffixes T_j, $j = 0 \bmod 3$ can be calculated using a linear time sorting, using the first character of T_j and the lexicographical order of T_{j+1} in the suffix array of T'.

7. The suffix array of T' and the order of all suffixes T_j, $j = 0 \bmod 3$, can be merged in linear time. The order of two suffixes is determined by comparing either the first or the first two characters of the suffixes as needed and then comparing the ranks of corresponding suffixes in the suffix array of T'.

The above algorithm constructs the suffix array in $O(n)$ time. Hon *et al.* [21] provided the first linear time construction algorithm for compressed suffix array which uses only $O(n)$ bits.

1.2   Suffix Tree Layout in Secondary Storage

The suffix tree of a set of strings is a compacted trie of all suffixes of all the strings, an example of the suffix tree can be found in Figure 1.2. Since the introduction of this data structure by Weiner [38], several linear time algorithms for in-memory construction of suffix trees have been designed, notable ones include McCreight's linear space algorithm [34], Ukkonen's on-line algorithm [37], and Farach's algorithm for integer alphabets [14]. To extend the scale of data that can be handled in-memory, Grossi and Vitter [20] developed compressed suffix trees and suffix arrays.

Most of the suffix tree construction algorithms make use of suffix links. Let u be a node in the suffix tree, and the concatenation of all the edge labels from the root to u be $c\alpha$, where c is a character and α is the remainder, possibly even a zero length string. Let v be a node in the suffix tree, and the concatenation of all the edge labels from the root to v be α. A suffix link is a pointer from u to v (see Figure 1.2 for an example).
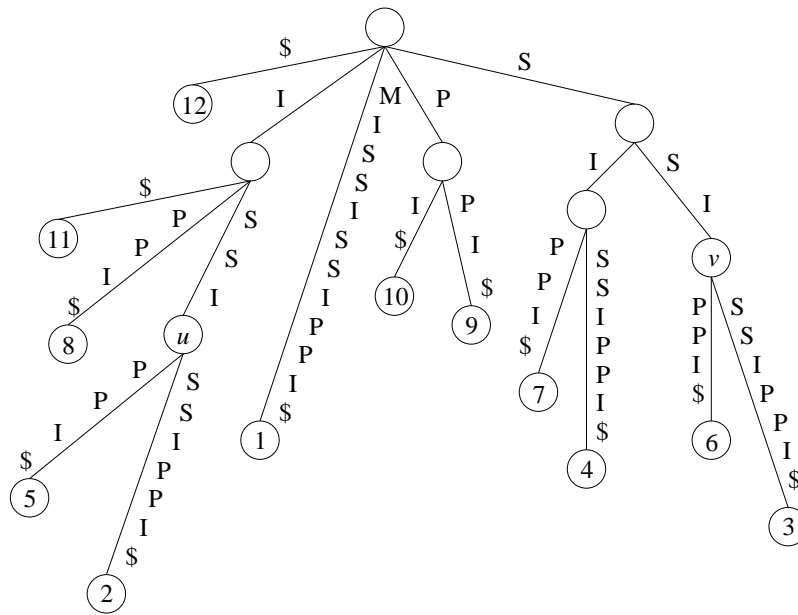
Figure 1.2 The suffix tree for the string "MISSISSIPPI$".

McCreight's linear time suffix tree construction algorithm [34] is perhaps one of the most popular and most implemented linear time algorithms. It constructs the final suffix tree by incrementally inserting the suffixes one by one. Let T be a string of size n.

1. The first suffix T_1 is inserted into an empty tree.

2. Suppose T_i was just inserted, and node u is the parent of the leaf node representing T_i. If u has an suffix link, then it is used. Otherwise, the suffix link stored in u's parent is used. Let v be the node pointed by the suffix link.

3. From v travel down by only comparing the first character of the edge label with the corresponding characters from suffix T_{i+1}, until the string depth is string depth of u minus one.

4. From this position compare every character of the edge label with corresponding characters from suffix T_{i+1}.
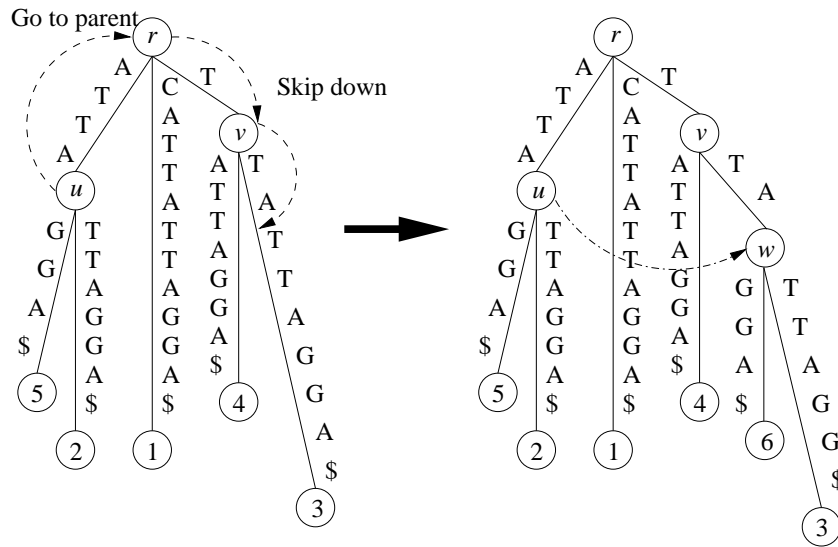
Figure 1.3   Inserting a new suffix in the McCreight's Algorithm.

5. Go downwards in the tree until either there is a mismatch in the edge, or the internal node does not have a child with the appropriate first character.

   - If there is a mismatch in the edge, a new node is inserted at that position and a new leaf representing suffix T_{i+1} is added. A suffix link is added from u to the newly created node.

   - If the internal node does not have a child with the appropriate first character a new leaf is attached to the internal node.

6. Steps 2 to 5 are repeated until all suffixes are inserted into the tree.

   Figure 1.3 shows the insertion of a new leaf during the construction of the suffix tree in McCreight's algorithm. It corresponds to Steps 2 through 5 in the above description.

   In the last few years, there has been significant research on disk-based storage of suffix trees for exploiting their utility on ever growing sequence databases. Many algorithms and strategies have been proposed to reduce the number of disk accesses during the construction of suffix trees [6, 7, 16, 22, 36]. Of these, only Farach *et al.* provided a construction algorithm for

secondary storage that achieves the optimal worst case bound of $\Theta\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$ disk accesses (where M is the size of main memory).

- In [6] the authors reduced the number of disk accesses by noticing that nodes near the root are accessed more often. Thus they provided a buffering strategy that tries to keep the nodes near the root in the main memory. Since the depth of a node is not always known, the authors chose to use the string depth of a node as a predictor and show that it works well for nodes near the root of the suffix tree.

- In [7] the authors use the fact that during suffix tree construction suffix links are used, so two nodes linked by a suffix link will likely be accessed consecutively. Therefore, they reduced the number of disk accesses by putting nodes that are linked by suffix links in the same disk page.

- In [22] the authors abandon the linear time construction algorithms, because the use of suffix links produces unpredictable disk access pattern. Instead the authors uses a naive construction algorithm of the suffix tree which runs in $O(n^2)$ time.

- In [36] the authors first partition all the suffixes using k characters. This decreases the number of suffixes need to be considered at once, thus reducing the space needed during the construction. The subtree that contains all the suffixes that share the same first k characters can than be constructed in memory. The authors also proposed memory management procedures to further reduce the number of disk accesses needed.

While these algorithms and techniques focused on suffix tree construction in secondary storage, the problems of searching and updating suffix trees (insertion/deletion of all suffixes of a string) on disks have not received as much attention. An interesting solution is provided by Clark and Munro [12] that achieves efficient space utilization. This is used to obtain a bound on disk accesses for substring search as a function of the height of the tree. In the worst case, the height of a suffix tree can be proportional to the length of the text indexed, although it is rarely the case and Clark and Munro's approach provides good experimental performance. To

date, algorithms with provably good worst-case performance for substring searches and updates for suffix trees in secondary storage are not known. To overcome these theoretical limitations, Ferragina and Grossi have proposed the string B-tree data structure [17, 18]. String B-trees provide the best known bounds for the worst-case number of disk access required for queries and updates. It was not known if the same performance bounds can be achieved with suffix trees, a problem that is resolved affirmatively in this thesis The unbalanced nature of suffix trees appears to be a major obstacle to designing efficient disk-based algorithms.

1.3   Self-adjusting String Data Structures

Like most indexing structures, suffix trees are built with the intention that they will be queried many times. Therefore, it is very important to devise algorithms that not only guarantee the worst case performance of a single query, but also provide good performance for a large sequence of queries collectively. In 1985, Sleator and Tarjan [35] created the self-adjusting binary tree by using a "splay" process, and proved that it produces the optimal number of disk accesses for a large sequence of queries. Since the publication of their ground breaking paper, the splaying technique has received wide attention.

However, the splaying process involves promoting a node in the tree to be the new root of the tree, and therefore is not suitable for suffix trees. Indeed, Sleator and Tarjan had left the development of a self-adjusting data-structure for text data as an open question. This open question has been partially resolved by Ciriani *et al.* [11], who provided a randomized algorithm that achieves the optimal number of disk accesses with high probability. Their method utilizes self-adjusting skip lists on top of a static suffix array. The problem of developing a deterministic algorithm for static and dynamic texts remained open.

1.4   Contributions of this dissertation

In this thesis, we presenting a direct linear time algorithm for constructing suffix arrays over integer alphabets [27, 29]. Contemporaneous to our result, Kärkkäinen *et al.* [24] and Kim *et al.* [26] also discovered suffix array construction algorithms with linear time complexity. All three

algorithms are very different and are important because they elucidate different properties of strings, which could well be applicable for solving other problems. An important distinguishing feature of our algorithm is that it uses only $8n$ bytes plus $1.25n$ bits for a fixed size alphabet. Our algorithm is based on a unique recursive formulation where the subproblem size is not fixed but is dependent on the properties of the string. Recently, Hon *et al.* [21] discovered a linear time construction algorithm for compressed suffix array.

It is well known that the suffix tree of a string can be constructed from the sorted order of its suffixes and the *lcp* array [15]. Because the *lcp* array can be inferred from the suffix array in linear time [25], our algorithm can also be used to construct suffix trees in linear time for large integer alphabets, and of course, for the special case of fixed size alphabets. Our algorithm is simpler and more space efficient than Farach's linear time algorithm for constructing suffix trees for integer alphabets. In fact, it is simpler than linear time suffix tree construction algorithms for fixed size alphabets [34, 37, 38]. A noteworthy feature of our algorithm is that it does not construct or use suffix links, resulting in additional space advantage.

Secondly, we propose a new suffix tree layout scheme, and present algorithms with provably good worst-case bounds on disk accesses required for search and update operations, while maintaining our layout [30]. Let n denote the number of leaves in the suffix tree, and B denote the size of a disk block. We provide algorithms that

- search for a pattern p in $O(|p|/B + \log_B n)$ disk accesses,
- insert (all suffixes of) a string of length m in $O(m \log_B(n + m))$ disk accesses, and
- delete (all suffixes of) a string of length m in $O(m \log_B n)$ disk accesses.

Since suffix tree construction can be achieved by starting from an empty tree and inserting strings one after another, the number of disk accesses needed for suffix tree construction is $O(n \log_B n)$. Our results provide the same worst-case performance as string B-trees, thus showing that suffix trees can be stored on disk and searched as efficiently as string B-trees.

Lastly, we resolve the open problem of developing a deterministic self-adjusting string data structure with optimal number of I/O accesses by designing a self-adjusting suffix tree layout that optimizes the total number of disk accesses for a sequence of queries. The main

difficulty is that while a number of valid alternative topologies exist for binary search trees, allowing continual adjustment suited to the flow of queries, the suffix tree topology is fixed and unbalanced to begin with. We overcome this limitation by proposing a layout scheme that creates a mapping of suffix tree nodes to disk blocks. While the tree topology remains fixed, the mapping of the layout can be adjusted to the sequence of queries, producing the desired performance bound. We begin with the layout scheme we proposed in [30], which balances the number of disk accesses required for any root to leaf path. Besides being optimal for a large[1] sequence of queries, our layout also have the following advantages:

1. We show that a "slow" moving promotion of the nodes works as well as the dramatic promotion to the root, which is a radical departure from existing self-adjusting algorithms and data structures, and can potentially be applied to develop more efficient "splaying" heuristics.

2. In practice, the self-adjusting data structures do not perform as well as balanced trees except in cases where a few of the leaves are accessed significantly more frequently than others [8, 39]. Our layout scheme is balanced in its initial state, thus combining the advantages of both types allows all the suffixes that share the same first k characters to be of data structures.

3. Besides suffix trees, our scheme can also be used for PATRICIA trees and potentially other data structures where the topology of the data structure cannot be altered.

4. Because of the topology of the suffix tree, our layout has the ability to reduce the number of disk accesses needed for a set of non-identical queries that share the same prefix.

The rest of the dissertation is organized as follows. In Chapter 2 we describe in detail our linear time suffix array construction algorithm for main memory and discuss briefly its implementation. In Chapter 3 we describe a layout scheme for suffix trees in secondary storage

---

[1]A sequence of queries is considered to be "large" if the number of queries is greater than the number of leaves in the suffix tree.

and analyze its performance in terms of number of disk accesses. We also present a self-adjusting variant of the layout scheme in Chapter 4, and we show that it is optimal for a large sequence of queries. Chapter 5 concludes the dissertation with some directions for future research.

