Advanced Text Indexing Techniques

Johannes Fischer

SS 2009
1 Suffix Trees, -Arrays and -Trays

1.1 Recommended Reading


1.2 Suffix Trees

In this section we will introduce suffix trees, which, among many other things, can be used to solve the string matching task (find pattern $P$ of length $m$ in a text $T$ of length $n$ in $O(n + m)$ time). There are other methods (Boyer-Moore, e.g.), which solve this task in the same time. So why do we need suffix trees in the context of string matching?

The advantage of suffix trees over the other string-matching algorithms (Boyer-Moore, KMP, etc.) is that suffix trees are an index of the text. So, if $T$ is static and there are several patterns to be matched against $T$, the $O(n)$-task for building the index needs to be done only once, and subsequent matching-tasks can be done in $O(m)$ time. If $m \ll n$, this is a clear advantage over the other algorithms.

Throughout this section, let $T = t_1 t_2 \ldots t_n$ be a text over an alphabet $\Sigma$ of size $|\Sigma| =: \sigma$.

**Definition 1.** A compact $\Sigma^+$-tree is a rooted tree $S = (V, E)$ with edge labels from $\Sigma^+$ that fulfills the following two constraints:

- For all $v \in V$: all outgoing edges from $v$ start with a different $a \in \Sigma$.
- Apart from the root, all nodes have out-degree $\neq 1$.

**Definition 2.** Let $S = (V, E)$ be a compact $\Sigma^+$-tree.

- For $v \in V$, $\overline{v}$ denotes the concatenation of all path labels from the root of $S$ to $v$.
- $|\overline{v}|$ is called the string-depth of $v$ and is denoted by $d(v)$.
- $S$ is said to display $\alpha \in \Sigma^*$ iff $\exists v \in V, \beta \in \Sigma^*: \overline{v} = \alpha \beta$.
- If $\overline{v} = \alpha$ for $v \in V, \alpha \in \Sigma^*$, we also write $\overline{\alpha}$ to denote $v$.
- words($S$) denotes all strings in $\Sigma^*$ that are displayed by $S$: words($S$) = \{ $\alpha \in \Sigma^*$ : $S$ displays $\alpha$ \}
- For $i \in \{1, 2, \ldots, n\}$, $t_i t_{i+1} \ldots t_n$ is called the $i$-th suffix of $T$ and is denoted by $T_{i...n}$. In general, we use the notation $T_{i...j}$ as an abbreviation of $t_i t_{i+1} \ldots t_j$. 

1
Example 1.

\( S: \Sigma = \{A, C, G, T\} \)

\[ \begin{align*}
\text{depth} &= 1 \\
\text{string-depth} &= 2
\end{align*} \]

\( w = ACA \)

\( v = C \)

\[ \begin{align*}
\pi &= A \\
\tau &= C
\end{align*} \]

\( \text{words}(S) = \{\epsilon, A, AC, ACA, ACG, ACGA, C\} \)

We are now ready to define suffix trees.

**Definition 3.** Let \( \text{substring}(T) \) denote the set of all substrings of \( T \), \( \text{substring}(T) = \{T_{i...j} : 1 \leq i \leq j \leq n\} \). The suffix tree of \( T \) is a compact \( \Sigma^+ \)-tree \( S \) with \( \text{words}(S) = \text{substring}(T) \).

For several reasons, we shall find it useful that each suffix ends in a leaf of \( S \). This can be accomplished by adding a new character \$ \notin \Sigma \) to the end of \( T \), and build the suffix tree over \( T\$ \).

From now on, we assume that \( T \) terminates with a $, and we define $ to be lexicographically smaller than all other characters in \( \Sigma \): $ < a \) for all \( a \in \Sigma \). This gives a one-to-one correspondence between \( T \)'s suffixes and the leaves of \( S \), which implies that we can \textit{label the leaves} with a function \( l \) by the start index of the suffix they represent: \( l(v) = i \iff \tau = T_{i...n} \) for all leaf nodes \( v \). This also explains the name “suffix tree.”

**Implementation Remark:** The outgoing edges at internal nodes \( v \) of the suffix tree can be implemented in two fundamentally different ways:

1. as arrays of size \( \sigma \)
2. as arrays of size \( s_v \), where \( s_v \) denotes the number of \( v \)'s children

**Example 2.** Suffix tree implemented in the first way:
Example 3. Suffix tree implemented in the second way:

Approach (1) has the advantage that the outgoing edge whose edge label starts with $\alpha \in \Sigma$ can be located in $O(1)$ time, but the complete suffix tree uses space $O(n\sigma)$, which can be as bad as $O(n^2)$. Hence, we assume that approach (2) is used, which implies that locating the correct outgoing edge takes $O(\log \sigma)$ time (using binary search). Note that the space consumption of approach (2) is always $O(n)$, independent of $\sigma$. We state a final theorem, which we proved in the lecture Advanced Methods for Sequencing Analysis:

**Theorem 1.** The suffix tree for a text of length $n$ over an integer alphabet can be built in $O(n)$ time.

A note on the alphabet size: Alphabets can be classified into different types, according to their size $\sigma$:

1. **Constant** alphabets with $\sigma = O(1)$.
2. **Good-natured** alphabets with $\sigma = o(n/\log n)$.
3. **Integer** alphabets with $\sigma = O(n)$.

4. **Unbounded** alphabets, where no upper bound on $\sigma$ exists.

This list clearly forms a hierarchy; e.g., integer alphabets subsume constant and good-natured alphabets. In this lecture, all results are valid for good-natured alphabets, unless stated otherwise. Note that the restriction is not too severe, as alphabets usually “grow” much slower than the corresponding text (if they grow at all), so it is usually safe to assume $\sigma = o(n/\log n)$.

### 1.3 Searching in Suffix Trees

Let $P$ be a pattern of length $m$. Throughout the whole lecture, we will be concerned with the two following problems:

**Problem 1. Counting:** Return the number of matches of $P$ in $T$. Formally, return the size of $O_P = \{i \in [1, n] : T_{i...i+m-1} = P\}$

**Problem 2. Reporting:** Return all occurrences of $P$ in $T$, i.e., return the set $O_P$.

**Example 4.**

$$T = \text{ACCTTCCT}$ \quad \Sigma = \{A, G, T, C, $\}$

With suffix trees, the **counting-problem** can be solved in $O(m \log \sigma)$ time: traverse the tree from the root downwards, in each step locating the correct outgoing edge, until $P$ has been scanned completely. More formally, suppose that $P_{1...i-1}$ have already been parsed for some $1 \leq i < m$, and our position in the suffix tree $S$ is at node $v$ ($v = P_{1..i-1}$). We then find $v$’s outgoing edge $e$ whose label starts with $P_i$. This takes $O(\log \sigma)$ time. We then compare the label of $e$ character-by-character with $P_{i...m}$, until we have read all of $P$ ($i = m$), or until we have reached position $j \geq i$ for which $P_{1...j}$ is a node $v'$ in $S$, in which case we continue the procedure at $v'$. This takes a total of $O(m \log \sigma)$ time. Suppose the search procedure has brought us successfully to a node $v$, or to the incoming edge of node $v$. We then output the size of $S_v$, the subtree of $S$ rooted at $v$. This can be done in constant time, assuming that we have labeled all nodes in $S$ with their subtree sizes. This answers the **counting query.** For the **reporting query**, we output the labels of all leaves in $S_v$ (recall that the leaves are labeled with text positions).

**Theorem 2.** The suffix tree allows to answer counting queries in $O(m \log \sigma)$ time, and reporting queries in $O(m \log \sigma + |O_P|)$ time.
1.4 Suffix- and LCP-Arrays

We will now introduce two arrays that are closely related to the suffix tree, the suffix array \( A \) and the lcp-array \( H \).

**Definition 4.** The suffix array \( A \) of \( T \) is a permutation of \( \{1, 2, \ldots, n\} \) such that \( A[i] \) is the \( i \)-th smallest suffix in lexicographic order: \( T_{A[i-1]} \ldots n < T_{A[i]} \ldots n \) for all \( 1 < i \leq n \).

The second array \( H \) builds on the suffix array:

**Definition 5.** The LCP-array \( H \) of \( T \) is defined such that \( H[1] = 0 \), and for all \( i > 1 \), \( H[i] \) holds the length of the longest common prefix of \( T_{A[i]} \ldots n \) and \( T_{A[i-1]} \ldots n \).

**Example 5.** The suffix array \( A \) and the lcp-array \( H \) for the string ACCTTCCT$:

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
T &=& T^1 T^2 T^3 T^4 T^5 T^6 T^7 T^8 T^9 \\
A &=& A^9 A^1 T^6 T^7 T^3 T^8 T^5 T^4 \\
H &=& H^0 H^0 H^1 H^1 H^1 H^1 H^1 H^1 H^1 \\
\end{array}
\]

**Lemma 3.** Both the lcp-array \( H \) and the suffix array \( A \) can be computed in \( O(n) \) time.

The following observations relate the suffix array \( A \) with the suffix tree \( S \).

**Observation 1.** If we do a lexicographically-driven depth-first search through \( S \) (visit the children in lexicographic order of the first character of their corresponding edge-label), then the leaf-labels seen in this order give the suffix-array \( A \).

**Example 6.**

\[
T = ACCTTCCT$
\]
Definition 6. Given a tree $S = (V,E)$ and two nodes $v,w \in V$, the lowest common ancestor of $v$ and $w$ is the deepest node in $S$ that is an ancestor of both $v$ and $w$. This node is denoted by $\text{lca}(v,w)$.

Observation 2. The string-depth of the lowest common ancestor of the leaves labeled $A[i]$ and $A[i-1]$ is given by the corresponding entry $H[i]$ of the LCP-array, in symbols: for all $i > 1 : H[i] = d(\text{lca}(T_{A[i]}...n, T_{A[i-1]}...n))$.

In summary, these observations give a deep connection between $S$ and $H/A$.

### 1.5 Searching in Suffix Arrays

We can use a plain suffix array $A$ to search for a pattern $P$, using the ideas of binary search, since the suffixes in $A$ are sorted lexicographically and hence the occurrences of $P$ in $T$ form an interval in $A$. The algorithm below performs two binary searches. The first search locates the starting position $s$ of $P$’s interval in $A$, and the second search determines the end position $r$. A counting query returns $r - s + 1$, and a reporting query returns the numbers $A[s], A[s+1], \ldots, A[r]$.

**Algorithm 1**: function $\text{SAsearch}(P_{1..m})$

```plaintext
l ← 1; r ← n + 1;
while $l < r$ do
    $q ← \lfloor \frac{l+r}{2} \rfloor$;
    if $P \geq_{\text{lex}} T_{A[q]}...\min(A[q]+m-1,n)$ then
        $l ← q + 1$;
    else
        $r ← q$;
    end
s ← l; l--; r ← n;
while $l < r$ do
    $q ← \lceil \frac{l+r}{2} \rceil$;
    if $P =_{\text{lex}} T_{A[q]}...\min(A[q]+m-1,n)$ then
        $l ← q$;
    else
        $r ← q - 1$;
    end
return $[s, r]$;
```

Note that both while-loops in Alg. 1 make sure that either $l$ is increased or $r$ is decreased, so they are both guaranteed to terminate. In fact, in the first while-loop, $r$ always points one position behind the current search interval, and $r$ is decreased in case of equality (when $P = T_{A[q]}...\min(A[q]+m-1,n)$). This makes sure that the first while-loop finds the leftmost position of $P$ in $A$. The second loop works symmetrically.

**Theorem 4.** The suffix array allows to answer counting queries in $O(m \log n)$ time, and reporting queries in $O(m \log n + |O_P|)$ time.
1.6 Range Minimum Queries (RMQs)

**Definition 7.** Given an array $H[1,n]$ of integers (or any other objects from a totally ordered universe) and two indices $1 \leq i \leq j \leq n$, $\text{RMQ}_H(i,j)$ returns the position of the minimum in $H[i,j]: \text{RMQ}_H(i,j) = \text{argmin}_{i \leq k \leq j} H[k]$.

**Example 7.**

\[
H = \begin{array}{ccccccc}
3 & 2 & 5 & 4 & 3 & 1 & 3 & 2 \\
\end{array}
\]

In the lecture Advanced Methods for Sequence Analysis, we proved:

**Theorem 5.** A static array $H$ can be preprocessed in linear time into a data structure of size $O(n)$ that allows to answer RMQs on $H$ in constant time.

1.7 Longest Common Prefixes and Suffixes

An indispensable tool in pattern matching are efficient implementations of functions that compute longest common prefixes and longest common suffixes of two strings (usually suffixes or prefixes of the same string).

**Definition 8.** Given two strings $T$ and $T'$ in $\Sigma^*$, $\text{lcp}(T,T')$ denotes the length of their longest common prefix, in symbols: $\text{lcp}(T,T') = \text{max}\{k \geq 0 : T_1 \ldots k = T'_1 \ldots k\}$.

**Example 8.** $\text{lcp}(\text{ACAG}, \text{ACC}) = 2$.

Note that $\text{lcp}(\cdot)$ only gives the length of the matching prefix; if one is actually interested in the prefix itself, this can be obtained by $T_1 \ldots \text{lcp}(T,T')$, or by $T'_1 \ldots \text{lcp}(T,T')$.

As mentioned above, we will be particularly interested in longest common prefixes of suffixes from the same string $T$:

**Definition 9.** For a text $T$ of length $n$ and two indices $1 \leq i,j \leq n$, $\text{lcp}_T(i,j)$ denotes the length of the longest common prefix of the suffixes starting at position $i$ and $j$ in $T$, in symbols: $\text{lcp}_T(i,j) = \text{lcp}(T_A[i] \ldots n, T_A[j] \ldots n) = \text{lcp}(A[i], A[i-1])$. Here and in the remainder of this chapter, $A$ is again the suffix array of text $T$.

**Example 9.** $T = \text{ATTAAATATC}$

$\text{lcp}_T(2,7) \triangleq \text{lcp}(\text{TTAAATATC}, \text{TATC}) = 1$

$\text{lcp}_T(4,5) \triangleq \text{lcp}(\text{AAATATC}, \text{AATATC}) = 2$

Note that the LCP-array $H$ from Sect. 1.4 holds the lengths of longest common prefixes of lexicographically consecutive suffixes: $H[i] = \text{lcp}(T_{A[i]} \ldots n, T_{A[i-1]} \ldots n) = \text{lcp}(A[i], A[i-1])$. Here and in the remainder of this chapter, $A$ is again the suffix array of text $T$.

But how do we get the LCP-values of suffixes that are not in lexicographic neighborhood? The key to this is to employ RMQs over the LCP-array, as shown in the next lemma.

**Definition 10.** The inverse suffix array $A^{-1}$ is defined by $A^{-1}[A[i]] = i$ for all $1 \leq i \leq n$. 


Example 10.

