Thesis

Space Efficient String Search Algorithms and Data Structures

Submitted by

Pratik Deoghare

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2015

Master's Committee:

Advisor: Ross McConnell
Co-Advisor: Wim Bohm

Alexander Hulpke

Abstract

Space Efficient String Search Algorithms and Data Structures

We address the problem of finding all the occurrences of a pattern of length m in a text of length n. We introduce two new data structures, called the *Sloppy Suffix Array* and the *Super Sloppy Suffix Array*. These data structures are space efficient, easy to understand and implement. Sloppy suffix arrays can be built faster than suffix arrays with just one array of n integers; the search algorithms for both have the same time complexity of $O(m \log n)$. We also give a space efficient representation for positional de Bruijn graphs using suffix arrays.

Table of Contents

List of Figures

CHAPTER 1

Introduction

Given a string X let $|X|$ denotes its length. We want to find all the occurrences of string P in string T. We call string P the pattern and T the text. $|P| = m$ and $|T| = n$. Let $occ(X)$ be the set of occurrences of a string X in text T.

The string will be searched repeatedly, so it makes sense to build a data structure. There are two algorithms associated with such a data structure. The construction algorithm builds the data structure from T. The search algorithm finds all the occurrences of P in T with the help of the data structure.

This problem is far from solved for all the cases [1]. There has been a lot of interest in space efficient data structures due to bioinformatics. Many existing data structures hide constants in their asymptotic space bounds. Thrashing can begin if the data structure does not fit entirely inside the RAM. There are algorithms that work in limited space. These algorithms tend to be very complicated, making the implementations bug-prone.

We design data structures with these things in mind. In this paper, we give data structures that are space efficient and easy to implement. To keep hidden constants in sight and calculate exact space used by the data structures we avoid using asymptotic notation for space bounds.

Definition 1. *Let $h(T)$ be the length of the longest substring X of T that occurs at least $|X|$ times in T. [2]*

We use $d(T)$ to denote the smallest power of 2 greater than or equal to $h(T) + 1$. The expected value of $h(T)$, hence of $d(T)$, is $O(\log n)$ for a randomly generated string of size n. For string $T = bananababa$, $h(T) = 2$ because ba occurs 3 times and $|ba| = 2$.

Definition 2. *Let $h(i,T)$ be the length of the longest substring X of T that starts at position i in T and occurs at least $|X|$ times in T. [3]*

Thus, the average value of $h(i,T)$ is

$$\hat{h}(T) = \frac{1}{n}\sum_{i=1}^{n} h(i,T).$$

We use $d(i,T)$ to denote smallest power of 2 that is greater than or equal to $h(i,T)+1$.

Let

$$\hat{d}(T) := \frac{1}{n}\sum_{i=1}^{n} d(i,T).$$

We use the following notation. Given a set $\mathcal{S}$, let $\mathcal{S}[i]$ denote i^{th} element of the set. We think of strings as ordered set of characters. Then we have following definition $T[i..j] := \{T[k] \mid i \leq k \leq j\}$. We call $T[i..j]$ a *slice* of T. The prefix of string T that ends at position i is $\mathit{prefix}(i,T) := T[1..i]$. The suffix of string T that begins at position i is $\mathit{suffix}(i,T) := T[i..n]$.

Older papers assume 32-bit representation of integers. For modern applications 32 bits is not enough. Assuming 64-bit representation of integers makes us overstate the space required. This makes it difficult to compare space complexity of the data structures. Since most algorithms build data structures that are indexes of integers and pointers are also integers, it makes sense to use number of integers used by an algorithm/data structure as the measure of space complexity. Throughout this document we represent an array of n integers as an array of bits, where each integer takes only $\lceil \log n \rceil$ bits. Similarly, we represent a string as an array of characters where each character uses $\lceil \log \sigma \rceil$.

In this document, a general set $\mathcal{S}$ of strings is represented as an array of strings. Associated with each string is an integer which is the index of the string in $\mathcal{S}$. When sorting $\mathcal{S}$, we don't move strings around; we move the associated indices. Strings (or prefixes of

a certain length) are used as keys by the sorting algorithm. This representation requires $\sum_{s \in \mathcal{S}} |s| \lceil \log |\sigma| \rceil + |\mathcal{S}| \lceil \log |\mathcal{S}| \rceil$ bits where σ is the alphabet. Our work is inspired by applications in bioinformatics where the alphabet sizes are typically small. Therefore, we consider alphabet size to be constant.

When dealing with a set of suffixes of string T, the set is represented using T and the set of indices is an array of integers, where each integer is the starting location of the suffix in T. This set contains integers from 1 to $|T|$. This takes $|T| \lceil \log |\sigma| \rceil + |T| \lceil \log |T| \rceil$ bits. While comparing suffixes of unequal length, we assume that there is a special character \$ at the end of any string and that \$ is lexicographically smaller than all the characters.

CHAPTER 2

Liturature Survey

2.1. Tries

Definition 3 (Trie)**.** *A trie on alphabet σ is a rooted tree with the following properties.*

(1) Each edge is labeled with a character.

(2) For each node u and each character $c \in \sigma$, there is at most one edge with label c from u to a child of u.

The *path label* of a node u in a trie is the string spelled out by the path from the root of the trie to u. The name trie comes from *re**trie**val* and is pronounced like *try*. It can be used to store a arbitrary set $\mathcal{S}$ of strings. Because of the second property of tries, there is a one-to-one correspondence between nodes of a trie and path labels. They can be used interchangeably. A trie is a tree, because there is only one path from each node to any other node (Figure 2.1). When the set that the trie represents is the set of suffixes of a string T we call it a *suffix trie* (Figure 2.2).

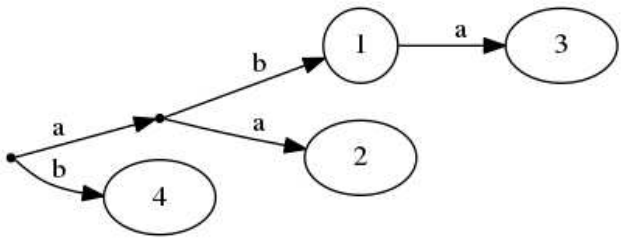


Figure 2.1. The trie of $\mathcal{S} = \{ab, aa, aba, b\}$.

Insertion of a new string q in a trie can be performed by starting at the root, following the edges that are labeled by characters of q and creating new edges corresponding to characters if they are not already the path label of any node of the trie. There is a node corresponding to every string of the set. At the node representing string q, we install a pointer that points

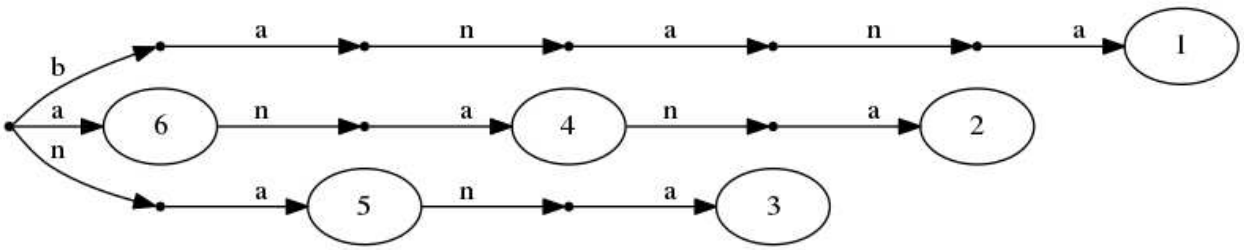


FIGURE 2.2. The suffix trie of $T = banana$.

to q. At nodes that represent no string in the set $\mathcal{S}$ we install a pointer that points to NULL. All leaf nodes point to some string in the set. Clearly, this can be done in $O(|q|)$ time. Thus, insertion in a trie is linear in the length of the string. Inserting all the strings in a set into the trie completes construction of the trie for the set of strings.

