Repetition- and linearity-aware rank/select dictionaries

Paolo Ferragina    Giovanni Manzini    Giorgio Vinciguerra

ISAAC 2021
Compressed rank/select dictionaries

• Given a set \( A \) of \( n \) elements over an integer universe \( 0, 1, \ldots, u \)
  1. Store them in compressed form
  2. Implement \( \text{rank}(x) \): number of elements in \( A \) which are \( \leq x \)
  3. Implement \( \text{select}(i) \): return the \( i \)th smallest element in \( A \)

• Well-studied building block of succinct data structures for texts, genomes, graphs, etc.

\[
\text{rank}(12) = 3
\]

\[
\text{select}(7) = 40
\]
Two sources of compressibility

Repetitiveness
We exploit them both

Approximate linearity
[Boffa et al., ALENEX ’21]

Many nonlinear points
⇓
Use piecewise linear $\varepsilon$-approx.

\[
\begin{array}{cccccccccc}
A & 2 & 3 & 5 & 6 & 13 & 14 & 16 & 17 & 20 & 24 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{Gap string} & 2 & 1 & 2 & 1 & 7 & 1 & 2 & 1 & 3 & 4 & \ldots \\
\end{array}
\]

Difference between adjacent values

Store just a “back reference”

Errors smaller than a given integer $\varepsilon$
Exploiting repetitiveness and approx. linearity

1. Build on two known repetition-aware methods
   • Lempel-Ziv parsing, LZ-End [Kreft and Navarro, TCS 2013]
   • Block tree [Belazzougui et al., JCSS 2021]

2. Augment them to use linear $\varepsilon$-approximations with corrections

3. Show how to support $\textit{rank}$ and $\textit{select}$ in space bounded by the high-order entropy or a repetitiveness measure of the gaps
The $LZ_\varepsilon$ parsing

- Already processed into phrases
- New phrase of the parsing

Gap string

- Find longest earlier occurrence ending at phrase boundary
- Compute longest linear $\varepsilon$-approximation

Diagram:
- $A[i + k]$ for $k = 0, 1, 2, 3, 4$
- Corrections
- $O(\log n)$ bias
The $LZ_\varepsilon$ parsing

For the new phrase we store

- Indexes $i$, $j$, and $r$
- Slope and intercept of the line
- Array of $j - i + 1$ corrections, $O(\log \varepsilon)$ bits each
Queries in the $LZ_\varepsilon$ parsing

$LZ^\rho_\varepsilon$: Introduce a trade-off parameter $\rho > 0$ to shorten the phrase head and make queries faster.
**LZ$^\rho_\varepsilon$ bounds**

- No worse than a traditional LZ-parsing
- No worse than LA-vector in space

Let $\sigma = \text{number of distinct values in the gap string}$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Select time</strong></td>
<td>$O(\log^{1+\rho} n)$</td>
</tr>
<tr>
<td><strong>Rank time</strong></td>
<td>$O(\log^{1+\rho} n + \log \varepsilon)$</td>
</tr>
<tr>
<td><strong>Space in bits</strong></td>
<td>$nH_k(\text{gap string}) + O(n/\log^{\rho} n) + o(n \log \sigma) + \text{space for tails}$</td>
</tr>
</tbody>
</table>

Exploit repetitions

Exploit approximate linearity
The block-$\varepsilon$ tree

• Start with a standard block tree construction on the gap string

<table>
<thead>
<tr>
<th>$A$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>10</th>
<th>13</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>21</th>
<th>23</th>
<th>27</th>
<th>30</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>38</th>
<th>39</th>
<th>41</th>
<th>43</th>
<th>45</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap string</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
The block-$\varepsilon$ tree

- Start with a standard block tree construction on the gap string
- Assign to each node the bit cost of encoding its subtree
- Prune subtrees that are better compressed by linear $\varepsilon$-approximations

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>Gaps string</th>
<th>Prune via segment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 4 6 10 13</td>
<td>1 1 2 2 4 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15 16 17 18 19 21</td>
<td>2 1 1 1 1 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23 27 30 32 33 34</td>
<td>2 4 3 2 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>38 39 41 43 45 49</td>
<td>4 1 2 2 2 4</td>
<td></td>
</tr>
</tbody>
</table>

Prune via segment
The block-$\epsilon$ tree

- Start with a standard block tree construction on the gap string
- Assign to each node the bit cost of encoding its subtree
- Prune subtrees that are better compressed by linear $\epsilon$-approximations
- Store topology, leaf linear $\epsilon$-approx., and left pointers of copied blocks
Block-\(\varepsilon\) tree bounds

- Based on the \(\delta\) repetitiveness measure on strings:\(^1,2,3\)
  \[
  \delta = \max\{d_k/k : k = 1,\ldots,n\}
  \]
  where \(d_k\) = number of distinct substrings of length \(k\) in the gap string

- Number of levels is \(h = \mathcal{O}\left(\log \frac{n}{\delta}\right)\)

<table>
<thead>
<tr>
<th></th>
<th>(\mathcal{O}(h))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select time</td>
<td></td>
</tr>
<tr>
<td>Rank time</td>
<td>(\mathcal{O}\left(\log \log \frac{u}{\delta} + h + \log \varepsilon\right))</td>
</tr>
<tr>
<td>Space in bits</td>
<td>(\mathcal{O}\left(\delta \log \frac{u}{\delta} \log u\right))</td>
</tr>
</tbody>
</table>

---

1. Raskhodnikova et al., Algorithmica (2013)
2. Christiansen et al., TALG (2020)
3. Kociumaka et al., LATIN '20
Experiments with the block-\(\varepsilon\) tree

• Compared with LA-vector, and a standard block tree built on the characteristic bitvector \(B[0,u]\) of the input \(A[1,n]\)

• Datasets: postings lists, positions of symbols in texts (DNA). Various \(n/u\)

• LA-vector is \(10.51\times\) faster in select and \(4.69\times\) faster in rank than block tree, but no clear winner in space

• Block-\(\varepsilon\) tree:
  o wrt LA-vector, it is always slower in select and in rank
  o wrt block tree, it is \(2.19\times\) faster in select, either faster (\(1.32\times\)) or slower (\(1.27\times\)) in rank
  o has the best space in 2/12 datasets, and the second-best space in 7/12 datasets

{combination of repetitiveness and approximate-linearity makes sense}{Block-\(\varepsilon\) tree achieves a good compromise by exploiting both regularities}

Code available at github.com/gvinciguerra/BlockEpsilonTree
Conclusions

• Exploit both repetitiveness and approx. linearity in rank/select dictionaries

• $\text{LZ}_\varepsilon^\rho$ parsing
  ▪ Combine backward copies and linear $\varepsilon$-approximations
  ▪ Space complexity bounded by the $k$th order entropy

• Block-$\varepsilon$ tree
  ▪ Optimise block tree by compressing areas with high approximate linearity
  ▪ Space-time bounds based on the $\delta$ repetitiveness measure
  ▪ Experimentally achieves a good compromise between block trees and LA-vectors

• Future work
  ▪ Implement $\text{LZ}_\varepsilon^\rho$
  ▪ Relation of approximate linearity with other compressibility measures