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
A &=& 4 & 5 & 6 & 8 & 1 & 10 & 3 & 7 & 9 & 2 \\
A^{-1} &=& 5 & 10 & 7 & 1 & 2 & 3 & 8 & 4 & 9 & 6 \\
\end{array}
\]

Lemma 6. Let \(i \neq j\) be two indices in \(T\) with \(A^{-1}[i] < A^{-1}[j]\) (otherwise swap \(i\) and \(j\)). Then

\[\text{LCP}(i, j) = H[\text{RMQ}_H(A^{-1}[i] + 1, A^{-1}[j])].\]

Proof. First note that any common prefix \(\omega\) of \(T_{i...n}\) and \(T_{j...n}\) must be a common prefix of \(T_{A[i]...n}\) and \(T_{A[j]...n}\) and must hence start with \(\omega\). Let \(m = \text{RMQ}_H(A^{-1}[i] + 1, A^{-1}[j])\) and \(\ell = H[m]\). By the definition of \(H\), \(T_{i...i+\ell-1}\) is a common prefix of all suffixes \(T_{A[k]...n}\) for \(A^{-1}[i] \leq k \leq A^{-1}[j]\). Hence, \(T_{i...i+\ell-1}\) is a common prefix of \(T_{i...n}\) and \(T_{j...n}\).

Now assume that \(T_{i...i+\ell}\) is also a common prefix of \(T_{i...n}\) and \(T_{j...n}\). Then, by the lexicographic order of \(A\), \(T_{i...i+\ell}\) is also a common prefix of \(T_{A[m-1]...n}\) and \(T_{A[m]...n}\). But \(|T_{i...i+\ell}| = \ell + 1\), contradicting the fact that \(H[m] = \ell\) tells us that \(T_{A[m-1]...n}\) and \(T_{A[m]...n}\) share no common prefix of length more than \(\ell\).

Lemma 6 implies that with the inverse suffix array \(A^{-1}\), the LCP-array \(H\), and constant-time RMQs on \(H\), we can answer LCP-queries for arbitrary suffixes in \(O(1)\) time.

1.8 Accelerated Search in Suffix Arrays

The simple binary search (Alg. 1) may perform many unnecessary character comparisons, as in every step it compares \(P\) from scratch. With the help of the LCP-function from the previous section, we can improve the search in suffix arrays from \(O(m \log n)\) to \(O(m + \log n)\) time. The idea is to remember the number of matching characters of \(P\) with \(T_{A[l]...n}\) and \(T_{A[r]...n}\), if \([l : r]\) denotes the current interval of the binary search procedure. Let \(\lambda\) and \(\rho\) denote these numbers,

\[\lambda = \text{LCP}(P, T_{A[l]...n})\] and \(\rho = \text{LCP}(P, T_{A[r]...n}).\]

Initially, both \(\lambda\) and \(\rho\) are 0. Let us consider an iteration of the first while-loop in function \(\text{SAsearch}(P)\), where we wish to determine whether to continue in \([l : q]\) or \([q, r]\). (Alg. 1 would actually continue searching in \([q + 1, r]\) in the second case, but this minor improvement is not possible in the accelerated search.) We are in the following situation:

\[
\begin{array}{cccccccccccc}
A &=& l & q & r \\
\lambda &|& P_1 & | & P_1 & | & P_1 & | \rho \\
\vdots & & \vdots & & \vdots & & \vdots & \\
\vdots & & \vdots & & \vdots & & \vdots & \\
\vdots & & \vdots & & \vdots & & \vdots & \\
P_\lambda & & P_r & & P_r & & P_r & \\
\end{array}
\]

8
Without loss of generality, assume $\lambda \geq \rho$ (otherwise swap). We then look up $\xi = \text{LCP}(A[l], A[q])$ as the longest common prefix of the suffixes $T_{A[l]...n}$ and $T_{A[q]...n}$. We look at three different cases:

1. $\xi > \lambda$

   \[
   A = \begin{array}{ccc}
   l & q & r \\
   \xi & & \\
   \end{array}
   \]

   Because $P_{\lambda + 1} >_{\text{lex}} T_{A[l]+\lambda} = T_{A[q]+\lambda}$, we know that $P >_{\text{lex}} T_{A[q]...n}$, and can hence set $l \leftarrow q$, and continue the search without any character comparison. Note that $\rho$ and in particular $\lambda$ correctly remain unchanged.

2. $\xi = \lambda$

   \[
   A = \begin{array}{ccc}
   l & q & r \\
   \xi & & \\
   \end{array}
   \]

   In this case we continue comparing $P_{\lambda+1}$ with $T_{A[q]+\lambda}$, $P_{\lambda+2}$ with $T_{A[q]+\lambda+1}$, and so on, until $P$ is matched completely, or a mismatch occurs. Say we have done this comparison up to $P_{\lambda+k}$. If $P_{\lambda+k} >_{\text{lex}} T_{A[q]+\lambda+k-1}$, we set $l \leftarrow q$ and $\lambda \leftarrow k - 1$. Otherwise, we set $r \leftarrow q$ and $\rho \leftarrow k - 1$.

3. $\xi < \lambda$

   \[
   A = \begin{array}{ccc}
   l & q & r \\
   \xi & & \\
   \end{array}
   \]

   First note that $\xi \geq \rho$, as $\text{LCP}(A[l], A[r]) \geq \rho$, and $T_{A[q]...n}$ lies lexicographically between $T_{A[l]...n}$ and $T_{A[r]...n}$. So we can set $r \leftarrow q$ and $\rho \leftarrow \xi$, and continue the binary search without any character comparison.
This algorithm either halves the search interval (case 1 and 3) without any character comparison, or increases either \( \lambda \) or \( \rho \) for each successful character comparison. Because neither \( \lambda \) nor \( \rho \) are ever decreased, and the search stops when \( \lambda = \rho = m \), we see that the total number of character comparisons (= total work of case 2) is \( O(m) \). So far we have proved the following theorem:

**Theorem 7.** Together with \( \text{lcp}-\)information, the suffix array supports counting and reporting queries in \( O(m + \log n) \) and \( O(m + \log n + |O_P|) \) time, respectively (recall that \( O_P \) is the set of occurrences of \( P \) in \( T \)).

### 1.9 Suffix Trays

We now show how the \( O(m \log \sigma) \)-algorithm for suffix trees and the \( O(m + \log n) \)-algorithm can be combined to obtain a faster \( O(m + \log \sigma) \) search-algorithm. The general idea is to start the search in the suffix tree where some additional information has been stored to speed up the search, and then, at an “appropriate” point, continue the search in the suffix array with a sufficiently small interval. Note that the accelerated search-algorithm from Sect. 1.8, if executed on a sub-interval \( I = A[x, y] \) of \( A \) instead of the complete array \( A[1, n] \), runs in \( O(m + \log |I|) \) time. This is \( O(m + \log \sigma) \) for \( |I| = \sigma^{O(1)} \).

We first classify the nodes in \( T \)'s suffix tree \( S \) as follows:

1. Node \( v \) is called **heavy** if the numbers of leaves below \( v \) is at least \( \sigma \).
2. Otherwise, node \( v \) is called **light**.

![Diagram of heavy and light nodes in a suffix tree](image)

Note that all heavy nodes in \( S \) have only heavy ancestors by definition, and hence the heavy nodes form a connected subtree of \( S \). Heavy nodes \( v \) are further classified into

(a) **branching**, if at least two of \( v \)'s children are heavy,

(b) **non-branching**, if exactly one of \( v \)'s children is heavy,

(c) **terminal**, if none of \( v \)'s children are heavy.

**Example 11.**
Lemma 8. The number of branching heavy nodes is $O\left(\frac{n}{\sigma}\right)$.

Proof: First count the terminal heavy nodes. By definition, every heavy node has $\geq \sigma$ leaves below itself. Note that every leaf in $S$ must be below exactly one terminal heavy node. For the sake of contradiction, suppose there were more than $\frac{n}{\sigma}$ terminal heavy nodes. Then the tree $S$ would contain $\frac{n}{\sigma} \sigma = n$ leaves. Contradiction! Hence, the number of terminal heavy nodes is at most $\frac{n}{\sigma}$.

Now look at the subtree of $S$ consisting of terminal and branching heavy nodes only. This is a tree with at most $\frac{n}{\sigma}$ leaves, and every internal node has at least two children. For such a tree we know that the number of internal nodes is bounded by the number of leaves. The claim follows.

We now augment the suffix tree with additional information at every heavy node $v$:

(a) At every branching heavy node $v$, we store an array $B_v$ of size $\sigma$, where we store pointers to $v$’s children according to the first character on the corresponding edge (like in the “bad” suffix tree with $O(n\sigma)$ space!).

(b) At every non-branching heavy node $v$, we store a pointer $h_v$ to its single heavy child.

(c) All terminal heavy nodes, and all light children of branching or non-branching heavy nodes are called interval nodes. Every interval node $v$ is augmented with its suffix-interval $I_v$, which contains the start- and end-position in the suffix array $A$ of the leaf-labels below $v$. Furthermore, the suffix-intervals of adjacent light children of a non-branching heavy node are contracted into one single interval. Thus, non-branching heavy nodes $v$ have at most 3 children (one heavy child $h_v$, and one interval node each to the left/right of $h_v$).

Everything in the tree below the interval nodes can be deleted. The resulting tree, together with the suffix array $A$, is called the suffix tray.

Example 12. The suffix tray for $T = ATTAAATATC$ is shown in the following picture (the dotted part has been deleted from the tree):
Lemma 9. The size of the resulting data structure is $O(n)$.

Proof: From Lemma 8 we know that there are only $O\left(\frac{n}{\sigma}\right)$ heavy branching nodes. So the total space for heavy branching nodes is $O\left(\frac{n}{\sigma}\sigma\right) = O(n)$. All other information is constant at each node. The claim follows.

Now we describe the search procedure. We start as with normal suffix trees and try reading $P$ from the root downwards. There are three different cases to consider (assume the characters $P_1 \ldots i-1$ have already been matched, and we have arrived at node $v = P_1 \ldots i-1$ of the suffix tray):

(a) At branching heavy nodes $v$, we can find the the correctly labeled edge (i.e., the edge whose label starts with $P_i$) in $O(1)$ time, by consulting $B_{v_i}$. 

(b) At non-branching heavy nodes $v$, we first check if the search continues at the heavy child, by comparing the next character $P_i$ with the first character $a \in \Sigma$ on the incoming edge $(v, h_v)$. If this is the case ($P_i = a$), we compare all characters on the edge $(v, h_v)$ to the pattern and, if necessary, continue the search procedure at the heavy child $h_v$. If $P_i < a$, we continue at the (only) interval node left of $h_v$ with case (c). Otherwise ($P_i \neq a$), there are two possibilities. If $P_i > a$, we do the same for the only interval node right of $h_v$ with case (c).

(c) At interval nodes $v$, we switch to the suffix array search algorithm from section 1.8, using $I_v$ as the start interval.

Lemma 10. The length of the intervals stored at the interval nodes is $O(\sigma^2)$.

Proof: The intervals of at most $\sigma - 1$ light nodes are contracted into a single interval, and each light node has at most $\sigma - 1$ leaves below itself.

Theorem 11. The suffix tray supports counting and reporting queries in $O(m + \log \sigma)$ and $O(m + \log \sigma + |O_P|)$ time, respectively.
Proof: We either advance by one character in $P$ with a constant amount of work, or we arrive at an interval node $v$, where we perform the accelerated binary search in $O(m + \log |I_v|) = O(m + \log \sigma^2) = O(m + 2 \log \sigma) = O(m + \log \sigma)$ time, by Lemma 10.

2 The Burrows-Wheeler Transformation

The Burrows-Wheeler Transformation was originally invented for text compression. Nonetheless, it was noted soonly that it is also a very useful tool in text indexing. In this Chapter, we introduce the transformation and briefly review its merits for compression. The subsequent chapter on backwards search will then explain how it is used in the indexing scenario.

2.1 Recommended Reading


2.2 The Transformation

Definition 11. Let $T_1...n$ be a text of length $n$, where $T_n = \$ is a unique character lexicographically smaller than all other characters in $\Sigma$. Then the $i$-th cyclic shift of $T$ is $T_{i...n}T_{1...i-1}$. We denote it by $T(i)$.

Example 13.

\[
1 2 3 4 5 6 7 8 9 10 \\
T = CACAACCAC$
\]

\[
T^{(6)} = CCAC$CACAA
\]

The Burrows-Wheeler Transformation (BWT) is obtained by the following steps:

1. Write all cyclic shifts $T(i)$, $1 \leq i \leq n$, column-wise next to each other.
2. Sort the columns lexicographically.
3. Output the last row. This is $T^{\text{BWT}}$.

Example 14.
The text $T^{\text{BWT}}$ in the last row is also denoted by $L$ (last), and the text in the first row by $F$ (first). Note:

- Every row in the BWT-matrix is a permutation of the characters in $T$.
- Row $F$ is a sorted list of all characters in $T$.
- In row $L = T^{\text{BWT}}$, similar characters are grouped together. This is why $T^{\text{BWT}}$ can be compressed more easily than $T$.

### 2.3 Construction of the BWT

The BWT-matrix needs not to be constructed explicitly in order to obtain $T^{\text{BWT}}$. Since $T$ is terminated with the special character $\$$, which is lexicographically smaller than any $a \in \Sigma$, the shifts $T^{(i)}$ are sorted exactly like $T$’s suffixes. Because the last row consists of the characters preceding the corresponding suffixes, we have

$$T_i^{\text{BWT}} = T_{A[i]-1} = T_{n[A[i]]}$$

where $A$ denotes again $T$’s suffix array, and $T_0$ is defined to be $T_n$ (read $T$ cyclically!). Because the suffix array can be constructed in linear time (shown in the lecture Advanced Methods in Sequence Analysis), we get:

**Theorem 12.** The BWT of a text length-$n$ text over an integer alphabet can be constructed in $O(n)$ time. 

**Example 15.**
2.4 The Reverse Transformation

The amazing property of the BWT is that it is not a random permutation of T’s letters, but that it can be transformed back to the original text T. For this, we need the following definition:

**Definition 12.** Let F and L be the strings resulting from the BWT. Then the last-to-front mapping LF is a function $LF : [1, n] \rightarrow [1, n]$, defined by


(Remember that $T^{(A[i])}$ is the i’th column in the BWT-matrix, and $(T^{(A[i])})^{(n)}$ is that column rotated by one character downwards.)

Thus, $LF(i)$ tells us the position in F where $L[i]$ occurs.

**Example 16.**

$$T = CACAACCAC\$$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>$$</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>C</td>
<td>$$</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>$$</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>$$</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>$$</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>$$</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

$F \text{ (first)} \rightarrow F^{\text{BWT}} \Rightarrow L \text{ (last)}$

**Observation 3.** Equal characters preserve the same order in F and L. That is, if $L[i] = L[j]$ and $i < j$, then $LF(i) < LF(j)$. To see why this is so, recall that the BWT-matrix is sorted lexicographically. Because both the $LF(i)$’th and the $LF(j)$’th column start with the same character...
a = L[i] = L[j], they must be sorted according to what follows this character a, say α and β. But since i < j, we know α <_{\text{lex}} β, hence LF(i) < LF(j).

