Chapter 3

Data Structure Compression


- The goal in data structure compression is to represent the structure in small space, but at the same time preserve its *functionality*.

**Example 3.0.1**

- There are \( \binom{2n}{n} / (n + 1) \) different binary trees of \( n \) nodes. Hence it is possible to construct a bijection \( f: \text{binary trees} \rightarrow \{0, 1, \ldots, \binom{2n}{n} / (n + 1) - 1\} \).

- For each binary tree \( T \) one can efficiently compute \( f(T) \), and vice versa [KM90].

- Coding \( f(T) \) as binary number is optimal, if all binary trees are as likely.

- However, the coding is not functionality preserving, because one cannot implement procedures such as “proceed to the left child of node \( v \)” without decompressing the whole encoding.
3.1 Functionality preserving compression

- In the sequel, we consider only functionality preserving compression.
- Our focus is on trees, tables, and on some special structures useful for Bioinformatics applications.

3.1.1 Concrete and abstract optimization

- **Concrete optimization** looks for the smallest representation among the given set of possible representations for a data structure.
- **Abstract optimization** looks for the smallest representation in any representation that implements the abstract definition of the data structure.

**Example 3.1.1**

- Minimization of an automata is concrete optimization; the data type for a node remains the same, but the overall amount of nodes is minimized.
- Representation of an automaton with fewest number of bits, so that one can still efficiently simulate the scanning of the automata, is abstract optimization.

**Example 3.1.2**

- Suffix trie can be compressed into a suffix automaton or DAWG. This is concrete optimization, although the data type of the node will be slightly altered.
- Suffix tree can be compressed into a compact DAWG, using concrete optimization.
- Suffix array can be compressed into compact suffix array [Mak03]:
  - Index range \([i, j]\) is replaced by a link to index range \([i', j']\), iff \(Pos[i] = Pos[i'] + 1, Pos[i + 1] = Pos[i' + 1] + 1, \ldots, Pos[j] = Pos[j'] + 1\).
  - This is also concrete optimization, as the original array can be seen as the least compressed array.
  - The functionality of the structure is preserved, as one can still use binary search on a properly implemented compact suffix array.
3.2 Abstract optimization of a tree

- Let us consider abstract optimization of a tree.

- **Motivation**: A tree of $n$ nodes represented using pointers requires $O(n \log n)$ bits. For many applications this is too much.

- We restrict to binary trees.

- We will next show how to represent a binary tree using $2n + o(n)$ bits so that the transitions from parent to children and back can be simulated in constant time

- The result is somewhat surprising, as the, only slightly smaller, optimal encoding looses all the functionality of the structure.

- Same technique can be used for encoding general trees [Jac89]; $2n + o(n)$ bits is enough in this case too.

3.2.1 From binary tree to bit-vector

- Binary tree can be represented as a bit-vector, as a so-called *Zak’s sequence*.
  
  - Let us mark all nodes of the tree with bit 1.
  
  - We add new leaves marked with 0 bits, so that all nodes marked with 1 bits (also leaves) have exactly two children.
  
  - We read the the bits marked in the nodes in levelwise order: This gives a sequence of $2n + 1$ bits, where $n$ is the number of nodes in the original tree.

- **Example.** a binary tree extended with new nodes and nodes marked with bits 0 and 1.
• Reading the above extended tree in levelwise order gives the sequence:

```
  1 1 1 1 0 1 1 1 0 0 0 1 0 0 0 0 0 0
  1 2 3 4 5 6 7 8 9
```

• Let $B$ be the bit-vector produced by the above mapping.

• We have the following equations to compute the children and parent of the $i$-th node:

\[
\begin{align*}
\text{leftchild}(i) &= 2 \cdot \text{rank}(B, i) \\
\text{rightchild}(i) &= 2 \cdot \text{rank}(B, i) + 1 \\
\text{parent}(i) &= \text{select}(B, \lfloor i/2 \rfloor),
\end{align*}
\]

where

- $\text{rank}(B, i)$ tells how many 1-bits there is up to position $i$ in $B$, and
- $\text{select}(B, i)$ tells which position contains the $i$-th 1-bit.

• (Notice the connection to heap sorting [Wil69], where perfect binary tree is represented as an array: The Zaks sequence $B$ of a perfect binary tree has $n$ 1-bits in the beginning, and for these holds $\text{rank}(B, i) = i$ and $\text{select}(B, i) = i$.)

