1 Succinct Data Structures (ctd.)

1.1 Select-Queries

A slightly different approach, compared to rank, is used for select. \( B \) represents the bit-vector with \( |B| = n \) and let \( k = \lfloor \log^2 n \rfloor \). A new table \( N \) is defined, which stores the \((k \cdot i)\)’th occurrence of a 1-bit in \( B \). Alternatively, \( N[i] = \text{select}_1(B, ik) \). As we can see, table \( N[1, \frac{n}{k}] \) divides \( B \) into blocks of different sizes, whereas each block contains \( k \) 1’s. An example is given in Figure 1, where an abstract bit-vector \( B \) is divided into blocks with \( k \) 1’s in each.

Example 1. Let \( N = [17, 28, 36, 53, \ldots] \) and \( k = 8 \). In this case, the 8\(^{th} \) 1 would be at index 17 in \( B \), the 16\(^{th} \) 1 at index 28 and so on.

The resulting blocks are grouped as follows:

**Definition 1.** A long block spans more than \( k^2 = \Theta(\log^4 n) \) positions in \( B \).

Its number is limited by \( \frac{n}{\log^2 n} \). Therefore, the answers for select-queries within all long blocks can be stored explicitly in a table: \( \text{LongBlock}[0, \frac{n}{\log^2 n}][1, k] \), where \( \text{LongBlock}[i][j] = \text{select}_1(B, ik + j) \). Moreover, the LongBlock table is indexed by potential block numbers, because we do not know how many long blocks there are before a given position. Therefore, we imagine that a long block begins at every \( k^2 \) position. Select-queries to long blocks can be responded completely based on this structure.

**Definition 2.** A short block spans \( \leq k^2 \) positions in \( B \).

It contains \( k \) 1-bits and it spans \( \leq k^2 \) positions in \( B \) at most. We divide their range of arguments into sub-ranges of: \( k' = \lfloor \log^2 k \rfloor = \Theta(\log^2 \log n) \). Then, a table \( N'[1, \frac{n}{k'}] \) is defined, whereas the answers to select-queries for multiples of \( k' \) (relative to the end of the previous block) are stored.
In table $N'$, a $\perp$ symbol indicates if we are in a long block. The formal definition of $N'$ is:

$$N'[i] = \text{select}_1(B, ik') - (N[i k'], \frac{ik'}{k})$$

for $i$th 1 and the subtrahend, representing the end of the block.

Table $N'$ divides the blocks into miniblocks, each containing $k'$ 1-bits. A miniblock is called long if it spans more than $s = \frac{\sqrt{k}}{2} = \frac{\log n}{2}$ positions in $B$, and short otherwise. Analogous to the long blocks, the answers to all select-queries are stored explicitly for all long miniblocks, relative to the beginning of the corresponding short block. The table $\text{LongMiniBlock}[0, \frac{n}{s}][1, k']$ is indexed by the potential long miniblock numbers, because the number of long miniblocks up to a given position is unknown.

Finally, a lookup table is stored for the short miniblocks because of its relative small size.

**Definition 3** (Lookup table for small miniblocks). The lookup table for short miniblocks is defined as follows: $\text{Inblock}[0, 2^s - 1][1, k']$, where $\text{Inblock}[\text{pattern}] = \text{select}_1(\text{pattern}, i)$ for all bit-patterns of length $s$ and $\forall 1 \leq i \leq k'$.

Based on this table, a select-query within a short miniblock $B[b, i]$ can be answered by looking at $\text{Inblock}[B[b, b + s - 1]][i]$. We should keep in mind that short miniblock could be shorter than $s$. In this case, a padding with arbitrary bits in the end to match exactly $s$ bits does not affect the select query answer.

The query procedure follows the description of the data structure. Note that we can determine if (mini-) blocks are long or short by inspecting adjacent elements of $N$ (or $N'$) and checking if they differ by more than $k^2$ (or $\frac{\sqrt{k}}{2}$).

In the following, it is verified that the required bit space of the defined structures for the select query are succinct.