This observation allows us to compute the LF-mapping without knowing the suffix array of T.

**Definition 13.** Let T be a text of length n over an alphabet Σ, and let L = T^{\text{BWT}} be its BWT.

- Define \( C : \Sigma \rightarrow [1, n] \) such that \( C(a) \) is the number of occurrences in T of characters that are lexicographically smaller than \( a \in \Sigma \).
- Define \( \text{occ} : \Sigma \times [1, n] \rightarrow [1, n] \) such that \( \text{occ}(a, i) \) is the number of occurrences of \( a \) in \( L \)'s length-\( i \)-prefix \( L[1, i] \).

**Lemma 13.** With the definitions above, \[ LF(i) = C(L[i]) + \text{occ}(L[i], i). \]

**Proof:** Follows immediately from the observation above. \( \square \)

This gives rise to the following algorithm to recover T from \( L = T^{\text{BWT}} \).

1. Scan \( L = T^{\text{BWT}} \) and compute array \( C[1, \sigma] \).
2. Compute the first row \( F \) from \( C \).
3. Compute \( \text{occ}(L[i], i) \).
4. Recover T from right to left: we know that \( T_n = \$ \), and the corresponding cyclic shift \( T^{(n)} \) appears in column 1 in BWT. Hence, \( T_{n-1} = L[1] \). Shift \( T^{(n-1)} \) appears in column \( LF(1) \), and thus \( T_{n-2} = L[LF(1)] \). This continues until the whole text has been recovered:

\[ T_{n-k} = L[LF(LF(\ldots (LF(1)) \ldots))] \]

\[ k-1 \text{ applications of LF} \]

**Example 17.**
2.5 Compression

Storing $T^{\text{BWT}}$ plainly needs the same space as storing the original text $T$. However, because equal characters are grouped together in $T^{\text{BWT}}$, we can compress $T^{\text{BWT}}$ in a second stage. We review two different compression methods in this section.

2.5.1 Move-to-front (MTF) & Huffman Coding

- Initialize a list $Y$ containing each character in $\Sigma$ in alphabetic order.

- In a left-to-right scan of $T^{\text{BWT}}, (i = 1, \ldots, n)$, compute a new array $R[1,n]$:  
  - Write the position of character $T_i^{\text{BWT}}$ in $Y$ to $R[i]$.
  - Move character $T_i^{\text{BWT}}$ to the front of $Y$.

- Encode the resulting string/array $R$ with any kind of reversible compressor, e.g. Huffman, into a string $R$.

Example 18.
Observation 4. MTF produces “many small” numbers for equal characters that are “close together” in $T_{BWT}$. These can be compressed using an order-0 compressor, e.g. Huffman, as in the next example.

Example 19.

Both steps (Huffman & MTF) are easy to reverse.

2.5.2 Run-Length Encoding

We can also directly exploit that $T_{BWT}$ consists of many equal-letter runs. Each such run $a^\ell$ can be encoded as a pair $(a, \ell)$ with $a \in \Sigma, \ell \in [1,n]$.

Example 20.
3 Backwards Search and FM-Indices

We are now going to explore how the BW-transformed text is helpful for (indexed) pattern matching. Indices building on the BWT are called FM-indices, most likely in honour of their inventors P. Ferragina and G. Manzini. From now on, we shall always assume that the alphabet $\Sigma$ is good-natured: $\sigma = o(n/\log \sigma)$.

3.1 Recommended Reading


3.2 Model of Computation and Space Measurement

For the rest of this lecture, we work with the word-RAM model of computation. This means that we have a processor with registers of width $w$ (usually $w = 32$ or $w = 64$), where usual arithmetic operations (additions, shifts, comparisons, etc.) on $w$-bit wide words can be computed in constant time. Note that this matches all current computer architectures. We further assume that $n$, the input size, satisfies $n \leq 2^w$, for otherwise we could not even address the whole input.

From now on, we measure the space of all data structures in bits instead of words, in order to be able to differentiate between the various text indexes. For example, an array of $n$ numbers from the range $[1, n]$ occupies $n \lceil \log n \rceil$ bits, as each array cell stores a binary number consisting of $\lceil \log n \rceil$ bits. As another example, a length-$n$ text over an alphabet of size $\sigma$ occupies $n \lceil \log \sigma \rceil$ bits. In this light, all text indexes we have seen so far (suffix trees, suffix arrays, suffix trays) occupy $O(n \log n + n \log \sigma)$ bits. Note that the difference between $\log n$ and $\log \sigma$ can be quite large, e.g., for the human genome with $\sigma = 4$ and $n = 3.4 \times 10^9$ we have $\log \sigma = 2$, whereas $\log n \approx 32$. So the suffix array occupies about 16 times more memory than the genome itself!

3.3 Backward Search

We first focus our attention on the counting problem (p. 4); i.e., on finding the number of occurrences of a pattern $P_{1...m}$ in $T_{1...n}$. Recall from Chapter 2 that

- $A$ denotes $T$’s suffix array.
- $L/F$ denotes the first/last row of the BWT-matrix.
- $LF(\cdot)$ denotes the last-to-front mapping.
- $C(a)$ denotes the number of occurrences in $T$ of characters lexicographically smaller than $a \in \Sigma$.
- $occ(a, i)$ denotes the number of occurrences of $a$ in $L[1, i]$.

Our aim is identify the interval of $P$ in $A$ by searching $P$ from right to left (= backwards). To this end, suppose we have already matched $P_{i+1...m}$, and know that the suffixes starting with $P_{i+1...m}$ form the interval $[s_{i+1}, e_{i+1}]$ in $A$. In a backwards search step, we wish to calculate the interval $[s_i, e_i]$ of $P_{i...m}$. First note that $[s_i, e_i]$ must be a sub-interval of $[C(P_i) + 1, C(P_i + 1)]$, where $(P_i + 1)$ denotes the character that follows $P_i$ in $\Sigma$. 

19
So we need to identify, from those suffixes starting with $P_i$, those which continue with $P_{i+1...m}$. Looking at row $L$ in the range from $s_{i+1}$ to $e_{i+1}$, we see that there are exactly $e_i - s_i + 1$ many positions $j \in [s_{i+1}, e_{i+1}]$ where $L[j] = P_i$.

From the BWT decompression algorithm, we know that characters preserve the same order in $F$ and $L$. Hence, if there are $x$ occurrences of $P_i$ before $s_{i+1}$ in $L$, then $s_i$ will start $x$ positions behind $C(P_i) + 1$. This $x$ is given by $\text{occ}(P_i, s_{i+1} - 1)$. Likewise, if there are $y$ occurrences of $P_i$ within $L[s_{i+1}, e_{i+1}]$, then $e_i = s_i + y - 1$. Again, $y$ can be computed from the $\text{occ}$-function.
Algorithm 2: function backwards-search($P_1...m$)

\[
s \leftarrow 1; e \leftarrow n; \\
\text{for } i = m...1 \text{ do} \\
\quad s \leftarrow C(P_i) + \text{occ}(P_i, s - 1) + 1; \\
\quad e \leftarrow C(P_i) + \text{occ}(P_i, e); \\
\quad \text{if } s > e \text{ then} \\
\quad \quad \text{return "no match"}; \\
\text{end} \\
\text{return } [s, e];
\]

This gives rise to the following, elegant algorithm for backwards search:

The reader should compare this to the “normal” binary search algorithm in suffix arrays. Apart from matching backwards, there are two other notable deviations:

1. The suffix array $A$ is not accessed during the search.
2. There is no need to access the input text $T$.

Hence, $T$ and $A$ can be deleted once $T^{\text{BWT}}$ has been computed. It remains to show how array $C$ and occ are implemented. Array $C$ is actually very small and can be stored plainly using $\sigma \log n$ bits.\(^1\) Because $\sigma = o(n / \log n)$, $|C| = o(n)$ bits. For occ, we have several options that are explored in the rest of this chapter. This is where the different FM-Indices deviate from each other. In fact, we will see that there is a natural trade-off between time and space: using more space leads to a faster computation of the occ-values, while using less space implies a higher query time.

**Theorem 14.** With backwards search, we can solve the counting problem in $O(m \cdot t_{\text{occ}})$ time, where $t_{\text{occ}}$ denotes the time to answer an occ(·)-query.

### 3.4 First Ideas for Implementing Occ

For answering $\text{occ}(c, i)$, there are two simple possibilities:

1. Scan $L$ every time an $\text{occ}(\cdot)$-query has to be answered. This occupies no space, but needs $O(n)$ time for answering a single $\text{occ}(\cdot)$-query, leading to a total query time of $O(mn)$ for backwards search.

2. Store all answers to $\text{occ}(c, i)$ in a two-dimensional table. This table occupies $O(n\sigma \log n)$ bits of space, but allows constant-time $\text{occ}(\cdot)$-queries. Total time for backwards search is optimal $O(m)$.

For more practical implementation between these two extremes, let us define the following:

**Definition 14.** Given a bit-vector $B[1,n]$, $\text{rank}_1(B, i)$ counts the number of 1’s in $B$’s prefix $B[1,i]$. Operation $\text{rank}_0(B, i)$ is defined similarly for 0-bits.

\(^1\)More precisely, we should say $\sigma \lceil \log n \rceil$ bits, but we will usually omit floors and ceilings from now on.
We shall see presently that a bit-vector $B$, together with additional information for constant-time RANK-operations, can be stored in $n + o(n)$ bits. This can be used as follows for implementing OCC: For each character $c \in \Sigma$, store an indicator bit vector $B_c[1, n]$ such that $B_c[i] = 1$ iff $L[i] = c$. Then

$$\text{OCC}(c, i) = \text{RANK}_1(B_c, i).$$

The total space for all $\sigma$ indicator bit vectors is thus $\sigma n + o(\sigma n)$ bits. Note that for reporting queries, we still need the suffix array to output the values in $A[s, e]$ after the backwards search.

**Theorem 15.** With backwards search and constant-time RANK operations on bit-vectors, we can answer counting queries in optimal $O(m)$ time. The space (in bits) is $\sigma n + o(\sigma n) + \sigma \log n$. □

**Example 21.**

<table>
<thead>
<tr>
<th>1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L = \text{CCCCAAAC}$A</td>
</tr>
<tr>
<td>$B_$ = 0000000010</td>
</tr>
<tr>
<td>$B_A = 0000111001$</td>
</tr>
<tr>
<td>$B_C = 1111000100$</td>
</tr>
</tbody>
</table>

### 3.5 Compact Data Structures on Bit Vectors

We now show that a bit-vector $B$ of length $n$ can be augmented with a data structure of size $o(n)$ bits such that RANK-queries can be answered in $O(1)$ time. First note that

$$\text{RANK}_0(B, i) = i - \text{RANK}_1(B, i),$$

so considering $\text{RANK}_1$ will be enough.

We conceptually divide the bit-vector $B$ into blocks of length $s = \lfloor \frac{\log n}{2} \rfloor$ and super-blocks of length $s' = s^2 = \Theta(\log^2 n)$.

The idea is to decompose a RANK$_1$-query into 3 sub-queries that are aligned with the block- or super-block-boundaries. To this end, we store three types of arrays:

1. For all of the $\lfloor \frac{n}{s'} \rfloor$ super-blocks, $M'[i]$ stores the number of 1’s from $B$’s beginning up to the end of the $i$’th superblock. This table needs order of

$$\frac{n}{s'} \times \log n = O\left(\frac{n}{\log n}\right) = o(n)$$

bits.
2. For all of the \( \lfloor \frac{n}{s} \rfloor \) blocks, \( M[i] \) stores the number of 1’s from the beginning of the superblock in which block \( i \) is contained up to the end of the \( i \)’th block. This needs order of
\[
\frac{n}{s} \times \log s' = O \left( \frac{n \log \log n}{\log n} \right) = o(n)
\]
bits of space.

3. For all bit-vectors \( V \) of length \( s \) and all \( 1 \leq i \leq s \), \( P[V][i] \) stores the number of 1-bits in \( V[1,i] \). Because there are only \( 2^s = 2^{\log n} \) such vectors \( V \), the space for table \( P \) is order of
\[
\frac{2^{\log n}}{\#\text{possible blocks}} \times s \times \log s = O \left( \sqrt{n \log n \log \log n} \right) = o(n)
\]
bits.

**Example 22.**

\[
s = 3 \quad s' = 9
\]
\[
B = \begin{array}{cccccccccccc}
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0
\end{array}
\]
\[
M' = 6 \quad 10 \quad 14
\]
\[
M = 1 \quad 3 \quad 6 \quad 0 \quad 3 \quad 4 \quad 1 \quad 3 \quad 4
\]
\[
P: \begin{array}{c|cccc}
V & 1 & 2 & 3 \\
--- & --- & --- & ---
\end{array}
\]
\[
\begin{array}{cccccccccccc}
000 & 0 & 0 & 0 & 001 & 0 & 0 & 1 & 010 & 0 & 1 & 1 & 011 & 0 & 1 & 2 & 100 & 1 & 1 & 1 & 101 & 1 & 1 & 2 & 110 & 1 & 2 & 2 & 111 & 1 & 2 & 3
\end{array}
\]

A query \( \text{RANK}_1(B,i) \) is then decomposed into 3 sub-queries, as seen in the following picture:

\[
B = \begin{array}{cccccccccccc}
1 & 2 & 3 & \cdots & \cdots & \cdots & \cdots & \cdots & i & \cdots & \cdots
\end{array}
\]

1. superblock query: precomputed in \( M' \)
2. block-query: precomp. in \( M \)
3. in-block-query: precomp. in \( P \)
Thus, computing the block number as $q = \left\lfloor \frac{i-1}{s} \right\rfloor$, and the super-block number as $q' = \left\lfloor \frac{i-1}{s'} \right\rfloor$, we can answer

$$\text{RANK}_1(B, i) = M'[q'] + M[q] + P[B[qs + 1, (q + 1)s]]_{i's \ block} \qquad [i - qs]_{\text{index in block}}$$

in constant time.

**Example 23.** Continuing the example above, we answer $\text{RANK}_1(B, 17)$ as follows: the block number is $q = \left\lfloor \frac{17-1}{3} \right\rfloor = 5$, and the super-block number is $q' = \left\lfloor \frac{17-1}{9} \right\rfloor = 1$. Further, $i$'s block is $B[5 \times 3 + 1, 6 \times 3] = B[16, 18] = 001$, and the index in that block is $17 - 5 \times 3 = 2$. Hence, $\text{RANK}_1(B, 17) = M'[1] + M[5] + P[001][2] = 6 + 3 + 0 = 9$.

This finishes the description of the data structure for $O(1)$ rank-queries. In addition to that, we also define the inverse of RANK, a function that will be helpful in subsequent chapters:

**Definition 15.** Given a bit-vector $B[1, n]$, $\text{SELECT}_1(B, i)$ returns the position of the $i$'th 1-bit in $B$, or $n + 1$ if $B$ contains less than $i$ 1's. Operation $\text{SELECT}_0$ is defined similarly.

