Manber and Myers Suffix Array

Hannes Kruse : 70451368
Mohamed Amin Hmaied : 70461945
Achraf Benhmed : 70466187

15.06.2020

University : Ostfalia University of Applied Science

Contents

1 Introduction

Exact string matching is a basic step used by many algorithms in computational biology: Given a pattern $P = P[1 :: m]$, and a text $S = S[1 :: n]$, we want to find all occurrences of P in S.
This can be done with exact string matching algorithms in time $O(m + n)$. These algorithms perform form of text preprocessing. In this way it is often possible to exclude or ignore parts of the text while searching. But as long as $m = O(1)$, the running time for this class of algorithms cannot be $O(n)$.
In order to achieve a sublinear search time, we have to preprocess the text. This operation is useful in scenarios where the text is relatively constant over time, and we will search for many different patterns.
To resolve this problem we will need data structures that offer a certain structure and methodology when trying to resolve the problem:

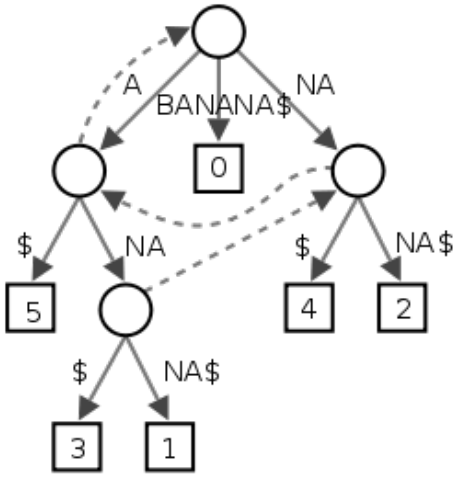
- Suffix tries: a naive structure offering a simple tree to query a string
- Suffix trees: a compressed trie containing all the suffixes of the given text as their keys and positions in the text

- Suffix array: a conceptually simple data structure for on-line string searches, saving suffix positions in arrays

2 Historic Context: Suffix Trees

In this paper we will mainly discuss suffix arrays, their purpose and logic as well as evaluating the efficiency of suffix arrays. But we first need to understand suffix trees. All suffixes of a text are stored in a suffix tree, starting with the shortest suffix, the empty string, up to the complete text as a single suffix. The substring can therefore be found in the tree without first having to search for the string preceding the substring in the text.
Suffix trees work efficiently because the length of the text and the duration of the substring search only increases linearly with the length of the search string.

2.1 Building Suffix Trees



Suffix trees have been applied to fundamental string problems such as finding the longest repeated substring, finding all squares or repetitions in a string, computing substring statistics, approximate string matching, and string comparison. They have also been used to address other types of problems such as text compression, compressing assembly code, inverted indices, and analyzing genetic sequences.
Let $T = T[1..n]$ be a text of length n over a fixed alphabet Σ. A suffix tree for T is a tree with n leaves and the following properties:

a. Every internal node other than the root has at least two children.

b. Every edge is labeled with a nonempty substring of T.

c. The edges leaving a given node have labels starting with different letters.

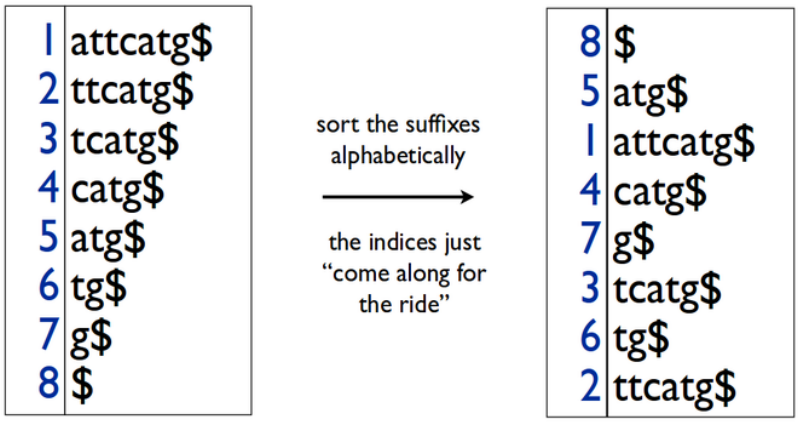
d. The concatenation of the labels of the path from the root to leaf i spells out the ith suffix $T[i...n]$ of T. We denote $T[i...n]$ by T_i.

2.2 Differences between Suffix Trees and Suffix Arrays

Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms and improved cache locality. A recent suffix tree implementation requires 15-20 Bytes per character. For suffix arrays, as few as 5 bytes are sufficient (with some tricks).