• Next section describes how $\text{rank()}$ and $\text{select()}$ operations can be computed in constant time, after the bit-vector $B$ is preprocessed suitably. This preprocessing attaches an $o(n)$ size dictionary to $B$.

• Assuming that $\text{rank()}$ and $\text{select()}$ can be computed in constant time, we have the e.g. the following result:

- A binary tree $T$ can be represented in space $2n + o(n)$ bits so that the question “Is there a path in $T$ starting with binary string $X$?”, can be answered in time $O(|X|)$.

• The result can be extended to general trees. Then one also has to store separately the information associated with the edges. If, for example, each edge is associated with an alphabet symbol, one needs $O(n \log \sigma)$ bits in addition to the $2n + o(n)$ bits representing the hierarchy of the tree.
3.2.2 Bit-vector operations \textit{rank} and \textit{select}

\textit{rank}-operation in constant time:
- Storing all values $\text{rank}(B, i)$ would take $O(n \log n)$ bits, where $n = |B|$.

\textit{Partial solution 1:}
- Let us store each $\ell$-th $\text{rank}(B, i)$ as is and scan the rest of the bits (at most $\ell$), during the query. We then have an array $\text{first}$, where $\text{first}[i/\ell] = \text{rank}(B, i)$ when $i \mod \ell = 0$ ($/$ is here integer division).
- If we choose $\ell = (\lceil \log n \rceil)^2$, we need (about) $n \log n/(\log^2 n) = n/(\log n)$ bits space for the array $\text{first}$.
- We can answer $\text{rank}(B, i)$ in $O(\log^2 n)$ time: $\text{rank}(B, i) = \text{first}[i/\ell] + \text{rank}(B, \ell \ast (i/\ell) + 1, i)$, where $\text{rank}(B, i', i)$ computes the amount of 1-bits in the range $B[i' \ldots i]$.

\textit{Partial solution 2:}
- Let us store more answers. We store inside each area of length $\ell$ answers for each $k$-th position (how many 1-bits from the start of the are). We obtain an array $\text{second}$, where $\text{second}[i/k] = \text{rank}(B, \ell \ast (i/k) + 1, i)$, when $i \mod k = 0$.
- This uses overall space $n \log \ell/k$ bits. Choosing $k = [\log n]$ gives $O(n \log \log n(\log n))$ bits space usage.
- Now we can answer $\text{rank}(B, i)$ in $O(\log n)$ time, as $\text{rank}(B, i) = \text{first}[i/\ell] + \text{second}[i/k] + \text{rank}(B, k \ast (i/k) + 1, i)$.

\textit{Final solution:}
- We use so-called \textit{four Russians Trick} to improve the $O(\log n)$ query time into constant. This is based on an observation that there are only $\sqrt{n}$ bit-vectors of length $k/2 = [\log n]/2$.
- We store for each position $j$ in each of the $(\log n)/2$ size bit-vector $C$ a value $\text{rank}(C, j)$ as is. This takes overall $O(\sqrt{n} \log n \log n)$ bits.
- Let a table $\text{third}[0 \ldots \sqrt{n} - 1][0 \ldots [\log n]/2 - 1]$ store the above values, where the first index equals $C$ as an integer.
- Let $c_i$ and $d_i$ be the first and second half of the bit-vector $\text{rank}(B, k \ast (i/k), i)$ as $k/2$ bit integers (one has to zero the first
bit of $c_i$, because that bit is already taken into account in the table \textit{first}). We obtain the final formula to compute $\text{rank}(B, i)$

$$
\text{rank}(B, i) = \text{first}[i/\ell] + \text{second}[i/k] \\
+ \text{third}[c_i][\min(i \mod k, k/2 - 1)] \\
+ \text{third}[d_i][\max((i \mod k) - k/2, -1)], \quad (3.1)
$$

where $\text{third}[d_i][-1] = 0$.

- Integers $c_i$ and $d_i$ can be read in constant time from the bit-vector $B$, if the model of computation is chosen properly (RAM-model, where $w = \Omega(\log n)$). E.g. using C-language, $B$ can be represented as an array \texttt{unsigned B[n/32+1]}. Then each $c_i$ and $d_i$ can be read from the bit-vectors representing integers $B[i/32]$ and $B[i/32+1]$ (Exercise: write a C-program that reads $c_i$ and $d_i$ in constant time).

**Theorem 3.2.1** Bit-vector rank operation for a given bit-vector $B[1 \ldots n]$ can be supported in constant time on a RAM-model when the size of the computer word is $w = \Omega(\log n)$. In addition to the bit-vector $B$, one needs a dictionary of size $o(n)$ to support the operation.