Note that $\text{RANK}_1(B, \text{SELECT}(B, i)) = i$. The converse $\text{SELECT}(B, \text{RANK}(B, i))$ is only true if $B[i] = 1$. Note also that $\text{SELECT}_0$ cannot be computed easily from $\text{SELECT}_1$ (as it was the case for RANK), so $\text{SELECT}_1$ and $\text{SELECT}_0$ have to be considered separately.

Solving $\text{SELECT}$-queries is only a little bit more complicated than solving $\text{RANK}$-queries. We divide the range of arguments for $\text{SELECT}_1$ into subranges of size $\kappa = \lceil \log^2 n \rceil$, and store in $N[i]$ the answer to $\text{SELECT}_1(B, i\kappa)$. This table $N[1, \lceil \frac{n}{2} \rceil]$ needs $O(\frac{n}{\kappa} \log n)$ bits, and divides $B$ into blocks of different size, each containing $\kappa$ 1's (apart from the last).

A block is called **long** if it spans more than $\kappa^2 = \Theta(\log^4 n)$ positions in $B$, and **short** otherwise. For the long blocks, we store the answers to all $\text{SELECT}_1$-queries explicitly. Because there are at most $\frac{n}{\log^2 n}$ long blocks, this requires

$$O\left(\frac{n}{\kappa^2 \log n}\right) = O\left(\frac{n}{\log^4 n} \times \frac{\log^2 n}{\kappa^2 \log n} \times \frac{\log n}{\log n}\right) = O\left(\frac{n}{\log n}\right) = o(n) \text{ bits.}$$

Short blocks contain $\kappa$ 1-bits and span at most $\kappa^2$ positions in $B$. We divide again their range of arguments into sub-ranges of size $\kappa' = \lceil \log^2 \kappa \rceil = \Theta(\log^2 \log n)$. In $N'[i]$, we store the answer to $\text{SELECT}_1(B, i\kappa')$, relative to the beginning of the block where $i$ occurs:

$$N'[i] = \text{SELECT}_1(B, i\kappa') - N[\left\lfloor \frac{i\kappa' - 1}{\kappa} \right\rfloor \text{ block before } i].$$
Because the values in \( N' \) are in the range \([1, \kappa^2]\), table \( N'[1, \lceil n/\kappa \rceil] \) needs
\[
O \left( \frac{n}{\kappa^2 \log \kappa^2} \right) = O \left( \frac{n}{\log^2 \log n} \right) = o(n)
\]
bits. Table \( N' \) divides the blocks into miniblocks, each containing \( \kappa' \) 1-bits.

Miniblocks are **long** if they span more than \( \sqrt{\kappa^2} = \Theta(\log n) \) bits, and **short** otherwise. For long miniblocks, we store again the answers to all **select**-queries explicitly, relative to the beginning of the corresponding block. Because the miniblocks are contained in short blocks of length \( \leq \kappa \), the answer to such a **select**-query takes \( \log \kappa \) bits of space. Thus, the total space for the long miniblocks is
\[
O \left( \frac{n}{\sqrt{\kappa}} \times \frac{\kappa'}{\#\text{arguments}} \times \log \kappa \right) = O \left( \frac{n \log^3 \log n}{\log n} \right) = o(n)
\]
bits.

Finally, because short miniblocks are of length \( \frac{\log n}{2} \), we can use a global lookup table (analogous to \( P \) in the solution for **rank**) to answer **select**\(_1\)-queries within short miniblocks.

Answering **select**-queries is done as follows. TODO!!!

The structures need to be duplicated for **select**\(_0\). We summarize this section in the following theorem.

**Theorem 16.** An \( n \)-bit vector \( B \) can be augmented with data structures of size \( o(n) \) bits such that
\[
\text{**rank**}_b(B, i) \quad \text{and} \quad \text{**select**}_b(B, i)
\]
can be answered in constant time \( (b \in \{0,1\}) \).

### 3.6 Wavelet Trees

Armed with constant-time **rank**-queries, we now develop a more space-efficient implementation of the \( \text{oc}\)\_function, sacrificing the optimal query time. The idea is to use a wavelet tree on the BW-transformed text.

The wavelet tree of a sequence \( L[1, n] \) over an alphabet \( \Sigma[1, \sigma] \) is a balanced binary search tree of height \( O(\log \sigma) \). It is obtained as follows. We create a root node \( v \), where we divide \( \Sigma \) into two halves \( \Sigma_l = \Sigma[1, \lceil \sigma/2 \rceil] \) and \( \Sigma_r = \Sigma[\lceil \sigma/2 \rceil + 1, \sigma] \) of roughly equal size. Hence, \( \Sigma_l \) holds the lexicographically first half of characters of \( \Sigma \), and \( \Sigma_r \) contains the other characters. At \( v \) we store a bit-vector \( B_v \) of length \( n \) (together with data structures for \( O(1) \) **rank**-queries), where a
'0' of position $i$ indicates that character $L[i]$ belongs to $\Sigma_l$, and a '1' indicates the it belongs to $\Sigma_r$. This defines two (virtual) sequences $L_v$ and $R_v$, where $L_v$ is obtained from $L$ by concatenating all characters $L[i]$ where $B_v[i] = 0$, in the order as they appear in $L$. Sequence $R_v$ is obtained in a similar manner for positions $i$ with $B_v[i] = 1$. The left child $l_v$ is recursively defined to be the root of the wavelet tree for $L_v$, and the right child $r_v$ to be the root of the wavelet tree for $R_v$. This process continues until a sequence consists of only one symbol, in which case we create a leaf.

**Example 24.**

$L=$CCCCAAAC$A$  $\Sigma=\{$$,$A,C$\}$

```
\begin{center}
\begin{tikzpicture}
  \node at (0,0) {\texttt{L}};
  \node at (1,-1) {\texttt{CCCAAAC}};
  \node at (1,-3) {\texttt{AAAAC}};
  \node at (2,-3) {\texttt{AAA}};
  \node at (3,-5) {\texttt{C}};
  \node at (4,-5) {\texttt{C}};
  \node at (5,-5) {\texttt{C}};
  \node at (6,-5) {\texttt{C}};
  \node at (7,-5) {\texttt{C}};

  \draw (0,0) -- (1,-1) node[midway,above] {$\Sigma_l = \{$\} $\Sigma_r = \{A\}$};
  \draw (1,-1) -- (1,-3) node[midway,above] {$\Sigma_l = \{$,A\}$ $\Sigma_r = \{C\}$};
  \draw (1,-3) -- (2,-3) node[midway,above] {\texttt{A}};
  \draw (2,-3) -- (3,-5) node[midway,above] {\texttt{A}};
  \draw (3,-5) -- (4,-5) node[midway,above] {\texttt{C}};
  \draw (4,-5) -- (5,-5) node[midway,above] {\texttt{C}};
  \draw (5,-5) -- (6,-5) node[midway,above] {\texttt{C}};
  \draw (6,-5) -- (7,-5) node[midway,above] {\texttt{C}};

  \draw (1,-1) -- (2,-1) node[midway,above] {1111000100};
  \draw (1,-3) -- (2,-3) node[midway,above] {11101};
  \draw (2,-3) -- (3,-5) node[midway,above] {11101};

  \node at (1,0) {\texttt{WT}};
  \node at (2,0) {$\Rightarrow$};
  \node at (3,0) {1111000100};
  \node at (4,0) {11101};
  \node at (5,0) {1};
  \node at (6,0) {1};
  \node at (7,0) {1};

\end{tikzpicture}
\end{center}
```

Note that the sequences themselves are not stored explicitly; node $v$ only stores a bit-vector $B_v$ and structures for $O(1)$ rank-queries.

**Theorem 17.** The wavelet tree for a sequence of length $n$ over an alphabet of size $\sigma$ can be stored in $n \log \sigma \times (1 + o(1))$ bits.

**Proof:** We concatenate all bit-vectors at the same depth $d$ into a single bit-vector $B_d$ of length $n$, and prepare it for $O(1)$-rank-queries (see Sect. 3.5). Hence, at any level, the space needed is $n + o(n)$ bits. Because the depth of the tree is $\lceil \log \sigma \rceil$ the claim on the space follows. In order to “know” the sub-interval of a particular node $v$ in the concatenated bit-vector $B_d$ at level $d$, we can store two indices $\alpha_v$ and $\beta_v$ such that $B_d[\alpha_v, \beta_v]$ is the bit-vector $B_v$ associated to node $v$. This accounts for additional $O(\sigma \log n)$ bits. Then a rank-query is answered as follows ($b \in \{0, 1\}$):

$$\text{rank}_b(B_v, i) = \text{rank}_b(B_d, \alpha_v + i - 1) - \text{rank}_b(B_d, \alpha_v - 1),$$

where it is assumed that $i \leq \beta_v - \alpha_v + 1$, for otherwise the result is not defined.

How does the wavelet tree help for implementing the $\text{occ}$-function? Suppose we want to compute $\text{occ}(c, i)$, i.e., the number of occurrences of $c \in \Sigma$ in $L[1, i]$. We start at the root $r$ of the wavelet tree, and check if $c$ belongs to the first or to the second half of the alphabet. In the first case, we know that the $c$’s are “stored” in the left child of the root, namely $L_r$. Hence, the number of $c$’s in $L[1, i]$ corresponds to the number of $c$’s in $L_r[1, \text{rank}_0(B_r, i)]$. If, on the hand, $c$ belongs to the second half of the alphabet, we know that the $c$’s are “stored” in the subsequence $R_r$ that corresponds to the right child of $r$, and hence compute the number of occurrences of $c$ in $R_r[1, \text{rank}_1(B_r, i)]$ as the number of $c$’s in $L[1, i]$. This leads to the following recursive procedure for computing $\text{occ}(c, i)$, to be invoked with $\text{WT-occ}(c, i, 1, \sigma, r)$, where $r$ is the root of the wavelet tree. (Recall that we assume that the characters in $\Sigma$ can be accessed as $\Sigma[1], \ldots, \Sigma[\sigma]$.)
Algorithm 3: function WT-occ($c, i, \sigma_l, \sigma_r, v$)

```
if $\sigma_l = \sigma_r$ then
    return $i$;
end

$\sigma_m = \lfloor \frac{\sigma_l + \sigma_r}{2} \rfloor$;
if $c \leq \Sigma[\sigma_m]$ then
    return WT-occ($c$, RANK0($B_v, i$), $\sigma_l, \sigma_m, l_v$);
else
    return WT-occ($c$, RANK1($B_v, i$), $\sigma_m + 1, \sigma_r, r_v$);
end
```

Due to the depth of the wavelet tree, the time for $\text{WT-occ}(\cdot)$ is $O(\log \sigma)$. This leads to the following theorem.

**Theorem 18.** With backward-search and a wavelet-tree on $T^{\text{RWT}}$, we can answer counting queries in $O(m \log \sigma)$ time. The space (in bits) is

$$\left| C \right| + \text{space for } \alpha_v \text{'s} + \text{wavelet tree} + \text{RANK data structure}$$

\[ \begin{aligned}
O(\sigma \log n) & \quad + \quad n \log \sigma & \quad + \quad o(n \log \sigma) \\
\end{aligned} \]

4 Compressed Suffix Arrays

Until now, for enumerating the occurrences of a search pattern $P$ in $T$, we still have to sacrifice the $O(n \log n)$ bits for the suffix array $A$. Note that all other structures for backward-searching (array $C$ and the wavelet-tree for computing occ) occupy $O(n \log \sigma)$ bits, the same as the text $T$. We will show in this section that $O(n \log \sigma)$ bits suffice also for representing $A$. The drawback of this compressed suffix array is that the time for retrieving an entry from $A$ is not constant any more, but rises from $O(1)$ to $O(\log^* n)$, for some arbitrarily small constant $0 < \epsilon \leq 1$.

4.1 Recommended Reading


4.2 The $\psi$-Function

The most important component of the compressed suffix array (abbreviated as CSA henceforth) is a function $\psi$ that allows us to “jump one character forward” in the suffix array.

**Definition 16.** Define $\psi: [1, n] \rightarrow [1, n]$ such that $\psi(i) = j \Leftrightarrow A[j] = A[i] + 1$, where position $n+1$ is interpreted as the first position in $T$ (read text circularly!).
Example 25.

\[\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
A= & 16 & 4 & 14 & 2 & 7 & 12 & 5 & 9 & 15 & 3 & 1 & 8 & 13 & 6 & 11 & 10 \\
\psi= & 11 & 7 & 9 & 10 & 12 & 13 & 14 & 16 & 1 & 2 & 4 & 8 & 3 & 5 & 6 & 15 \\
\end{array}\]

Note the similarity of the \(\psi\)-function to suffix links in suffix trees: both “cut off” the first character of the corresponding substring.

We remark that \(\psi\) is actually the inverse function of the \(l_f\)-mapping from Sect. 3.3: while \(\psi\) allows us to move from suffix \(T_{A[i]}\ldots n\) to \(T_{A[i]+1}\ldots n\), with \(l_f\) we can move from \(T_{A[i]}\ldots n\) to \(T_{A[i]-1}\ldots n\) (recall Def. 14 in Sect. 3.3), in symbols:

\[\psi(l_f(i)) = i = l_f(\psi(i))\,.
\]

This is also the reason why \(\psi\) is increasing in areas where the corresponding suffixes start with the same character. For instance, in Ex. 25 we have that all suffixes from \(A[2,9]\) start with letter \(A\); and indeed, \(\psi[2,9] = [7,9,10,12,13,14,16]\) is increasing. This is summarized in the following lemma, which can be proved similarly as Observation 3.

**Lemma 19.** If \(i < j\) and \(T_{A[i]} = T_{A[j]}\), then \(\psi(i) < \psi(j)\).

This lemma will be used in Sect. 4.6 to store \(\psi\) in a space-efficient form.

### 4.3 The Idea of the Compressed Suffix Array

We now present the general approach to store \(A\) in a space-efficient form. Instead of storing every entry in \(A\), in a new bit-vector \(B_0[1,n]\) we mark the positions in \(A\) where the corresponding entry in \(A\) is even:

\[B_0[i] = 1 \iff A[i] \equiv 0 \pmod{2}.
\]

Bit-vector \(B_0\) is prepared for \(O(1)\) rank-queries (Sect. 3.5). We further store the \(\psi\)-values at positions \(i\) with \(B_0[i] = 0\) in a new array \(\psi_0[1, \lceil\frac{n}{2}\rceil]\). Finally, we store the even values of \(A\) in a new array \(A_1[1, \lceil\frac{n}{2}\rceil]\), and divide all values in \(A_1\) by 2.

**Example 26.**

\[\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
A= & 16 & 4 & 14 & 2 & 7 & 12 & 5 & 9 & 15 & 3 & 1 & 8 & 13 & 6 & 11 & 10 \\
B_0= & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\psi_0= & 12 & 14 & 16 & 12 & 4 & 3 & 6 \\
A_1= & 8 & 2 & 7 & 1 & 6 & 4 & 3 & 5 \\
\end{array}\]

Now, the three arrays, \(B_0\), \(\psi_0\) and \(A_1\), completely substitute \(A\): to retrieve value \(A[i]\), we first check if \(B_0[i] = 1\). If so, we know that \(A[i]/2\) is stored in \(A_1\), and that the exact position in \(A_1\) is given by the number of 1-bits in \(B_0\) up to position \(i\). Hence, \(A[i] = 2A_1[\text{rank}_1(B_0, i)]\).

