Repetition- and linearity-aware rank/select dictionaries

Paolo Ferragina    Giovanni Manzini    Giorgio Vinciguerra

(Università di Pisa)

(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

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

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

Repetitiveness

Approximate linearity

\[ A = 2 \ 3 \ 5 \ 6 \ 13 \ 14 \ 16 \ 17 \ 20 \ 24 \ \ldots \]

Difference between adjacent values

Gap string

\[ A = 2 \ 1 \ 2 \ 1 \ 7 \ 1 \ 2 \ 1 \ 3 \ 4 \ \ldots \]

Store just a "back reference"

We exploit them both

Many nonlinear points

Use piecewise linear \( \epsilon \)-approx.

Errors smaller than a given integer \( \epsilon \)

\[ O(\log \epsilon) \ \text{bits} \]

[Boffa et al., ALENEX '21]
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 $\text{LZ}_\varepsilon$ 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 showing the process of LZ$\varepsilon$ parsing with a gap string and arrows indicating the process steps.](image)
The $\text{LZ}_\varepsilon$ parsing

Already processed into phrases

New phrase of the parsing

Gap string

$r$th phrase

Find longest earlier occurrence ending at phrase boundary

Compute longest linear $\varepsilon$-approximation

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 $\text{LZ}_\varepsilon$ parsing

$\text{LZ}_\rho^\varepsilon$: Introduce a trade-off parameter $\rho > 0$ to shorten the phrase head and make queries faster.

If $t \in \text{Phrase head}$, then we must recursively unroll the source phrase.

If $t \in \text{Phrase tail}$, then we can answer directly with the linear $\varepsilon$-approx

$\text{select}(t)$

$\text{rank}(A[t])$

**Diagram:**
- Gap string: 1 3 4 4 2 1 2 2 3 4 4 2 1 2 2 5 1 3 3 4 ...
- Phrase head: 3 4 4 2 1 2 2
- Phrase tail: 5 1 3 3 4
- New phrase of the parsing
**LZ**$_\varepsilon$ **bounds**

No worse than a traditional LZ-parsing

No worse than LA-vector in space

Let $\sigma$ = 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>$\mathcal{O}(\log^{1+\rho} n)$</td>
</tr>
<tr>
<td><strong>Rank time</strong></td>
<td>$\mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$</td>
</tr>
<tr>
<td><strong>Space in bits</strong></td>
<td>$nH_k$ (gap string) + $\mathcal{O}(n / \log^\rho n)$ + $o(n \log \sigma)$ + 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></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>(A)</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>
<td></td>
</tr>
<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
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
- Store topology, leaf linear $\varepsilon$-approx., and left pointers of copied blocks
Block-$\varepsilon$ tree bounds

• Based on the $\delta$ repetitiveness measure on strings:\textsuperscript{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 $h = \mathcal{O} \left( \log \frac{n}{\delta} \right)$

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

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

• Compared with LA-vector, and a standard block tree

• Datasets: postings lists, positions of symbols in texts (DNA, URLs)

• LA-vector is $10.5\times$ faster in select and $4.7\times$ faster in rank than block tree, but no clear winner in space

• Our block-$\varepsilon$ tree:
  o wrt LA-vector, it is always slower in select and in rank
  o wrt block tree, it is $2.2\times$ faster in select, either faster ($1.3\times$) or slower ($1.3\times$) 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

Code available at github.com/gvinciguerra/BlockEpsilonTree
Conclusions

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

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

• Block-$\epsilon$ 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 $LZ^\rho$
  ▪ Relation of approximate linearity with other compressibility measures