3 Idea behind Suffix Arrays

Suffix arrays were introduced by Manber and Myers in 1989 (and published in 1993). The idea is based on building a sorted array of all suffixes of a given string and its definition is similar to that of a suffix tree which is a compressed trie of all suffixes of the given text.
The suffix array is based on a lexicographic sorting of all the possible suffixes of a certain text. All suffixes starting with the same character will be collected and grouped together in a bucket. This first step is explained in the following figure:



3.1 Space Efficiency and Longer Preprocessing Times

Udi Manber and Gene Myers introduced suffix arrays as an efficient substitute of suffix trees. The sorting of suffixes in suffix arrays can be done in $O(n\ log\ n)$ time worst case and $O(n)$ space which is a better solution and a revolutionary improvement at the time.

4 Suffix Array Creation

Manber and Myers describe the creation of a suffix array using essentially 5n bytes of storage and a time-complexity of $O(n\ log\ n)$. The output of the algorithm is stored in two integer arrays POS and PRM. The array POS holds the index of a suffix at a given rank, that is $POS[i]$ is the ith smallest suffix at the lexicographic rank i. PRM is the inverse array of POS and describes the rank of a suffix at index i. Three additional arrays are used to hold supplemental data. The integer array $Count$, which is used in one of the sorting stages, holds the offset of the next free index in a sorting bucket. That is $Count[i]$ + starting position of a bucket = first free index in the bucket. Two boolean arrays BH and $B2H$ hold true values if a suffix at a position i is the smallest in its BH (first stage sort) or $B2H$ (h stage sort) bucket. Each array is sized equally to the length of the input text.

4.1 Sorting

Sorting is done in two distinct stages. The first stage consists of a bucket sort comparing only the first character of each suffix. Inductively the second stage sorts the suffixes in $log_2(n)$ iterations, doubling the length h of the compared prefix each time. Thus, after h is greater than or equal to the length of the input text, all suffixes are sorted according to the $\leq_H$-order.
The H-order is described as a lexicographic sorting order. A string is lexicographical smaller than some other string if one or more of its characters are smaller according to their position in the alphabet. The greater and equal relations are defined likewise.

4.2 Case Study: Creating a suffix array

This section describes some aspects of the implemented case study. The algorithm described by Manber and Myers was implemented using Java. The program flow is divided into three main methods. First the alphabet of the input text is computed, the result is used in the first stage sorting method, sorting the suffixes into buckets according to their first symbol. The last h-stage-sort method sorts all suffixes according to the 2H-stage described in the paper by Manber and Myers. After the sorting is done the resulting POS array can be used for searching the suffixes of the provided input text.

4.2.1 Computing the alphabet

The first described method which computes the alphabet is used to get the size of each bucket. For each character of the input text a tree map, mapping a character to the bucket size, is queried if an entry with a bucket already exists. If this is the case the size of the bucket is incremented, else the character is added to the map with a bucket size of one.
A tree map was used because it guarantees a sorted order of its keys.

Listing 1: TreeMap initialization

```
// TreeMap implementation with explicit Char comparator
this.alphabet = new TreeMap<>(Character::compareTo);
```

The explicit use of the character comparator makes all keys in the tree sorted according to the lexicographic order. 1

4.2.2 First Stage Sort

Sorting all suffixes according to their first character into one array seemed challenging at first, but it turns out this is relatively easily done in $O(|\Sigma| * n)$ time. For each character of the input text, the whole alphabet is traversed in order. If the current character does not match the key of the alphabet entry the size of the computed bucket is added to a temporary offset. After traversing the alphabet the starting index of the corresponding bucket is computed. It is essential that the alphabet is sorted lexicographical as this order defines which character bucket is the first in the POS array.
The start of the bucket is marked as such in the BH array. As the starting index of the bucket may already be assigned to a suffix a second loop increments a counter until a free position is reached. Values for POS and PRM arrays are set at the end.

Listing 2: First stage sort comparing the first character of each suffix

```
for (char x : textChars) {
  int positionOfStartOfBucket = 0;
  int firstFreeIndexInBucket;

  for (each alphabet entryset) {
    if (x != entry.getKey()) {
      int bucketSizeOffset = entry.getValue();
      positionOfStartOfBucket += bucketSizeOffset;
    } else {
      break;
    }
  }

  ...
  // Mark BH array, compute offsets
  ...

  POS[firstFreeIndexInBucket] = currentSuffix;
  PRM[currentSuffix] = firstFreeIndexInBucket;
  currentSuffix++;
}
```

4.2.3 H-Stage Sort

The final POS array is sorted in the second (h-Stage) sort method. For each iteration with h starting at 1, h is doubled as long as $h < n$, that is h is smaller than the length of the text. At the start of each iteration first the bucket intervals are computed, these are needed as suffixes are sorted bucket wise. The intervals of a bucket are given in the BH array, if the value of an i in BH is true, the end of the current bucket has been reached. 3