A search algorithm is just like the construction algorithm above. Start at the root. Keep following outgoing edges labeled from current node and current character. If no outgoing edge exists from current node on the current character then $s \notin \mathcal{S}$. If you end up at a node u that points to NULL and s is exhausted, then there exists some $t \in \mathcal{S}$ of which s is a prefix. This is useful when searching for all the occurrences of a string in T. The subtree below node u gives us all the strings in $\mathcal{S}$ of which s is a prefix. If $\mathcal{S}$ is the set of suffixes of T then the subtree contains pointers to all the occurrences of $s \in T$. Searching can be done in $O(|s|)$ time. This algorithm finds whether s occurs as a prefix of any string in S.

The construction algorithm for a trie takes

$$O(\sum_{s \in S} |s|)$$

time. Hence to construct a suffix trie takes

$$O(n(n+1)/2) = O(n^2)$$

time. A worst case space requirement for a trie is $O(n^2)$, for strings of the form $a^{n/2}b^{n/2}$ [4] (Figure 2.3). This space requirement is prohibitive for many applications of practical interest.

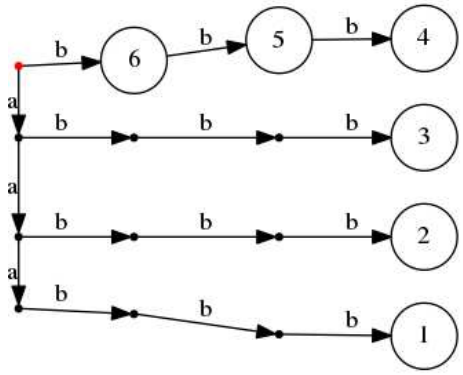


FIGURE 2.3. Worst case suffix trie for $T = a^{n/2}b^{n/2}$ for $n = 6$.

Despite the disadvantages, tries have found some real world applications. If set of strings is small and you want to filter out strings that are not in the set, then tries are good choice of data structure. They are used in the high performance messaging library ZeroMQ. There is a large number of machines that broadcast messages and every machine is subscribed to a small subset of machines to receive updates from. Each message contains identifier of the sender. Subscribers need to decide quickly if the message is from one of the machines it is subscribed to or not. Tries are used to decide if a given sender is in the subscribed list, independently of the size of the subscribed list [5].

2.2. Suffix Trees

We can merge two edges into one and label the new edge by concatenation of the labels of the original edges. This is called *path compression*. To reduce the space requirement of a suffix trie, we compress the paths that don't branch and connect two nodes that point to some string in the set represented (i.e. non-NULL nodes). This gives us the suffix tree. We

label each edge with a string instead of a character. This reduces the number of the internal nodes in the tree. Compare the suffix trie and the suffix tree of $T = banana$ in Figure 2.4,2.5.

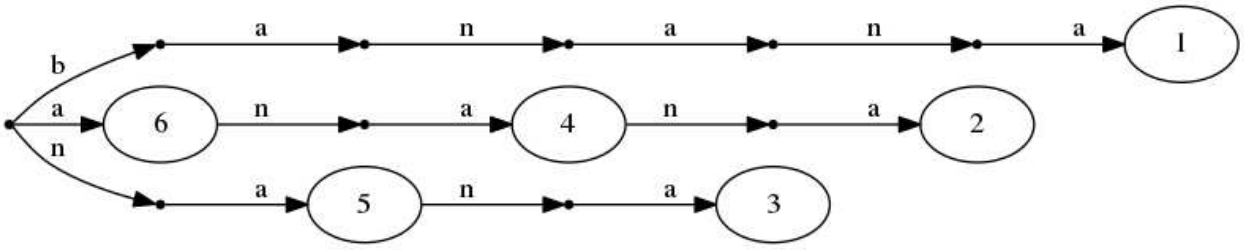


FIGURE 2.4. Suffix trie of $T = banana$

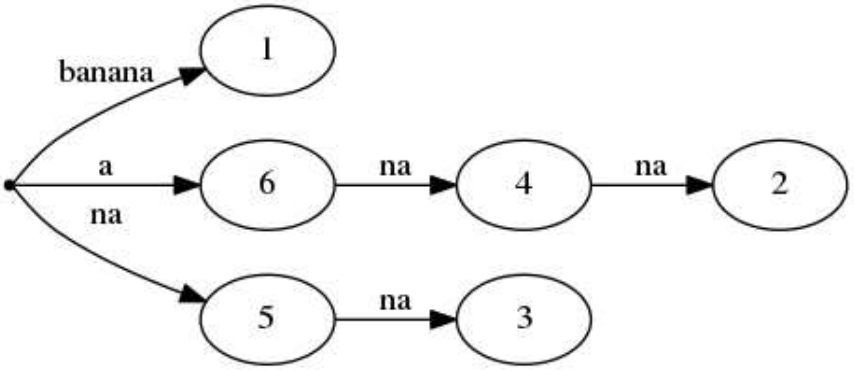


FIGURE 2.5. Suffix tree of $T = banana$

There are at least n nodes in this tree, each corresponding to a suffix. The label of each edge is an integer tuple, which is the start and end position of the label of the node in T. Hence it is necessary to have $4n$ integers. There can be at most n internal nodes and hence the worst case space requirement is $5n$ integers which is $O(n)$.

Path compression improves space complexity to $O(n)$ for suffix trees vs $O(n^2)$ for suffix tries. The suffix tree can be constructed by first constructing the suffix trie and compressing paths. However, this construction algorithm uses $O(n^2)$ space. There are algorithms that construct the suffix tree without constructing the suffix trie first, and require only $O(n)$ working space [6–9].

2.3. Position Heaps

The position heap data structure was first described in [2].

Definition 4 (Position Heap)**.** *Position heap for a text T is created by inserting suffixes of T into a trie by the following two rules:*

(1) A shorter suffix is inserted before a longer suffix.

(2) While inserting a suffix s in a position heap we insert shortest prefix of s that is not already in the trie.

Let $T = bananababa$. The position heap for $T' = ananababa$ is shown in Figure 2.6 where the suffix *bananababa* is being inserted.

```
10 9 8 7 6 5 4 3 2 1
 b a n a n a b a b a
```

ban is the shortest prefix of suffix *bananababa* that is not already in the position heap, so we insert *ban* in the trie and label the node 10. See Figure 2.6. Its easy to see that insertion into position heap is linear in length of the string being inserted.

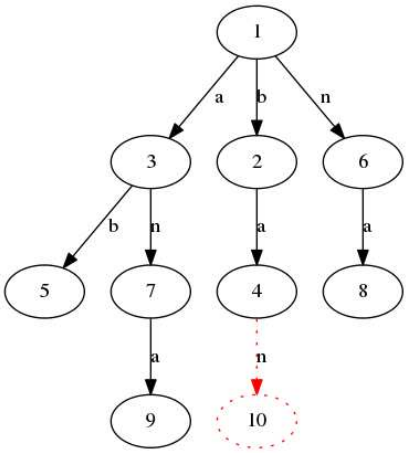


Figure 2.6. Inserting *bananababa* into the position heap.

