1 Pattern Matching in Texts

1.1 Overview

There are many applications which consist of or regard text editing. For these, it is often important to search within a long text for a shorter text, a so-called pattern. We consider an algorithm which is highly suitable to search the same text for several patterns consecutively. In a case like this, it is helpful to do a slightly costly preprocessing step on the text, in order to make the following searches for the patterns much more efficient.

A word w is a sequence of character from a fixed alphabet Σ (with a constant size of the alphabet $|\Sigma|$, e.g. ASCII characters with $|\Sigma| = 256$). The length (number of characters) of w is denoted by $|w|$, and the ith character of w is denoted by $w[i]$ (for $0 \leq i < |w|$). The notation $w[i..j]$ stands for the subword of w, beginning at the ith character and ending at the jth. A subword $w[i..|w|-1]$ is called a *suffix* of w. A suffix is *proper* if $i \neq 0$.

The given text is a (very long) word X. Let $|X| = n$. We assume the text to end with the special character '\$', which does not occur anywhere else in the text or in the search patterns. '\$' is called a *sentinel.* This assumption guarantees that no suffix can be the start (prefix) of another new suffix of X. (A suffix which is a proper prefix of another suffix is called *nested.*)
The suffix $X[i..n-1]$ of X is denoted by X_i.
A search pattern Y of length m occurs in X if there exists an i such that $X[i..i+m-1] = Y$.
We want to have a data structure, which can be constructed in time $O(n \log n)$, and allows us to search X for an arbitrary pattern Y of length m in time $O(m \log n)$.

A suitable data structure for this is a *suffix array.* We show how suffix arrays can be constructed in an efficient and direct way in linear-logarithmic time (i.e. $n \log n$). This construction is by the original article of Udi Manber and Gene Myers [1]. The sorting method could be called a Recursive Forward Bucket Sort.
Given a text of length n, we construct an integer array **Pos** of length n, containing the indices (the starting position) of the lexicographically sorted suffixes.

Text	b	c	c	a	a	b	a	b	a	\$
Index	0	1	2	3	4	5	6	7	8	9

Sorted index	9	8	3	6	4	7	5	0	2	1
Suffix	\$	a\$	aababa\$	aba\$	ababa\$	ba\$	baba\$	bccaababa\$	caababa\$	ccaababa\$

Figure 1: Suffix array for the text “bccaababa\$”

1.2 Search

Searching for a pattern Y is done in two steps. First, we find the leftmost index L_Y in **Pos** such that Y is a prefix of the $\mathbf{Pos}[L_Y]$-th suffix. Afterwards, we find the corresponding rightmost index R_Y. Because the suffix array **Pos** is ordered lexicographically, binary search is used for both these searches.

For the first step, i.e. to find the leftmost index L_Y in **Pos** with Y as prefix of the suffix at $\mathbf{Pos}[L_Y]$, we begin with the indices $L = 0$ and $R = n - 1$.
The invariant here is that the suffix at position $\mathbf{Pos}[L]$ is lexicographically smaller than Y, and the suffix at position $\mathbf{Pos}[R]$ is lexicographically larger than or equal to Y.

We terminate if $R - L = 1$. In each step, we consider $\mathbf{Pos}[M]$. If the suffix at $\mathbf{Pos}[M]$ is lexicographically larger than or equal to Y, then we set $R := M$, otherwise we set $L := M$. The index M is chosen according to the binary search. See Figure 2.

```
Input: text $X$, pattern $Y$, suffix array Pos.
Output: $L_Y$.
Algorithm:
    if $Y$ is lexicographically smaller than $X_{\mathbf{Pos}[0]}$ then
        return $0$;
    else if $Y$ is lexicographically larger than or equal to $X_{\mathbf{Pos}[|X|-1]}$ then
        return $|X|$;
    else
        $L := 0$;
        $R := |X| - 1$;
        while $R - L > 1$ do
            $M := \lceil (L + R)/2 \rceil$;
            if $Y$ is lexicographically smaller than $X_{\mathbf{Pos}[M]}$ then
                $R := M$;
            else
                $L := M$;
        return $R$;
```

Figure 2: Binary search for the leftmost index L_Y in **Pos**, such that Y is a prefix of the suffix at the position $\mathbf{Pos}[L_Y]$.

