1 Introduction

Definition 1.1
Let $\Sigma = \{0, \dots, \sigma - 1\}$ be a finite, ordered set. The elements of Σ are called *characters* or *symbols* and Σ is called an *alphabet* of size σ.

Definition 1.2
A *string* S is a sequence of characters from an alphabet Σ.

- We usually use $n = |S|$ to be the length of the string.
- The i-th character of S is $S[i]$. Indices are 0-based.
- The substring from the i-th to the j-th character is $S[i..j]$.
- A substring with $i = 0$ is called *prefix*. A substring with $j = n - 1$ is called *suffix*.
- The i-th suffix is $S[i..n-1]$.

1.1 Suffix Tries

Definition 1.3
Let $S = \{S_0, S_1, \dots, S_{N-1}\}$ be a set of strings over an alphabet Σ. A *trie* $\mathcal{T}$ is a tree, where each node represents a different prefix in the set S. The root represents the empty prefix ε. Vertex u representing prefix Y is a child of vertex v representing prefix X, if and only if $Y = Xc$ for some character $c \in \Sigma$. The edge (v, u) is then labeled c.
If S is the set of all suffixes of a string T, the trie is called *suffix trie*.

Example 1.4
Figure 1.1 shows the suffix trie for the string "banana\$". The dollar sign "\$" is a sentinel that does not appear elsewhere in the text. This guarantees, that no suffix is a prefix of another suffix and the suffix trie therefore has $n + 1$ leaves.

To construct a trie over string set $S = \{S_0, \dots, S_{N-1}\}$, we need $\mathcal{O}(|S_0| + \dots + |S_{N-1}|)$ steps. This bound is tight: If all characters are pairwise distinct in all strings and no two strings share a character, than the number of different prefixes and therefore vertices is given by $1 + \sum_{i=0}^{N-1} |S_i|$, where the additional 1 represents the empty prefix ε.
The time needed to search for a string T of length $m = |T|$ in the trie depends on the implementation of the tree. If the children of each vertex are stored in a list, the time is in $\mathcal{O}(m\sigma)$. If the children are stored in a sorted array (using the order of the characters

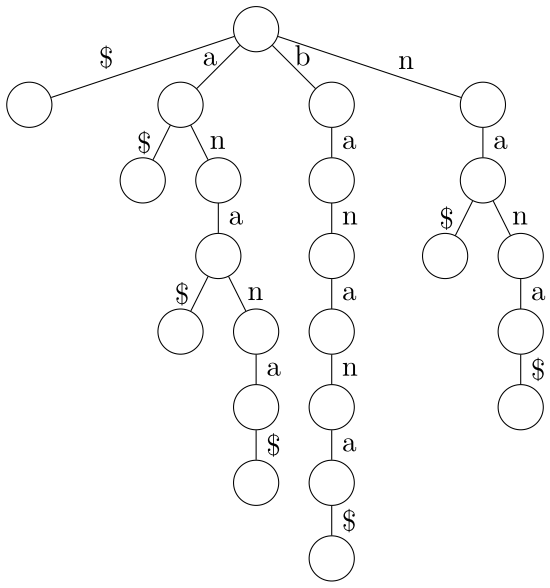

Figure 1.1: The suffix trie for the string "banana$"

in the alphabet), the time is in $\mathcal{O}(m \log \sigma)$. By using a hash table and perfect hashing, the time is in $\mathcal{O}(m)$.

The space needed to store the suffix trie $\mathcal{T}$ for a string of length n is in $\mathcal{O}(n^2 \log \sigma + n^2 \log n)$ bits. The first summand is the space needed to store the $\mathcal{O}(n^2)$ edge labels of one character $c \in \Sigma$ each. The second summand is the space needed to store the pointers to the children of each node.

1.2 Suffix Trees

Definition 1.5
A *suffix tree* $\mathcal{T}$ for a string S is the suffix trie of S where each unary path is converted into a single edge. Those edges are labeled with the concatenation of the characters from the replaced edges. The leaves of the suffix tree store the text position where the corresponding suffix starts.

Example 1.6
Figure 1.2 shows the suffix tree for the string "banana$". It contains only 11 vertices compared to the 23 vertices of the suffix trie.

The suffix array can be constructed in time $\mathcal{O}(n)$ with algorithms by Weiner[4], McCreight[2] or Ukkonen[3]. It needs $\mathcal{O}(n \log n + n \log \sigma)$ bits. The first summand is the space needed for the pointers to the children and the indices stored in the leaves. The second summand is the space needed for the edge labels. To achieve this space, the edge

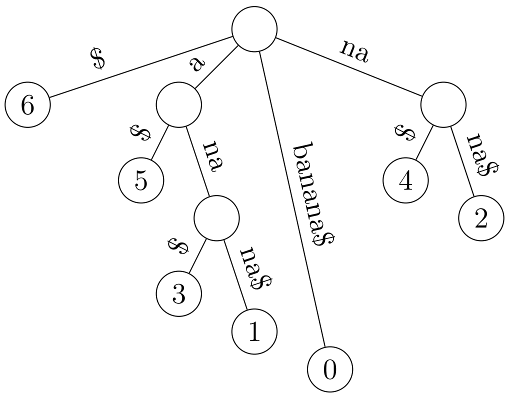

Figure 1.2: The suffix tree for the string "banana$".

labels must not be stored explicitly. Instead we can store pointers to the first and last position of the label in the text.