Note that positions in the tree satisfy the min heap property (label of root is smaller than or equal to labels of its children) hence the name position heap.

To search for all the occurrences of some pattern P, we find longest prefix X of P that is in the position heap. Then there are two cases,

(1) $P = X$: In this case all the nodes in the subtree below X are labeled with occurrences of P. All the nodes on the path leading up to X are potential occurrences and need to be checked one by one to confirm if P really occurs at that location.

(2) $P > X$: In this case all the nodes on the path leading up to X are labeled with potential occurrences and we check each of them individually to see if its really an occurrence of P.

We give example of searching in a position heap below. Let $P = ba$. See Figure 2.7. In this case X is ba and so 4,10 are occurrences of P. 1, 2 are potential occurrences of P. We lookup the positions 1, 2 and see that P occurs only at position 2. So P occurs at $\{2, 4, 10\}$.

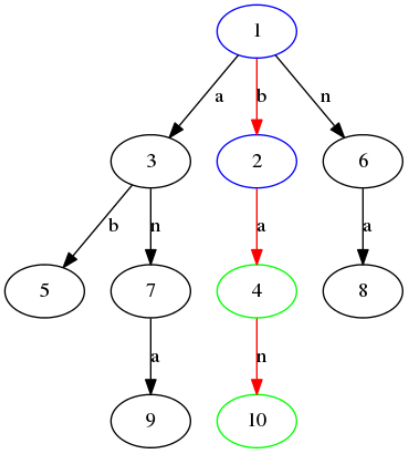


FIGURE 2.7. The positions 4, 10 are definite occurrences of ba and 1,2 are potential occurrences

Since the height can be at most $2h(T)$ and we insert n suffixes into the position heap construction takes $O(nh(T))$ time.

While searching for a pattern P we find longest prefix X of P that is in the position heap. At each of $O(|X|)$ ancestors of X we perform an $O(m)$ time operation to check if P occurs at that position. The number of ancestors is $\min\{m, h(T)\}$. So the search algorithm takes $O(m \min\{m, h(T)\} + |occ(P)|)$ time.

In a position heap, each suffix has a node corresponding to it and a parent. There are n suffixes so the space requirement is at least $2n$ integers.

Any σ-ary tree can be represented by a binary tree. Each node stores data corresponding to that node, a pointer to the leftmost child and a pointer to the right sibling. This requires $3n$ integers space (assuming data stored at each node is an integer). This representation doesn't save us any space for trees in general. For position heaps we know that data corresponding to each node is a position of a suffix. The positions of the suffixes are integers from 1 to n. This allows us to represent the position heap in only $2n$ integers. We store the left child and right sibling pointers for a node with data i at index i in the left child array and the right sibling array respectively.

Succinct representation of the position heap of $T = bananababa$ is shown below.

Index	1	2	3	4	5	6	7	8	9	10
Left child array	3	4	5	10	0	8	9	0	0	0
Right sibling array	0	6	2	0	7	0	0	0	0	0

We can do even better with position heaps. They can be built in $O(n)$ time and the search time can be improved to $O(m + |occ(P)|)$. This requires more space. Interested readers should refer to the original paper [2]. There also is online construction algorithm for position heaps [10].

2.4. Suffix Arrays

All the data structures above used pointers and hence a lots of storage space was required. Suffix arrays (SA) don't use any pointers, so they are a space efficient alternative to the above data structures. Searching in suffix arrays takes $O(m \log n)$ time instead of $O(m)$. We can speed up searches to take only $O(m + \log n)$ time, but that increases the space requirements.

Suffix arrays are constructed by sorting the suffixes of a string T. We store only the positions of these suffixes in an array of integers, so the space requirement is just n integers. Since comparison of two suffixes takes $O(n)$ time in the worst case, suffix arrays can be built in $O(n^2 \log n)$ time. We can reduce this to $O(n \log n)$ time using extra space. By traversing suffix tree in inorder the suffix array can be constructed in $O(n)$ time. There are many algorithms to build suffix array they fall into different classes according to algorithmic tricks they use [11].

Searching in suffix array for patter P of length m can be performed in $O(m \log n)$ time using binary search, where each comparison takes $O(m)$ time.

Definition 5 (Bucket)**.** *Let a k-bucket be a set of strings sorted using their k-prefixes as the sort keys.*

Let A be an algorithm that produces a $2i$-bucket of set $\mathcal{S}$ given its i-bucket in $O(n)$ time where $n = |\mathcal{S}|$. Then we can use A to build the suffix array in $O(n \log n)$ time as follows,

(1) Construct a 1-bucket of suffixes. This takes $O(n \log n)$ time.

(2) Then use A to get a $2i$-bucket from i-bucket for $i \in \{1, 2, 4, \ldots, 2^{\lceil \log_2 n \rceil}\}$

We have to use A only $\log n$ times. This shows we can successfully use A to build SA in $O(n \log n)$ time. The authors give an algorithm A in the paper [12]. It requires an extra array of n integers which brings space requirement of the algorithm to $2n$ integers.

CHAPTER 3

Space Efficient Representation of Positional de Bruijn Graphs

The positional de Bruijn graphs are useful in bioinformatics. They were introduced in [13].

A k-mer is simply a string of length k. Given a string R we can get $|R| - k + 1$ k-mers from it by extracting the string of length k corresponding to each position $0 \leq i \leq |R|-k+1$. A tuple $(R[i..i+k], i)$ corresponding to a k-mer and its position within R is called positional k-mer.

Definition 6 (Positional de Bruijn Graphs)**.** *Given a set $\mathcal{R}$ of strings the positional de Bruijn graph is $G_{k,\Delta} := (V, E)$ where $V :=$ positional k-mers extracted from each string $R \in \mathcal{R}$. There is an edge connecting $(u, p), (v, q) \in V$ if $u[2..k] = v[1..k-1]$ and $|p-q| \leq \Delta$.*

The naive representation of this graph is space consuming. For a set of strings $\mathcal{R}$ we give,

(1) A representation of this graph that uses only $2|V|$ integers space other than space required by $\mathcal{R}$.

(2) An algorithm to build it in $O(k|V|)$ time from $\mathcal{R}$.

(3) An algorithm to find the neighbors of $v \in V$ in $O(|\sigma|k \log |V|)$ time for alphabet σ.

We can represent each $v \in V$ as a tuple (i, j) of integers, where i is the index of the k-mer and j is the index of corresponding string $R \in \mathcal{R}$.

To build the data structure we sort the tuples using k-mer (i, j) as primary key and i as secondary key. Since the alphabet size is fixed we can do sorting by the primary key in

$O(k|V|)$ time using the radix sort. Similarly sorting by the secondary key can be done in $O(k|V|)$ time.

To search for the neighbors of a $(u, p) \in V$ we first binary search for strings from $\{c + u[1..k-1], u[1..k] + c | c \in \sigma\}$ in the data structure and then binary search for p to find the positions that are within Δ distance of p. This takes $O(|\sigma| k \log |V|)$ time.