Listing 3: Computing bucket intervals

```
// Loops from bucket start to bucket end
while (sizeOfBucket < textLength && !BH[sizeOfBucket]) {
  sizeOfBucket++;
}
```

Next the PRM array is reset to point to the leftmost cell of the h-bucket containing the *ith* suffix. This is done by looping through the previously computed intervals and then setting the value of PRM to the left interval boundary for each entry in the bucket. 4

Listing 4: Resetting the PRM array

```
// For each interval
// l = start of bucket (left interval boundary)
for (int l = 0; l < textLength; l = intervals[l]) {
  nextFreeIndexInBucket[l] = 0;

  // For each suffix in bucket
  // assign left boundary (l) to the suffix in bucket
  // c \in [l, r]
  for (int c = l; c < intervals[l]; c++) {
    PRM[POS[c]] = l;
  }
}
```

The rest of the h-stage sort is implemented using the pseudo-code given by Manber and Myers. Basically all the suffixes whose $h + 1$ through $2h$ symbols equal the unique h-prefix of the current h-bucket are moved to the top. Moved suffixes are marked in the $B2H$ array. In the final phase of each iteration the POS array is updated to represent the new sorting order. BH is set to $B2H$ if $B2H$ is true at the given position, in essence this means that BH array now holds all new left boundaries of each bucket.

4.2.4 Performance

The implemented algorithm was tested with a plain-text version of Moby Dick. The text has around 1.250.000 characters. Each of the three described methods outputs the total time of computation. A typical output can be seen in listing 5.

n	Alphabet	First Stage	h-Stage	Time total	Memory
≈1.250.000	69ms	9s	5,1s	14,2s	49MB
≈2.500.000	130ms	59,8s	18s	78s	79MB
≈5.000.000	225ms	3,8min	56,6s	4,8min	155MB
≈10.000.000	490ms	15,6min	3,5min	19,2min	223MB
≈20.000.000	-	-	-	>50min	-

Table 1: Case Study test cases

Listing 5: Case study console output

```
java -jar .\suffixarray-1.0.jar -f ..\mobydick.txt
Computing the alphabet took 70ms
First stage sort took 9100ms
h stage sort took 5223ms

SA created in 14393 ms
Used memory 49MB
```

Four more tests were done with the same Moby Dick text but doubling the text length each time by appending the whole text. Testing was done on an Intel 8300H mobile CPU with 2.3GHz. The results are shown in Table 1.

4.2.5 Searching

Searching using a keyword in context method without utilising lcp information takes at most $1ms$. An example search for *water* can be seen in listing 6.

Listing 6: Keyword in context search for *water*

```
Type a substring to search for:
water

posed that the watery circle surrou
ver the entire watery circumference
n walls of the watery defile in whi
d dip into the watery glens and hol
essions of the watery horizon; whil
, an unstaked, watery locality, sou
ence, the wild watery loneliness of
hased over the watery moors, and sl
world of ours— watery or otherwise;
le and see the watery part of the w
...
```

n	First Stage
≈1.250.000	573ms
≈2.500.000	982ms
≈5.000.000	1,8s
≈10.000.000	3,6s
≈20.000.000	7,5s

Table 2: Refactored first stage running times

4.2.6 Refactoring

Looking at the numbers from table 1, the complexity for the first stage sort is not linear but quadratic. The code segment in listing 7 was identified in generating a quadratic complexity for the first stage sort.

Listing 7: Quadratic complexity in first stage sort

```
// Check if the current index is already in use
// If so increment and check the next index
while (POS[firstFreeIndexInBucket] != -1) {
  firstFreeIndexInBucket++;
}
```

While this approach seemed reasonable for small input sizes it is problematic for larger ones. The loop tries to find the first free index in a bucket, that is, it loops from each bucket start to the first entry with a value of -1. Now for each character the corresponding bucket is searched beginning from the start, as the sorting progresses, the first free index will move further to the end, making the loop run longer.
A fix was implemented making the while loop redundant. For each bucket the current offset of the first free index is saved, thus requiring only retrieving the current offset from a map, and writing an incremented number back. The new linear running times for the first stage can be seen in table 2.

5 Searching in Suffix Arrays

5.1 Binary Search with LCP information

Now we have a suffix array ready with the table suftab which contains the suffixes in a sorted order. We want to find all instances of a string $P = p1...pm$ of length $m < n$ in S.