**Table $N$** The $N$ table can have $\frac{n}{k}$ entries as maximum in the case that $B$ only contains 1's. Moreover, one stored index requires $\leq \log n$ bits. We get: $|N| = \frac{n}{k} \log n = O\left(\frac{n \log n}{\log n}\right) = o(n)$

**Table $\text{LongBlock}$** LongBlock consists of $\frac{n}{k^2}$ entries, because of one entry for each potential long block. Moreover, it has $k$ columns in order to store the positions of the $k$ 1's, which are inside the block. A table cell requires $\log n$ bits. To sum up, the bit space results in: $|\text{LongBlock}| = \frac{n}{k^2} \cdot k \cdot \log n = O\left(\frac{n \log n}{\log n}\right) = o(n)$

**Table $N'$** The analysis is similar to $N$ by just using the definition for $k'$: $|N'| = \frac{n}{k'} \cdot \log k^2 = \frac{4n \log \log n}{\log \log n} = O\left(\frac{n \log \log n}{\log \log n}\right) = o(n)$

**LongMiniBlock** The table consists of $\frac{n}{s}$ potential miniblock entries and for each, with indices for $k'$ 1's, relative to the ending of the previous block. Thus, we get: $|\text{LongMiniBlock}| = \frac{n}{s} \cdot k' \cdot \log k^2 = \frac{n}{\sqrt{k}} \cdot \log^2 k \cdot \log k^2 = O\left(\frac{n \log^3 \log n}{\log n}\right) = o(n)$

**Inblock** Inblock as a lookup table contains $2^s$ different patterns, including $k'$ indices for the position of the $i$th 1 ($0 < i < k'$): $|\text{Inblock}| = 2^s \cdot k' \cdot \log s = O\left(\sqrt{n} \cdot \log^3 \log n\right) = o(n)$

As can be seen from the analysis, all defined structures require a bit space in $o(n)$ and thus, they are succinct.
Example 2. An example for the select structures is given in Figure 2. In the upper part, the bit vector $B$ is shown, including the borders for the different block types and indices. The three red-dashed separators indicate the potential borders of long blocks. Moreover, there is presented a long miniblock in darker blue and three short miniblocks following. Below, one can see the parameters $k$, $k'$ etc. and the tables LongBlock, LongMiniBlock and Inblock, which are associated by means of corresponding colors with the blocks in the bit-vector.

![Figure 2: Example of select](image)

1.2 Findclose

Again, let $B$ be a balanced string of $2n$ parentheses.

**Problem 1.** We have to define a succinct data structure that requires $o(n)$ bits of additional space to answer findclose queries in $O(1)$ time for any balanced string of length $2N \leq 2n$.

**Example 3.** The following table presents a balanced parentheses string. An example query for findclose would be: findclose(1) = 6.

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Pattern</th>
<th>(Inblock)</th>
<th>(LongMiniBlock)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LongBlock</td>
<td>000000</td>
<td>$k = 8, k^2 = 16$</td>
<td>$k' = 4, \sqrt{k} = 6$</td>
</tr>
<tr>
<td>ShortBlock</td>
<td>011111</td>
<td>$N = 17, 28, 36, 53$</td>
<td>$N' = \perp$</td>
</tr>
<tr>
<td>ShortBlock</td>
<td>011110</td>
<td>$</td>
<td>7, 11</td>
</tr>
<tr>
<td>LongBlock</td>
<td>111111</td>
<td>$</td>
<td>N</td>
</tr>
</tbody>
</table>

If $N = O\left(\frac{n}{\log n}\right)$, we can precompute the answers for all $2N$ parentheses, using $O\left(\frac{n}{\log n}\right) = o(n)$ bits in total. Otherwise, we construct the structures as follows: we divide $B$ into equal-sized blocks of length $s$, where: $s = \left\lfloor\frac{\log n}{2}\right\rfloor$.

**Definition 4.** Let $b(p) = \left\lfloor\frac{p}{s}\right\rfloor$ denote the block in which parenthesis $p$ lies. Moreover, we let $\mu(p)$ denote the matching parenthesis of $p$: $\mu(p) = \begin{cases} \text{findclose}(p) & B[p] = '(' \\ \text{findopen}(p) & \text{else} \end{cases}$

\(^1\)A new variable $N$ is introduced, because the findclose operation is applied on the bit-vectors $B'$ and $B''$ as well (defined later) and their length is $\leq n$. 

3
Definition 5. Let’s call \( p \) far, if \( b(\mu(p)) \neq b(p) \) (the matching parenthesis for \( p \) is located in another block) or near, if \( b(\mu(p)) = b(p) \).