CHAPTER 4

Sloppy Suffix Arrays

Constructing the a suffix array without extra space requires $O(n^2 \log n)$ time. Searching in a suffix array requires $O(m \log n)$ time with extra space this can be improved to $O(m + \log n)$ time [12].

In this section we present a new data structure called the *Sloppy Suffix Arrays (SSA)* that requires only n integers of space and can be constructed in $O(d(T)n \log n)$ time. For a randomly generated string expected time complexity of the construction algorithm is $O(n \log^2 n)$. We also present some experimental results about the SSA. First we define some concepts that are needed to talk about the SSA.

Definition 7 (Bucket)**.** *Let a k-bucket be a set of strings sorted using their k-prefixes as the sort keys.*

We also say that the depth of a k-bucket is k. For example, the 2-bucket of the set of suffixes of *banana* is,

```
6 a |
2 an|ana
4 an|a
1 ba|nana
5 na|
3 na|na
```

Let **BuildBucket**(k, S) be an algorithm that builds the k-bucket from an unsorted set of strings S in place. Clearly, **BuildBucket**(k, S) takes $O(k|S| \log |S|)$ time to build k-bucket from S.

Definition 8 (Block)**.** *For a k-bucket B let a p-block be a maximal slice $B[i..j]$ that shares a common k-prefix and has size at most p.*

A bucket is k-perfect iff it is a k-bucket and has no i-block for $i > k$. For example, the 2-bucket of suffixes of *banana* is 2-perfect but not 1-perfect. Let **IsPerfect**(k, B) be an algorithm that checks if a bucket B is k-perfect. Obviously, it takes $O(k|B|)$ time to check if B is k-perfect.

4.1. Construction Algorithm

Now we give an algorithm to construct Sloppy Suffix Array (SSA) from T.

Definition 9. *The SSA of a string T is the $d(T)$-perfect bucket built from the set of suffixes of T.*

Algorithm: BuildSSA

Input: Text T

Output: SSA(T) the sloppy suffix array of T.

1: $B \leftarrow$ Set of suffixes of T
2: $k \leftarrow 1$
3: $B \leftarrow$ **BuildBucket**(k, B) // In place sort.
4: **while** $k \leq n$ **do**
5: **if IsPerfect**(k, B) **then**
6: **return** B
7: **else**
8: $k \leftarrow 2k$
9: **end if**
10: **end while**

Notice that when strings fall into a k-block they will stay together in $2k$-bucket. From the definition of $d(T)$ we can see that when the set of suffixes of a string is sorted by $d(T)$ characters each string falls into a $d(T)$-block. After sorting by at least $d(T)$-prefixes B is $d(T)$-perfect. Hence, the algorithm terminates and produces the SSA of T.

This gives us runtime of the construction algorithm. The time complexity is

$$\begin{aligned}
&\sum_{k\in\{1,2,4,\ldots,d(T)\}} kn\log n + kn \\
&= (n\log n + n) \sum_{k\in\{1,2,4,\ldots,d(T)\}} k \\
&= (2d(T)-1)(n\log n + n) \\
&= O(d(T)n\log n)
\end{aligned}$$

We use in-place sorting algorithm so only $O(1)$ extra space is used at any point that gives space complexity of n integers. The expected value of $d(T)$ is $O(\log n)$ on a randomly generated string, so the expected runtime of the algorithm is $O(n\log^2 n)$.

4.2. Search Algorithm

Search for a pattern P of length m in SSA can be performed in $O(m\log n)$ time. Note that this is also the time complexity of searching in the suffix arrays when no extra space or auxiliary data structures are allowed. We give recursive version of the algorithm because it is easy to analyze. It can easily be converted into a iterative version with the same complexity, which will allow us to save stack space used for function calls. **FindSSA** finds all the occurrences of string P in T given SSA(T).

Algorithm: FindSSA

Input: Pattern P of length m

Output: All the occurrences of P in T.

1: $S_1 =$ **BinarySearch**(P) // Returns all the occurances of P.

2: **if** $|P| \leq d(T)$ **then**

3: **return** S_1

4: **else**

5: $q = \min\{d(T), |P| - d(T)\}$

6: $S_2 =$ **FindSSA**$(P[q+1\,..\,|P|])$

7: $S_3 = \{x - q | x \in S_2\}$

8: **return** $S_1 \cap S_3$

9: **end if**

When $m > d(T)$ is not a multiple of $d(T)$ we search for the last $d(T)$ characters of P to guarantee that **BinarySearch** returns at most $d(T)$ hits. No string of length $d(T)$ occurs more than $d(T)$ times so **BinarySearch** never returns more than $d(T)$ occurrences. Each binary search for a string of length $d(T)$ takes $O(d(T) \log n)$ time ($\log n$ comparisons; each comparison takes $d(T)$ time).

Using a naive algorithm, the intersection of two sets of size at most $d(T)$ can be done in $O(d(T) \log d(T))$ time (We can sort one set and binary search elements of the other in the sorted array). If we use indices as the secondary key while building SSA, the search algorithm will always return a sorted set. A simple Merge like algorithm can intersect sorted sequences in $O(d(T))$. Note that we can not use compressed representation of integers with intersection

algorithm if we want to take advantage of processor instructions. Fast implementation of the algorithms is of great practical interest.

The running time of the above algorithm is given by following recurrence,

$$\text{Time taken} = \text{Recursive call} + \text{Binary Search} + \text{Set Intersection}$$

$$\begin{aligned} T(m) &= T(m - d(T)) + O(d(T)\log n) + O(d(T)\log d(T)) \\ &= T(m - d(T)) + O(d(T)\log n) \\ &= \sum_{i=1}^{m/d(T)+1} d(T)\log n \\ &= \frac{m + d(T)}{d(T)} d(T)\log n \\ T(m) &= O(m\log n) \end{aligned}$$

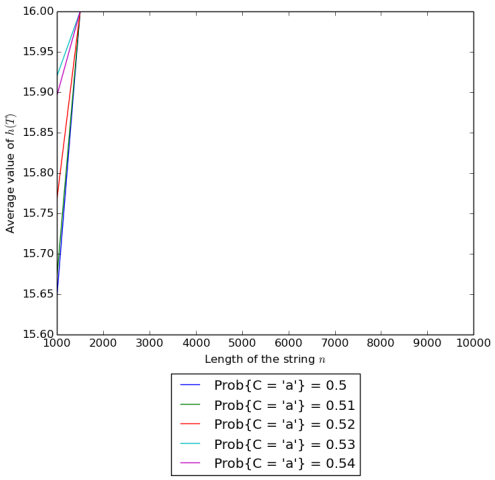
To report all the occurrences of P we need extra $O(|occ(P)|)$ time so the time complexity is $O(m\log n + |occ(P)|)$. Search time in SSA is independent of $d(T)$.