In practice, a suffix tree needs more than 20 times the space of the original text. Based on the required functionality, this can even be worse.

1.3 Bitvectors

Definition 1.7
A *bitvector* B is an array of bits that are compactly stored. A bitvector supports the following operations:

- access(i) returns the i-th element in B.
- rank(i) returns the number of set bits in the prefix $[0..i-1]$ of B.
- select(i) returns the position of the i-th set bit.

Theorem 1.8
The rank*-Operation on bitvectors can be done constant time and $o(n)$ additional space.[1]*

Proof Let B be a bitvector of length n. Precompute the following information:

1. Divide B into *superblocks* $SB_1, \dots, SB_{\lceil \frac{n}{L} \rceil}$ of size L.
2. For each superblock SB_j now store $\sum_{i=0}^{(j-1)L-1} B[i]$. This is the number of set bits in the first j superblocks. For each superblock this needs $\mathcal{O}(\log n)$ bits of space.
3. Now further divide each superblock B_j into blocks $B_1^j, \dots, B_{\lceil \frac{L}{S} \rceil}^j$ of size S.
4. For each block B_k^j of superblock SB_j store $\sum_{i=(j-1)L}^{(j-1)L+kS-1} B[i]$. This is the number of set bits in the first k blocks of superblock SB_j. For each block this needs $\mathcal{O}(\log L)$

bits.

For a given rank(i) query we now calculate the corresponding superblock SB_j and block B_k. We use the precomputed sums to get the number of set bits in all superblocks before SB_j and all blocks before B_k. All thats missing now is the number of set bits in block B_k until position i. If the blocks are small enough, this information can be precomputed efficiently for all possible blocks and positions (Four-Russians-Trick).

We still need to choose L and S:

$$L = \log^2 n \tag{1.1}$$

$$S = \frac{1}{2} \log n \tag{1.2}$$

The total amount of space needed is in $\mathcal{O}(\frac{n}{\log n} + \frac{n \log\log n}{\log n} + \sqrt{n} \log n \log\log n)$ bits. ■

1.4 Wavelet Trees

Definition 1.9
A *wavelet tree* is a compact datastructure that stores a sequence S generalizes the operations of a bitvector to an arbitrary alphabet.

- access(i) returns the i-th element of the sequence.
- rank$_q$(i) returns the number of occurrences of q in the prefix $S[0..i-1]$.
- select$_q$(i) returns the position of the i-th occurrence of q in S.

The root of the wavelet tree stores the whole sequence. Each vertex recursively divides its sequence to its two children. The left child contains the first half of the remaining alphabet, the right child contains the second half of the remaining alphabet. A bitvector in every vertex stores the corresponding child for each element.

Example 1.10
Figure 1.3 shows the wavelet tree for the string "abracadabra".

Lemma 1.11
A wavelet tree can be stored in $n\lceil \log \sigma \rceil$ bits space.

Proof The wavelet tree has height $\lceil \log \sigma \rceil$ and stores n bits on every layer (maybe even less on the last layer). Therefore $n\lceil \log \sigma \rceil$ bits are needed to store the bitvectors. A wavelet tree can be implemented fully via bitvectors and does not need any pointers. ■

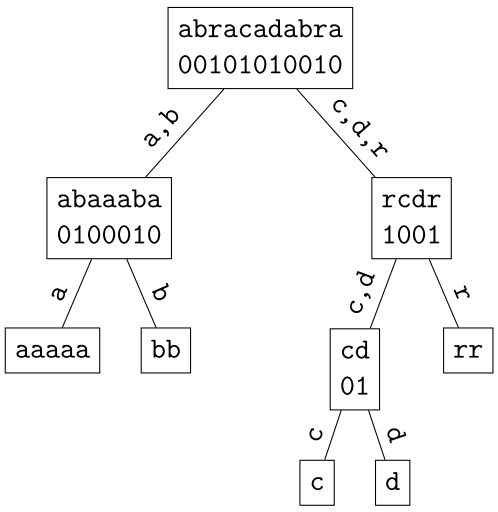

Figure 1.3: The wavelet tree for the string "abracadabra".

Index

Bibliography

[1] G. Jacobson. “Space-efficient static trees and graphs”. In: *30th Annual Symposium on Foundations of Computer Science*. Oct. 1989, pp. 549–554. DOI: `10.1109/SFCS.1989.63533`.

[2] Edward M. McCreight. “A Space-Economical Suffix Tree Construction Algorithm”. In: *J. ACM* 23.2 (Apr. 1976), pp. 262–272. ISSN: 0004-5411. DOI: `10.1145/321941.321946`. URL: `http://doi.acm.org/10.1145/321941.321946`.

[3] E. Ukkonen. “On-line construction of suffix trees”. In: *Algorithmica* 14.3 (Sept. 1995), pp. 249–260. ISSN: 1432-0541. DOI: `10.1007/BF01206331`. URL: `https://doi.org/10.1007/BF01206331`.

[4] Peter Weiner. “Linear Pattern Matching Algorithms”. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. SWAT ’73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11. DOI: `10.1109/SWAT.1973.13`. URL: `https://doi.org/10.1109/SWAT.1973.13`.