Near Parentheses:
A lookup table NearFindClose\([0, 2^s - 1] : [0, s - 1] \) is precomputed such that:

\[
\text{NearFindClose}[\text{pattern}][i] = \begin{cases} 
\text{findclose}(\text{pattern}, i) & \text{if } \text{pattern}[i] \text{ is near} \\
\bot & \text{if } \text{pattern}[i] \text{ is far}
\end{cases}
\]

\( \forall \text{patterns}, \forall i : 1 \leq i \leq s \) such that \( \text{pattern}[i] = ' \).'

Far Parentheses:

Definition 6. Consider \( p \) as an opening far parenthesis and let \( q \) be the immediate predecessor of \( p \) which is also an opening far parenthesis. An opening parenthesis \( p \) is called an opening pioneer if: \( b(\mu(p)) \neq b(\mu(q)) \). A closing pioneer is defined symmetrically. A pioneer is either opening or closing. Note that the match of a pioneer is not necessarily a pioneer itself. The root is always a pioneer.

Example 4. Figure 3 shows an example of a pioneer \( p \) with \( s = 5 \).

The number of pioneers is size of: \#pioneer = \( |B'| = O(\frac{N}{\log n}) \), because there can be at most one pair \((p, \mu(p))\) per pair of blocks such that \( p \) is in the one block and \( \mu(p) \) in the other one, and \( p \) or \( \mu(p) \) is a pioneer: Imagine a graph, where nodes are represented by blocks and edges represent a pioneer and its match. We can see that the resulting graph is planar, because matching parentheses cannot cross. Hence, its size is linear in the number of blocks, which is \( O(\frac{N}{\log n}) \).

We construct a data structure \( B' \), which represents a substring of \( B \), but only consisting of pioneers and their matches. To tell whether a parenthesis \( p \) is stored in \( B' \), the pioneers and their matches are marked in a bitmap piofam\([0, 2N - 1] \) and prepared for \( O(1) \) rank- and select-queries. To keep the space within \( o(n) \), we need the following theorem, which will be proved in a further section.

Theorem 1. Sparse Bitmap Theorem
A bitmap \( B \) of length \( N \) containing \( u \leq N \) 1’s can be represented in \( O(u \log \frac{N}{u}) + o(N) \) bits of space, such that subsequent rank- and select-queries on \( B \) can be answered in \( O(1) \) time.

The same structure is stored recursively for \( B' \) such that \( |B''| = O(\frac{N}{\log^2 n}) \) with its corresponding bitmap piofam’. In this stage, all answers can be precomputed.

Additionally, we need two more lookup tables for the final algorithm.
Definition 7. $\Delta_{\text{Excess}}[0, 2^s - 1]|0, s - 1|0, s - 1]$ represents a lookup table to find differences in excess level, defined as follows.

$$\Delta_{\text{Excess}}[\text{pattern}] [i][j] = \text{excess}(\text{pattern}, j) - \text{excess}(\text{pattern}, i)$$

Definition 8. Leftmost $\Delta[\text{pattern}][\Delta][i] = \min \{j \leq i : \text{excess}(j) - \text{excess}(i) = \Delta \}$ with $0 \leq i < s$

By now, all relevant data structures for the findclose operation are defined. A bit space analysis will verify that the structures are succinct.

**Vector $B'$** We have already shown that there exist $\#\text{pioneers} = |B'| = \mathcal{O}(\frac{N}{\log n})$, which can be stored in $o(n)$ bits.

**Bitmap piofam** Based on the sparse bitmap theorem and for $u = \mathcal{O}(\frac{N}{\log n})$, piofam requires a bit space of: $|\text{piofam}| = \mathcal{O}(\frac{N}{\log n} \cdot \log \log n) + o(N) = o(n)$.

**Vector $B''$** $B''$, which contains all pioneer families of $B'$, requires a bit space of: $|B''| = \mathcal{O}(\frac{N}{\log^2 n}) = o(n)$.

**Bitmap piofam'** For the corresponding pioneer bitmap for $B''$, we set $u = \mathcal{O}(\frac{N}{\log^2 n})$, which results in: $|\text{piofam'}| = \mathcal{O}(\frac{N}{\log^2 n} \cdot \log(\log n \cdot \frac{\log^2 n}{N})) + o(N) = \mathcal{O}(\frac{N}{\log^2 n} \cdot \log \log n) + o(N) = o(n)$.