\textit{select}-operation in constant time:

- Notice that \textit{select} can be implemented in $O(\log n)$ time by making a binary search on the \textit{rank}-dictionary.
- Constant time solution is possible using techniques like above [Mun96, Cla96].
  - The solution is somewhat more complicated.
  - Constant is \textit{asymptotic}: about $16(\log \log n)^4/(\log n)$. In practice always larger than 300.
  - Some small changes to the scheme gives a practical $O(\log \log n)$ time solution (without large constant factors).

**Theorem 3.2.2** Bit-vector select operation for a given bit-vector $B[1 \ldots n]$ can be supported in constant time on a RAM-model when the size of the computer word is $w = \Omega(\log n)$. In addition to the bit-vector $B$, one needs a dictionary of size $o(n)$ to support the operation.
3.3 Wavelet trees

- **Wavelet tree** generalizes **rank** and **select** queries to sequences from any alphabet size [GGV03].
- Let us denote by $\text{rank}_s(T, i)$ the count of symbols $s$ up to position $i$ in $T$, and by $\text{select}_s(T, j)$ the position of the $j$-th $s$ in $T$. Here $T \in \Sigma^*$.
- We will next show several ways to reduce the problem of **rank**/**select** computation on general sequences to computation on binary sequences.

3.3.1 Linear representation

- Let us represent a string $T[1\ldots n] \in \Sigma^*$ as $\sigma$ binary strings $B_s[1\ldots n]$ for all $s \in \Sigma$, such that $B_s[i] = 1$ if $T[i] = s$ otherwise $B_s[i] = 0$.
- Now, $\text{rank}_s(T, i) = \text{rank}(B_s, i)$ and $\text{select}_s(T, j) = \text{select}(B_s, j)$.
- After preprocessing binary strings $B_s$ for **rank**/**select** queries, we can answer $\text{rank}_s(T, i)$ and $\text{select}_s(T, j)$ in constant time using $\sigma n (1 + o(1))$ bits of space.

3.3.2 Balanced representation

- Consider a perfectly balanced binary tree where each node corresponds to a subset of the alphabet.
- The children of each node partition the node subset into two. A bitmap $B_v$ at the node $v$ indicates to which children does each sequence position belong. Each child then handles the subsequence of the parent’s sequence corresponding to its alphabet subset. The root of the tree handles the sequence $T[1\ldots n]$. The leaves of the tree handle single alphabet symbols and require no space.
- To answer query $\text{rank}_s(T, i)$, we first determine to which branch of the root does $s$ belong. If it belongs to the left, then we recursively continue at the left subtree with $i \leftarrow \text{rank}_0(B_{\text{root}}, i)$. Otherwise we recursively continue at the right subtree with $i \leftarrow \text{rank}_1(B_{\text{root}}, i)$. The value reached by $i$ when we arrive at the leaf that corresponds to $s$ is $\text{rank}_s(T, i)$. 
• The character $t_i$ at position $i$ is obtained similarly, this time going left or right depending on whether $B_v[i] = 0$ or 1 at each level, and finding out which leaf we arrived at.

• Query $select_s(T, j)$ is answered by traversing the tree bottom-up.

• The above hierarchical structure is called wavelet tree [GGV03].

• The size of the structure is $n \log \sigma (1 + o(1))$ bits.

• Wavelet tree supports rank/select queries in $O(\log \sigma)$ time.

### 3.3.3 Huffman shaped wavelet trees

• Instead of using perfectly balanced binary tree to form the hierarchy of the wavelet tree, one can use the Huffman tree hierarchy.

• It is then easy to see that the size of the structure reduces to $(H_0(T) + 1)(n + o(n))$.

• The Huffman tree is more skewed than the balanced tree, which affects the running time of rank/select queries.

• Huffman shaped wavelet tree supports rank/select queries in $O(\log n)$ time.

### 3.4 Compressed full-text indexes

• We have seen how to represent a tree in a succinct way.

• Generalizing the result to suffix trees is non-trivial, but possible.

• Succinct representation of suffix arrays is however much easier.