CHAPTER 2.   LINEAR TIME SUFFIX ARRAY CONSTRUCTION

Consider a string $T = t_1 t_2 \ldots t_n$ over the alphabet $\Sigma = \{1 \ldots n\}$. Without loss of generality, assume the last character of T occurs nowhere else in T, and is the lexicographically smallest character. We denote this character by '\$'. Let $T_i = t_i t_{i+1} \ldots t_n$ denote the suffix of T starting with t_i. To store the suffix T_i, we only store the starting position number i. For strings α and β, we use $\alpha \prec \beta$ to denote that α is lexicographically smaller than β. Throughout this paper the term *sorted order* refers to lexicographically ascending order.

A high level overview of our algorithm is as follows: We classify the suffixes into two types, S and L. Suffix T_i is of type S if $T_i \prec T_{i+1}$, and is of type L if $T_{i+1} \prec T_i$. The last suffix T_n does not have a next suffix, and is classified as both type S and type L. The positions of the type S suffixes in T partitions the string into a set of substrings. We substitute each of these substrings by its rank among all the substrings and produce a new string T'. This new string is then recursively sorted. The suffix array of T' gives the lexicographic order of all type S suffixes. Then the lexicographic order of all suffixes can be deduced from this order.

We now present complete details of our algorithm. The following lemma allows easy identification of type S and type L suffixes in linear time.

Lemma 2.0.1 *All suffixes of T can be classified as either type S or type L in $O(n)$ time.*

Proof Consider a suffix T_i $(i < n)$.

  Case 1: If $t_i \neq t_{i+1}$, we only need to compare t_i and t_{i+1} to determine if T_i is of type S or type L.

  Case 2: If $t_i = t_{i+1}$, find the smallest $j > i$ such that $t_j \neq t_i$.

  if $t_j > t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are of type S.

```
T       M I  S  S  I  S  S  I  P  P  I  $
Type    L  S  L  L  S  L  L  S  L  L  L  L/S
Pos     1  2  3  4  5  6  7  8  9  10 11 12
```

Figure 2.1   The string "MISSISSIPPI$" and the types of its suffixes.

if $t_j < t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are of type L.

Thus, all suffixes can be classified using a left to right scan of T in $O(n)$ time.   □

The type of each suffix of the string MISSISSIPPI$ is shown in Figure 2.1. An important property of type S and type L suffixes is, if a type S suffix and a type L suffix both begin with the same character, the type S suffix is always lexicographically greater than the type L suffix. The formal proof is presented below.

Lemma 2.0.2 *A type S suffix is lexicographically greater than a type L suffix that begins with the same first character.*

Proof Suppose a type S suffix T_i and a type L suffix T_j are two suffixes that start with the same character c. We can write $T_i = c^k c_1 \alpha$ and $T_j = c^l c_2 \beta$, where c^k and c^l denotes the character c repeated for $k, l > 0$ times, respectively; $c_1 > c$, $c_2 < c$; α and β are (possibly empty) strings.

Case 1: If $k < l$ then c_1 is compared to a character c in c^l. Then $c_1 > c \Rightarrow T_j \prec T_i$.

Case 2: If $k > l$ then c_2 is compared to a character c in c^k. Then $c > c_2 \Rightarrow T_j \prec T_i$.

Case 3: If $k = l$ then c_1 is compared to c_2. Since $c_1 > c$ and $c > c_2$, then $c_1 > c_2 \Rightarrow T_j \prec T_i$.

Thus a type S suffix is lexicographically greater than a type L suffix that begins with the same first character.   □

Corollary 2.0.3 *In the suffix array of T, among all suffixes that start with the same character, the type S suffixes appear after the type L suffixes.*

Proof Follows directly from Lemma 2.0.2. □

Let A be an array containing all suffixes of T, not necessarily in sorted order. Let B be an array of all suffixes of type S, sorted in lexicographic order. Using B, we can compute the lexicographically sorted order of all suffixes of T as follows:

1. Bucket all suffixes of T according to their first character in array A. Each bucket consists of all suffixes that start with the same character. This step takes $O(n)$ time.

2. Scan B from right to left. For each suffix encountered in the scan, move the suffix to the current end of its bucket in A, and advance the current end by one position to the left. More specifically, the move of a suffix in array A to a new position should be taken as swapping the suffix with the suffix currently occupying the new position. After the scan of B is completed, by Corollary 2.0.3, all type S suffixes are in their correct positions in A. The time taken is $O(|B|)$, which is bounded by $O(n)$.

3. Scan A from left to right. For each entry $A[i]$, if $T_{A[i]-1}$ is a type L suffix, move it to the current front of its bucket in A, and advance the front of the bucket by one. This takes $O(n)$ time. At the end of this step, A contains all suffixes of T in sorted order.

In Figure 2.2, the suffix pointed by the arrow is moved to the current front of its bucket when the scan reaches the suffix at the origin of the arrow. The following lemma proves the correctness of the procedure in Step 3.

Lemma 2.0.4 *In step 3, when the scan reaches $A[i]$, then suffix $T_{A[i]}$ is already in its sorted position in A.*

Proof By induction on i. To begin with, the smallest suffix in T must be of type S and hence in its correct position $A[1]$. By inductive hypothesis, assume that $A[1], A[2], \ldots, A[i]$ are the first i suffixes in sorted order. We now show that when the scan reaches $A[i+1]$, then the suffix in it, i.e., $T_{A[i+1]}$ is already in its sorted position. Suppose not. Then there exists a suffix referenced by $A[k]$ ($k > i+1$) that should be in $A[i+1]$ in sorted order, i.e., $T_{A[k]} \prec T_{A[i+1]}$.

T		M	I	S	S	I	S	S	I	P	P	I	$
Type			S			S			S				S
Pos		1	2	3	4	5	6	7	8	9	10	11	12

Order Of Type S suffixes

12	8	5	2

Bucket: $ I M P S

After Step 2

12	11	8	5	2	1	9	10	3	4	6	7

Sorted Order

12	11	8	5	2	1	10	9	7	4	6	3
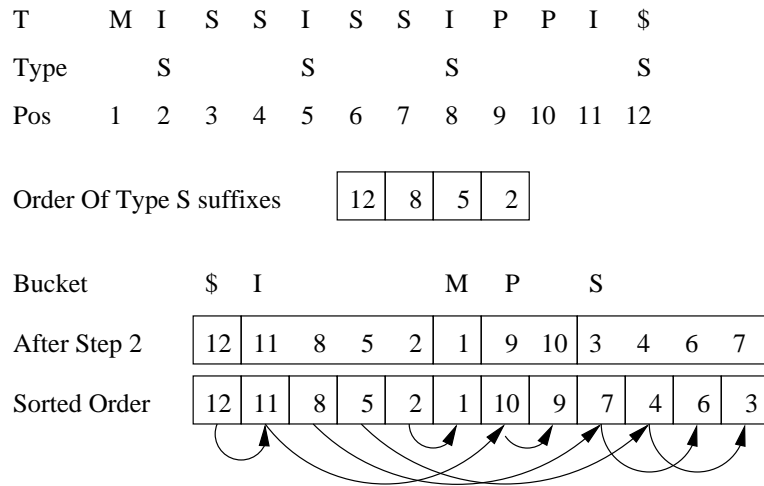
Figure 2.2   Illustration of how to obtain the sorted order of all suffixes, from the sorted order of type S suffixes of the string MISSISSIPPI$.

As all type S suffixes are already in correct positions, both $T_{A[k]}$ and $T_{A[i+1]}$ must be of type L. Because A is bucketed by the first character of the suffixes prior to step 3, and a suffix is never moved out of its bucket, $T_{A[k]}$ and $T_{A[i+1]}$ must begin with the same character, say c. Let $T_{A[i+1]} = c\alpha$ and $T_{A[k]} = c\beta$. Since $T_{A[k]}$ is type L, $\beta \prec T_{A[k]}$. From $T_{A[k]} \prec T_{A[i+1]}$, $\beta \prec \alpha$. Since $\beta \prec T_{A[k]}$, and the correct sorted position of $T_{A[k]}$ is $A[i+1]$, β must occur in $A[1] \ldots A[i]$. Because $\beta \prec \alpha$, $T_{A[k]}$ should have been moved to the current front of its bucket before $T_{A[i+1]}$. Thus, $T_{A[k]}$ can not occur to the right of $T_{A[i+1]}$, a contradiction.   □

So far, we showed that if all type S suffixes are sorted, then the sorted position of all suffixes of T can be determined in $O(n)$ time. In a similar manner, the sorted position of all suffixes of T can also be determined from the sorted order of all suffixes of type L. To do this, we bucket all suffixes of T based on their first characters into an array A. We then scan the sorted order of type L suffixes from left to right and determine their correct positions in A by moving them to the current front of their respective buckets. We then can A from right to left and when $A[i]$ is encountered, if $T_{A[i]-1}$ is of type S, it will be moved to the current en of its bucket.

Once the suffixes of T are classified into type S and type L, we choose to sort those type

of suffixes which are fewer in number. Without loss of generality, assume that type S suffixes are fewer. We now show how to recursively sort these suffixes.

Define position i of T to be a type S position if the suffix T_i is of type S, and similarly to be a type L position if the suffix T_i is of type L. The substring $t_i \dots t_j$ is called a type S substring if both i and j are type S positions, and every position between i and j is a type L position.

Our goal is to sort all the type S suffixes in T. To do this we first sort all the type S substrings. The sorting generates buckets where all the substrings in a bucket are identical. The buckets are numbered using consecutive integers starting from 1. We then generate a new string T' as follows: Scan T from left to right and for each type S position in T, write the bucket number of the type S substring starting from that position. This string of bucket numbers forms T'. Observe that each type S suffix in T naturally corresponds to a suffix in the new string T'. In Lemma 2.0.5, we prove that sorting all type S suffixes of T is equivalent to sorting all suffixes of T'. We sort T' recursively.

We first show how to sort all the type S substrings in $O(n)$ time. Consider the array A, consisting of all suffixes of T bucketed according to their first characters. For each suffix T_i, define its *S-distance* to be the distance from its starting position i to the nearest type S position to its left (excluding position i). If no type S position exists to the left, the *S-distance* is defined to be 0. Thus, for each suffix starting on or before the first type S position in T, its *S-distance* is 0. The type S substrings are sorted as follows (illustrated in Figure 2.3):

1. For each suffix in A, determine its *S-distance*. This is done by scanning T from left to right, keeping track of the distance from the current position to the nearest type S position to the left. While at position i, the *S-distance* of T_i is known and this distance is recorded in array $Dist$. The *S-distance* of T_i is stored in $Dist[i]$. Hence, the *S-distances* for all suffixes can be recorded in linear time.

2. Let m be the largest *S-distance*. Create m lists such that list j $(1 \le j \le m)$ contains all the suffixes with an *S-distance* of j, listed in the order in which they appear in array A.

```
T        M  I  S  S  I  S  S  I  P  P  I  $        Step 3. Sort all type S substrings
Type        S        S        S        S          Original
Pos      1  2  3  4  5  6  7  8  9 10 11 12             ┌──┬─────────┐
A       [12│ 2  5  8 11│ 1│ 9 10│ 3  4  6  7]          │12│ 2  5  8 │
                                                        └──┴─────────┘
Step 1. Record the S−distances                    Sort according to list 1
Pos      1  2  3  4  5  6  7  8  9 10 11 12             ┌──┬─────────┐
Dist     0  0  1  2  3  1  2  3  1  2  3  4             │12│ 8  2  5 │
                                                        └──┴─────────┘
Step 2. Construct S−distance Lists                Sort according to list 2
        1   [9 │ 3  6]                                  ┌──┬─────────┐
                                                        │12│ 8  2  5 │
        2  [10 │ 4  7]                                  └──┴─────────┘
                                                   Sort according to list 3
        3   [5 │ 8 11]                                  ┌──┬─────────┐
                                                        │12│ 8  2  5 │
        4  [12]                                         └──┴─────────┘
                                                   Sort according to list 4
                                                        ┌──┬─────────┐
                                                        │12│ 8  2  5 │
                                                        └──┴─────────┘
```
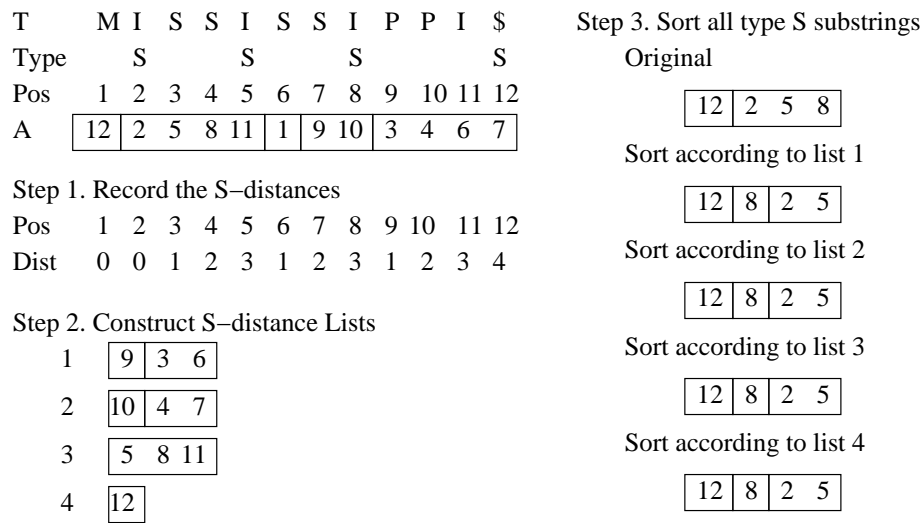
Figure 2.3   Illustration of the sorting of type S substrings of the string MISSISSIPPI$.