After the termination of the algorithm, the number of occurrences of Y in X can be calculated by $R_Y - L_Y + 1$. Their left end points (i.e. their starting positions) are given by $\mathbf{Pos}[L_Y], \mathbf{Pos}[L_Y + 1], \ldots, \mathbf{Pos}[R_Y]$.

In total, the suffix array **Pos** allows us to find all occurrences of a search pattern Y within X in time $O(|Y| \log |X|)$.

1.3 Index creation: sorting the suffixes

The sorting of the suffixes is done in $\log n$ phases. In phase h (starting at 0), the suffixes are sorted in such a way that the lexicographic order with respect to the first 2^h characters is correct. For this, we use the information of the previous phase and the fact that we are sorting suffixes.
The suffixes are put into several *buckets*, such that all suffixes within one bucket coincide on the first 2^h characters. Then we sort all suffixes within a bucket.
Suppose we have two suffixes X_j and X_k from a bucket, with the first 2^h characters being identical for both suffixes, i.e. $X[j..j+2^h] = X[k..k+2^h]$. We have to compare the next 2^h characters. But these are exactly the same as the first 2^h characters of X_{j+2^h} and X_{k+2^h}. By assumption, we know their relative order from the previous phase and only need to find the positions of the indices $j+2^h$ and $k+2^h$ in the array **Pos**.
Because we want to have a cost of only $O(n)$ per phase, we cannot simply sort the items of a bucket with an $n \log n$ sorting algorithm. Instead, we go through the suffix array from front to back bucket-wise and carry on the ordering of the suffixes to the other buckets. This means, that if X_k is an element of the first bucket, than X_{k-2^h} must come before all other elements within its bucket which correspond to suffixes in higher buckets.

Depending on how cleverly the buckets are managed, we need less or more space. We use two additional boolean arrays **BucketStart** and **BucketStart2** and two integer arrays **BucketSize** and **Suf** (all of length n).
In addition, we use an array **Bucket** of length $|\Sigma|$, describing the up to $|\Sigma|$ buckets.
BucketStart$[i]$ contains TRUE if a bucket in **Pos** starts at position i, and FALSE otherwise. **Suf** reverses **Pos**, i.e. it holds that **Suf**$[$**Pos**$[i]] = i$. The other two arrays store temporary values.

During an initialization phase the suffixes are sorted with a Radix sorting step with respect to the first character. This gives us the arrays **Pos**, **Suf**, and **BucketStart** for the first phase in time $O(n)$.
After the second loop we have that, for an arbitrary character $d \in \Sigma$, $i :=$ **Bucket**$[d]$ holds the last occurrence of d in X (or -1). **Pos**$[i]$ holds a reference to the position of the next-to-last occurrence of d in X (or -1).
This chain is passed in the third loop and then, in **Suf**$[i]$ we store the position of the ith suffix in the lexicographic order with respect to the first character (in Figure 3 this is c, which is incremented by one for each suffix). See Figure 3 for the initialization phase.