Since suftab is in $\leq m$ order, it follows that P matches a suffix S_i iff $i = suftab[k]$ for some k in $[LP; RP]$. Hence a simple binary search can find LP and RP. Each comparison in the search needs $O(m)$ character comparisons, and we can find all instances in the string in time $O(m\ log\ n)$.

For example if we search for $P = aca$ in the text $S = acaaacatat\$$ then LP = 3 and RP = 4. We then generate the value LP and RP, by setting $(L; R)$ to $(1; n)$ and updating the borders of this interval. We find LP with the sequence: $(1; 11) \Rightarrow (1; 6) \Rightarrow (1; 4) \Rightarrow (1; 3) \Rightarrow (2; 3)$. Hence LP = 3.

1	aaacatat$
2	aacatat$
3	acaaacatat$
4	acatat$
5	atat$
6	at$
7	caaacatat$
8	catat$
9	tat$
10	t$
11	$

$lcp(i, j)$ is the size of the longest common prefix among all the suffixes mentioned in positions i and j of suftab.
For $S = aabaacatat$ the $lcp(1, 2)$ is the length of the longest common prefix of $aabaacata$ and $aacata$ which is 2. With the help of lcp, the objective of one redundant character comparison per iteration of the search is met.

If $l = r$:

- We compare P to the suffix of $suftab[M]$ as before starting from position $mlr + 1$, since in this case the minimum of l and r is also the maximum of the two and no redundant character comparisons are made.

If $l \neq r$, we have three cases , we assume $l > r$:

- Case 1: $lcp(L, M) > l$

  Then the common prefix of the suffixes of $suftab[L]$ and $suftab[M]$ is longer than the common prefix of P and the Suffix of $suftab[L]$. Therefore, P agrees with the Suffix of $suftab[M]$ up through character l. Or to put it differently, characters $l + 1$ of the Suffixes of $suftab[L]$ and $suftab[M]$ are identical and lexicographical less than character $l+1$ of P. Hence any possible starting position must start to the right of M in $suftab$. So in this case no examination of P is needed. L is set to M and l and r remain unchanged.

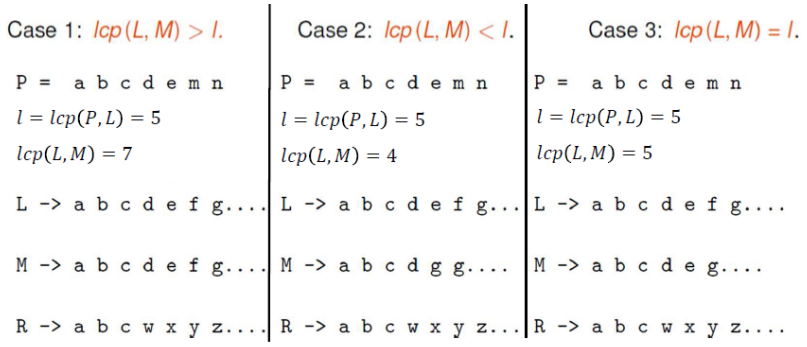
- Case 2: $lcp(L, M) < l$

  Then the common prefix of suffix $suftab[L]$ and $suftab[M]$ is smaller than the common prefix of $suftab[L]$ and P. Therefore P agrees with

$suftab[M]$ up through character $lcp(L, M)$. The $lcp(L, M) + 1$ characters of P and $suftab[L]$ are identical and lexically less than the character $lcp(L, M) + 1$ of $suftab[M]$. Hence any possible starting position must start left of M in $suftab$. So in this case again no examination of P is needed. R is set to M, r is changed to $lcp(L, M)$, and l remains unchanged.

- Case 3: $lcp(L, M) = l$

  Then P agrees with $suftab[M]$ up to character l. The algorithm then compares P to $suftab[M]$ starting from position $l+1$ considering a lexical rule. In the usual manner the outcome of the comparison determines which of L and R change along with the corresponding change of l and r.



5.2 Computing the lcp values

We went through the fast search in a suffix array assuming that we know the lcp values for all pairs (i, j).
We can compute these values by going through two steps defined as follows:

a. Compute the lcp values for pairs of suffixes adjacent in suftab using an array height of size n.

b. Compute the lcp values for the internal nodes of the fixed binary search tree used in the search for LP and RP using the array height.

In Conclusion, the most important thing was computing the array height , i.e. the lcp values of adjacent suffixes in suftab.

6 Conclusion

To conclude our work, we can say that Manber and Myers Suffix array is an efficient Algorithm. This technique opened the doors to other algorithms to be discovered like Suffix Cactus which is a cross between suffix tree and suffix array. Its size and its performance in searches lies between those of the suffix tree and the suffix array. Structurally, the suffix cactus can be seen either as a compact variation of the suffix tree or as an augmented suffix array.