This can be done by scanning A from left to right in linear time, referring to $Dist[A[i]]$ to put $T_{A[i]}$ in the correct list.

3. We now sort the type S substrings using the lists created above. The sorting is done by repeated bucketing using one character at a time. To begin with, the bucketing based on first character is determined by the order in which type S suffixes appear in array A. Suppose the type S substrings are bucketed according to their first $j-1$ characters. To extend this to j characters, we scan list j. For each suffix T_i encountered in the scan of a bucket of list j, move the type S substring starting at t_{i-j} to the current front of its bucket, then move the current front to the right by one. After a bucket of list j is scanned, new bucket boundaries need to be drawn between all the type S substrings that have been moved, and the type S substrings that have not been moved. Because the total size of all the lists is $O(n)$, the sorting of type S substrings only takes $O(n)$ time.

The sorting of type S substrings using the above algorithm respects lexicographic ordering of type S substrings, with the following important exception: If a type S substring is the prefix of another type S substring, the bucket number assigned to the shorter substring will

be larger than the bucket number assigned to the larger substring. This anomaly is designed on purpose, and is exploited later in Lemma 2.0.5.

As mentioned before, we now construct a new string T' corresponding to all type S substrings in T. Each type S substring is replaced by its bucket number and T' is the sequence of bucket numbers in the order in which the type S substrings appear in T. Because every type S suffix in T starts with a type S substring, there is a natural one-to-one correspondence between type S suffixes of T and all suffixes of T'. Let T_i be a suffix of T and $T'_{i'}$ be its corresponding suffix in T'. Note that $T'_{i'}$ can be obtained from T_i by replacing every type S substring in T_i with its corresponding bucket number. Similarly, T_i can be obtained from $T'_{i'}$ by replacing each bucket number with the corresponding substring and removing the duplicate instance of the common character shared by two consecutive type S substrings. This is because the last character of a type S substring is also the first character of the next type S substring along T.

Lemma 2.0.5 *Let T_i and T_j be two suffixes of T and let $T'_{i'}$ and $T'_{j'}$ be the corresponding suffixes of T'. Then, $T_i \prec T_j \Leftrightarrow T'_{i'} \prec T'_{j'}$.*

Proof We first show that $T'_{i'} \prec T'_{j'} \Rightarrow T_i \prec T_j$. The prefixes of T_i and T_j corresponding to the longest common prefix of $T'_{i'}$ and $T'_{j'}$ must be identical. This is because if two bucket numbers are the same, then the corresponding substrings must be the same. Consider the leftmost position in which $T'_{i'}$ and $T'_{j'}$ differ. Such a position exists and the characters (bucket numbers) of $T'_{i'}$ and $T'_{j'}$ in that position determine which of $T'_{i'}$ and $T'_{j'}$ is lexicographically smaller. Let k be the bucket number in $T'_{i'}$ and l be the bucket number in $T'_{j'}$ at that position. Since $T'_{i'} \prec T'_{j'}$, it is clear that $k < l$. Let α be the substring corresponding to k and β be the substring corresponding to l. Note that α and β can be of different lengths, but α cannot be a proper prefix of β. This is because the bucket number corresponding to the prefix must be larger, but we know that $k < l$.

Case 1: β is not a prefix of α. In this case, $k < l \Rightarrow \alpha \prec \beta$, which implies $T_i \prec T_j$.

Case 2: β is a proper prefix of α. Let the last character of β be c. The corresponding position in T is a type S position. The position of the corresponding c in α must be a type L

position.

Since the two suffixes that begin at these positions start with the same character, by Corollary 2.0.3, the type L suffix must be lexicographically smaller then the type S suffix. Thus, $T_i \prec T_j$.

From the one-to-one correspondence between the suffixes of T' and the type S suffixes of T, it also follows that $T_i \prec T_j \Rightarrow T'_{i'} \prec T'_{j'}$. $\qquad \square$

Corollary 2.0.6 *The sorted order of the suffixes of T' determines the sorted order of the type S suffixes of T.*

Proof Let $T'_{i'_1}, T'_{i'_2}, T'_{i'_3}, \ldots$ be the sorted order of suffixes of T'. Let $T_{i_1}, T_{i_2}, T_{i_3}, \ldots$ be the sequence obtained by replacing each suffix $T'_{i'_k}$ with the corresponding type S suffix T_{i_k}. Then, $T_{i_1}, T_{i_2}, T_{i_3}, \ldots$ is the sorted order of type S suffixes of T. The proof follows directly from Lemma 2.0.5. $\qquad \square$

Hence, the problem of sorting the type S suffixes of T reduces to the problem of sorting all suffixes of T'. Note that the characters of T' are consecutive integers starting from 1. Hence our suffix sorting algorithm can be recursively applied to T'.

If the string T has fewer type L suffixes than type S suffixes, the type L suffixes are sorted using a similar procedure − Call the substring $t_i, \ldots, t_j$ a type L substring if both i and j are type L positions, and every position between i and j is a type S position. Now sort all the type L substrings and construct the corresponding string T' obtained by replacing each type L substring with its bucket number. Sorting T' gives the sorted order of type L suffixes.

Thus, the problem of sorting the suffixes of a string T of length n can be reduced to the problem of sorting the suffixes of a string T' of size at most $\lceil \frac{n}{2} \rceil$, and $O(n)$ additional work. This leads to the recurrence

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

Theorem 2.0.7 *The suffixes of a string of length n can be lexicographically sorted in $O(n)$ time and space.*

2.1 Space Requirement

We now consider the space requirement of our suffix array construction algorithm. The algorithm can be decomposed into the following parts:

1. Classifying the types of all suffixes.

2. Sorting all suffixes according to their first character.

3. Constructing m lists according to the *S-distance* of each suffix, and the sorted order of their first character.

4. Sorting all type S substrings by repeated bucketing using the m lists.

5. Constructing a new string T' according to the bucket numbers of type S substrings.

6. Recursively applying our algorithm, and obtaining the sorted order of type S suffixes.

7. Constructing the suffix array from the sorted order of all type S suffixes.

Except for Step 4, the calculation of space requirement for each of the steps listed above is straightforward, and offers little room for improvement by using a more efficient implementation. Therefore we limit the focus of our analysis to efficient implementation of Step 4.

As mentioned previously, the sorting of all type S substrings is done by repeated bucketing using one character at a time. Suppose the type S substrings are bucketed according to their first $j - 1$ characters. To extend this to j characters, we scan list j. For each suffix T_i encountered, move the type S substring starting at t_{i-j} to the current front of its bucket and advance the current front by one.

In Manber and Myers' algorithm [33], the suffixes are also moved to the front of their respective buckets in each iteration. However, their space-efficient scheme does not apply to our algorithm because every suffix will be moved at most once in each iteration of their algorithm. On the other hand, a type S substring may be moved multiple times in each

recursion step of our algorithm. In order to achieve $O(n)$ runtime, we must be able to locate the current front of the bucket containing a given type S substring in constant time.

Let array C be an array containing all type S substrings, bucketed according to their first characters. A type S substring is denoted by its starting position in T. Array C can be generated by copying from array A computed in Step 2. Let R be an array of size n, such that if $C[i] = j$, then $R[j] = k$ where k is the position of the end of the bucket containing j. R can be constructed by a right to left scan of C. Let $lptr$ be an array of the same size as C, such that if i is the last position of a bucket in C, then $lptr[i] = j$ where j is the current front of that bucket. For all other positions k, $lptr[k] = -1$.

Each of the m lists is itself bucketed according to the first character of the suffixes. As previously mentioned, for each suffix T_i encountered in the scan of a bucket in list j, type S substring starting at t_{i-j} is moved to the current front of its bucket. The bucket containing t_{i-j} can be found by referring to $R[i-j]$, and the current front of its bucket can then be found by referring to $lptr[R[i-j]]$. The current front is advanced by incrementing $lptr[R[i-j]]$. Note that the effect of moving a type S substring starting at t_{i-j} is achieved by adjusting the values of $R[i-j]$ and $lptr[R[i-j]]$ instead of actually moving it in C.

After scanning an entire bucket of list j, all the elements of C that have been moved should be in a new bucket in front of their old bucket. To accomplish this, we note that the $lptr$ at the end of each old bucket in C is pointing to the current front of the old bucket, which is immediately next to the last element of the new bucket. Thus the bucket of list j is scanned again. For suffix T_i encountered in the scan, type S substring starting at t_{i-j} is moved into the new bucket by first setting $R[i-j] = lptr[R[i-j]] - 1$, then we set $lptr[R[i-j]] = R[i-j]$ if $lptr[R[i-j]] = -1$ or decrement $lptr[R[i-j]]$ by one otherwise.

It is easy to see that all the values of R and $lptr$ are set correctly at the end of the second scan. The amount of work done in this step is proportional to the size of all the m lists, which is $O(n)$. Two integer arrays of size n and two integer arrays of size at most $\lceil \frac{n}{2} \rceil$ are used. Assuming each integer representation takes 4 bytes of space, the total space used in this step is $12n$ bytes. Note that it is not necessary to actually move the type S substrings in C as the

final positions of type S substrings after sorting can be deduced from R. In fact, we construct T' directly using R. Array C is only needed to initialize R and $lptr$. We can initialize R from C, then discard C, and initialize $lptr$ from R, thus further reducing the space usage to $10n$ bytes. However, this reduction is not necessary as construction of m lists in Step 3 requires $12n$ bytes, making it the most space-expensive step of the algorithm.

To construct the m lists, we use a stable counting sort on A using the S-$distance$ as the key. The total amount of space used in this part of the algorithm is 3 integer arrays - one for A, one for the m lists, and a temporary array. The fact that we discard almost all arrays before the next recursion step of our algorithm except the strings, and that each subsequent step uses only half the space used in the previous step, makes the construction of the m lists in the first iteration the most space consuming stage of our algorithm.

It is possible to derive an implementation of our algorithm that uses only three integer arrays of size n and three boolean arrays[1] (two of size n and one of size $\lceil \frac{n}{2} \rceil$). The space requirement of our algorithm is $12n$ bytes plus $\frac{3}{2}n$ bits. This compares favorably with the best space-efficient implementations of linear time suffix tree construction algorithms, which still require $20n$ bytes [4]. Hence, direct linear time construction of suffix arrays using our algorithm is more space-efficient.

In case the alphabet size is constant, it is possible to further reduce the space requirement by eliminating the calculation of the m lists in the first iteration. This is possible because the type S substrings can be sorted character by character as individual strings in $O(n)$ time if the alphabet size is constant. This reduces the space required to only $8n$ bytes plus $0.5n$ bits for the first iteration. Note that this idea cannot be used in subsequent iterations because the string T' to be worked on in the subsequent iterations will still be based on integer alphabet. So we resort to the traditional implementation for this and all subsequent iterations. As a result, the space requirement for the complete execution of the algorithm can be reduced to $8n$ bytes plus $1.25n$ bits. This is competitive with Manber and Myers' $O(n \log n)$ time algorithm for suffix array construction [33], which requires only $8n$ bytes. In many practical applications,

---

[1]The boolean arrays are used to mark bucket boundaries, and to denote the type of each suffix.

the size of the alphabet is a small constant. For instance, computational biology applications deal with DNA and protein sequences, which have alphabet sizes of 4 and 20, respectively.

2.2   Reducing the Size of T'

In this section, we present an implementation strategy to further reduce the size of T'. Consider the result of sorting all type S substrings of T. Note that a type S substring is a prefix of the corresponding type S suffix. Thus, sorting type S substrings is equivalent to bucketing type S suffixes based on their respective type S substring prefixes. The bucketing conforms to the lexicographic ordering of type S suffixes. The purpose of forming T' and sorting its suffixes is to determine the sorted order of type S suffixes that fall into the same bucket. If a bucket contains only one type S substring, the position of the corresponding type S suffix in the sorted order is already known.

Let $T' = b_1 b_2 \ldots b_m$. Consider a maximal substring $b_i \ldots b_j$ ($j < m$) such that each b_k ($i \leq k \leq j$) contains only one type S substring. We can shorten T' by replacing each such maximal substring $b_i \ldots b_j$ with its first character b_i. Since $j < m$ the bucket number corresponding to '$' is never dropped, and this is needed for subsequent iterations. It is easy to directly compute the shortened version of T', instead of first computing T' and then shortening it. Shortening T' will have the effect of eliminating some of the suffixes of T', and also modifying each suffix that contains a substring that is shortened. We already noted that the final positions of the eliminated suffixes are already known. It remains to be shown that the sorted order of other suffixes are not affected by the shortening.

Consider any two suffixes $T'_k = b_k \ldots b_m$ and $T'_l = b_l \ldots b_m$, such that at least one of the suffixes contains a substring that is shortened. Let $j \geq 0$ be the smallest integer such that either b_{k+j} or b_{l+j} (or both) is the beginning of a shortened substring. The first character of a shortened substring corresponds to a bucket containing only one type S substring. Hence, the bucket number occurs nowhere else in T'. Therefore $b_{k+j} \neq b_{l+j}$, and the sorted order of $b_k \ldots b_m$ and $b_l \ldots b_m$ is determined by the sorted order of $b_k \ldots b_{k+j}$ and $b_l \ldots b_{l+j}$. In other words, the comparison of any two suffixes never extends beyond the first character of a

Table 2.1   Algorithms and their descriptions.

Name	Description
Manber and Myers	Manber and Myers' original algorithm [33].
Sadakane	Larsson and Sadakane's algorithm [32].
Two-stage suffix sort	Itoh and Tanaka's two-stage suffix sorting algorithm [23].
Multikey Quicksort	Sorting suffixes as individual strings using ternary Quicksort [10].
Our algorithm	The algorithm presented in this paper.

shortened substring.