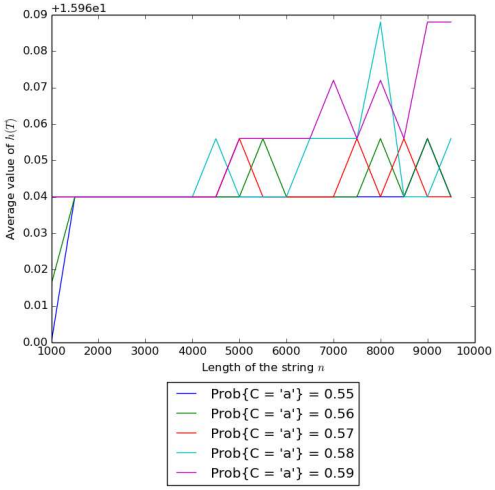
4.3. Experiments

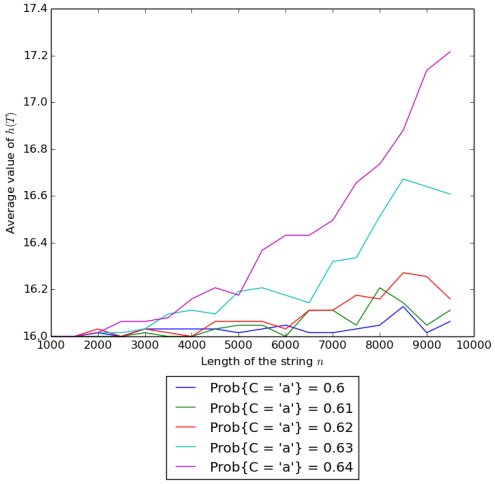
We generated random binary strings of lengths ranging from 1000 to 9500 in increments of 500. For each length and for each $p \in 0.5, 0.51, \ldots, 0.99$ we picked characters from distribution $p = Prob\{C =' a'\}$ and $1 - p = Prob\{C =' b'\}$ and generated 1000 strings. These strings were used to calculate the average value of $d(T)$ for a string of given length and p.

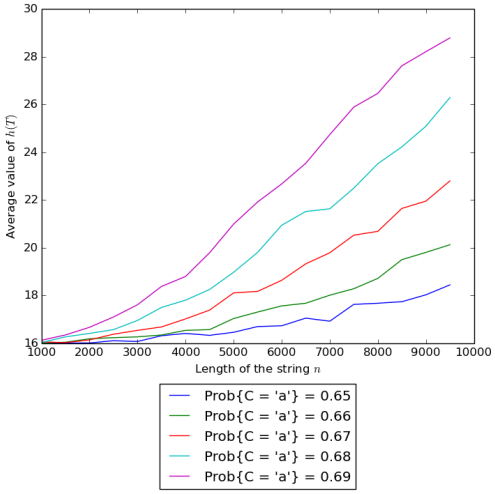
We can see from the plots that the value of $d(T)$ barely exceeds 10% of the string length even for highly skewed distributions $0.9 < p < 0.97$. Only when $p \geq 0.98$ $d(T)$ takes bigger

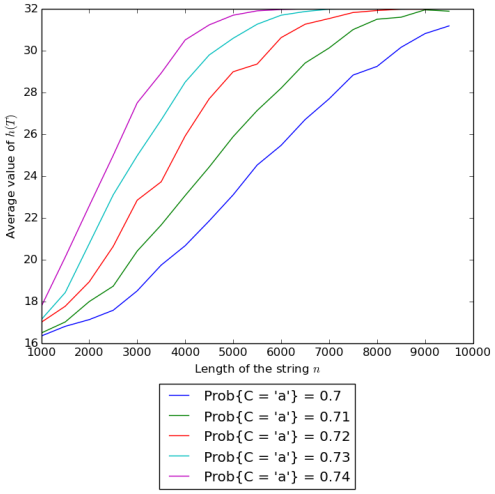
values. Such strings are unlikely to arise in practice because they contain little information. A reasonably complex organism is likely to have complex DNA and therefore small $d(T)$ value. For E.coli genome of length 4707963, $d(T)$ is merely 32. It is important to observe that long repeated substrings do not change the value of $d(T)$ by a lot unless they occur too many times. Hence, we can expect the construction algorithm to be fast in practice.

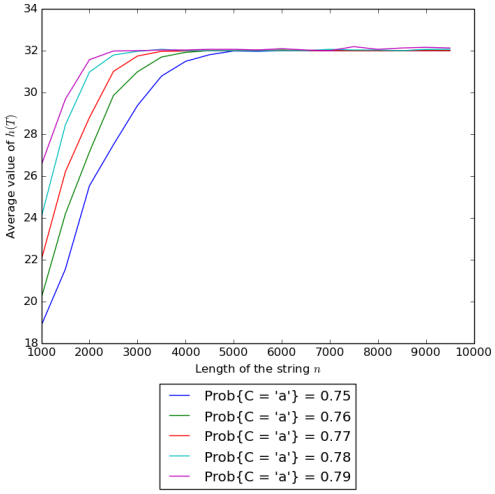

Average value of $h_i(T)$
Length of the string n
Prob{C = 'a'} = 0.5
Prob{C = 'a'} = 0.51
Prob{C = 'a'} = 0.52
Prob{C = 'a'} = 0.53
Prob{C = 'a'} = 0.54

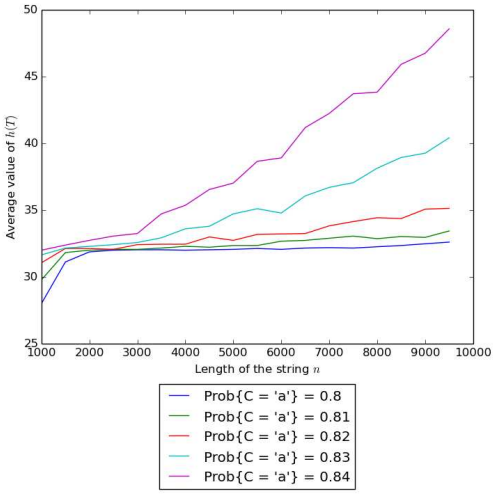

+1.596e1
Average value of $h_i(T)$
Length of the string n
Prob{C = 'a'} = 0.55
Prob{C = 'a'} = 0.56
Prob{C = 'a'} = 0.57
Prob{C = 'a'} = 0.58
Prob{C = 'a'} = 0.59

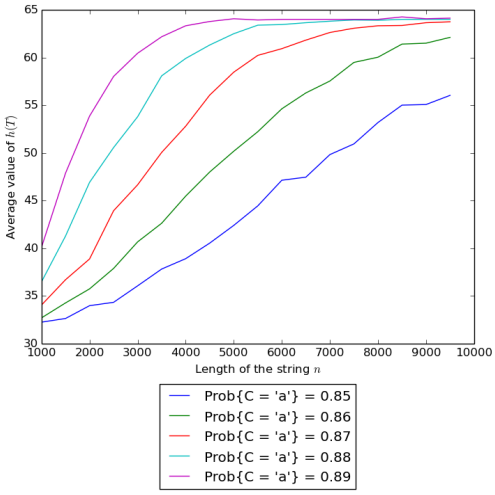

Average value of $h_i(T)$
Length of the string n
Prob{C = 'a'} = 0.6
Prob{C = 'a'} = 0.61
Prob{C = 'a'} = 0.62
Prob{C = 'a'} = 0.63
Prob{C = 'a'} = 0.64