If, on the other hand, \(B_0[i] = 0\), we follow \(\psi(i)\) in order to get to the position of the \((A[i] + 1)\)st suffix, which must be even (and is hence stored in \(A_1\)). The value \(\psi(i)\) is stored in \(\psi_0\), and its position therein is equal to the number of 0-bits in \(B_0\) up to position \(i\). Hence, \(A[i] = A[\psi_0(\text{rank}_0(B_0, i))] - 1\), which can be calculated be the mechanism of the previous paragraph.

As we shall see later, \(\psi_0\) can be stored very efficiently (basically using \(O(n \log \sigma)\) bits). Hence, we have almost halved the space with this approach (from \(n \log n\) bits for \(A\) to \(\frac{n}{2} \log \frac{n}{2}\) for \(A_1\)).
4.4 Hierarchical Decomposition

We can use the idea from the previous section recursively in order to gain more space: instead of representing $A_1$ plainly, we replace it with bit-vector $B_1$, array $\psi_1$ and $A_2$. Array $A_2$ can in turn be replaced by $B_2, \psi_2$, and $A_3$, and so on. In general, array $A_k[1,n_k]$, with $n_k = \frac{n}{2^k}$, implicitly represents $T$’s suffixes that are a multiple of $2^k$, in the order as they appear in the original array $A_0 := A$.

Example 27.

$$\begin{array}{c}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
A = & 16 & 4 & 14 & 2 & 7 & 12 & 5 & 9 & 15 & 3 & 1 & 8 & 13 & 6 & 11 & 10
\end{array}$$

$$\begin{array}{c}
\psi_0 = 12 & 14 & 16 & 1 & 2 & 4 & 3 & 6 \\
B_0 = 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & \\
A_1 = 8 & 2 & 7 & 1 & 6 & 4 & 3 & 5
\end{array}$$

$$\begin{array}{c}
\psi_1 = 1 & 2 & 6 & 5 \\
B_1 = 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
A_2 = 4 & 1 & 3 & 2
\end{array}$$

... etc.

$A_k$ can be seen as a suffix array of a new string $T^k$, where the $i$’th character of $T^k$ is the concatenation of $2^k$ characters $T_{i2^k...(i+1)2^k-1}$ (we assume that $T$ is padded with sufficiently enough $\$-$characters). This means that the alphabet for $T^k$ is $\Sigma^{2^k}$, i.e., all $2^k$-tuples from $\Sigma$.

Example 28. $A_2 = [4,1,3,2]$ can be regarded as the suffix array of

$$T^2 = (AATA) (CATT) (ATAC) (\\$\\$\\$).$$

This way, on level $k$ we only store $B_k$ and $\psi_k$. Only on the last level $h$ we store $A_h$. We choose $h = \lceil \log \log_\sigma \frac{n}{\log n} \rceil$ such that the space for storing $A_h$ is

$$O(n_h \log n_h) = O(n_h \log n) = O \left( \frac{n}{2^h} \log n \right) = O \left( \frac{n \log \sigma}{\log \frac{n}{\log n}} \log n \right) = O(n \log \sigma) \text{ bits.}$$

However, storing $B_k$ and $\psi_k$ on all $h$ levels would take too much space. Instead, we use only a constant number of $1 + \frac{1}{\epsilon}$ levels, namely $0, h\epsilon, 2h\epsilon, \ldots, h$ (constant $0 < \epsilon \leq 1$).

Example 29.
T = C A C A A T A C A T T A T A C $
A_0 = 16 4 14 2 7 12 5 9 15 3 1 8 13 6 11 10
ψ_0 = 9 10 12 14 16 1 2 4 3 5 6 15
\psi_2 = 4 1 3
B_2 = 1 0 0 0
A_4 = 1

Hence, bit-vector \( B_k \) has a '1' at position \( i \) iff \( A_k[i] \) is a multiple of \( 2^{h \epsilon + k} \).

Given all this, we have the following algorithm to compute \( A[i] \), to be invoked with \( \text{lookup}(i, 0) \).

**Algorithm 4: function \( \text{lookup}(i, k) \)**

```
if \( k = h \) then
  return \( A_h[i] \);
end

if \( k = \omega_k \) then
  return \( n_k \);
end

if \( B_k[i] = 1 \) then
  return \( 2^{h \epsilon} \text{lookup}(\text{rank}_1(B_k, i), k + h \epsilon) \);
else
  return \( \text{lookup}(\psi_k(\text{rank}_0(B_k, i), k)) - 1 \);
end
```

Here, \( \omega_k \) stores the position of the last suffix, i.e., \( A_k[\omega_k] = n_k \). Checking if \( i = \omega_k \) is necessary in order to avoid following \( \psi_k \) from the last suffixes to the first, because this would give incorrect results.

**Example 30.** \( A[15] = \text{lookup}(15, 0) = \text{lookup}(\psi_0(11), 0) - 1 = \text{lookup}(6, 0) - 1 = 2^2 \text{lookup}(3, 2) - 1 = 2^2(\text{lookup}(\psi_2(2), 2) - 1) - 1 = 2^2(\text{lookup}(1, 2) - 1) - 1 = 2^2(4 - 1) - 1 = 11 \)

To analyze the running time of the \( \text{lookup} \)-procedure, we first note that on every level \( k \), we need to follow \( \psi_k \) at most \( 2^{h \epsilon} \) times until we hit a position \( i \) with \( B_k[i] = 1 \) (second case of the last if-statement). Because the number of “implemented” levels, \( 1 + \frac{1}{\epsilon} \), is constant (remember \( \epsilon \) is constant!), the total time of the \( \text{lookup} \)-procedure is

\[
O\left(2^{h \epsilon}\right) = O\left(\left(2^{\log \log \sigma n}\right)^{\epsilon}\right) = O\left(\log^{\epsilon} n\right),
\]

which is sub-logarithmic for \( \epsilon < 1 \).
4.5 Elias-Codes

For coding the \(\psi\)-values in a space efficient form, we will use Elias-\(\gamma\) and Elias-\(\delta\) codes, which we present in this section. Let us write \((x)_{2}\) for the binary representation of integer \(x \geq 1\). Also \((x)_{1}\) denotes the unary representation of \(x\), which consists of \(x - 1\) 0’s, followed by a single 1. For example, \((5)_{2} = 101\) and \((5)_{1} = 00001\).

The Elias-\(\gamma\) code of a number \(x\), denoted by \((x)_{\gamma}\), is defined as follows: first, write the length of the binary representation of \(x\) in unary, i.e., write bits \((|x|_{2})_{1}\). Then append the bits from \((x)_{2}\), with the first (leftmost) ‘1’ being omitted. For example, the first five \(\gamma\)-codes (representing the numbers 1, 2, \ldots, 5) are 1, 010, 011, 00100 and 00101. The length in bits is

\[
|\(x)_{\gamma}| = \left\lfloor \log x \right\rfloor + 1 + \left\lfloor \log x \right\rfloor .
\]

The \(\delta\)-code is obtained in a similar manner, but instead of encoding \(|x|_{2}\) in unary, we encode it with the \(\gamma\)-code. That is, we first write \(|(x)_{2})_{\gamma}|\), and then append \((x)_{2}\), again with the trailing '1' being omitted. Examples of \(\delta\)-codes are shown in the following table.

**Example 31.**

<table>
<thead>
<tr>
<th>(x)</th>
<th>((x)_{\delta})</th>
<th>(x)</th>
<th>((x)_{\delta})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>9</td>
<td>00100001</td>
</tr>
<tr>
<td>2</td>
<td>0100</td>
<td>10</td>
<td>00100010</td>
</tr>
<tr>
<td>3</td>
<td>0101</td>
<td>11</td>
<td>00100011</td>
</tr>
<tr>
<td>4</td>
<td>01100</td>
<td>12</td>
<td>00100100</td>
</tr>
<tr>
<td>5</td>
<td>01101</td>
<td>13</td>
<td>00100101</td>
</tr>
<tr>
<td>6</td>
<td>01110</td>
<td>14</td>
<td>00100110</td>
</tr>
<tr>
<td>7</td>
<td>01111</td>
<td>15</td>
<td>00100111</td>
</tr>
<tr>
<td>8</td>
<td>00100000</td>
<td>16</td>
<td>001010000</td>
</tr>
</tbody>
</table>

The size of the \(\delta\)-code is

\[
|\(x)_{\delta}| = \left\lfloor \log \left(\left\lfloor \log x \right\rfloor + 1\right)\right\rfloor + \left\lfloor \log \left(\left\lfloor \log x \right\rfloor + 1\right)\right\rfloor + \left\lfloor \log x \right\rfloor + 2\left\lfloor \log \left(\left\lfloor \log x \right\rfloor + 1\right)\right\rfloor + 1 \text{ bits.}
\]

4.6 Storing \(\psi\)

Let us first concentrate on level 0, i.e., on storing \(\psi_0\). From Lemma 19, we know that \(\psi\) is piecewise increasing in areas \(A[l, r]\) where the suffixes start with the same character (i.e., where \(T_{A[i]} = T_{A[j]}\) for all \(i, j \in [l, r]\)). Let \([l, r]\) be one such area. Instead of storing \(\psi_0[l, r]\) plainly, we first compute the differences \(\Delta_0[i] = \psi_0[i] - \psi_0[i - 1]\) for \(l < i \leq r\). This produces a list of positive integers from the range \([1, n]\), which will be encoded space-efficiently in a subsequent step. In general, we define

\[
\Delta_0[i] = \begin{cases} 
\psi_0[i] - \psi_0[i - 1] & \text{if } T_{A_0[i]} = T_{A_0[i-1]}, \\
\psi_0[i] & \text{otherwise.}
\end{cases}
\]
Example 32.

\[ \Delta_0 = 9 \quad 1 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 2 \quad 3 \quad 2 \quad 1 \quad 9 \]

These \( \Delta \)-values are now encoded with Elias \( \delta \)-code; the resulting bit stream is called \( S_0 \).

Example 33.

\[ \Delta_0 = 9 \quad 1 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 2 \quad 3 \quad 2 \quad 1 \quad 9 \]

\[ S_0 = 00100001 \ 1 \ 0100 \ 0100 \ 0100 \ 1 \ 1 \ 0100 \ 0101 \ 0100 \ 1 \ 0010001 \]

In general, because \( A_k \) can be regarded as the suffix array of a text \( T^k \), we can compress \( \psi_k \) on levels \( k > 0 \) by the same mechanism, i.e., by using Elias \( \delta \)-codes on the list of differences of consecutive \( \psi_k \)-values. We therefore define

\[
\Delta_k[i] = \begin{cases} 
\psi_k[i] - \psi_k[i-1] & \text{if } T^k_{A_k[i]} = T^k_{A_k[i-1]} \\
\psi_k[i] & \text{otherwise.}
\end{cases}
\]

How can we decompress the \( \psi_k \)-values from the stream \( S_k \) of \( \delta \)-encoded \( \Delta_k \)-values? For this purpose we store \( \psi_k[i] \) explicitly if either position \( i \) marks the beginning of a new character in \( T^k \) (second case in the definition of \( \Delta_k \)), or if the length of the encoded bit-stream since the last sampled \( \psi_k \)-value exceeds \( s = \frac{\log n}{2} \) bits. To implement this, we introduce three new arrays:

1. \( D_k \) is a bit vector such that \( D_k[i] = 1 \) iff \( \psi_k[i] \) is sampled. \( D_k \) is enhanced with data structures for constant-time \( \text{rank} \) and \( \text{select} \) queries.

2. \( R_k \) is an array that stores the sampled values of \( \psi_k \). All \( \psi_k \)-values stored in \( R_k \) are removed from the bit-stream \( S_k \) (they need not to be stored twice!).

3. \( P_k \) is a bit stream of the same size as \( S_k \) and marks those positions in \( S_k \) with a ‘1’ where a \( \delta \)-encoded \( \Delta_k \)-value starts. \( P_k \) is prepared for \( O(1) \) \( \text{select} \) queries. Then \( \text{select}_1(P_k, i) \) points to the \( i \)’th \( \Delta_k \)-value \( S_k[i] \).

Example 34. Assuming \( s = 5 \), we have the following structures:

\[
\begin{align*}
\psi_0 &= 9 \quad 10 \quad 12 \quad 14 \quad 16 \quad 1 \quad 2 \quad 4 \quad 3 \quad 5 \quad 6 \quad 15 \\
\Delta_0 &= 9 \quad 1 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 2 \quad 3 \quad 2 \quad 1 \quad 9 \\
D_0 &= 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \\
R_0 &= 9 \quad 14 \quad 1 \quad 3 \quad 15 \\
P_0 &= 11000 \quad 1000 \quad 11000 \quad 10001 \\
S_0 &= (00100001) \quad 10100 \quad (0100) \quad (0100) \quad (1) \quad 10100 \quad (0101) \quad 01001 \quad 00100001
\end{align*}
\]

We can decode \( \psi_k[i] \) as follows. First compute the number of sampled \( \Delta_k \)-values up to position \( i \) by \( y = \text{rank}_1(D_k, i) \). Then check if \( \Delta_k[i] \) is represented explicitly (\( D_k[i] = 1 \)), and return \( R_k[y] \) in this case. Otherwise (\( D_k[i] = 0 \)), compute the greatest index \( j \) such that \( \psi_k \) is sampled by \( j = \text{select}_1(D_k, y) \). The result is then \( R_k[y] = \Delta_k[j] \), plus the sum of the \( (i-j) \) values \( \Delta_k[j+1], \ldots, \Delta_k[i] \) that follow \( \Delta_k[j] \) in \( S_k \). Note that \( D_k[j+1] = 0 \), and that the 0’s in \( D_k \) correspond to the 1’s in \( P_k \). As \( \Delta_k[j+1] \) is the \( z \)’th encoded \( \Delta_k \)-value in \( S_k \), with \( z = \text{rank}_0(D_k, j+1) \to \text{rank}_0(D_k, j+1) \).
1) \( j + 1 - \text{RANK}_1(D_k, j + 1) = j + 1 - y \), we thus go to position \( \text{SELECT}_1(P_k, z) \) in \( S_k \), from where we decode the values \( \Delta_k[j + 1], \ldots, \Delta_k[i] \), and return \( R_k[y] + \sum_{i=j+1}^{i} \Delta_k[i] \) as the result \( \psi_k[i] \). This decoding is possible because the \( \delta \)-code is prefix-free (no codeword is a prefix of a different codeword).

To compute this sum in \( O(1) \) time, we use again the Four-Russians-Trick: in a global lookup-table \( G \), for all bit-vectors \( V \) of length \( s \) and all positions \( i \in [1, s] \), \( G[V][i] \) stores the answer to \( \sum_{j=1}^{i} y_j \), if we interpret \( V \) as a sequence of \( \delta \)-encoded values \( y_1, y_2, \ldots \). Note that some values in \( G \) are undefined, because not at all positions \( i \in [1, s] \) there ends a \( \delta \)-encoded value in \( V \), and not all bit-vectors \( V \) represent a correct sequence of \( \delta \)-codes, but these values will never be accessed by the algorithm.