2.3   Related Work

In this section we compare our algorithm with some of the other suffix array construction algorithms. Since the introduction of suffix array by Manber and Myers [33], several algorithms for suffix array construction have been developed. Some of these algorithms are aimed at reducing the space usage, while others are aimed at reducing the runtime. Table 2.3 contains the names and descriptions of the algorithms used in our comparison. Table 2.3 lists the space requirement, time complexity, and restrictions on alphabet size. It is immediately clear that space is sacrificed for better time complexity. We also note that for the case of constant size alphabet, our algorithm has a better runtime, while maintaining similar memory usage compared to algorithms by Manber and Myers [33], and Larsson and Sadakane [32]. Kurtz [31] has developed a space-efficient way of constructing and storing suffix trees. Although on average it only uses 10.1 bytes per input character, it has a worst case of 20 bytes per input character. Indeed, some of the techniques used in his implementation can be applied to our algorithm as well, and this will lead to further reduction of space in practice.

Note that we have not included a comparison of the space required by other linear time algorithms [24, 26] in Table 2.3. To achieve optimal space usage for our algorithm, it is very important that implementation techniques outlined in Section 3 are properly utilized. Due to the recent discovery of these results, a thorough space analysis of the other two linear time

Table 2.2   Comparison of different algorithms.

Algorithm	Space (bytes)	Time Complexity	Alphabet Size
Manber and Myers	$8n$	$O(n \log n)$	Arbitrary
Sadakane	$8n$	$O(n \log n)$	Arbitrary
Two-stage suffix sort	$4n$	$O(n^2)$	$1 \ldots n$
Two-stage suffix sort	$4n$	$O(n^2 \log n)$	Arbitrary
Multikey Quicksort	$4n$	$O(n^2 \log n)$	Arbitrary
Our algorithm	$12n$	$O(n)$	$1 \ldots n$
Our algorithm	$8n$	$O(n)$	Constant

algorithms is not yet available in the published literature. Our analysis indicates that our space requirement would be lower than the space required by Park *et al.*'s algorithm [26] and is the same as the space required for Kärkkäinen and Sanders' algorithm [24]. All three algorithms depend on recursively reducing the problem size - to half the original size for Park *et al.*'s algorithm, to two thirds of the original size for Kärkkäinen and Sanders' algorithm, and to at most half the original size for our algorithm. The worst-case of reducing the problem size to only half will be realized when the number of type S and type L suffixes are the same. This, coupled with the reduction technique presented in Section 4, will significantly reduce the number of levels of recursion required. For example, in an experiment to build a suffix array on the genome of *E. Coli* which is approximately 4 million base pairs (characters) long, we found the number of levels of recursion required is only 8 compared to the 22 that would be required by recursively halving.

CHAPTER 3.   SUFFIX TREE DISK LAYOUT

Consider a set S of strings with total length n. Without loss of generality, we assume that all characters are drawn from the alphabet Σ, except for the last character each string, which a special characters $\$ \notin \Sigma$. Let $s \in S$ be a string of length m. We use $s[i]$ to denote the i-th character of s, where $1 \leq i \leq m$, and let $s[i..j]$ denote the substring $s[i]s[i+1]\ldots s[j]$, where $1 \leq i < j \leq m$. The i-th suffix of s, $s[i..m]$, is denoted by s_i. The suffix tree of the set of strings S, abbreviated $ST(S)$, is a compacted trie of all suffixes of all strings in S. The term *generalized suffix tree* is commonly used when dealing with multiple strings, while the term *suffix tree* is usually used when dealing with a single string. For convenience, we use the term suffix tree to denote either case.

3.1   Static Layout

For a node v in $ST(S)$, the string depth of v is the total length of all edge labels on the path from the root to v. The size of a subtree whose root is node v is the total number of leaves in that subtree, and is referred to as $size(v)$. If v is a leaf node then $size(v) = 1$. The *rank* of a node v, denoted $rank(v)$, is i if and only if $C^i \leq size(v) < C^{i+1}$, for some integer constant $C > 1$ of choice. Nodes u and v belong to the same partition if all nodes on the undirected path between u and v have the same $rank$. It is easy to see that the entire suffix tree is partitioned into disjoint parts. Figure 3.1(a) shows an example of a suffix tree, and one of its partitions.

The rank of a partition $\mathcal{P}$ is the same as the rank of the nodes in $\mathcal{P}$, i.e. $rank(\mathcal{P}) = rank(v)$ for any v in $\mathcal{P}$. Node v in $\mathcal{P}$ is a leaf of $\mathcal{P}$ if and only if none of v's children in $ST(S)$ is in $\mathcal{P}$. Node u is termed the root of $\mathcal{P}$ if and only if u's parent is not a node in $\mathcal{P}$. Nodes with

Figure 3.1 (a) The suffix tree of the string CATTATTAGGA$. The nodes of an example partition are drawn as concentric circles. Braching nodes of the partition are labeled as u, v, w. (b) The skeleton partition tree of the example partition.

concentric circles in Figure 3.1(a) are belong to an example partition. Node u in $\mathcal{P}$ is called a branching node if two or more of its children in $ST(S)$ are also in $\mathcal{P}$. All other nodes are referred to as non-branching nodes. From Figure 3.1(a) we see that a partition need not be a compacted trie. For each partition $\mathcal{P}$, a compacted trie is constructed containing the root of $\mathcal{P}$ (which may not be a branching node), all the branching nodes and all the leaves. This resulting compacted trie is referred to as the skeleton partition tree of $\mathcal{P}$, or $\mathcal{T}_\mathcal{P}$, Figure 3.1(b) show the corresponding skeleton partition tree of the partition in Figure 3.1(a).

While the number of nodes in any partition $\mathcal{P}$ could be as large as $C^{rank(\mathcal{P})+1} - C^{rank(\mathcal{P})}$, which depends on $rank(\mathcal{P})$, the size of its skeleton partition tree $\mathcal{T}_\mathcal{P}$ is $O(C)$ independent of $rank(\mathcal{P})$. First we show that the number of leaves of the partition is limited, then we used the fact that the skeleton partition tree is a compacted tries to show that the number of nodes in it is bounded by C rather than $rank(\mathcal{P})$.

Lemma 3.1.1 *There are at most $C - 1$ leaves in a partition.*

Proof Let $\mathcal{P}$ be a partition that has $C' \geq C$ leaves, and node u be its root. Since $size(u) \geq C^i \cdot C' \geq C^i \cdot C = C^{i+1}$, $rank(u) > rank(\mathcal{P})$, a contradiction. $\qquad\square$
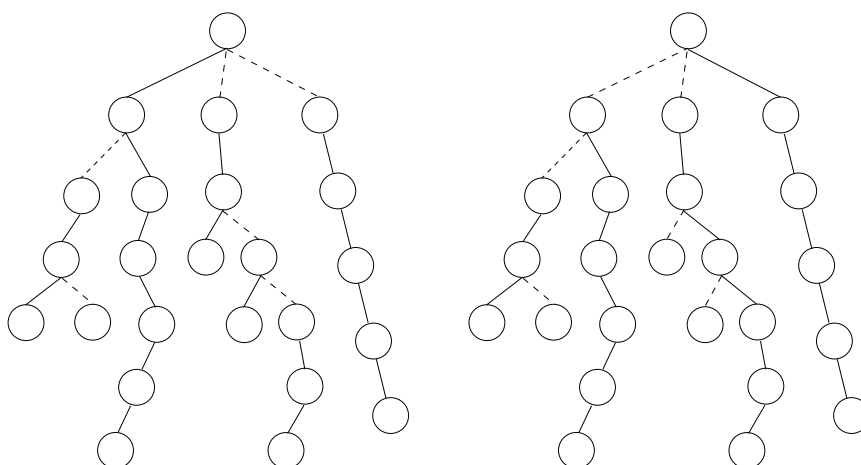
Figure 3.2   Two valid decompositions of the same partition. In each figure, solid edge between two nodes represents that they belong to the same *component*, while dashed lines means they belong to two different components.

Lemma 3.1.2 *For a partition $\mathcal{P}$, the number of nodes in $\mathcal{T}_\mathcal{P}$ is at most $2C - 2$.*

Proof By Lemma 3.1.1 there are at most $C - 1$ leaf nodes in a skeleton partition tree. Therefore, there can be at most $C - 2$ branching nodes. In addition, the root node may not be a branching node. So the total number of nodes in a skeleton partition tree is at most $2C - 2 = O(C)$. $\square$

Since the size of the skeleton partition tree of a partition $\mathcal{P}$ is independent of $rank(\mathcal{P})$, an appropriate C can be chosen so $\mathcal{T}_\mathcal{P}$ can be stored in one or a constant number of disk blocks. However, the number of nodes in $\mathcal{P}$ may be large and cannot be stored in one single disk block. Therefore, it is decomposed into multiple components and each of the component can be stored in segments across multiple disk blocks.

A *component* of a partition $\mathcal{P}$ consists of all the nodes on a path between a node v and a leaf of the partition. A valid decomposition of a partition $\mathcal{P}$ is a partitioning of all $\mathcal{P}$'s nodes into disjoint components. For any given partition, there could be many valid decompositions. Figure 3.2 shows two valid decompositions of the same partition.

By the definition of a partition $\mathcal{P}$, and its skeleton partition tree $\mathcal{T}_\mathcal{P}$. Every node $u \in \mathcal{T}_\mathcal{P}$ has a corresponding copy of itself in a component of $\mathcal{P}$. We refer to the copy of u in $\mathcal{T}_\mathcal{P}$ be $u_{\mathcal{T}_\mathcal{P}}$, and the copy of u in one of the components $\mathcal{P}$ as $u_\mathcal{P}$. For a non-branching node v that is neither the root nor a leaf of its partition, there is only one copy of v in one of the components of $\mathcal{P}$, we denote it as $v_\mathcal{P}$ to be consistent. Note that if two partitions $\mathcal{P}$ and $\mathcal{Q}$ are being mentioned simultaneously, then nodes from $\mathcal{P}$ will be denoted as $u_{\mathcal{T}_\mathcal{P}}$ or $u_\mathcal{P}$, while nodes from $\mathcal{Q}$ will be denoted as $u_{\mathcal{T}_\mathcal{Q}}$ or $u_\mathcal{Q}$. Since for every node $u_\mathcal{P}$ it is also a node in the conceptual suffix tree $ST(S)$, we use u to denote the conceptual node u in the suffix tree. For example, the children of u refers to all the children of the node u in the suffix tree, regardless whether they are in the same partition. This notation is also applicable for all nodes $u_{\mathcal{T}_\mathcal{P}}$.

In order to facilitate various tree operation, the following information is store in each node $v_{\mathcal{T}_\mathcal{P}}$ of the skeleton partition tree:

1. The string depth of $v_{\mathcal{T}_\mathcal{P}}$.

2. A pointer to the parent of $v_{\mathcal{T}_\mathcal{P}}$ in $\mathcal{T}_\mathcal{P}$. If $v_{\mathcal{T}_\mathcal{P}}$ is the root of the partition, then it points to the parent of $v_\mathcal{P}$, $u_\mathcal{Q}$[1].

3. A pointer to $v_\mathcal{P}$ in one of the components of $\mathcal{P}$, i.e. the copy of itself in a component of $\mathcal{P}$.

4. A representative suffix s_i if $v_{\mathcal{T}_\mathcal{P}}$ is a leaf of the partition, where s_i is a suffix represented by one of the leaves in the suffix tree under v.

5. For each child $u_{\mathcal{T}_\mathcal{P}}$ of $v_{\mathcal{T}_\mathcal{P}}$ in $\mathcal{T}_\mathcal{P}$ store in $v_{\mathcal{T}_\mathcal{P}}$

   (a) A pointer to $u_{\mathcal{T}_\mathcal{P}}$.

   (b) The leading character of the first edge label on the path from v to u.

   (c) A pointer to $w_\mathcal{P}$, where $w_\mathcal{P}$ is the child of $v_\mathcal{P}$ and the ancestor of $u_\mathcal{P}$, i.e. $w_\mathcal{P}$ is the first node on the path from $v_\mathcal{P}$ to $u_\mathcal{P}$.

---

[1]Note that if $v_{\mathcal{T}_\mathcal{P}}$ is the root of the partition, then its parent pointer points to a location on disk where the actually location of the parent $u_\mathcal{Q}$ is stored. This way when the physical location of $u_\mathcal{Q}$ is changed, only one disk access to this location is needed to modify the "parent pointers" of all its children. For simplicity and consistency we refer this two-pointer combination as the parent pointer.

For each node $v_{\mathcal{P}}$ of the partition, the following information is stored:

1. The string depth of $v_{\mathcal{P}}$.

2. A pointer to the next node in the component. By the definition of a component, there is only one such node.

3. A pointer to the leaf[2] of the component $w_{\mathcal{T}_{\mathcal{P}}}$, note that the pointer points to the copy of the leaf in $\mathcal{T}_{\mathcal{P}}$.

4. For each child $u_{\mathcal{T}_{\mathcal{Q}}}$ that is not in the same partition store in $v_{\mathcal{P}}$

   (a) A pointer to $u_{\mathcal{T}_{\mathcal{Q}}}$, note that $u_{\mathcal{T}_{\mathcal{Q}}}$ is the root of $\mathcal{Q}$.

   (b) The leading character of the edge label between v and u.

3.2   Substring Search

Given a pattern p and the suffix tree for a set of strings S, the substring matching problem is to locate a position i and a string $s \in S$ such that $s[i..i+|p|-1] = p$ where $|p|$ is the length of p, or conclude that it is impossible to find such a match. In a suffix tree we match p character by character with edge labels of the suffix tree until we can proceed no longer, or until all p's characters have been exhausted in which case a match is found.