• We will next see how to exploit the Burrows-Wheeler transform and wavelet trees to develop a compressed suffix array.
3.4.1 Backward search on Burrows-Wheeler

- Let $L$ denote the Burrows-Wheeler transformed text, i.e. $L$ is the last column of matrix $M$. Let $F$ denote the sorted order of symbols of the text, i.e. $F$ is the first column of $M$.
- Let us formalize a fact we have learned before: We can get text $T$ given $L$. Note the following properties [BW94]:
  a. Given the $i$-th row of $M$, its last character $L[i]$ precedes its first character $F[i]$ in the original text $T$, that is, $T = \ldots L[i] F[i] \ldots$.
  b. Let $L[i] = s$ and let $r_i$ be the number of occurrences of $s$ in $L[1,i]$. Take the row $M[j]$ as the $r_i$-th row of $M$ starting with $s$. Then the character corresponding to $L[i]$ in the first column $F$ is located at $F[j]$ (this is called the LF mapping: $LF(i) = j$). This is because the occurrences of character $s$ are sorted both in $F$ and $L$ using the same criterion: by the text following the occurrences.
- The BWT can then be reversed as follows:
  1. Compute the array $C[1, \sigma]$ storing in $C[s]$ the number of occurrences of characters $\{s, 1, \ldots, s - 1\}$ in the text $T$. Notice that $C[s] + 1$ is the position of the first occurrence of $s$ in $F$ (if any).
  2. Define the LF mapping as follows: $LF(i) = C[L[i]] + \text{rank}_{L[i]}(L, i)$, where $\text{rank}_{s}(L, i)$ is the number of occurrences of character $s$ in the prefix $L[1,i]$.
  3. Reconstruct $T$ backwards as follows: let $p$ be the row of $M$ spelling $T$ ($p$ is stored separately), then for $n, \ldots, 1$ do $T[i] \leftarrow L[p]$ and $p \leftarrow LF[p]$.

Binary search on Suffix Array.

- Recall the suffix array $Pos[1\ldots n]$.
- Given $Pos[1\ldots n]$, the occurrences of the pattern $P = p_1 p_2 \ldots p_m$ can be counted in $O(m \log n)$ time: The occurrences form an interval $Pos[sp, ep]$ such that suffixes $t_{Pos[i]} t_{Pos[i]+1} \ldots t_n$, for all $sp \leq i \leq ep$, contain the pattern $P$ as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$. Once the interval is obtained, the starting positions of the $occ$ occurrences can be listed in $O(occ)$ time.
Backward search.

- We exploit the connection of suffix array and reverse Burrows-Wheeler transform to derive a pattern search algorithm that works in $O(m)$ steps.
- The algorithm for counting the pattern occurrences is shown below.

**Algorithm** Count($P[1 \ldots m], L[1 \ldots n]$)

1. $i \leftarrow m$
2. $sp \leftarrow 1; ep \leftarrow n$
3. while $(sp \leq ep)$ and $(i \geq 1)$ do
   4. $s \leftarrow P[i];$
   5. $sp \leftarrow C[s] + \text{rank}_s(L, sp - 1) + 1$
   6. $ep \leftarrow C[s] + \text{rank}_s(L, ep)$
   7. $i \leftarrow i - 1$
8. if $(ep < sp)$ then return “not found”
else return “found $(ep - sp + 1)$ occurrences”.

- The correctness of the above algorithm is easy to see by induction: At each phase $i [sp, ep]$ gives the maximal interval of suffix array $Pos$ pointing to suffixes prefixed by $P[i \ldots m]$.

**Time and Space Analysis.**

- The time requirement of the $\text{Count}()$ algorithm is clearly $O(m)$ if function $\text{rank}_s()$ can be computed in constant time.
  - Using the results from Sect., we can support $\text{rank}_s()$ in constant time using $\sigma n(1 + o(1))$ bits of space, OR
  - we can support $\text{rank}_s()$ in $O(\log \sigma)$ time using $n \log \sigma(1 + o(1))$ bits of space, OR
  - we can support $\text{rank}_s()$ in $O(\log n)$ time using $(H_0 + 1)(n + o(n))$ bits of space.
- Other space-time tradeoffs appear in the literature.
Remarks.

- Notice that so far we can only support counting queries. We are not able to locate the occurrence positions.
- Similar techniques can be used to support locating in $O(\text{occ} \log^{1+\epsilon} n)$ time, for any given $\epsilon > 0$, using $o(n)$ extra bits of space.

3.4.2 Compressed suffix arrays using compression boosting

- It is possible to combine compression boosting and the backward search algorithm to obtain $nH_k + o(n)$ space complexity with the same query times as above.
- (See [FMMN04] for details.)

3.4.3 Simplified compressed suffix arrays using sampling and integer codes

- Although the backward search algorithm on Burrows-Wheeler transformed text is conceptually nice and easy, we have used several complex techniques to obtain the result, e.g. rank queries, wavelet trees, etc.
- We will now learn that one can simulate the backward search without any use of complex data structures [Sad00,MNS04].