**Example 35.**

\[
G:
\begin{array}{c|ccccc}
V & 1 & 2 & 3 & 4 & 5 \\
00000 & - & - & - & - & - \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
10100 & 1 & 3 & - & - & - \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
11111 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

\[ s = 5 \]

### 4.7 Space Analysis

We now analyze the space requirement of the compressed suffix array. Recall that on level \( k < h \), we store bit-vectors \( B_k, D_k, S_k \), and \( P_k \) (plus some data structures for \( \text{RANK} \) and \( \text{SELECT} \)), and array \( R_k \). On level \( h \), we only store \( A_h \), which needs \( O(n \log \sigma) \) bits. Thus it remains to be shown that an level \( k < h \) the space is \( O(n \log \sigma) \) bits. Then the total space on all \( 1 + \frac{1}{\epsilon} \) levels is \( O(\frac{1}{\epsilon} n \log \sigma) \) bits.

The bit-vectors \( B_k \) and \( D_k \) are certainly of size \( O(n) \) bits each, as they are never longer than \( n \), the length of the text. Actually, the total size of all \( B_k \)'s can be bounded by \( 2n \) bits, because the length of the \( B_k \)-vectors is at least halved from one level to the next:

\[
\sum_{k=0}^{h-1} |B_k| = \sum_{k=0}^{h-1} n_k = \sum_{k=0}^{h-1} n \frac{1}{2^k} = n \sum_{k=0}^{h-1} \frac{1}{2^k} \leq n \sum_{k=0}^{\infty} \frac{1}{2^k} = 2n .
\]

The total size of the \( D_k \)'s is even smaller. Together with the data structures for constant-time \( \text{RANK} \)- and \( \text{SELECT} \)-queries, the space for all \( B_k \)'s and \( D_k \)'s can be upper bounded by \( 4n + o(n) \) bits in total.

Let us now analyze the space for the bit-stream \( S_k \) on a fixed level \( k < h \). For simplicity, we assume that \( S_k \) stores all \( \Delta_k \)-values, also those that are stored explicitly in \( R_k \) and thus deleted from \( S_k \). Let \( n_k^c \) denote the number of positions in \( \psi_k \) corresponding to suffixes that start with the same character \( c \in \Sigma^k \), and let \( \Delta_k[i, n_k^c] \) denote the corresponding sub-array in \( \Delta_k \). Thus,
by Lemma 19, $S_k$ stores at most $\sigma^{2k}$ increasing sequences from the range $[1, n_k]$, each encoded by $\delta$-codes of the differences $\Delta_k$. Therefore, the space is

$$|S_k| = \sum_{c \in \Sigma^{2k}} \sum_{i=1}^{n_c} \left( \lfloor \log \Delta_k^c[i] \rfloor + 2 \lfloor \log (\lfloor \log \Delta_k^c[i] \rfloor + 1) \rfloor + 1 \right)$$

$$= \sum_{c \in \Sigma^{2k}} \sum_{i=1}^{n_c} \left( \lfloor \log \Delta_k^c[i] \rfloor + 2 \log \log \Delta_k^c[i] \right) + O(n_k)$$

$$\leq \sum_{c \in \Sigma^{2k}} \sum_{i=1}^{n_c} \left( \log \frac{n_k}{n_k^c} + 2 \log \log \frac{n_k}{n_k^c} \right) + O(n_k)$$

$$= \sum_{c \in \Sigma^{2k}} n_c^c \left( \log \frac{n_k}{n_k^c} + 2 \log \log \frac{n_k}{n_k^c} \right) + O(n_k)$$

$$\leq \sum_{c \in \Sigma^{2k}} n_c^c \left( \log \sigma^{2k} + 2 \log \log \sigma^{2k} \right) + O(n_k)$$

$$= \left( \log \sigma^{2k} + 2 \log \log \sigma^{2k} \right) \sum_{c \in \Sigma^{2k}} n_c^c + O(n_k)$$

$$= \left( 2^k \log \sigma + 2 \log 2^k \log \sigma \right) n_k + O(n_k)$$

$$= \left( 2^k \log \sigma + 2 \log 2^k \log \sigma \right) \frac{n}{2^k} + O(n_k)$$

$$= n \log \sigma + O(n \log \log \sigma) \text{ bits.}$$

Here, both inequalities follow from the fact that the sum of logarithms is largest when the values are spread evenly over the interval: if $\sum_{i=1}^{m} x_i \leq x$ for a sequence of $m$ real numbers with $x_i \geq 1$ for all $i$, then $\sum_{i=1}^{m} \log x_i \leq \sum_{i=1}^{m} \log \frac{x}{m}$.

Because $P_k$ is of the same size as $S_k$, we can upper bound the space for $P_k$ (including the data-structure for SELECT) by $O(n \log \sigma)$ bits.

Finally, the array $R_k$ of sampled values consist of

$$|R_k| = \left( \frac{\Sigma^{2k}}{\log n} \right) + \frac{|S_k|}{\log n} \times \frac{\log n_k}{\text{value from } [1, n_k]}$$

$$= \left( \sigma^{2k} + \frac{n \log \sigma}{\log n} \right) \log n_k$$

$$\leq O \left( \left( \sigma^{2k} + \frac{n \log \sigma}{\log n} \right) \log n \right)$$

$$= O \left( \left( \frac{n}{\log n} + \frac{n \log \sigma}{\log n} \right) \log n \right)$$

$$= O(n \log \sigma) \text{ bits.}$$

34
We summarize this section in a final theorem:

**Theorem 20.** The suffix array $A$ of a text of length $n$ over an alphabet of size $\sigma$ can be stored in $O(\frac{1}{\epsilon} n \log \sigma)$ bits such that retrieving an arbitrary entry $A[i]$ from the suffix array with $1 \leq i \leq n$ takes $O(\log \frac{1}{\epsilon} n)$ time.

5 Compressed Texts with Random Access

We have already seen how to compress a text $T = t_1 \ldots t_n$, e.g., with dictionary-based algorithm like the LZ-factorization, or with algorithms based on the BWT (see Chapter 2). The drawback of these methods is that the compressed text does not allow random access to the original characters in $T$: to read a single character $t_i$, $1 \leq i \leq n$, the whole text has to be decompressed first in the worst case. In this chapter we show how to compress $T$ such that $O(1)$-random access to any character $t_i$ is still guaranteed.

5.1Recommended Reading


5.2Empirical Entropy of Texts

Before presenting the details of the compression scheme, let us first define a measure of compressibility of a given text $T$. In coding theory, it is often assumed that $T$ is generated by a source that emits characters with varying probabilities; the entropy of that source is then taken as the measure of compressibility for all strings generated by this source. This concept, however, has several problems in the analysis of algorithms. As an example, two texts $T$ and $T'$ can be generated by the same source, but still have a completely different degree of compressibility. For this and other reasons, we shall resort to a slightly different concept, the $k$-th order empirical entropy, which has proved to be useful in the context of algorithm analysis, as it provides a worst-case measure for the performance of algorithms, but still relates the space usage to the text compressibility.

We first define the empirical entropy of order 0.

**Definition 17.** Let $T$ be a text of size $n$ over an alphabet $\Sigma$, and let $n_c$ denote the number of occurrences of character $c \in \Sigma$ in $T$. Then

$$H_0(T) = \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \log \frac{n}{n_c}$$

is called $T$’s empirical entropy of order zero.

Note that in the above definition, $\frac{n_c}{n}$ denotes the relative frequency of character $c$ in $T$, and $\log \frac{n}{n_c} = -\log \frac{n_c}{n}$ denotes the least possible number of bits needed to encode $c$, if more frequent letters are to be assigned shorter codewords. In this sense $H_0(T)$ denotes the average number of
bits needed to encode character $c$, and hence $nH_0(T)$ is a lower bound on the output of a contextless compressor, when applied to text $T$. Here, “contextless” means that the compressor assigns a fixed codeword $w(c)$ to every character $c \in \Sigma$, and then outputs $w(t_1), w(t_2), \ldots, w(t_n)$ as the compressed text, as opposed to adapting the encoding of $t_i$ to the context, say $T_{i-k\ldots i+k}$ for some $k$, where $t_i$ appears. The Huffman-code from Sect. 2.5.1, when applied directly to $T$ (as opposed to the transformed text $T^{\text{bwt}}$), is an example of such a compressor, and indeed it can be shown that the output of Huffman Coding is bounded by $O(nH_0(T))$.

Example 36.

\[
T = AA \ldots A = A^n, \quad H_0(T) = 0
\]
\[
T' = ACACAC \ldots AC = (AC)^\frac{n}{2}, \quad H_0(T') = 1
\]
\[
T'' = CCC \ldots CG \ldots G = C^\frac{n}{2}G^\frac{n}{2}, \quad H_0(T'') = 1
\]
\[
T''' = CCC \ldots CA = C^{n-1}A, \quad H_0(T''') \approx 1
\]

In general, we have $0 \leq H_0 \leq \log \sigma$, and the worst case $H_0 = \log \sigma$ appears when all characters have the same relative frequency, as in $T'$ and $T''$ in the example above. In such a case, the best one can do (assuming $\log \sigma$ is integer) is to assign codewords of equal length $\log \sigma$ to every character $c \in \Sigma$. This is exactly what we do when we store the text in uncompressed form, when we always need $n \log \sigma$ bits to store $T$.

The next step is to consider a preceding context of length $k$.

**Definition 18.** For $s \in \Sigma^k$, let $T^s$ denote the text obtained from concatenating all characters following the occurrences of $s$ in $T$, in the order of $T$. Then the empirical entropy of order $k$ is defined as

\[
H_k(T) = \sum_{s \in \Sigma^k, T^s \neq \epsilon} \frac{|T^s|}{n} H_0(T^s)
\]

With this definition, $nH_k(T)$ is the best possible space that a compressor can achieve if it considers the length-$k$ context $T_{i-k\ldots i-1}$ before encoding $t_i$.

**Example 37.**

\[
T = A^n, \quad H_0(T) = H_1(T) = \ldots = 0
\]
\[
T' = (AC)^\frac{n}{2}, \quad H_0(T') = 1, H_1(T') = H_2(T') = \ldots = 0
\]
\[
T'' = ACGCACGCACGC \ldots
\]
\[
H_1(T'') = \frac{|T^A|}{n} H_0(T^A) + \frac{|T^C|}{n} H_0(T^C) + \frac{|T^G|}{n} H_0(T^G)
\]
\[
\approx \frac{|C|^\frac{n}{2}}{n} H_0(C^\frac{n}{2}) + \frac{|AG|^\frac{n}{2}}{n} H_0((AG)^\frac{n}{2}) + \frac{|C|^\frac{n}{2}}{n} H_0(C^\frac{n}{2})
\]
\[
= \frac{1}{2} \times 0 + \frac{1}{2} \times \frac{1}{2} + \frac{1}{4} \times 0
\]
\[
= 0
\]

In general, we have the following “hierarchy”:

\[
\log \sigma \geq H_0 \geq H_1 \geq H_2 \geq \cdots \geq 0
\]

This makes sense, as a compressor that considers a longer context (higher $k$) has “better chances” to predict the next character.
5.3 Compression Scheme

We partition $T$ into $2^n$ blocks $B_1, \ldots, B_{2^n}$ of size $s = \lfloor \frac{1}{2} \log_\sigma n \rfloor$. The $i$'th block is $B_i = T_{(i-1)s+1 \ldots is}$. As in previous chapters, this blocking is only done conceptually. Let $S$ denote the set of distinct blocks in $T$. Note that $|S| = O(\sigma^s) = O\left(n^{1/2}\right)$, as there are only $\sigma^s$ different substrings of size $s$.

Example 38.

$$T = GAT\{AC\{A\}CAT\{AC\}GAT\{ACA\}CAT\{TAC\}$$
$$s = 3$$
$$S = \{ACA, CAT, GAT, TAC\}$$

We now sort the elements in $S$ in decreasing order of their occurrence in $T$.

Example 39. Continuing the example from above, the sorted sequence (together with the frequencies) is TAC(4), ACA(2), GAT(2), CAT(1).

For a block $B_i$, let $r(B_i)$ be the rank of $B_i$ in the sorted sequence $S$. To get the compressed text $T'$, we replace $B_i$ by the binary string that has rank $r(B_i)$ in $C$, the canonical enumeration of all binary strings:

$$C = \epsilon, 0, 01, 10, 11, 000, 001, \ldots$$

Example 40.

$$T' = GAT\{TAC\{ACA\}CAT\{AC\}GAT\{ACA\}CAT\{TAC\}$$
$$T' = 1 \epsilon 0 \epsilon 1 0 00 \epsilon \epsilon$$

Hence, the compressed string is $T' = 101000$.

For decoding the contents of the blocks, we use again a global lookup table $P$: for $1 \leq i \leq |S|$, $P[i]$ stores the length-$s$ string $\alpha$ if $\alpha$ has rank $i$ in the sorted sequence $S$.

Example 41.

$$\begin{array}{c|c|c|c|c}
  i & 1 & 2 & 3 & 4 \\
  \hline
  P[i] & TAC & ACA & GAT & CAT \\
\end{array}$$

Table $P$ needs order of

$$\sigma^s \times s \times \log \sigma = \sqrt{n} \log n \log \sigma = o(n)$$

bits.

Finally, we group $\log n$ blocks into superblocks consisting of $s' = s \log n$ characters in $T$. In a new table $M'[1, \frac{n}{s}]$ we store in $M'[i]$ the beginning of the encoding of the $i$'th superblock in the compressed sequence $T'$. Also, in a new table $M[1, \frac{n}{s}]$, we store in $M[i]$ the beginning of the
encoding of the $i$'th block in $T'$, this time only relative to the beginning of the enclosing superblock. Table $M'$ needs

$$|M'| = O\left(\frac{n}{s} \times \log |T'|\right)$$

$$= O\left(\frac{n \log (n \log \sigma)}{\log n \log \sigma \log n}ight)$$

$$= O\left(\frac{n \log n}{\log n \log \sigma n}\right)$$

$$= O\left(n/\log \sigma n\right)$$

$$= o(n)$$

bits of space. (We used the upper bound of $O(n \log \sigma)$ for the length of $T'$ in this calculation, which makes sense as this is the size of the uncompressed text $T$, and $T'$ is certainly not longer.) Table $M$ needs

$$|M| = O\left(\frac{n}{s} \times \log (s' \log \sigma)\right)$$

$$= O\left(\frac{n \log (\log n \log \sigma \log \sigma n)}{\log \sigma n}\right)$$

$$= O\left(\frac{n \log \log n}{\log \sigma n}\right)$$

bits.