To search for a pattern p in a suffix tree with our proposed layout, a top-down traversal is used similar to the one used in substring search in normal suffix tree. The search begins with the partition containing the root of the suffix tree. For each partition $\mathcal{P}$ encountered, the ultimate goal is to identify the node in $\mathcal{P}$, where one of its children is the root of the correct next partition in the traversal. We refer to this node as the exit node of $\mathcal{P}$, denoted as $exit_{\mathcal{P}}$. If such an exit node cannot be found then there is no substring that matches the pattern p.

In order to locate $exit_{\mathcal{P}}$, its closest ancestor $v_{\mathcal{T}_{\mathcal{P}}}$ in $\mathcal{T}_{\mathcal{P}}$ is first located, i.e. $exit_{\mathcal{P}}$ lies on the path between $v_{\mathcal{T}_{\mathcal{P}}}$ and one of its children in $\mathcal{T}_{\mathcal{P}}$, or it is $v_{\mathcal{P}}$ itself. To achieve this, an over-matched descendant of $v_{\mathcal{T}_{\mathcal{P}}}$, $w_{\mathcal{T}_{\mathcal{P}}}$ is located. Then the representative suffix stored with

---

[2]Like the parent pointer of the root of a partition. This pointer actually points to a location with another pointer pointing to the actual location of the leaf.

one of the leaves under $w_{\mathcal{T}_{\mathcal{P}}}$ in $\mathcal{T}_{\mathcal{P}}$ is used to compare with the pattern p. The number of matched characters is used to identify $v_{\mathcal{T}_{\mathcal{P}}}$. This process works, because in order for $exit_{\mathcal{P}}$ to be a descendant of $v_{\mathcal{P}}$, $v_{\mathcal{T}_{\mathcal{P}}}$ must have a string depth less than or equal to the number of matched characters.

The to find the aforementioned over-matched node $w_{\mathcal{T}_{\mathcal{P}}}$, we use an algorithm that compare only the leading character of the path between two nodes in $\mathcal{T}_{\mathcal{P}}$. The algorithm takes three inputs, $\mathcal{T}_{\mathcal{P}}$ is the skeleton partition tree where $w_{\mathcal{T}_{\mathcal{P}}}$ is to be located; max_depth is used to limit the maximum string depth of $w_{\mathcal{T}_{\mathcal{P}}}$; and the pattern p being matched. The detail of this algorithm can be found in Algorithm 1.

Algorithm 1 Compare_ and_ Skip($\mathcal{T}_{\mathcal{P}}$, max_depth, p)

1: Load $\mathcal{T}_{\mathcal{P}}$ into memory

2: $w_{\mathcal{T}_{\mathcal{P}}} \leftarrow$ root of $\mathcal{T}_{\mathcal{P}}$

3: $\ell_w \leftarrow$ string depth of root of $\mathcal{T}_{\mathcal{P}}$

4: **if** $\ell_w < max\_depth$ **then**

5:     **return null**

6: **end if**

7: $continue \leftarrow$ **true**

8: **while** $continue$ **do**

9:     $continue \leftarrow$ **false**

10:     **if** one of $w_{\mathcal{T}_{\mathcal{P}}}$'s children $u_{\mathcal{T}_{\mathcal{P}}}$ whose leading character is $p[\ell_w + 1]$ **then**

11:         **if** String depth of $u_{\mathcal{T}_{\mathcal{P}}} \leq max\_depth$ **then**

12:             $v_{\mathcal{T}_{\mathcal{P}}} \leftarrow u_{\mathcal{T}_{\mathcal{P}}}$

13:             $\ell_v \leftarrow$ string depth of $u_{\mathcal{T}_{\mathcal{P}}}$ $continue \leftarrow$ **true**

14:         **end if**

15:     **end if**

16: **end while**

17: **return** $w_{\mathcal{T}_{\mathcal{P}}}$

Since only the leading character of the path between two nodes in the skeleton partition tree $\mathcal{T}_\mathcal{P}$ is compared with corresponding character from the pattern p, it is possible that an mismatch exists on the path ending at the resulting node $w_{\mathcal{T}_\mathcal{P}}$. To check for this possibility, the representative suffix associated with one of the leaves in $\mathcal{T}_\mathcal{P}$ under $w_{\mathcal{T}_\mathcal{P}}$ is compared with the corresponding characters of p. The number of characters matched during the comparison is used to identify $v_{\mathcal{T}_\mathcal{P}}$ – the closest ancestor of $exit_\mathcal{P}$ in $\mathcal{T}_\mathcal{P}$. Algorithm 2 gives a detailed description of how to locate $v_{\mathcal{T}_\mathcal{P}}$, it takes three inputs the skeleton partition tree $\mathcal{T}_\mathcal{P}$; ℓ, the number of character of p matched so far; and the pattern p. The output of the algorithm is the node $v_{\mathcal{T}_\mathcal{P}}$ (if it exists) and ℓ, the number of character of p matched at the end of the algorithm.

Algorithm 2 Locate_ Nearest_ Ancestor($\mathcal{T}_\mathcal{P}$, ℓ, p)

1: $v_{\mathcal{T}_\mathcal{P}} \leftarrow$ Compare_ and_ Skip($\mathcal{T}_\mathcal{P}$, ∞, p)

2: $\ell_v \leftarrow$ string depth of $v_{\mathcal{T}_\mathcal{P}}$

3: Load the representative suffix of a leaf in $\mathcal{T}_\mathcal{P}$ under $v_{\mathcal{T}_\mathcal{P}}$, say s_i

4: $\ell_m \leftarrow$ # of matches between $s[i+\ell+1 \dots i+\ell_v]$ and $p[\ell+1 \dots \ell_v]$

5: $\ell \leftarrow \ell + \ell_m$

6: **return** (Compare_ and_ Skip($\mathcal{T}_\mathcal{P}$, ℓ, p), ℓ)

After Algorithm 2 one of the following cases is true.

1. Algorithm 2 returns **null**; this situation happens if and only if the total number of characters matched with p is less than the string depth of the root of the partition $\mathcal{P}$. If p is exhausted at this point then a match is found between the root of the partition and its parent in the suffix tree, otherwise no match is possible.

2. If a node $v_{\mathcal{T}_\mathcal{P}}$ is found, and the string depth of $v_{\mathcal{T}_\mathcal{P}}$ is the same as ℓ, then either $v_\mathcal{P}$ is a match (if p is exhausted) or $v_\mathcal{P}$ is $exit_\mathcal{P}$. If p is not exhausted, then $v_\mathcal{P}$ is $exit_\mathcal{P}$ and use the pointer stored in $v_{\mathcal{T}_\mathcal{P}}$ to locate $v_\mathcal{P}$, and find an appropriate child to continue, i.e. use $p[\ell+1]$ and the leading character of the children to find the appropriate child, and repeat

the algorithm with the partition containing that child. If none of the leading character of the children matches $p[\ell + 1]$ then no match is possible.

3. If a node $v_{\mathcal{T}_\mathcal{P}}$ is found, and the string depth of $v_{\mathcal{T}_\mathcal{P}}$ is strictly less than ℓ, then one of the children $u_{\mathcal{T}_\mathcal{P}}$ of $v_{\mathcal{T}_\mathcal{P}}$ whose leading character matches $p[\ell + 1]$. This is because the representative suffix from a leaf under $u_{\mathcal{T}_\mathcal{P}}$ is used. The use the pointer to $w_\mathcal{P}$, where $w_\mathcal{P}$ is the child of $v_\mathcal{P}$ and the ancestor of $u_\mathcal{P}$ in the partition. Use the string depth stored in each node in the component, and the leading character to the next node to traverse the component. Suppose the search is at node $w_\mathcal{P}$,

   (a) If the string depth of $w_\mathcal{P}$ is less than ℓ, then we move to the next node in the component.

   (b) If the string depth of $w_\mathcal{P}$ is equal to ℓ, and one of the leading character on the edge from $w_\mathcal{P}$ to one of its children not in the same partition is the same as $p[\ell + 1]$, then use the pointer to that child to go to the next partition. If none of its children have the correct leading character, then no match is found.

   (c) If the string depth of $w_\mathcal{P}$ is greater than ℓ, and p is exhausted then a match is found between $w_\mathcal{P}$ and its parent. Otherwise no match can be found.

Lemma 3.2.1 *The substring search algorithm is correct, i.e. it identifies a node or a position in an edge label where the concatenation of all the edge labels from the root to that node or position is exactly p, or report that no match can be found.*

Proof Suppose there is a match with the pattern p, and let $u_\mathcal{Q}$ be the node where the concatenation of all the edge labels from the root to $u_\mathcal{Q}$ is exactly p. Then there is a unique path in the suffix tree that the searching algorithm must traverse in order to locate $u_\mathcal{Q}$. Let $\mathcal{P}$ be a partition, such that this unique path pass through $\mathcal{P}$. Let $exit_\mathcal{P}$ be the deepest node in $\mathcal{P}$ on the unique path, i.e. $exit_\mathcal{P}$ is on the unique path but none of its children in the same partition are on the same path. We show that the searching algorithm correctly identifies $exit_\mathcal{P}$.

Let $v_\mathcal{P}$ be the closest descendant of $exit_\mathcal{P}$ or $exit_\mathcal{P}$ itself, such that there is a $v_{\mathcal{T}_\mathcal{P}}$. Line 1 of Algorithm 2 correctly locates a descendant of $v_{\mathcal{T}_\mathcal{P}}$ in $\mathcal{T}_\mathcal{P}$. Because the parent of $v_{\mathcal{T}_\mathcal{P}}$ in

$\mathcal{T_P}$ is an ancestor of $exit_\mathcal{P}$, therefore, every leading character of the edge labels on path from the root of the partition to the parent of $v_{\mathcal{T_P}}$ matches corresponding characters of p. Since $exit_\mathcal{P}$ is between $v_{\mathcal{T_P}}$ and its parent in $\mathcal{T_P}$, then the leading character from the parent to $v_{\mathcal{T_P}}$ also matches with the corresponding characters from p. However, a descendant of $v_{\mathcal{T_P}}$ may be identified due to coincident.

Line 3 of Algorithm 2 correctly finds a leaf in $\mathcal{T_P}$ that is a descendant of $exit_\mathcal{P}$, because line 1 identifies $v_{\mathcal{T_P}}$ which is a descendant of $exit_\mathcal{P}$ so any leaf that is a descendant of $v_{\mathcal{T_P}}$ must also be a descendant of $exit_\mathcal{P}$. Since the leaf is a descendant of $exit_\mathcal{P}$, then ℓ after line 5 must be the same as the string depth of $exit_\mathcal{P}$. This is because if the string depth of $exit_\mathcal{P}$ is d, then any suffix in the subtree under $exit_\mathcal{P}$ must match at least d character with p. However, by the definition of $exit_\mathcal{P}$ the child of $exit_\mathcal{P}$ with the leading character matching the corresponding character in p is not in the partition $\mathcal{P}$. Therefore, a suffix from the subtree under any of the child of $exit_\mathcal{P}$ in $\mathcal{P}$ must not produce a match more than d.

Since Algorithm 2 correctly identifies $exit_\mathcal{P}$ if it exists, it is easy to verify that the three conditions given after Algorithm 2 correctly terminates or continues the search. Furthermore, if p matches a position in the middle of an edge instead of a node, then we can pretend that the characters from the incomplete matched edge label is not a part of p, and from above, the searching algorithm can correctly locate the truncated pattern, and by condition 1 and 3(c) the position in the edge can be found.

If a pattern p cannot be located in the suffix tree, then suppose p' is the longest prefix of p that can be found in the suffix tree. The searching algorithm can correctly locate p', and by the three conditions presented after algorithm 2 it is easy to see that the algorithm will report that no match is possible for the pattern p. $\qquad\square$

Lemma 3.2.2 *Substring searching can be done in $O(|p|/B + \log_B n)$ disk accesses if $|\Sigma| = O(n)$; where $|p|$ is the length of the pattern; B is the size of a disk block and n is the number of leaves in the suffix tree.*

Proof Assume the size of each node to be constant, then we can choose $C = O(B)$ such that all the nodes of a skeleton partition tree can be stored in one or a constant number of disk

blocks. The searching algorithm first attempts to find a node in $\mathcal{T}_\mathcal{P}$, not counting the disk accesses needed to compare the leading character with the corresponding characters from the pattern p, this step takes $O(1)$ disk accesses for each partition encountered. The algorithm will encounter at most $O(\log_C n)$ number of partitions, therefore the total number of disk access is $O(\log_C n) = O(\log_B n)$.

The total number of disk accesses needed to compare the leading characters of the edge labels with appropriate characters of p is not more than $O(|p|/B)$ for all partition we encountered. This can be done with a slight modification of the searching algorithm presented in Algorithm 2. In Algorithm 2 the over-matched node is identified by comparing the leading characters of the edge labels with p, then the match between the representative suffix and p is done. Instead, whenever a leading character comparison that will cause a new block of p to be loaded, compare the entire block of p with a representative suffix for all characters in the existing block. This ensures that for the entire searching algorithm only $O(|p|/B)$ number of disk accesses are used for the comparisons between p and the leading character of edge labels.

For each partition the algorithm encounters, parts of pattern p is compared with a representative suffix, we assume both are stored in logically consecutive disk blocks. Since by the algorithm no two characters of p are compared twice except the first mismatch character between p and the representative suffix of each partition, the number of disk accesses for all partition encountered is $O(|p|/B + \log_C n) = O(|p|/B + \log_B n)$.