$\Psi$ function.

- We represent the suffix array $Pos[1...n]$ by a sequence of numbers $\Psi(i)$, such that $Pos[\Psi(i)] = Pos[i] + 1$. Furthermore, the sequence is differentially encoded, $\Psi(i) - \Psi(i-1)$.
- Note that $\Psi$ values are increasing in the areas of $Pos$ where the suffixes start with the same character $a$, because $ax < ay$ iff $x < y$.
- We use the notation $R(X)$, for a string $X$, to denote the range of suffix array positions corresponding to suffixes that start with $X$. 
• The search goal is therefore to determine \( R(P) \).

**Backward search using \( \Psi \) function.**

• Since \( \text{Pos}[\Psi(i)] = \text{Pos}[i] + 1 \), it turns out that
\[
x \in R(P[i \ldots m]) \iff x \in R(P_i) \land \Psi(x) \in R(P[i + 1 \ldots m])
\]

• The set of suffix array positions \( x \) such that \( \Psi(x) \) is inside some range forms a continuous range of positions and can be binary searched inside \( R(P_i) \), at \( O(\log n) \) cost.

• We obtain the following backward search routine for counting queries:

**Algorithm** \( \text{CountPsisi}(P, C, \Psi) \):

\[
\begin{align*}
\text{left}_{m+1} &:= 1; \text{right}_{m+1} := n; \\
\text{for } i &:= m \text{ downto } 1 \text{ do begin} \\
\text{left}_i &:= \min\{ j \in [C[p_i] + 1, C[p_i + 1]], \Psi(j) \in [\text{left}_{i+1}, \text{right}_{i+1}] \}; \\
\text{right}_i &:= \max\{ j \in [C[p_i] + 1, C[p_i + 1]], \Psi(j) \in [\text{left}_{i+1}, \text{right}_{i+1}] \}; \\
\text{if } \text{left}_i > \text{right}_i &\text{ return } \text{"no occurrences found";}
\end{align*}
\]

\[
\text{return } \text{"right}_1 - \text{left}_1 + 1 \text{ occurrences found"}
\]

• Here \( C[] \) stores the cumulative character counts as defined before, and \( \text{min} \) and \( \text{max} \) stand for the binary searches.

• The algorithm is illustrated in Fig. 3.1.

**Integer coding of \( \Psi \) values.**

• If we represent \( \Psi \) values as such, we need \( O(n \log n) \) bits of space.

• Instead, we can store the differences \( \Psi(i + 1) - \Psi(i) \) using e.g. \( \delta \)-encoding, to obtain space requirement \( nH_0(1 + o(1)) \) bits (showing this is left as an exercise).

• However, we should not use more than \( O(\log n) \) to access each \( \Psi \) value, to obtain the \( O(m \log n) \) search time.

• To obtain good space requirement and fast access to \( \Psi \) simultaneously we do as follows:
A C G T
A C G T
A C G T
A C G T
A C G T

Figure 3.1: Searching for pattern $P = CCAGTA$ backwards (right-to-left). The situation after reading each character is plotted. The gray-shaded regions indicate the interval of the suffix array that contain the current pattern suffix. The computation of the new interval is illustrated in the second step (starting from right). The $\Psi$ values from the block of letter $G$ point to consecutive positions in the suffix array. Hence it is easy to binary search the top-most and bottom-most pointers that are included in the previous interval.

- We store each $\lfloor \log n \rfloor /2$-th $\Psi$ value as such, and for each of them a pointer to the corresponding position in the $\delta$-encoded sequence of $\Psi(i+1) - \Psi(i)$ values.

- Now, the binary search for the $\Psi$ value splits into two: First, we binary search among the sampled $\Psi$ values to find an initial position, then we decompress at most $\log n$ differences from the corresponding position in the $\delta$-encoded sequence to find the final $\Psi$ value.

- This procedure is illustrated in Fig. 3.2.

- Hence we obtained $O(m \log n)$ search time using $nH_0(1 + o(1))$ bits of space using only sampling and compression.
Figure 3.2: Binary search followed by sequential search. The top-most sampled value closest to the previous interval is found using binary search (indicated by the top-most solid arrow). Then the next $\Psi$ values are decoded until the first value inside the interval (if exists) is encountered (indicated by the top-most dashed arrow). The same is repeated to find the bottom-most sampled value and then the bottom-most encoded value.