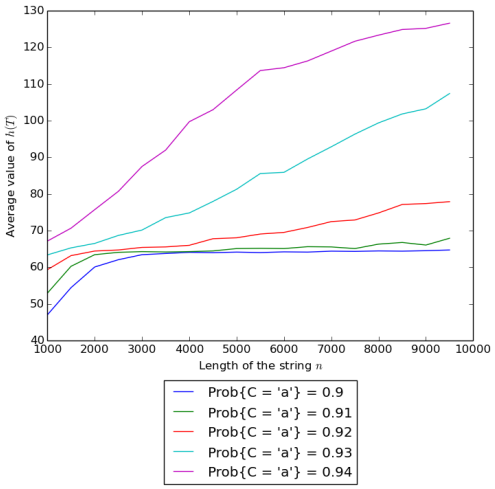

Average value of $h_i(T)$
Length of the string n
Prob{C = 'a'} = 0.65
Prob{C = 'a'} = 0.66
Prob{C = 'a'} = 0.67
Prob{C = 'a'} = 0.68
Prob{C = 'a'} = 0.69

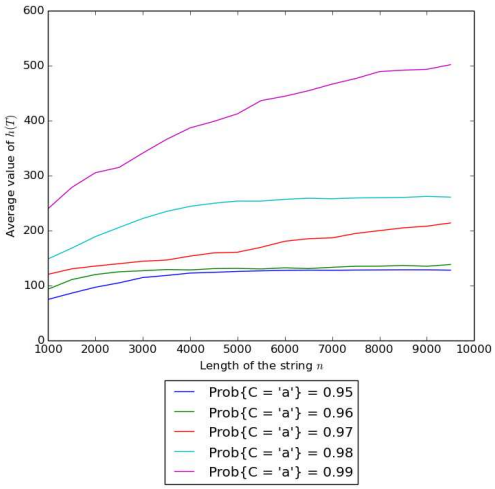

Average value of $h_i(T)$
Length of the string n
Prob{C = 'a'} = 0.7
Prob{C = 'a'} = 0.71
Prob{C = 'a'} = 0.72
Prob{C = 'a'} = 0.73
Prob{C = 'a'} = 0.74


Average value of $h_i(T)$
Length of the string n
Prob{C = 'a'} = 0.75
Prob{C = 'a'} = 0.76
Prob{C = 'a'} = 0.77
Prob{C = 'a'} = 0.78
Prob{C = 'a'} = 0.79


Average value of $h(T)$
Length of the string n
Prob{C = 'a'} = 0.8
Prob{C = 'a'} = 0.81
Prob{C = 'a'} = 0.82
Prob{C = 'a'} = 0.83
Prob{C = 'a'} = 0.84


Average value of $h(T)$
Length of the string n
Prob{C = 'a'} = 0.85
Prob{C = 'a'} = 0.86
Prob{C = 'a'} = 0.87
Prob{C = 'a'} = 0.88
Prob{C = 'a'} = 0.89


Average value of $h(T)$
Length of the string n
Prob{C = 'a'} = 0.9
Prob{C = 'a'} = 0.91
Prob{C = 'a'} = 0.92
Prob{C = 'a'} = 0.93
Prob{C = 'a'} = 0.94


Average value of $h(T)$
Length of the string n
Prob{C = 'a'} = 0.95
Prob{C = 'a'} = 0.96
Prob{C = 'a'} = 0.97
Prob{C = 'a'} = 0.98
Prob{C = 'a'} = 0.99

CHAPTER 5

Super Sloppy Suffix Arrays

To construct the SSA we sorted suffixes of T by first $d(T)$ characters. The construction algorithm took $O(d(T)n \log n)$ time. We give new data structure that has faster construction algorithm that takes $O(\hat{d}(T)n \log n)$ time by being *sloppier*. We call this data structure *Super Sloppy Suffix Array (SuperSSA)*. Searching in SuperSSA for a pattern of length m takes $O(m \log^2 n)$ time.

Construction of SuperSSA takes space for $n + \log n + O(1)$ integers but to bound the runtime of the search algorithm we need to augment SuperSSA with lookup table called Inverse SuperSSA which requires extra n integers. Hence, the total space requirement is $2n + \log n + O(1)$ integers. In the following sections we give algorithms to construct and search in SuperSSA.

5.1. Construction Algorithm

To construct SuperSSA we sort the set B of all suffixes of the text by the first k (initially $k = 1$) characters. Then remove groups of suffixes that form k-blocks into a k-bucket B_k. Note that this B_k will be perfect. This leaves us with fewer suffixes, which are the sorted by first $2k$ characters during the next iteration of the algorithm. We repeat the process until we run out of suffixes to sort. This procedure is outlined below. For exact implementation details see the pseudocode **BuildSuperSSA**.

(1) Let $B =$ Suffixes of T

(2) $k \leftarrow 1$

(3) While B is not empty

(a) Sort B by first k characters.

(b) Construct a perfect k-bucket from suffixes that form k-blocks by removing the blocks from B. See Figure 5.1

(c) $k \leftarrow 2k$.

Implementation of this algorithm needs two important subroutines after sorting the suffixes by the first k characters, one to find k-blocks and another to construct a perfect k-bucket using those blocks. Moreover, these subroutines have to work in constant extra space.

5.1.1. Finding Blocks. The k-blocks can be found by starting at the top of the k-bucket and scanning until a suffix s, that has a different k-prefix than starting suffix is found. If there are less than or equal to k suffixes between s and the starting point, we have found a k-block. Otherwise the location of s becomes new starting point and scanning continues. This is what **FindBlock** does and returns indices of start and end of the block. We need 2 integers to mark out a block one for the beginning and one for the end of the block. To avoid using space to mark all blocks, we move a block to its bucket as soon as it is found. This subroutine uses $O(1)$ extra space. Since the **FindBlock** eventually has to go through all the suffixes in bucket B, and compare first k characters for each the time complexity is $O(k|B|)$.

Algorithm: FindBlock

Input: A k-bucket B, k, index to start scanning t

Output: Start index, end index, (i, j) of k-block that starts at or after t. If no k-block is found return $(B.end + 1, B.end + 1)$.

1: $i \leftarrow t$

2: $j \leftarrow t$

```
3: while i ≤ B.end do
4:     while k-prefix(B_i) = k-prefix(B_j) do
5:         j ← j + 1
6:         if j = B.end then
7:             if j − i + 1 ≤ k then
8:                 return (i, j)
9:             end if
10:        end if
11:    end while
12:    if j − i ≤ k then
13:        return (i, j − 1)
14:    end if
15:    i ← j
16: end while
17: return (B.end + 1, B.end + 1)
```

5.1.2. Moving Blocks. Let B be a k-bucket. We want to find k-blocks in it and build a perfect k-bucket B_k without using extra space. This can be achieved by moving all the k-blocks to the top of B. We set $B_k.start$ to the top of B and keep updating $B_k.end$ as we move k-blocks. Once we have moved all the k-blocks to the top of B we have built B_k and suffixes from $B_k.end + 1$ to $B.end$ is the new B. This uses 2 integers for marking out the boundaries of B_k. There can be at most $\log n$ perfect buckets so overall $2 \log n$ integers will be required. See Figure 5.2 for working of the algorithm.