Finally, for each partition $\mathcal{P}$, a part of a component is accessed to locate the node $exit_\mathcal{P}$ and go to the next partition. Note that the number of nodes accessed in this part of the algorithm is never more than the number of characters compared, and since each component is a continuous sequence of nodes they can be stored sequentially on logically consecutive disk blocks. Therefore the total number of disk access is $O(|p|/B)$ for this part of the algorithm, and $O(|p|/B + \log_B n)$ for the entire algorithm.

The above proof is valid if all the nodes are of constant size. We observe that a linked list representation of the skeleton partition tree can be used, where each node $u_{\mathcal{T}_\mathcal{P}}$ points to the leftmost child, and each of the child point to its right sibling. This ensures that even if the

size of the alphabet $|\Sigma|$ is $O(n)$, the space required for each node of the skeleton partition tree is still constant. Furthermore, for each node store in the components, instead of storing all the pointers to children not in the same partition, we simply store a pointer at the node to a span of physically consecutive disk blocks that contains all the child pointers. This span of physically consecutive disk blocks should be able to contain $O(n)$ child pointers, then each of the child pointer can be accessed in $O(1)$ disk accesses. This will incur at most $O(1)$ for each partition encountered in the search. □

3.3 Updating the Suffix Tree

A dynamic suffix tree must support insertion, deletion, and modification of strings. Since a modification operation can be viewed as a deletion followed by an insertion, we concentrate our discussion on the insertion and deletion operations.

3.3.1 Insertion and deletion algorithm

To insert a string into the suffix tree with our layout, all the suffixes of the string is inserted one-by-one into the suffix tree. The process is the same as McCreight's algorithm [34] which we briefly summerize here. Suppose the first $i-1$ suffix are inserted into the suffix tree, and suffix s_{i-1} is inserted as a leaf of node v, we show how to insert suffix s_i;

1. If v have a suffix link to node u then go to node u, and skip to step 3.

2. If v does not have a suffix link yet, then go to v's parent in the suffix tree and use its suffix link to go to node u'; from u' locate one of the children of u' whose leading character of the edge label matches the corresponding character of suffix s_i, i.e. if the string depth of u' is $d_{u'}$, then the leading character is compared with character $s[i + d_{u'} + 1]$. Repeat this process until we reach node u'' that match either of the two following conditions.

   (a) The string depth of the node u'' is the same as node v; in this case set the suffix link of v to point to u'' and go to step 3.

(b) The child of u'' whose leading character of the edge label matches the corresponding character of suffix s_i has a string depth greater than v. In this case create a new node between u'' and the child with string depth equal to v; point v's suffix link at the new node and go to step 3.

3. Traverse down the suffix tree by matching every character on the edge label with its corresponding character from s_i, until the first mismatch.

   (a) If the algorithm stops at an internal node, then attach s_i as a new leaf of the internal node.

   (b) If the algorithm stops in the middle of an edge between w and its child w', then insert a new internal node in the middle of the edge where the mismatch occurred. Attach the new internal node as a child of w and the parent of w' and attach s_i as a new leaf of the new internal node.

To delete a string the same process is followed, except leaves and internal nodes are removed instead of added. The insertion and deletion algorithm for our layout is the same as McCreight's algorithm, except that after each suffix is inserted (deleted), the size and possibly some the rank of the ancestors of the inserted (deleted) leaf need to be changed as well. If the rank of a node is changed then it need to be moved to another partition. The maintenance of the correct the partition structure, and size information are discussed in Section 3.3.2 and Section 3.3.3, respectively. Here we provide an algorithm to traverse upwards in our layout, so that all the ancestors of a newly inserted (deleted) leaf can be visited.

Suppose the algorithm is at partition $\mathcal{P}$ after a leaf is inserted (deleted), then all the affected nodes can be visited if necessary by using the parent pointers stored with each node in $\mathcal{T}_\mathcal{P}$ and the appropriate pointers to individual components of the partition. Let $r_{\mathcal{T}_\mathcal{P}}$ be the root node of the current partition $\mathcal{P}$, we show how to access the partition $\mathcal{Q}$ containing the parent of $r_{\mathcal{T}_\mathcal{P}}$, and all the ancestors of $r_{\mathcal{T}_\mathcal{P}}$ in $\mathcal{Q}$.

1. The node $r_{\mathcal{T}_\mathcal{P}}$ contains a parent pointer, which points to node $u_\mathcal{Q}$.

2. In $u_{\mathcal{Q}}$, there is a pointer pointing at the last node of the component, $v_{\mathcal{Q}}$; and also the string depth of $u_{\mathcal{Q}}$, say d_u.

3. By the definition of a component, $v_{\mathcal{Q}}$ is a leaf of the partition $\mathcal{Q}$, therefore $v_{\mathcal{T_Q}} \exists \mathcal{T_Q}$.

4. By using the parent pointer and the string depth stored with each node of $\mathcal{T_Q}$, traverse upwards from $v_{\mathcal{T_Q}}$ to the node $w_{\mathcal{T_Q}}$, where $w_{\mathcal{T_Q}}$ is the first node with string depth less than or equal to d_u.

5. All the ancestor of $w_{\mathcal{T_Q}}$ in $\mathcal{T_Q}$ is an ancestor of $u_{\mathcal{Q}}$ and also the ancestor of the newly inserted (deleted) leaf. All the ancestors of $u_{\mathcal{Q}}$ between $w_{\mathcal{Q}}$ and $u_{\mathcal{Q}}$ can be found by using the pointer to the component stored with $w_{\mathcal{T_Q}}$; and all the ancestors of $u_{\mathcal{Q}}$ between the root of the partition and $w_{\mathcal{Q}}$ can be found by using the parent pointers and the appropriate pointer to different components.

3.3.2 Maintaining the structure of the partition

After a suffix is inserted to or deleted from the suffix tree, the size of all the ancestors of the leaf is changed by one while the size of all other nodes remain unchanged. This change in size may cause the rank of some of the ancestors to change as well. However, the number of ancestors whose rank are changed is limited.

Lemma 3.3.1 *The insertion of a new suffix into the suffix tree may increase the rank of a node by one only if it is an ancestor of the new leaf and is the root node of its partition. The ranks of all other nodes are unaffected.*

Proof If a node is not an ancestor of the new leaf, its size and hence its rank does not change. The size of each node that is an ancestor of the newly inserted leaf will increase by one. Consider a node v that is an ancestor of the new leaf. Suppose v is not the root of a partition and let r denote the root of the partition containing v. If $rank(v)$ were to increase, then $size(v) = C^i - 1$ just before the insertion. Since r is an ancestor of v, then $size(r) > size(v) \Rightarrow size(r) \geq C^i$, so r could not have been in the same partition as v, a contradiction. $\qquad \square$

Lemma 3.3.2 *The deletion of an existing suffix from the suffix tree may decrease the rank of a node by one only if it is an ancestor of the deleted leaf and is the leaf of its partition. The ranks of all other nodes are unaffected.*

Proof If a node is not an ancestor of the deleted leaf, then its size and rank does not change. Suppose v is not a leaf of its partition and $rank(v)$ decreases because of the deletion, then let u be a descendant of v, such that it is a leaf of the partition containing v. Since $rank(v)$ decreased by one after the deletion then $size(v) = C^i$ before the deletion, but u is a descendant of v, thus $size(u) \leq C^i - 1$ therefore is not a part of the same partition of v, a contradiction.

Assuming the size of the root and leaves of a partition is known at all time we discuss how to maintain the structure of the partitions after an insertion and deletion.

3.3.2.1   Insertion

When a new leaf is added to the suffix tree one of the following scenarios will be applicable.

1. If the new leaf is added to an existing internal node $v_{\mathcal{P}}$, and its rank is less than $v_{\mathcal{P}}$ then the new leaf forms a partition by itself and a child pointer and the appropriate leading character is added to $v_{\mathcal{P}}$.

2. If the new leaf have the same rank as $v_{\mathcal{P}}$, and $v_{\mathcal{P}}$ is a branching node in $\mathcal{P}$.

   (a) The new leaf is added to $\mathcal{P}$ as a component containing only of itself.

   (b) A node representing the new leaf is added to $\mathcal{T}_{\mathcal{P}}$ as a child of $v_{\mathcal{T}_{\mathcal{P}}}$. Appropriate information are set in the new node and $v_{\mathcal{T}_{\mathcal{P}}}$, these information is described in detail in Section 3.1.

3. If the new leaf have the same rank as $v_{\mathcal{P}}$, and $v_{\mathcal{P}}$ is a non-branching node in $\mathcal{P}$. Then after the insertion $v_{\mathcal{P}}$ will become a new branching node. The following steps will be taken,

   (a) The new leaf is added to $\mathcal{P}$ as a component containing only of itself.

(b) A new node $v_{\mathcal{T}_{\mathcal{P}}}$ is added in $\mathcal{T}_{\mathcal{P}}$ between two nodes, $u_{\mathcal{T}_{\mathcal{P}}}$ and its child $u'_{\mathcal{T}_{\mathcal{P}}}$. They can be located by use the pointer to the last node of the partition from $v_{\mathcal{P}}$, this pointer leads to a node $w_{\mathcal{T}_{\mathcal{P}}}$ which is a leaf in $\mathcal{T}_{\mathcal{P}}$; from $w_{\mathcal{T}_{\mathcal{P}}}$ travel upwards until the first node with a string depth less than $v_{\mathcal{P}}$ is found.

(c) A new node representing the new leaf is added in $\mathcal{T}_{\mathcal{P}}$ as a child of $v_{\mathcal{T}_{\mathcal{P}}}$.

4. If the new leaf is added to a new internal node $v_{\mathcal{P}}$ and the new internal node have the same rank as the new leaf. Then this new internal is a new branching node of partition $\mathcal{P}$. Assuming $v_{\mathcal{P}}$ need to inserted between $u_{\mathcal{P}}$ and its child $u'_{\mathcal{P}}$ then the following are done,

(a) The new internal node $v_{\mathcal{P}}$ is added to the component containing $u'_{\mathcal{P}}$. Note that it is possible that $u_{\mathcal{P}}$ and $u'_{\mathcal{P}}$ are not in the same component.

(b) The new node $v_{\mathcal{P}}$ is a branching node of $\mathcal{T}_{\mathcal{P}}$ therefore $v_{\mathcal{T}_{\mathcal{P}}}$ is added to the skeleton partition tree. The appropriate parent and child of $v_{\mathcal{T}_{\mathcal{P}}}$ can be found by the same process as described in Step 3(b).

(c) The new leaf is added as the only node in a new component of $\mathcal{P}$.

(d) A new node $w_{\mathcal{T}_{\mathcal{P}}}$ representing the new leaf is added to $\mathcal{T}_{\mathcal{P}}$ as the child of $v_{\mathcal{T}_{\mathcal{P}}}$.

5. If the new leaf is added to a new internal node $v_{\mathcal{P}}$, and $v_{\mathcal{P}}$ does not have the same rank as the new leaf. Then the new leaf is in a partition by itself as described in Step 1, and $v_{\mathcal{P}}$ is inserted in the component of partition $\mathcal{P}$ containing its other child.

After the new leaf is added, consider the partition $\mathcal{P}$ where the root of the partition $r_{\mathcal{P}}$ is an ancestors of the new leaf and its rank is increased by one. If the new rank of $r_{\mathcal{P}}$ is not the same as its parent, then it will become a new partition by itself. The treatment of this case is the same as case 1 in the list above. If the new rank of $r_{\mathcal{P}}$ is the same as its parent then treatment is similar to case 2 and 3 in the list above. One additional case is that the node $r_{\mathcal{P}}$ could be added under its parent in partition $\mathcal{Q}$ replacing it a the leaf, i.e. the parent continues to be a non-branching node, but cease to be a leaf of the partition. In this case, $r_{\mathcal{P}}$

is append to the end of the same component contain its parent as $r_{\mathcal{Q}}$. Then $r_{\mathcal{T}_{\mathcal{Q}}}$ replaces its parent in $\mathcal{T}_{\mathcal{Q}}$, by simply altering some of the information stored in the existing leaf representing the parent. Although $r_{\mathcal{T}_{\mathcal{Q}}}$ has replace its parent as the new last node of the component, the physical location is not changed, therefore, the pointer to the last node of the component need not be changed.

If the root of a partition is moved after the insertion of a suffix, it need to be removed from the component in the old partition; and all of its children in the old partition will become roots of different partitions. Thus the skeleton partition tree need to be split. We delay this discussion till after the deletion algorithm is presented, because the policy to maintain the skeleton partition trees need to take deletion into account as well.

3.3.2.2 Deletion

The process of deletion is similar to that of insertion, however, instead of added leaves and new internal nodes, they are deleted from the tree. For a string s McCreight's Algorithm for insertion is used to visit every suffix of s, but instead of adding a leaf that represents the suffix, the leaf is deleted. If after the deletion of the leaf its parent node have only one child left, then the parent node is deleted as well. The location of the parent is the same as in the insertion algorithm.

By Lemma 3.3.2, only a leaf can exit a component of the partition during deletion. If there are still nodes left in the component, then the appropriate information are copied to the location of the old leaf, and nothing else is changed. If the component has no nodes left, then pointers pointing to the leaf in its parent $u_{\mathcal{P}}$ is deleted; and the leaf in $\mathcal{T}_{\mathcal{P}}$ attached to $u_{\mathcal{T}_{\mathcal{P}}}$ is also deleted. If after the deletion $u_{\mathcal{T}_{\mathcal{P}}}$ became a non-branching node then it is also removed from $\mathcal{T}_{\mathcal{P}}$. When the leaf of a partition is moved from its current partition, unlike in the case of insertion, there are possibly multiple components that it can join, i.e. all the components containing one of its children with the same rank, and the decision can be made in any arbitrary manner. Note that we can find out if and which of $u_{\mathcal{P}}$'s children will have the same rank as $u_{\mathcal{P}}$ by keeping a list.