**Example 42.**

\[
\begin{align*}
T' &= \text{GATTACA|CATA|CAGTAC|CATAC|TAC}\|\text{GATACACATTACTAC} \\
T' &= 1 \epsilon 0 \epsilon 1 0 00 \epsilon \epsilon \\
M' &= 1 \ 2 \ 3 \ 4 \ 5 \\
M &= 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1
\end{align*}
\]

It can be shown that the compressed string $T'$ is of size $nH_k(T) + O\left(\frac{nk \log \sigma}{\log \sigma n}\right)$ bits, simultaneously over all $k \in o(\log \sigma n)$. The proof of this claim, however, requires more time than we have in this lecture.

We finally show how to retrieve an arbitrary character $t_i$ in $O(1)$ time. We compute the starting position of $t_i$’s block encoding in $T'$ as $a = M' \left[\left\lceil\frac{i-1}{s'}\right\rceil + 1\right] + M \left[\left\lceil\frac{i-1}{s}\right\rceil + 1\right]$, and the starting position of the next block as $b = M' \left[\left\lceil\frac{i-1+s'}{s'}\right\rceil + 1\right] + M \left[\left\lceil\frac{i-1+s}{s}\right\rceil + 1\right]$. Then we fetch the encoding of the block as $T'_{a...b-1}$, which can be done in $O(1)$ time, as the block encodings have size $O\left(\log (\sigma^s)\right) = O(\log n)$ bits.

To decode the real contents of $t_i$’s block, we need to consult $P$. We interpret $e = T'_{a...b-1}$ as a binary number, and thus need to find the rank of $e$ in the canonical enumeration $C$ of all binary strings. Note that if the length of $e$ is $l$, then there are $1 + 2 + 3 + 4 + \cdots + 2^{l-1} = 2^l - 1$ binary strings before $0^l$ in $C$, as the number of binary strings of length $x$ is $2^x$. Hence, $e$ has rank
\[ r = 2^l - 1 + e + 1 = 2^l + e \] in \( C \). Thus, we retrieve \( P[r] \) as the contents of \( t_i \)'s block, and return the \(((i - 1) \mod s) + 1\)'st character therein.

We summarize this chapter in a final theorem.

**Theorem 21.** A string \( T \) can be stored compressed in \( nH_k + O \left( \frac{n(k \log \sigma + \log \log n)}{\log \sigma n} \right) \) bits, simultaneously over all \( k \in o(\log \sigma n) \), such that constant time access to the characters in \( T \) is guaranteed.

### 6 Simulation of Suffix Trees

So far, we have seen compressed text indices that have only one functionality: locating all occurrences of a search pattern \( P \) in a text \( T \). In some cases, however, more functionality is required. From other courses you might know that many sequence-related problems are solved efficiently with suffix trees (e.g., computing tandem repeats, MUMs, ...). However, the space requirement of a suffix tree is huge: it is at least 40 times higher than the space of the DNA itself, using very proprietary implementations that support only a very small number of all conceivable suffix tree operations. In this chapter, we present a generic approach that allows for the simulation of all suffix tree operations, by using only compressed data structures. More specifically, we will build on the compressed suffix array from Chapter 4, and show how all suffix tree operations can be simulated by computations on suffix array intervals (the same intervals that we used for suffix trays). Space-efficient data structures that facilitate these computations will be handled in subsequent chapters.

#### 6.1 Recommended Reading


#### 6.2 Basic Concepts

The reader is encouraged to recall the definitions from Sect. 1.2. From now on, we regard the suffix tree as an abstract data type that supports the following operations.

**Definition 19.** A suffix tree \( S \) supports the following operations.

- \( \text{ROOT}() \): returns the root of the suffix tree.
- \( \text{ISLEAF}(v) \): true iff \( v \) is a leaf.
- \( \text{LOCATE}(v) \): returns \( l(v) \) if \( v \) is a leaf, and \text{NULL} otherwise.
- \( \text{ANCESTOR}(v, w) \): true iff \( v \) is an ancestor of \( w \).
- \( \text{SDEPTH}(v, w) \): returns \( d(v) \), the string-depth of \( v \).
- \( \text{COUNT}(v) \): the number of leaves in \( S_v \).
• **Parent**(*v*): the parent node of *v*.

• **FChild**(*v*): the alphabetically first child of *v*.

• **NSibling**(*v*): the alphabetically next sibling of *v*.

• **LCA**(*v*): the lowest common ancestor of *v* and *w*.

• **SLINK**(*v*): the suffix-link of *v* (the node *w* such that \( \pi = \alpha \pi \) for \( \alpha \in \Sigma \)).

• **Child**(*v*, *a*): node *w* such that the edge-label of \((v, w)\) starts with \( \alpha \in \Sigma \).

• **Edge-label**(*v*, *i*): the *i*’th letter on the edge \((Parent(v), v)\).

We recall from Chapter 1.2 that \( A \) denotes the suffix array, \( H \) the lcp-array, and \( \text{RMQ} \) a range minimum query. Because we will use compressed data structures, we will use variables \( t_A, t_\pi, t_H \) and \( t_{\text{RMQ}} \) for the access time to the corresponding array/function. E.g., with uncompressed (plain) arrays, we have \( t_A = t_\pi = t_H = t_{\text{RMQ}} = O(1) \), while with the CSA from Chapter 4 we have \( t_A = O(\log^* n) \) (constant \( \epsilon \)).

As in the case of suffix trays, we represent a node *v* in \( S \) by the interval \([l_v, r_v]\) such that \( A[l_v], \ldots, A[r_v] \) are exactly the labels of the leaves in \( S_v \). For such a representation we have the following basic lemma (from now on we assume \( H[1] = H[n + 1] = -1 \)):

**Lemma 22.** Let \([l_v, r_v]\) be the interval of an internal node *v*. Then

1. \( H[l_v] < d(v) \) and \( H[r_v + 1] < d(v) \).

2. For all \( k \in [l_v + 1, r_v] \): \( H[k] \geq d(v) \).

3. There is a \( k \in [l_v + 1, r_v] \) with \( H[k] = d(v) \).

**Proof:** Condition (1) follows because otherwise suffix \( T_{A[r_v + 1], \ldots, n} \) would start with \( \pi \), and hence leaves labeled \( A[l_v] \) or \( A[r_v + 1] \) would also be in \( S_v \). Property (2) follows because all suffixes \( T_{A[k], \ldots, n}, k \in [l_v, r_v] \), have \( \pi \) as their prefix, and hence \( \text{LCP}(T_{A[k], \ldots, n}, T_{A[k-1], \ldots, n}) \geq |\pi| = d(v) \) for all \( k \in [l_v + 1, r_v] \). For proving property (3), for the sake of contradiction assume \( H[k] > d(v) \) for all \( k \in [l_v + 1, r_v] \). Then all suffixes \( T_{A[k], \ldots, n}, k \in [l_v, r_v] \), would start with \( \pi a \) for some \( \alpha \in \Sigma \). Hence, \( v \) would only have one outgoing edge (whose label starts with \( \alpha \)), contradicting the fact \( S \) is a compact \( \Sigma^+ \)-tree.

As a side remark, this is actually an “if and only if” statement, as every interval satisfying the three conditions from Lemma 22 corresponds to an internal node.

**Definition 20.** Let \([l_v, r_v]\) be the interval of an internal node *v*. Any position \( k \in [l_v + 1, r_v] \) satisfying point (3) in Lemma 22 is called a \( d(v) \)-index of *v*.

Our aim is to simulate all suffix tree operations by computations on suffix intervals: given the interval \([l_w, r_w]\) corresponding to node *v*, compute the interval of \( w = f(v) \) from the values \( l_w \) and \( r_w \) alone, where \( f \) can be any function from Def. 19; e.g., \( f = \text{Parent} \). We will see that most suffix tree operations follow a generic approach: first locate a \( d(w) \)-index \( p \) of \( w \), and then search for the (yet unknown) delimiting points \( l_w \) and \( r_w \) of \( w \)’s suffix interval. For this latter task (computation of \( l_w \) and \( r_w \) from \( p \)), we need two more functions, called next and previous smaller value functions, defined as follows.
Definition 21. Given the LCP-array $H$ and an index $1 \leq i \leq n$, the previous smaller value function $PSV_H(i) = \max\{k < i : H[k] < H[i]\}$. The next smaller value function $NSV_H(i)$ is defined similarly for succeeding positions: $NSV_H(i) = \min\{k > i : H[k] < H[i]\}$.

We use $t_{PSV}$ to denote the time to compute a value $NSV_H(i)$ or $PSV_H(i)$. In what follows, we often use simply $PSV$ and $NSV$ instead of $PSV_H$ and $NSV_H$, implicitly assuming that array $H$ is the underlying array. The following lemma shows how these two functions can be used to compute the delimiting points $l_w$ and $r_w$ of $w$’s suffix interval:

Lemma 23. Let $p$ be a $d(w)$-index of an internal node $w$. Then $l_w = PSV(p)$, and $r_w = NSV(p) - 1$.

Proof: Let $l = PSV(p)$, and $r = NSV(p)$. We must show that all three conditions in Lemma 19 are satisfied by $[l, r - 1]$. Because $H[l] < H[p]$ by the definition of $PSV$, and likewise $H[r] < H[p]$, point (1) is clear. Further, because $l$ and $r$ are the closest positions where $H$ attains a smaller value, condition (2) is also satisfied. Point (3) follows from the assumption that $p$ is a $d(w)$-index. We thus conclude that $l_w = l$ and $r_w = r - 1$. \qed

6.3 Suffix Tree Operations

We now step through the operations from Def. 19 and show how they can be simulated by computations on the suffix array intervals. Let $[l_v, r_v]$ denote the interval of an arbitrary node $v$. The most easy operations are:

- $\text{ROOT}()$: returns the interval $[1, n]$.
- $\text{isLeaf}(v)$: true iff $l_v = r_v$.
- $\text{COUNT}(v)$: returns $r_v - l_v + 1$.
- $\text{Ancestor}(v, w)$: true iff $l_w \leq l_v$ and $r_v \leq r_w$.

Time is $O(1)$ for all four operations.

- $\text{LOCATE}(v)$: If $l_v \neq r_v$, return $\text{null}$. Otherwise, return $A[l_v]$ in $O(t_A)$ time.
- $\text{SDEPTH}(v)$: If $l_v = r_v$, return $n - A[l_v] + 1$ in time $O(t_A)$, as this is the length of the $A[l_v]$’th suffix. Otherwise from Lemma 22 we know that $d(v)$ is the minimum LCP-value in $H[l_v + 1, r_v]$. We hence return $H[\text{RMQ}_H(l_v + 1, r_v)]$ in time $O(t_{RMQ} + t_H)$.
- $\text{PARENT}(v)$: Because $S$ is a compact tree, either $H[l_v]$ or $H[r_v + 1]$ equals the string-depth of the parent-node, whichever is greater. Hence, we first set $p = \arg\max\{H[k] : k \in \{l_v, r_v + 1\}\}$, and then, by Lemma 23, return $[\text{PSV}(p), \text{NSV}(p) - 1]$. Time is $O(t_H + t_{PSV})$.
- $\text{FCHILD}(v)$: If $v$ is a leaf, return $\text{null}$. Otherwise, locate the first $d(v)$-value in $H[l_v, r_v]$ by $p = \text{RMQ}_H(l_v + 1, r_v)$. Here, we assume that RMQ returns the position of the leftmost minimum, if it is not unique. The final result is $[l_v, p - 1]$, and the total time is $O(t_{RMQ})$.
- $\text{NSibling}(v)$: First, compute $v$’s parent as $w = \text{PARENT}(v)$. Now, if $r_v = r_w$, return $\text{null}$, since $v$ does not have a next sibling in this case. If $r_v = r_w + 1$, then $v$’s next sibling is a leaf, so we return $[r_w, r_w]$. Otherwise, try to locate the first $d(w)$-value after $r_v + 1$ by $p = \text{RMQ}_H(r_v + 2, r_w)$. If $H[p] = d(w)$, we return $[r_v + 1, p - 1]$ as the final result. Otherwise ($H[p] > d(w)$), the final result is $[r_v + 1, w_r]$. Time is $O(t_H + t_{PSV} + t_{RMQ})$. 

41
• **LCA(v, w):** First check if one of v or w is an ancestor of the other, and return that node in this case. Otherwise, assume \( r_v < l_w \) (otherwise swap \( v \) and \( w \)). Let \( u \) denote the (yet unknown) LCA of \( v \) and \( w \), so that our task is to compute \( l_u \) and \( r_u \). First note that all suffixes \( T_{A[k]...n} \), \( k \in [l_v, r_v] \cup [l_w, r_w] \), must be prefixed by \( \pi \), and that \( u \) is the deepest node with this property. Further, because none of \( v \) and \( w \) is an ancestor of the other, \( v \) and \( w \) must be contained in subtrees rooted at two different children \( u' \) and \( u'' \) of \( u \), say \( v \in T_{u'} \) and \( w \in T_{u''} \). Because \( r_v \leq l_w \), we have \( r_{u'} \leq l_{u''} \), and hence there must be a \( d(u) \)-index in \( H \) between \( r_{u'} \) and \( l_{u''} \), which can be found by \( p = \text{RMQ}_H(r_v, l_w) \). The endpoints of \( u \)'s interval are again located by \( l_u = \text{PSV}(p) \) and \( r_u = \text{NSV}(p) - 1 \). Time is \( O(t_{\text{RMQ}} + t_{\text{PSV}}) \).

![Diagram of LCA](image)

• **SLINK(v):** If \( v \) is the root, return `null`. If \( v \) is a leaf, return \([\psi(l_v), \psi(l_v)]\), as the \( \psi \)-function at position \( l_v \) tells us where suffix \( T_{A[l_v]...n} \) can be found in \( A \). Otherwise, follow the suffix links of the leftmost and rightmost leaf in \( S_v \) by computing \( x = \psi(l_v) \) and \( y = \psi(r_v) \). Now note that for the string depth of the target node \( w = \text{SLINK}(v) \), it must hold \( d(w) = d(v) - 1 \), as exactly one character is “cut-off” from the beginning of \( v \) by following the suffix link. To locate a value with this string depth, we compute again \( p = \text{RMQ}(x + 1, y) \), as in the NSIBLING- and in the LCA-operation. Such a \( d(w) \)-index must exist between \( x \) and \( y \), as the leaves \([l_v, l_v] \) and \([r_v, r_v] \) are located in subtrees rooted at two different children of \( v \), and hence leaves \([x, x] \) and \([y, y] \) must also be below two different children of \( w \). Finally, we return \([\text{PSV}(p), \text{NSV}(p) - 1]\). Total time is \( O(t_{\psi} + t_{\text{RMQ}} + t_{\text{PSV}}) \).
6.4 Compressed LCP-Arrays

We now show how to reduce the space for the LCP-array $H$ from $n \log n$ to $O(n)$ bits. To this end, we first note that the LCP-value can decrease by at most 1 when moving from suffix $A[i]$ to $A[i+1]$ in $H$ (i.e., when enumerating the LCP-values in text order):

Lemma 24. For all $1 \leq i \leq n$, $H[\psi[i]] \geq H[i] - 1$.