We then begin with the sorting. There are $\lfloor \log n \rfloor$ phases, each taking $O(n)$ time. So assume that **Pos**, **Suf**, and **BucketStart** contain the right values after phase h, and consider phase $h+1$.
The left and right bucket boundaries for the current bucket are denoted by l and r, respectively.
We reset **BucketSize**$[l]$ to 0 for every l which is a left boundary of a bucket, as well as resetting **Suf**$[i]$ to the leftmost cell of the bucket containing the ith suffix (rather than setting **Suf**$[i]$ to the suffix's exact position within the bucket). Afterwards, the array **Pos** is scanned in an increasing order regarding the buckets, and we consider the prolonged suffix in the bucket beginning at $k :=$ **Suf**$[d]$ with $d :=$ **Pos**$[i] - 2^h$. The first

BucketSize$[k]$ elements are already set, so we insert the suffix into the next free cell, and change the temporary information accordingly. (We increment the **BucketSize**$[k]$ counter and set **BucketStart2**$[k]$ to TRUE to mark those suffixes that were moved.) Moreover, before we go on to the next bucket, we look again at all suffixes in the current bucket and reset **BucketStart2** such that only the cell corresponding to the leftmost suffix in the new bucket is set to TRUE. Thus, **BucketStart2** correctly marks the beginnings of the new buckets.
In the final loop, we update the **Pos** array, and set **BucketStart**. Figure 4 has the pseudo code for the iterative sorting.

All buckets can be traversed in linear time, hence one phase takes $O(n)$ time in total.

```
Input: Text $X$ of length $n$.
Output: Suffix ordering Suf for the first character,
        Initializing Suf, Pos,
        BucketStart, BucketStart2 and BucketSize.
Algorithm:
    Construct the arrays Suf, Pos,
    BucketStart, BucketStart2 and BucketSize;
    //Radix Sort with respect to the first character
    for each $c \in \Sigma$ do
        Bucket[$c$] := $-1$;
    for $i = 0 \ldots n-1$ do
        $b$ := Bucket[$X[i]$];
        Bucket[$X[i]$] := $i$;
        Pos[$i$] := $b$;
    $c := 1$;
    for each $d \in \Sigma$ do
        $i$ := Bucket[$d$];
        while $i \neq -1$ do
            $j$ := Pos[$i$];
            Suf[$i$] := $c$;
            if $i$ = Bucket[$d$] then
                BucketStart[$c$] := TRUE;
            else
                BucketStart[$c$] := FALSE;
            $c++$;
            $i := j$;
    //Initialize arrays
    BucketStart[$n$] := TRUE;
    for $i = 0 \ldots n-1$ do
        Pos[Suf[$i$]] := $i$;
```

Figure 3: Initializing sorting phase

```
Input: Text $X$ of length $n$, initialized arrays Suf, Pos,
       BucketStart, BucketStart2 and BucketSize.
Output: Suffix ordering in Pos.
Algorithm:
  for $h = 0 \dots \lfloor \log n \rfloor$ do
      for each Bucket $[l, r)$ do
          BucketSize$[l] := 0$;
          for $i = l \dots r - 1$ do
              Suf[Pos$[i]$] $:= l$;
      for each Bucket $[l, r)$ do
          for $i = l \dots r - 1$ do
              $d :=$ Pos$[i] - 2^h$;
              if $d < 0$ or $d \geq n$ then continue;
              $k :=$ Suf$[d]$;
              Suf$[d] := k +$ BucketSize$[k]$;
              BucketSize$[k] + +$;
              BucketStart2[Suf$[d]$] := TRUE;
          for $i = l \dots r - 1$ do
              $d :=$ Pos$[i] - 2^h$;
              if $d < 0$ OR $d \geq n$ OR NOT BucketStart2[Suf$[d]$] then
                  continue;
              $k := \min\{j : j >$ Suf$[d]$
                  AND (BucketStart$[j]$ OR NOT BucketStart2$[j]$)$\}$;
              for $j =$ Suf$[d] + 1 \dots k - 1$ do
                  BucketStart2$[j]$ := FALSE;
      for $i = 0 \dots n - 1$ do
          Pos[Suf$[i]$] $:= i$;
          BucketStart$[i]$ := BucketStart$[i]$ OR BucketStart2$[i]$;
```

Figure 4: Recursive Bucket Sort Phase

References

[1] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. COMPUT.*, 22(5):935–948, oct 1993.

[2] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of LNCS, pages 449–463. Springer, 2002.