3.3.2.3   Maintaining the skeleton partition tree

For each component in a partition $\mathcal{P}$, every node of the component have a pointer point to a location on disk containing another pointer to a leaf node $v_{\mathcal{T}_{\mathcal{P}}}$, where $v_{\mathcal{P}}$ is the last node of the component. We refer to the pointer stored at the location as the pointer to the leaf of the component, or simply pointer to the leaf. To move a skeleton partition tree, two processes are involved. 1) The skeleton partition tree is moved to a new disk block, and 2) all the pointers to the leaves of the components must be changed to point to the new location of the leaf in the new disk block. While the skeleton partition tree can be moved in a constant number of disk accesses; the pointers to the leaves could be stored in different disk blocks and in the worst case requires C number of disk accesses to change all of them.

In order to avoid C disk accesses all at once, we propose a way to fix all the pointers affected incrementally. The basic idea is to first copy one of the skeleton partition trees to a new disk block, and for each leaf of the skeleton partition tree in the old disk block, a pointer pointing to the location of the same leaf in the new disk block is stored. This way when a node in the component need to access the leaf of the component, it will first find the old location of the leaf, then follow the pointer to the new location of the leaf. Then in the subsequent insertions or deletions a few of the components are visited at a time and their pointers to the leaves are "*processed*," i.e. the pointers are changed to point to the location of the leaf in the new disk block. After the pointers to the leaves of all the components of one skeleton partition tree are processed, the same process is repeated for another skeleton partition tree. While the pointers to the leaves of a skeleton partition tree is been processed, we refer to this skeleton partition tree as in the process of being processed.

When a set of skeleton partition trees that need to be split from a disk block. It is possible that this set of skeleton partition trees are currently being merged into one disk block. If this is the case, choose the skeleton partition tree whose components are currently being processed, otherwise pick the skeleton partition tree with the least number of components. If the skeleton partition tree is being merged, then copy it back to the old disk block and fix the pointers to the leaves to point back to the old disk block. If the skeleton partition tree completely resides

in one disk block, then copy it to a new disk block.

When a set of skeleton partition trees to be merged into a single disk block. As with split, it is possible that this set of skeleton partition trees are currently being split into multiple disk blocks. However, only one of these skeleton partition trees is in the process of being processed. If this is the case, the skeleton partition tree being processed is chosen and continue to fix the remaining of its components' pointers to point to the new disk block. If every skeleton partition tree are in a disk block of their own, then the one with the smallest number of component is chosen, and is copied into the disk block containing other skeleton partition trees, or the skeleton partition tree with the largest number of components. The new root of all the skeleton partition trees being merged is added to the same disk block as the first chosen skeleton partition tree, i.e. the skeleton partition tree being processed or the skeleton partition tree with the largest number of components.

With a skeleton partition tree and its destination selected, the pointer to the leaf of each of its components are processed in subsequent insertion and deletions. Suppose a node $v_{\mathcal{P}}$ is an exit node of $\mathcal{P}$, i.e. from $v_{\mathcal{P}}$ the insertion or deletion algorithm moves to another partition $\mathcal{Q}$. Let $\mathcal{T} = \{\mathcal{T}_{\mathcal{Q}1}, \ldots, \mathcal{T}_{\mathcal{Q}k}\}$, $k \leq C$, be a set of skeleton partition tree that is being merged into one disk block or split into k disk blocks. Let $\mathcal{T}_{\mathcal{Q}i}$ be the skeleton partition tree that is currently being processed. If any node $u_{\mathcal{Q}j}$ such that $u_{\mathcal{Q}j}$ is the root of $\mathcal{T}_{\mathcal{Q}j} \in \mathcal{T}$ is visited, then two of $\mathcal{T}_{\mathcal{Q}i}$'s components are processed. Note that by this definition, even if $\mathcal{T}_{\mathcal{Q}j}$ has been completely processed the algorithm would still process two of $\mathcal{T}_{\mathcal{Q}i}$'s components. This only incurs constant number of disk accesses per partition visited, therefore it does not increase the over-all asymptotic performance of our algorithm.

The aforementioned incremental correction of the pointers only applies to partitions whose rank is strictly greater than zero. Because for any partition whose rank is zero, every node except the leaves are branching. Therefore, the subtree that is the partition is also the skeleton partition tree, so there is no need to store the components and the skeleton partition tree separately, and the pointer to the leaf of each component can be eliminated. So no process are needed, only sometime when they are visited the partition need to be moved to a different

block.

For a skeleton partition tree whose components are currently being processed, it might be possible that a subsequent insertion would cause this tree to split further, or merge with other skeleton partition trees that are also being processed. The accumulation of this could lead to a skeleton partition tree being fragmented into many different pieces, therefore affecting the performance of our algorithm. We show this scenario is not possible in the subsequent lemmas.

Lemma 3.3.3 *Let $\mathcal{T} = \{\mathcal{T}_{\mathcal{P}1}, \ldots, \mathcal{T}_{\mathcal{P}k}\}$ be a set of skeleton partition trees to be moved out of a disk block due to the removal of node $v_{\mathcal{P}}$ as the parent of their roots. The rank of the root of any $\mathcal{T}_{\mathcal{P}i} \in \mathcal{T}$ cannot increase until all the skeleton partition trees are processed.*

Proof Let $\mathcal{T} = \{\mathcal{T}_{\mathcal{P}1}, \ldots, \mathcal{T}_{\mathcal{P}k}\}$ be a set of skeleton partition trees affected by the same insertion. Let $r_{\mathcal{P}i}$ be the root of $\mathcal{T}_{\mathcal{P}i}$. Since $\mathcal{T}_{\mathcal{P}i}$ is being processed, then the rank of $r_{\mathcal{P}i}$ is greater than zero. Let the size of $r_{\mathcal{P}i}$ in the beginning when $\mathcal{T}$ start being processed be x, and the total size of the roots of all other skeleton partition tree be y at the same time. Since the parent of all the nodes has just been promoted then its size is C^j, where $j < 1$, and $C^{j-1} \leq x, y < C^j$. So at least $y \geq C$ number of insertion in the subtree under $r_{\mathcal{P}i}$ is required to cause its rank to increase. By our algorithm $\mathcal{T}$ will be processed at least $2y \geq 2C$ times due to the insertions, so all the skeleton partition trees should be processed by the time $r_{\mathcal{P}i}$'s rank is increased. $\square$

Lemma 3.3.4 *Let $\mathcal{T} = \{\mathcal{T}_{\mathcal{P}1}, \ldots, \mathcal{T}_{\mathcal{P}k}\}$ be a set of skeleton partition trees to be moved into of the same disk block due to the decrease in rank of node $v_{\mathcal{P}}$, the parent of their roots. The rank of the root of any $\mathcal{T}_{\mathcal{P}i} \in \mathcal{T}$ cannot increase until all the skeleton partition trees are processed.*

Proof The proof is the same as Lemma 3.3.3.

Lemmas 3.3.3 and 3.3.4 shows that while a set of partitions are being processed, none of them can be affect by a split internally until all of them are completely processed. Although, it is possible that for a set of skeleton partition trees that are currently being merged, they can be split back into individual trees due to the removal of their parent. This does not cause problem to our algorithm, because instead of merging the algorithm will start to split out the

already merged partitions, and still maintain the fact that only partition is being processed for the set of partitions.

Lemma 3.3.5 *Let $\mathcal{T} = \{\mathcal{T}_{\mathcal{P}1}, \ldots, \mathcal{T}_{\mathcal{P}k}\}$ be a set of skeleton partition trees being processed for merge. It is not possible to merge them with another set of skeleton partition trees currently being processed.*

Proof Assume to the contrary, that another set skeleton partition trees that are being processed is to be merged with $\mathcal{T} = \{\mathcal{T}_{\mathcal{P}1}, \ldots, \mathcal{T}_{\mathcal{P}k}\}$. They cannot be affected by the same merge operation because in that case they will be the same set of skeleton partition trees. Consider the node $v_{\mathcal{P}}$ that caused the partitions in $\mathcal{T}$ to be merged, $size(v_{\mathcal{P}})$ was $C^{i+1} - 1$ at the time of the merge. Since $\mathcal{T}$ is being processed, their rank must be greater than zero. In order for $\mathcal{T}$ to be merged with another set of partitions with the same rank, at least C deletion must occur under $v_{\mathcal{P}}$, this will cause $2C$ components to be processed. So $\mathcal{T}$ would be completely processed.

3.3.2.4 Maintaining the components

While the skeleton partition tree is guaranteed to fit in one or constant number of disk blocks, each individual component may not. However, since a component consists of all nodes on a path in the suffix tree, therefore each disk block can be used to store a consecutive segment of a component. During insertion eventually a disk block containing a consecutive segment of a component may be completely filled. In this case, the disk block containing the consecutive segment is marked for split; and a new disk block is obtained; copy the last two nodes in the segment to the new disk block. For every subsequent visit to these two disk blocks the last two nodes of the segment contained in the old disk block is moved to the new disk block, until $\frac{B}{2}$ nodes are moved, where B is the total number of nodes can be stored in one disk block. This is similar to the process Ferragina and Grossi used for string B-trees [18].

3.3.3   Maintaining the correct size information

Lemmas 3.3.1 and 3.3.2 limits the number of nodes that may be moved to another partition. However, it does not limit the number of nodes whose size are changed due to insertion. Consider that if a suffix tree is build for the string $a^n\$$ (the character a repeated n times followed by \$), and the new string $a^{n-1}b\$$ is being inserted. The size of $n-1$ internal nodes need to be changed, while there are at most $\log_C n$ number of nodes whose rank need to be changed. Therefore, it is impossible to keep track of the size of all nodes. In order to decrease the number of nodes whose size is changed after the insertion of a suffix we make the following observations.

1. During insertion it is enough to know the size of the root and the size of all the leaves for any given partition. Therefore only the size of a subset of nodes need to be known at all times. As long as this subset of nodes allows us to:

   (a) Calculate the size of the root and any leaf of any partition.

   (b) Calculated the size of a node that just became the root or a leaf of a partition.

2. if the suffix tree is build incrementally by inserting one suffix at a time, then for any branching node of a partition, it is possible to keep track the size of all but one of its children in the same partition. This is because when a non-branching node of the partition change into a branching node, then the new child was the root of a partition so its size is known, this is true with all subsequent children. If a node became a branching node due to a deletion in its subtree, and the merging of it with multiple of its children. Then the size of each of these children are know because they were the roots of their own partition.

With the two observations above we keep track of the size as follows,

1. If $v_{\mathcal{T_P}}$ is the root or a leaf of the partition.

2. Let $v_{\mathcal{T}_\mathcal{P}}$ be a branching node, and let $r_{\mathcal{T}_\mathcal{P}}$ be the root of the partition $\mathcal{P}$. If every node $u_\mathcal{P}$ on the path from $r_\mathcal{P}$ to $v_\mathcal{P}$, $u_{\mathcal{T}_\mathcal{P}} \in \mathcal{T}_\mathcal{P}$, i.e. every node is a part of the skeleton partition tree.

3. If $u_\mathcal{P}$ is a non-branching child of a node that satisfies either of the two rules above.

4. If $v_{\mathcal{T}_\mathcal{P}}$ is a branching node, but does not satisfy either of the first two rules, then the size of all but one of its children are correctly maintained at all times.

5. For every node in $\mathcal{P}$, the total size of all its children not in partition $\mathcal{P}$ is maintained.

If $r_\mathcal{P}$ is the root of a partition, any insertion or deletion of a leaf in its subtree $r_\mathcal{P}$ will be visited either during the search or when updating the partitions. So its size information can be updated if necessary, this is also true with all leaves and branching nodes whose size are known. So the first two conditions can be maintained. If the size of the nodes satisfying condition three are stored in their parents in the skeleton partition tree, then their size can be updated as well, since its possible to find out if a new leaf is added or deleted under a child. This method can also be used to satisfy condition four. Condition five can be done easily because if the size of a child of a non-branching node is changed, the non-branching node must be the exit node of the partition, thus it will be accessed and the total count can be updated.

When a partition is split it can be seen from the five conditions above, the size of the root of ever new partition are know. In the case the new root is a branching node nothing need to be done. But if the new root is a non-branching node, the size of it's child in the same partition can be easily calculated from its own size and the total size of all its children not in the partition. When multiple partitions are merged, then it can be seen all five conditions are observed.

When a non-branching node $v_\mathcal{P}$ of the partition became a branching node by the insertion of a new node $w_\mathcal{P}$, the existing child of $v_\mathcal{P}$, say $u_\mathcal{P}$ could be an branching node. Before the insertion of $w_\mathcal{P}$, $u_\mathcal{P}$ is a node satisfying condition four, however after the insertion it may satisfy condition two. If this is the case then the size of $v_\mathcal{P}$ is know because it satisfies condition three before the insertion, and since the total size of all its children not in the same partition was

known, the size of $u_{\mathcal{P}}$ can be calculated. If one of $u_{\mathcal{P}}$'s children is also a branching node, then its size can be calculated the same way.

Lemma 3.3.6 *To insert or delete one leaf in our layout takes $O(\log_C n)$ number of disk accesses, and to insert a string of length n into a suffix tree of size m takes $O(n \log_C(n + m))$ disk accesses; and to delete a string of length n into a suffix tree of size m takes $O(n \log_C m)$ disk access.*

Proof After the insertion or deletion of a leaf, all the partition that contains the ancestor of the leaf need to be updated; and there are at most $O(\log_C n)$ partitions. To process each partition at most $O(1)$ disk accesses are needed. Therefore to insert or delete a leaf takes $O(\log_C n)$ time.