Proof: If $H[i] = 0$, the claim is trivial. Hence, suppose $H[i] > 0$, and look at the two suffixes starting at positions $A[i]$ and $A[i-1]$, that must start with the same character. Suppose $T_{A[i]...n} = a\alpha$ and $T_{A[i-1]...n} = a\beta$ for $a \in \Sigma$, $\alpha, \beta \in \Sigma^*$.

Because the suffixes are sorted lexicographically in $A$, and $a\alpha >_{\text{lex}} s\beta$, we know $a >_{\text{lex}} \beta$, and that $\alpha$ and $\beta$ store a common prefix of length $H[i] - 1$, call it $\gamma$. Now note that all suffixes between $\beta$ and $\alpha$ in $A$ must also start with $\gamma$, as otherwise the suffixes would not be in lexicographically order.
In particular, suffix $T_{\psi(i)-1...n}$ must be prefixed by $\gamma$, and hence $H[\psi(i)] = \text{LCP}(\alpha, T_{\psi(i)-1...n}) \geq |\gamma| = H[i] - 1$.

From the above lemma, we can conclude that $I[1, n] = [H[A^{-1}[1]] + 1, H[A^{-1}[2]] + 2, H[A^{-1}[3]] + 3, \ldots, H[A^{-1}[n]] + n]$ is an array of increasing integers. Further, because no LCP-value can exceed the length of corresponding suffixes, we see that $H[A^{-1}[i]] \leq n - i + 1$. Hence, sequence $I$ must be in range $[1, n]$. We encode $I$ differentially: writing $\Delta[i] = I[i] - I[i - 1]$ for the difference between entry $i$ and $i - 1$, and defining $I[0] = 0$ for handling the border case, we encode $\Delta[i]$ in unary as $0^{\Delta[i]}1$. Let the resulting sequence be $S$.

Note that the number of 1’s in $S$ is exactly $n$, and that the number of 0’s is at most $n$, as the $\Delta[i]$’s sum up to at most $n$. Hence, the length of $S$ is at most $2n$ bits. We further prepare $S$ for constant-time \texttt{rank}$_0$- and \texttt{select}$_1$-queries, using additional $o(n)$ bits. Then $H[i]$ can be retrieved by

$$H[i] = \text{rank}_0(S, \text{select}_1(S, A[i])) - A[i].$$

This is because the \texttt{select}-statement points to the position of the terminating ‘1’ of $0^{\Delta[A[i]]}1$ in $S$, and the \texttt{rank}-statement counts the sum of $\Delta$-values before that position, which is $I[A[i]]$. From this, in order to get $H[i]$, we need to subtract $A[i]$, which has bin “artificially” added when deriving $I$ from $H$. We have proved:

**Theorem 25.** The LCP-array $H$ can be stored in $2n + o(n)$ bits such that retrieving an arbitrary entry $H[i]$ takes $t_H = O(t_A)$ time.

Note that with the CSA from Chapter 4, this means that we no more have constant-time access to $H$, as $t_A = O(\log^* n)$ in this case.

## 7 Succinct Data Structures for RMQs and PSV/NSV Queries

This chapter shows that $O(n)$ bits are sufficient to answer RMQs and PSV/NSV-queries in constant time. For our compressed suffix tree, we assume that all three queries are executed on the LCP-array $H$, although the data structures presented in this chapter are applicable to any array of ordered objects.

### 7.1 2-Dimensional Min-Heaps

We first define a tree that will be the basis for answering RMQs and NSV-queries. The solution for PSV-queries is symmetric. The following definition assumes that $H[n+1]$ is always the smallest value in $H$, what can be enforced by introducing a “dummy” element $H[n+1] = -\infty$. 

<table>
<thead>
<tr>
<th>T = C A C A A C C A C $</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 10 4 8 2 5 9 3 7 1 6</td>
</tr>
<tr>
<td>H = 0 0 1 2 2 0 1 2 3 1</td>
</tr>
<tr>
<td>I = 4 4 4 4 7 7 9 9 9 10</td>
</tr>
<tr>
<td>S = 00001 1 1 0001 1 001 1 1 01</td>
</tr>
</tbody>
</table>
Definition 22. Let $H[1, n + 1]$ be an array of totally ordered objects, with the property that $H[i] < H[i]$ for all $1 \leq i \leq n$. The 2-dimensional Min-Heap $\mathcal{M}_H$ of $H$ is a tree on $n$ nodes $1, \ldots, n$, defined such that $NSV(i)$ is the parent-node of $i$ for $1 \leq i \leq n$.

Note that $\mathcal{M}_H$ is a well-defined tree whose root is $n + 1$.

Example 43.

![Diagram of 2d-Min-Heap]

$H = \{-1, 0, 0, 3, 1, 2, 0, 1, 1\}$

From the definition of $\mathcal{M}_H$, it is immediately clear that the value $NSV(i)$ is given by the parent node of $i$ ($1 \leq i \leq n$). The next lemma shows that $\mathcal{M}_H$ is also useful for answering RMQs on $H$.

Lemma 26. For $1 \leq i < j \leq n$, let $l = \text{LCA}_{\mathcal{M}_H}(i, j)$. Then if $l = j$, $\text{RMQ}_H(i, j) = j$. Otherwise, $\text{RMQ}_H(i, j)$ is given by the child of $l$ that is on the path from $l$ to $i$.

Proof: “graphical proof”:

Example 44. Continuing the example above, let $i = 4$ and $j = 6$. We have $\text{LCA}_{\mathcal{M}_H}(4, 6) = 7$, and $5$ is the child of $7$ on the path to $4$. Hence, $\text{RMQ}_H(4, 6) = 5$.

7.2 Balanced Parentheses Representation of Trees

Any ordered tree $T$ on $n$ nodes can be represented by a sequence $B$ of $2n$ parentheses as follows: in a depth-first traversal of $T$, write an opening parenthesis '(' when visiting a node $v$ for the first time, and a closing parenthesis ')' when visiting $v$ for the last time (i.e., when all nodes in $T_v$ have been traversed).

Example 45. Building on the 2d-Min-Heap from the Example 43, we have $B = (())()(()())()$.
In a computer, a ‘(’ could be represented by a ‘1’-bit, and a ‘)’ by a ‘0’-bit, so the space for $B$ is $2n$ bits. Note further that this representation allows us to answer queries like $\text{RANK}(B, i)$ and $\text{SELECT}(B, i)$, by using only $o(n)$ additional space (see Sect. 3.5).

Note that the sequence $B$ is balanced, in the sense that in each prefix the number of closing parentheses is no more than the number of opening parenthesis, and that there are $n$ opening and closing parentheses each in total. Hence, this representation of trees is called balanced parentheses sequence (BPS).

We also need the following operation.

**Definition 23.** Given a sequence $B[1, 2n]$ of balanced parentheses and a position $i$ with $B[i] = ’)$’, $\text{ENCLOSE}(B, i)$ returns the position of the closing parenthesis of the nearest enclosing ‘(’-pair.

In other words, if $v$ is a node with closing parenthesis at position $i < 2n$ in $B$, and $w$ is the parent of $v$ with closing parenthesis at position $j$ in $B$, then $\text{ENCLOSE}(B, i) = j$. Note that $\text{ENCLOSE}(i) > i$ for all $i$, because of the order in which nodes are visited in a depth first traversal.

**Example 46.**

$$B = ( ( ) ( ) ( ) ( ( ( ) ) ( ) ) ( ) ( ) )$$

$$\text{ENCLOSE}$$

We state the following theorem without proof.

**Theorem 27.** There is a data structure of size $O\left(\frac{n \log \log n}{\log n}\right) = o(n)$ bits that allows for constant-time $\text{ENCLOSE}$-queries.

(The techniques are roughly similar to the techniques for $\text{RANK}$- and $\text{SELECT}$-queries.)

Now look at an arbitrary position $i$ in $B$, $1 \leq i \leq 2n$. We define the excess-value $E[i]$ at position $i$ as the number of opening parenthesis in $B[1, i]$ minus the number of closing parenthesis in $B[1, i]$. Note that the excess-values do not have to be stored explicitly, as

$$|E[i]| = \text{RANK}(B, i) - \text{RANK}(B, i) = i - \text{RANK}(B, i) - \text{RANK}(B, i) = i - 2\text{RANK}(B, i).$$

**Example 47.**

$$B = ( ( ) ( ) ( ) ( ( ( ) ) ( ) ) ( ) ( ) )$$

$$E = 1\ 2\ 1\ 2\ 1\ 2\ 1\ 2\ 1\ 2\ 1\ 0$$

Note:

1. $E[i] > 0$ for all $1 \leq i < 2n$
2. $E[2n] = 0$
3. If \( i \) is the position of the closing parenthesis of node \( v \), then \( E[i] \) is the depth of \( v \). (Counting starts at 0, so the root has depth 0.)

We also state the following theorem without proof.

**Theorem 28.** Given a sequence \( B \) of balanced parentheses, there is a data structure of size \( O\left(\frac{n\log\log n}{\log n}\right) = o(n) \) bits that allows to answer RMQs on the associated excess-sequence \( E \) in constant time.

(The techniques are again similar to RANK and SELECT: blocking and table-lookups. Note in particular that \( \frac{\log n}{2} \) excess-values \( E[x], E[x+1], \ldots, E\left[x + \frac{\log n}{2} - 1\right] \) are encoded in a single computer-word \( B\left[x, x + \frac{\log n}{2} - 1\right] \), and hence it is again possible to apply the Four-Russians-Trick!)

### 7.3 Answering Queries

We represent \( \mathcal{M}_H \) by its BPS \( B \), and identify each node \( i \) in \( \mathcal{M}_H \) by the position of its closing parenthesis in \( B \).

**Example 48.**

\[
\begin{align*}
\mathcal{M}_H & = \begin{array}{c}
\circ \\
\circ & \circ & \circ & \circ
\end{array} \\
B & = ( ( ( ) ( ( ( ( ) ( ))( ) ) ) ) ) \\
E & = 1 2 1 2 1 2 1 2 3 4 3 2 3 2 1 2 1 2 1 0
\end{align*}
\]

Note that the (closing parenthesis of) nodes appear in \( B \) in sorted order - this is simply because in \( \mathcal{M}_H \) node \( i \) has post-order number \( i \), and the closing parenthesis appear in post-order by the definition of the BPS. This fact allows us to jump back and forth between indices in \( H \) and positions of closing parentheses \( ')' \) in \( B \), by using rank- and select-queries in the appropriate sequences.

Answering NSV-queries is now simple. Suppose we wish to answer \( NSV_H(i) \). We then move to the position of the \( i \)'th \( ')' \) by

\[ x \leftarrow \text{SELECT}_j(B, i) , \]

and then call

\[ y \leftarrow \text{ENCLOSE}(B, x) \]

in order to move to the position \( y \) of the closing parenthesis of the parent \( j \) of \( i \) in \( \mathcal{M}_H \). The (yet unknown) value \( j \) is computed by

\[ j \leftarrow \text{RANK}_j(B, y) . \]

**Example 49.** We want to compute \( NSV(7) \). First compute \( x \leftarrow \text{SELECT}_j(B, 7) = 15 \), and then \( y \leftarrow \text{ENCLOSE}(15) = 20 \). The final result is \( j \leftarrow \text{RANK}_j(B, 20) = 10 . \)
Answering RMQs is only slightly more complicated. Suppose we wish to answer $\text{RMQ}_H(i, j)$ for $1 \leq i < j \leq n$. As before, we go to the appropriate positions in $B$ by

$$x \leftarrow \text{select}_j(B, i) \text{ and } y \leftarrow \text{select}_j(B, j).$$

We then compute the position of the minimum excess-value in the range $[x, y]$ by

$$z \leftarrow \text{RMQ}_E(x, y),$$

and map it back to a position in $H$ by

$$m \leftarrow \text{rank}_j(B, z).$$

This is the final answer.

**Example 50.** We want to compute $\text{RMQ}_H(4, 9)$. First, compute $x \leftarrow \text{select}_j(B, 4) = 11$ and $y \leftarrow \text{enclose}(B, 9) = 19$. The range minimum query yields $z \leftarrow \text{RMQ}_E(11, 19) = 15$. Finally, $m \leftarrow \text{rank}_j(B, 15) = 7$ is the result.

We now justify the correctness of this approach. First assume that $l = \text{LCA}_M_H(i, j)$ is different from $j$. Let $l_1, \ldots, l_k$ be the children of $l$, and assume $i \in T_{l_{\gamma}}$ and $j \in T_{l_{\delta}}$ for some $1 \leq \gamma < \delta \leq k$. By Lemma 26, we thus need to show that the position of the closing parenthesis of $l_{\gamma}$ is the position where $E$ attains the minimum in $E[x, y]$. 

**Example 51.**

Let $d - 1$ be the tree-depth of $l$, and let $B[a, b]$ denote the position of $B$ that “spells out” $T_l$ (i.e., $B[a, b]$ is the BPS of the sub-tree of $T$ rooted at $l$). Note that $a < x < y < b$, as $i$ and $j$ are both below $l$ in $T$.

Because $B[a]$ is the opening parenthesis of node $l$, we have $E[a] = d$. Further, because $B$ is balanced, we have $E[c] \geq d$ for all $a < c < b$. But $E$ assumes the values $d$ at the positions of the closing parenthesis of nodes $l_{\beta}$ (1 \leq \beta \leq k), in particular for $l_{\gamma}$. Hence, the leftmost minimum in $E[x, y]$ is attained at the position $z$ of the closing parenthesis of node $l_{\gamma}$, which is computed by an RMQ in $E$. The case where $l = j$ is similar (and even simpler to prove). Thus, we get:
Theorem 29. With a data structure of size $2n + o(n)$ bits, we can answer RMQs and NSV-queries on an array of $n$ ordered objects on $O(1)$ time.

The drawback of the 2d-Min-Heap, however, is that it is inherently asymmetric (as the parent-relationship is defined by the minimum to the right), and cannot be used for answering PSV-queries as well. For this, we have to build another 2d-Min-Heap $M_{RH}$ on the reversed sequence $H^R$, using another $2n + o(n)$ bits. (Note that an interesting side-effect of this $M_{RH}$ is that it would allow to compute the rightmost minimum in any query range, instead of the leftmost, which could have interesting applications in compressed suffix trees.)

If we plug all these structures into the compressed suffix tree from Chapter 6 (which was indeed the reason for developing the solutions for RMQs and PNSVs), we get:

Theorem 30. A suffix tree on a text of length $n$ over an alphabet of size $\sigma$ can be stored in $6n + |\text{CSA}| + o(n)$ bits of space ($|\text{CSA}|$ denotes the space of the underlying compressed suffix array), such that operations root, isLeaf, Count, Ancestor, FChild, SLink and lca take $O(1)$ time, operations locate, SDepth, Parent, NSibling and edge-label take $O(t_A)$ time ($t_A$ denotes the time to retrieve an element from the CSA), and Child takes $O(t_A \cdot \sigma)$ time.

Acknowledgments

The author wishes to thank Benjamin Albrecht, who did most of the typesetting work. Thanks also to the other students from Tübingen for their numerous corrections on this script.

49