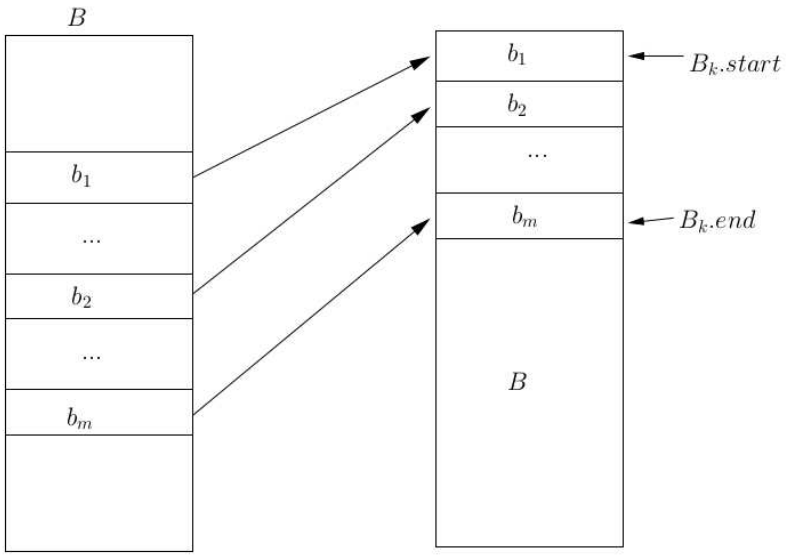


FIGURE 5.1. Constructing a perfect k-bucket B_k using k-blocks $b_1, b_2, \ldots, b_m$

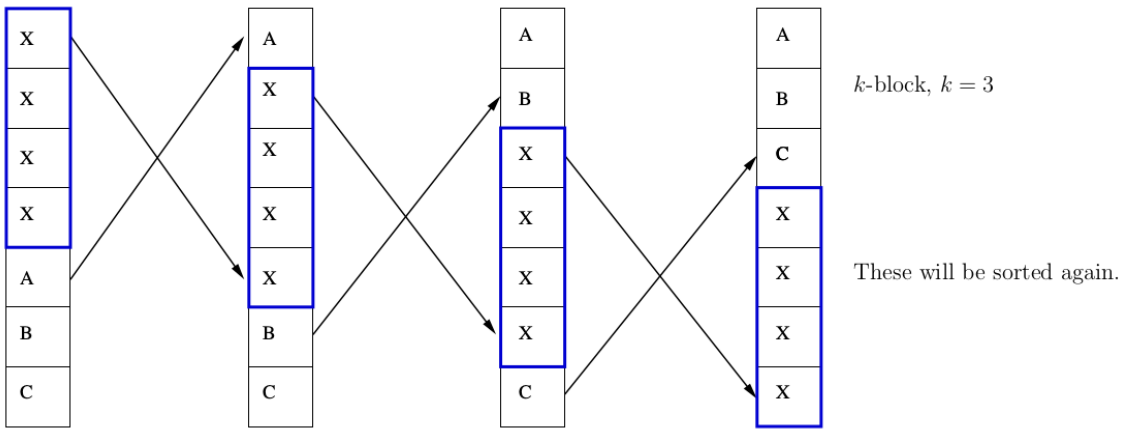


FIGURE 5.2. Working of MoveBlock

Algorithm: MoveBlock

Input: A k-bucket B, (i, j) the start index, the end index of block to be moved.

Output: Returns modified B where $B[i..j]$ is moved to the front of B.

1: $t \leftarrow B.start$

2: **while** $i \leq j$ **do**

3: Swap $B[t]$ with $B[i]$

4: $i \leftarrow i + 1$

5: $t \leftarrow t + 1$

6: **end while**

7: $B_k.start \leftarrow t$

8: **return** B

One issue is some suffixes that don't form k-blocks will become unsorted. But since those suffixes were not in any k-block they won't get into a k-bucket. This is not a problem because they will be sorted by $2k$ characters in the next phase of the algorithm and fall into their proper places.

5.1.3. Building Inverse SuperSSA. To make intersection faster in cases where there are too many occurrences we can we construct an index. This index answers where a suffix occurs in SuperSSA in $O(1)$ time. This index is called **Inverse Super Sloppy Suffix Array (ISuperSSA)**. Its defined as

$$ISuperSSA[SuperSSA[i]] = i, \quad \forall 1 \leq i \leq n.$$

We can store it in an array of n integers. And its definition gives us direct construction algorithm.

5.1.4. Analysis. It is easy to see that all the subroutines used only $O(1)$ extra space. Now we analyze the time complexity. **BuildBucket**(k, B) takes $O(k|B| \log |B|)$ time. **FindBlock** touches every suffix in B at constant number of times because once it finds a k-bucket $B[i..j]$ after index t next time it is called with index $j + 1$ as starting point so suffixes between t

and j are not touched again. **FindBlock** takes total $O(k|B|)$ time. Similarly **MoveBlock** touches the suffixes only $O(1)$ times and takes $O(|B|)$ time overall. The total time consumed between Step 13 and Step 22 is $O(k|B| + |B|)$ see pseudocode for **BuildSuperSSA**. **BuildBucket** which involves sorting dominates the runtime.

If a suffix $T[i..n]$ falls into bucket B_k then it was involved in $1 + 2 + 4 + \ldots + k = O(k)$ comparisons. It will fall into B_k if its k-prefix i.e. $T[i..k]$ occurs more than $k/2$ times but $\leq k$ times. In other words, $T[i..n]$ falls into bucket k if $k/2 < d(i, T) \leq k$, because that is when it forms a k-block. $T[i..n]$ is inspected upto only $O(d(i, T))$ characters. Average suffix comparison time is $O(\hat{d}(T))$. Which gives the runtime of $O(\hat{d}(T)n \log n)$ for SuperSSA construction algorithm . Since $\hat{d}(T) < d(T)$ building SuperSSA is faster than building SSA. Following calculation is another way of proving runtime of the construction algorithm. We use B_k for size of k-bucket B_k.

$$
\begin{aligned}
T(n) &= \sum_k k(n - \sum_{i<k} B_i) \log(n - \sum_{i<k} B_i) \\
&\leq \log n \sum_k k(n - \sum_i B_{i<k}) \\
&= \log n \sum_k k \sum_{i \geq k} B_i \\
&\leq \log n \sum_k 2kB_k \\
&= 2 \log n \sum_i d(i, T) \\
&= O(\hat{d}(T)n \log n)
\end{aligned}
$$

5.1.5. Pseudocode.

Algorithm: BuildSuperSSA(T)

Input: Text T

Output: SuperSSA(T)

1: $B \leftarrow$ Suffixes of T

2: $B.start \leftarrow 1$

3: $B.end \leftarrow n$

4: **for** $k \in \{1, 2, 4, \ldots, n\}$ **do**

5: $\quad B_k.start = B.end + 1$

6: $\quad B_k.end = B.end + 1$

7: **end for**

8: $k \leftarrow 1$

9: **while** $B \neq \phi$ **do**

10: $\quad B \leftarrow$ **BuildBucket**(k, B)

11: $\quad B_k.start \leftarrow B.start$

12: $\quad t \leftarrow B.start$

13: $\quad$ **while** $t \neq B.end + 1$ **do**

14: $\quad\quad (i, j) \leftarrow$ **FindBlock**(k, B, t)

15: $\quad\quad$ **if** $i = j = B.end + 1$ **then**

16: $\quad\quad\quad$ GOTO Step 23

17: $\quad\quad$ **end if**

18: $\quad\quad B \leftarrow$ **MoveBlock**(i, j, B)

19: $\quad\quad B.start \leftarrow B.start + (j - i) + 1$

20: $B_k.end \leftarrow B.start - 1$

21: $t \leftarrow t + j + 1$

22: **end while**

23: $k \leftarrow 2k$

24: **end while**

25: **return** $\{B_1, B_2, B_4, \ldots, B_n\}$

5.2. Search Algorithm

To search in SuperSSA we start with prefix of P of size i (initially set to 1) and search for it in bucket B_i. If it doesn't occur in B_i we we search for prefix of size $2i$ in bucket B_{2i}. Once a prefix $|x|$ is found we recursively find occurrences of the $y := suffix(|x| + 1, P)$. We subtract $|x|$ from each element of $occ(y)$ and find *intersection* of the resulting set with $occ(x)$. This gives us all the occurrences of P. For example, while searching for $P = abcdefgh$ if $occ(abc) = \{3, 11, 25\}$ and $occ(defgh) = \{5, 27, 99\}$ then $occ(P) = \{3, 25\}$.