To insert a string using McCreight's algorithm the position of the first leaf to be inserted takes $O(n + \log_C n)$ disk accesses to locate. The total disk accesses needed to locate the position for subsequent suffixes are no more than $O(n)$. To process all the partition containing the ancestors takes $O(n \log_C(n + m))$ disk accesses. This is the same for deletion except the size of the tree is n in the worst case, not $n + m$, therefore the number of disk accesses is $O(n \log_C n)$.

CHAPTER 4.   SELF-ADJUSTING LAYOUT

Like most indexing structures, suffix trees are built with the intention that they will be queried many times. Therefore, it is very important to devise algorithms that not only guarantee the worst case performance of a single query, but also provide good performance for a large sequence of queries collectively. In 1985, Sleator and Tarjan [35] created the self-adjusting binary tree by using a "splay" process, and proved that it produces the optimal number of disk accesses for a large sequence of queries. The splaying technique has received wide attention since the publication of their ground breaking paper.

4.1   Self-adjusting layout

For each node v of the suffix tree, instead of using the number of leaves in the subtree ($size_v$), we define the **score** of a node v (denoted as $score_v$) as follows; for any leaf v of the suffix tree, $score_v$ is one plus the number of times v is accessed in a sequence of queries. If v is an internal node, then the score is defined as the sum of the score of all the leaves in the subtree under v. Similarly, we define $rank_v = \lfloor \log_C(score_v) \rfloor$.

Consider a set of N queries $\{q_1, q_2, \ldots, q_N\}$ consisting of substring searches, suffix insertions and/or suffix deletions. For a node v in the suffix tree, every substring search and suffix insertion query will only cause $score_v$ to increase by one. However, $score_v$ could decrease by more than one during suffix deletion operations.

Lemma 4.1.1 *The insertion of a new suffix into the suffix tree may increase the rank of a node by one only if it is an ancestor of the new leaf and is the root node of its partition. The ranks of all other nodes are unaffected.*

Proof Same as the proof for Lemma 3.3.1.

Lemma 4.1.2 *When a leaf v is removed from a suffix tree, then the rank of a node u will not change if u is not an ancestor of v, or it is a branching node in its partition, or the ancestor of a branching node in its partition.*

Proof Suppose leaf v is removed from the suffix tree. For any node u that is not an ancestor of v in the suffix tree, u's score and rank are not changed. If $u_{\mathcal{P}}$ is a branching node and an ancestor of v, then the score and rank of at least one child of $u_{\mathcal{P}}$ also in $\mathcal{P}$ is not affected, and since $u_{\mathcal{P}}$ must have a rank equal to or greater than any of its children, $u_{\mathcal{P}}$'s rank is unaffected. If $u_{\mathcal{P}}$ is the ancestor of a branching node $w_{\mathcal{P}}$ and an ancestor of v, $w_{\mathcal{P}}$'s rank is not affected by v's deletion, so $u_{\mathcal{P}}$'s rank must not change either. $\qquad\square$

Although more than one node can change their partition under suffix deletion, but the size of skeleton partition tree and the number of components in the new tree is still bounded by C, therefore our delayed merging technique will still give the same asymptotic bound.

4.2 Self-adjusting performance

Given a sequence of N queries, we assume without loss of generality that all queries are successful and end at one of the leaf nodes of the suffix tree. If this sequence of queries is known beforehand and the suffix tree is partitioned accordingly, then the number of disk accesses needed to answer any of the queries will be $O\left(\frac{|p|}{C} + rank(r) - rank(v)\right) = O\left(\frac{|p|}{C} + \log_C \frac{N}{score(v)}\right)$ where $|p|$ is the length of the query, r is the the root node of the suffix tree, and v is the leaf where the query ends. Let $p_1, p_2, \ldots, p_N$ be the sequence of queries, and let $\{v_1, v_2, \ldots, v_M\}$ be the set of leaves in the suffix tree. Over the entire sequence of queries, the performance of our layout is

$$O\left(\sum_{i=1}^{N} \frac{|p_i|}{C} + \sum_{j=1}^{M} score(v) \log_C \frac{N}{score(v)}\right) \tag{4.1}$$

This is the optimal performance for a given sequence of queries for any data structure indexing strings [11]. We now show that this worst case performance can be achieved even if the sequence of queries are is known beforehand.

Theorem 4.2.1 *Let $P = p_1, p_2, \ldots, p_N$ be a sequence of N queries, and S be the set of numbers $\{s_1, s_2, \ldots, s_M\}$ such that s_i is the number of times leaf v_i is accessed by patterns in P. Then*

$$O\left(\sum_{i=1}^{N} \frac{|p_i|}{C} + \sum_{j=1}^{M} s_j \log_C \frac{N}{s_j}\right)$$

number of disk accesses are needed to answer all queries in P.

Proof Since the sequence of queries is not known beforehand, we calculate how many more disk accesses are needed than the ideal scenario. Consider a suffix tree in our layout where all leaves have an initial score of one. Let v_i be a leaf node which will be accessed s_i times in the sequence of N queries. The ideal number of disk accesses is $O(\frac{|p|}{C} + \log_C N - \log_C s_i)$ for each of the s_i times v is accessed. For the first C queries made that end at v_i the number of disk accesses is $O\left(\frac{|p|}{C} + rank(r) - rank(v)\right) = O\left(\frac{|p|}{C} + \log_C N\right)$ which requires $O(\log_C s_i)$ more disk accesses than ideal. For the next $C^2 - C$ number of queries, the number of disk accesses is $O(\log_C s_i - 1)$ more than the ideal number of disk accesses, and so on. The sum of this telescoping series is

$$B \log_B k + (B^2 - B)(\log_B s_i - 1) + \ldots = B + B^2 \ldots + B^{\log_B s_i} = O(s_i)$$

Therefore the total number of disk accesses for all of the s_i times that v_i is accessed is $O\left(s_i \frac{|p|}{B} + s_i \log_B \frac{N}{s_i} + s_i\right)$, and for the sequence of queries P, the number of disk accesses needed is

$$O\left(\sum_{i=1}^{N} \frac{|p_i|}{B} + \sum_{j=1}^{M} s_j \left(\log_B \frac{N}{s_j} + \Theta(1)\right)\right) = O\left(\sum_{i=1}^{N} \frac{|p_i|}{B} + \sum_{j=1}^{M} s_j \log_B \frac{N}{s_j}\right)$$

$\square$

So even with the slow promotion of the leaf, the optimal disk access bound can be achieved.

4.3 Discussion

Let q_i and q_j be two queries from the sequence of N queries, that share a common prefix p. Let v_i and v_j be the two leaves where q_i and q_j will end, respectively. Then the score of

the lowest common ancestor of v_i and v_j, say v, is at least the sum of the scores of v_i and v_j. Therefore even if q_i and q_j are infrequent queries compared to others queries in the sequence, v could still have a high rank, thus potentially reducing the number of disk accesses needed to find v_i and v_j. This is also true for a sequence of such queries.

For a balanced tree there will always be $\Theta\left(\frac{|p|}{B} + \log_B M\right)$ number of disk accesses, where M is the number of leaves in the tree. For any self-adjusting data structure for strings in secondary storage, suppose that every leaf is accessed equal number of times, then the worst case performance of the self-adjusting data structure is the same as the balanced tree, because $\frac{N}{k} = M$. But if the self-adjusting data structure is not balanced to start with, then it will take some time for it to became as efficient as the balanced tree. Therefore self-adjusting data structure will not be as effective as the balanced tree.

From Equation 4.1 we can also observe that if a leaf is accessed B times more frequently than other leaves in the tree in the sequence of N queries, it will only save a total of B disk accesses compared to the balanced tree. So if we assume that the self-adjusting data structure is not balanced in the beginning, then it would require a very skewed data set to offset the initial inefficiency of the self-adjusting data structure.

This observation provides an explanation for the results in [8, 39], where the authors were surprised that self-adjusting data structures do not perform nearly as well as balance trees, except for very skewed data sets. However, if the suffix tree is built with our layout scheme, then it will be a balanced tree, potentially avoiding the initial inefficiencies. But it should be noted that all self-adjusting data structures will incur an overhead for the adjustment which will also affect their performance.

CHAPTER 5.   DISCUSSION AND FUTURE RESEARCH DIRECTIONS

In this dissertation we presented a linear time algorithm for sorting the suffixes of a string over an integer alphabet, or equivalently, for constructing the suffix array of the string. Our algorithm can also be used to construct suffix trees in linear time. Apart from being one of the first direct algorithms for constructing suffix arrays in linear time, the simplicity and space advantages of our algorithm are likely to make it useful in suffix tree construction as well. An important feature of our algorithm is that it breaks the string into substrings of variable sizes, while other linear time algorithms break the string into substrings of a fixed size. Although our algorithm has a linear worst case run-time, some of the fastest sorting algorithm for integers are not linear. Therefore some future research should focus on improving the performance of linear time suffix array construction algorithm.

We also presented a new tree layout scheme for secondary storage and showed in detail how to use this layout scheme to store suffix trees in secondary storage. We also provided algorithms with provably good worst-case performance for search and update operations. The performance of our algorithms when applied to suffix trees matches what can be obtained by the use of string B-trees, a data structure specifically designed to efficiently support string operations on secondary storage.

Finally, we presented a self-adjusting variant of the suffix tree layout scheme in secondary storage, which allows insertion and deletion of strings. We showed that our layout scheme is optimal for a sequence of queries. This settles the question first proposed by Sleator and Tarjan [35] of constructing an optimal self-adjusting data structure for strings. Our layout scheme can also be applied to PATRICIA trees and possibly to many other data structures whose toplogy is uniquely determined by the data being indexed and cannot be altered. However, we also call

the benefit of non-balanced self-adjusting data structures for strings into question. We argued that due to the overhead needed to readjust itself, self-adjusting data structures are not likely to perform as well as balanced trees except for very skewed data sets. But since the initial state of our layout scheme is a balanced tree, and it only readjusts itself very infrequently it may perform well in practice.

Suffix trees are extensively used in biological applications. As our scheme provides how to efficiently store and operate on them in secondary storage that is competitive with some of the best available alternatives, the research presented provides justification for using suffix trees in secondary storage as well. It is important to compare how the presented algorithms compare in practice with other storage schemes developed so far (those with and without provable bounds on disk accesses), and such work remains to be carried out. Some of the possible work in the future includes: 1) improve the suffix tree construction algorithm so that it is optimal for secondary storage and 2) provide a cache oblivious version of the layout scheme.

BIBLIOGRAPHY

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proc. 2nd Workshop on Algorithms in Bioinformatics*, pages 449–463, 2002.

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

[3] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. *Handbook of Computational Molecular Biology (Chapman & All/Crc Computer and Information Science Series)*, chapter 7: "Enhanced Suffix Arrays and Applications". Chapman & Hall/CRC, 2005.

[4] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. 9th International Symposium on String Processing and Information Retrieval*, pages 31–43, 2002.

[5] S. Aluru. *Handbook of Computational Molecular Biology (Chapman & All/Crc Computer and Information Science Series)*. Chapman & Hall/CRC, 2005.

[6] S. J. Bedathur and J. R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proc. 20th International Conference on Data Engineering*, pages 720–731, 2004.

[7] S. J. Bedathur and J. R. Haritsa. Search-optimized suffix-tree storage for biological applications. In *Proc. 12th IEEE International Conference on High Performance Computing*, pages 29–39, 2005.

[8] J. Bell and G. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software - Practice and Experience*, 23(4):369–382, 1993.

[9] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. Genbank. *Nucleic Acids Research*, 34(Database issue):D16–D20, 2006.

[10] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.

[11] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *Proc. 43rd Annual Symposium on Foundations of Computer Science*, pages 219–227, 2002.

[12] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

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

[14] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[15] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proc. 23rd International Colloquium on Automata, Languages and Programming*, pages 550–561, 1996.

[16] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[17] P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382, 1996.

[18] 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.

[19] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures & Algorithms*, chapter 5: "New indices for text: PAT trees and PAT arrays", pages 66–82. Prentice Hall, 1992.

[20] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Annual ACM Symposium on Theory of Computing*, pages 397–406, 2000.

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

[22] E. Hunt, M. P. Atkinson, and R. W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11(3):256–271, 2002.

[23] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix array. In *Proc. String Processing and Information Retrieval Symposium & International Workshop on Groupware*, pages 81–88, 1999.

[24] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, pages 943–955, 2003.

[25] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium Combinatorial Pattern Matching*, pages 181–92, 2001.

[26] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium Combinatorial Pattern Matching*, pages 186–199, 2003.

[27] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium Combinatorial Pattern Matching*, pages 200–210, 2003.

[28] P. Ko and S. Aluru. *Handbook of Computational Molecular Biology (Chapman & All/Crc Computer and Information Science Series)*, chapter 6: "Suffix Tree Applications in Computational Biology". Chapman & Hall/CRC, 2005.

[29] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[30] P. Ko and S. Aluru. Obtaining provably good performance from suffix trees in secondary storage. In *Proc. 17th Annual Symposium Combinatorial Pattern Matching*, pages 72–83, 2006.

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

[32] N.J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, 1999.

[33] U. Manber and G. Myers. Suffix arrays: a new method for on-line search. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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

[35] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[36] S. Tata, R.A. Hankins, and J.M. Patel. Practical suffix tree construction. In *Proc. 13th International Conference on Very Large Data Bases*, pages 36–47, 2004.

[37] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995.

[38] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[39] H. E. Williams, J. Zobel, and S. Heinz. Self-adjusting trees in practice for large text collections. *Software - Practice and Experience*, 31(10):925–939, 2001.