**Table NearFindClose** The lookup table contains an entry for each of the $2^s$ patterns, one entry stores $\mathcal{O}(s)$ indices for near parentheses and each index requires $\log s$ bits. Therefore, a bit space of $\mathcal{O}(2^s \cdot s \cdot \log s) = o(n)$ is required.

**$\Delta_{\text{Excess}}$ and Leftmost $\Delta$** Similar to NearFindClose, both tables have entries for $2^s$ patterns, $\mathcal{O}(s^2)$ rows and a cell with $\log s$ bits. If the three tables are combined, they require: $\mathcal{O}(2^s \cdot s^2 \cdot \log s) = o(n)$. Because both tables and NearFindClose use the same patterns, it is even possible to precompute a combined lookup table.

As has been shown for all data structures, each requires a bit space in $o(n)$ and thus, they are still succinct. In the following, we continue with the definition of the algorithm for the operation findclose.$(p)$.

**Definition 9.** Operation findclose.$(p)$

1. Based on the lookup table, determine whether $p$ is far:

   (a) $p$ is near $\rightarrow$ The table NearFindClose gives the answer.

   (b) $p$ is far, then calculate the number of members in the pioneer family $B'$ up to $p$ by $q \leftarrow \text{rank}_1(\text{piofam}, p)$ and the position of this parenthesis in $B'$ by $p^* \leftarrow \text{select}_1(\text{piofam}, q)$, which is an opening parenthesis and the immediate previous pioneer. Using the recursive structure for $B'$, we find that $j \leftarrow \text{findclose}(B', q - 1)$ is the match of $q$ in $B'$, which can be mapped back to a position in $B$ by $\mu(p^*) = \text{select}_1(\text{piofam}, j + 1)$.

2. Since the first far parenthesis in each block is stored in $B'$, $b(p) = b(p^*)$. Via a table lookup, the excess level difference $\Delta$ between $p^*$ and $p$ is determined. Let $b = \left\lceil \frac{p}{s} \right\rceil$ and set $\Delta \leftarrow \Delta_{\text{Excess}}[B[bs, (b+1)s-1]][p-bs][p^*-bs]$.

$q - 1$ because rank starts at 1.
3. The change between $\mu(p)$ and $\mu(p^*)$ must be $\Delta$ and $\mu(p)$ is the leftmost position in $\mu(p^*)$’s block with this property (same excess difference). Thus, we can use the Leftmost $\Delta$ lookup table, where $b' = \lceil \frac{\mu(p^*)}{s} \rceil$ is $\mu(p^*)$’s block (hence, also $\mu(p)$’s block) by

$$\mu(p) = b' \cdot s + \text{Leftmost } \Delta \cdot [B[b' \cdot s, (b' + 1)s - 1]] \cdot [\Delta] [\mu(p^*) - b' \cdot s]$$

**Example 5.** Finally, an example of findclose is presented. The required data structures are visible in Figure 4. Separators are indicating block borders for $s = 5$. Our example query is: findclose(4). As we can see from a lookup in NearFindClose in Figure 4. Separators are indicating block borders for $s = 5$. Our example query is: findclose(4). Finally, an example of findclose is presented. The required data structures are visible in Figure 4. Separators are indicating block borders for $s = 5$. Our example query is: findclose(4). Based on $q$, the immediate previous pioneer $p^*$ is calculated. Moreover, the closing parenthesis for $p^*$ is at $j = 2$ and $\mu(p^*) = 6$ can be determined. The excess difference between $p^*$ and $p$ is: $\Delta = 1$ ($\Delta$Excess, last pattern, $i = 1$ for index 4 in pattern). At the end, the $\Delta$Leftmost table is accessed for $\Delta = 1$. In the pattern, $\mu(p^*)$ is at index 1 and the table returns a 0. We can finally calculate $\mu(p) = \lceil \frac{6}{5} \rceil \cdot 5 + 0 = 5$.

![Excess](image)

**Figure 4:** Example data structures of findclose

![Pattern](image)

**Figure 5:** Example lookup table of findclose
References