5.2.1. Analysis. First we analyze the search algorithms complexity. For a string of length m we binary search in at most $\log n$ buckets of size at most n, so it clearly takes $O(m \log n)$ time for each of the $\log n$ buckets. This gives us runtime of $O(m \log^2 n)$.

Now let us analyze the complexity of intersection. Let A and B be the sets of integers (in our case these are the sets of occurrences of the substrings of the pattern) we can sort A and binary search for occurrences of $b \in B$ and output b if it is found in A. This $O(|A| \log |A| + |B| \log |A|)$ time to find the intersection. We can use this algorithm for intersection as long as the number of occurrences of the string is proportional to its length. But this is not true when searching in SuperSSA for string x that does not occur in any

k-bucket for $k \leq |x|$. In this case, x can have more than $|x|$ occurances. We can get arround this problem by using ISuperSSA.

Intersection Algorithm using ISuperSSA Let X be the set of occurrences of $P[1..i]$ a small set and Y be set of occurrences of the part of the pattern under consideration i.e $P[i+1..m]$ a large set. Y is represented as a set of $\log n$ intervals i.e. lower and upper bound locations of $P[i+1..m]$ in each bucket as shown in Figure 5.3.

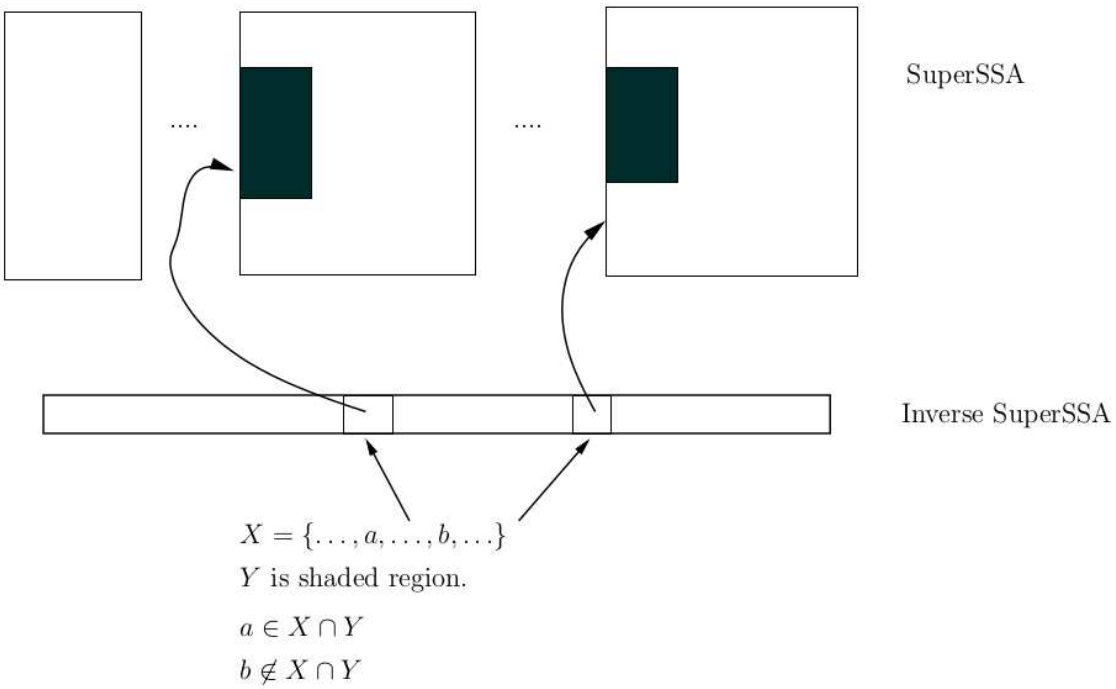


FIGURE 5.3. Intersection using Inverse SuperSSA

(1) For $x \in X$

(a) $j \leftarrow ISuperSSA[x+i]$

(b) If j doesn't fall within any of the intervals that specify Y remove it from X.

This algorithm takes $O(|X| \log n)$ time. The complexity of search for pattern of length m in SuperSSA of a string of length n is,

Search time P = Search time for suffix of P + Search time for prefix of P of size m_i

+ Time for intersection of occurances

$$T(m) = T(m - m_i) + O(m_i \log^2 n) + O(m_i \log n)$$

$$= \sum_i m_i O(log^2 n)$$

$$= O(m \log^2 n)$$

To report all the occurrences of P we need extra $O(|occ(P)|)$ time. Hence, we can search in SuperSSA in $O(m \log^2 n + |occ(P)|)$ time.

5.2.2. Pseudocode.

Algorithm: $FindSuperSSA$

Input: Pattern P

Output: All the occurrences of P

1: $t \leftarrow 1$

2: **while** $t \leq d(T)$ **do**

3: **if** $t \leq |P|$ **then**

4: $X \leftarrow BinarySearch(P[1..t], B_t)$

5: **else**

6: **return** $\bigcup_{l>|P|} BinarySearch(P, B_l)$

7: **end if**

8: **if** $X = \phi$ **then**

9: $t \leftarrow 2t$

10: **continue**

11: **else**

12: **break**

13: **end if**

14: **end while**

15: **return** $X \cap FindSuperSSA(P[t+1..m])$

Bibliography

[1] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[2] A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo, "Position heaps: A simple and dynamic text indexing data structure," *Journal of Discrete Algorithms*, vol. 9, no. 1, pp. 100–121, 2011.

[3] R. M. M. Andrzej Ehrenfeucht, *Chapter 30, String Searching, Handbook of data structures and applications*. CRC Press, 2004.

[4] B. Langmead, "Suffix tries and trees,"

[5] "To trie or not to trie," 2014.

[6] P. Weiner, "Linear pattern matching algorithms," in *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pp. 1–11, IEEE, 1973.

[7] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM (JACM)*, vol. 23, no. 2, pp. 262–272, 1976.

[8] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[9] M. Farach, "Optimal suffix tree construction with large alphabets," in *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pp. 137–143, IEEE, 1997.

[10] G. Kucherov, "On-line construction of position heaps," *Journal of Discrete Algorithms*, vol. 20, pp. 3–11, 2013.

[11] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys (CSUR)*, vol. 39, no. 2, p. 4, 2007.

[12] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[13] R. Ronen, C. Boucher, H. Chitsaz, and P. Pevzner, "Sequel: improving the accuracy of genome assemblies," *Bioinformatics*, vol. 28, no. 12, pp. i188–i196, 2012.