Advanced Data Structures

Lecture 12: Sparse Sets and Variable Bit-Length Arrays

Florian Kurpicz
Recap: Learned Data Structures

- use piece-wise linear approximation
- store corrections
- compress everything

Open Questions

- are $y$-intersections monotonic increasing
- are $\log u + \log n$ bits enough to store slope
Recap: The PGM-Index [FV20]

- now $S$ has to be stored
- how do we access elements in $S$
  - e.g., predecessor
  - trick used before requires too much space
- store key instead position
- recurs on first keys of each segment

For Queries

- $\epsilon = \Theta(B)$
- load $2\epsilon + 1$ blocks per level

$S = \langle 10, 18, 22, 24, 27, 31, 43 \rangle$
Variable Bit-Length Arrays

- not all elements require the same space
- arrays with \( w \) bits per element can waste space
- e.g., integers can be encoded with space proportional to their size

**Definition: Variable Bit-Length Data**

Let \( a[1..n] \) be an array containing entries of size \( |a[i]| \) bits for \( i \in [1, n] \).
Variable Bit-Length Arrays

- not all elements require the same space
- arrays with \( w \) bits per element can waste space
- e.g., integers can be encoded with space proportional to their size

Definition: Variable Bit-Length Data

Let \( a[1..n] \) be an array containing entries of size \( |a[i]| \) bits for \( i \in [1, n] \).
Sampling (1/2)

- encode \( a \) using close to \( N = \sum_{i=1}^{n} |a[i]| \) bits
Sampling (1/2)

- encode $a$ using close to $N = \sum_{i=1}^{n} |a[i]|$ bits

**Definition: Sampling**
Sample the starting position of every $k$-th element in array $s$.

$$a = \begin{bmatrix} 011 & 11 & 11 & 10 & 010 & 010 & 011 & 0001 & 0011 \end{bmatrix}$$

$$s = \begin{bmatrix} 1 & 6 & 10 & 16 & 23 \end{bmatrix}$$
Sampling (1/2)

- encode $a$ using close to $N = \sum_{i=1}^{n} |a[i]|$ bits

**Definition: Sampling**
Sample the starting position of every $k$-th element in array $s$.

**Lemma: VLA with Sampling**
Using sampling, storing $a$ requires $N + O(n \log N/k)$ bits of space. Accessing a single element requires $O(k)$ time.
Sampling (1/2)

- encode $a$ using close to $N = \sum_{i=1}^{n} |a[i]|$ bits

**Definition: Sampling**

Sample the starting position of every $k$-th element in array $s$.

**Lemma: VLA with Sampling**

Using sampling, storing $a$ requires $N + O(n \log N/k)$ bits of space. Accessing a single element requires $O(k)$ time.

- space can be reduced using Elias-Fano coding
- access time depends on input size unless $k = O(1)$
Definition: Two-Level Sampling

In addition to sampling every $k$-th element, also sample the offset to the closest preceding sampled element for each non-sampled element.

Lemma: VLA with Two-Level Sampling

Using two-level sampling, storing $a$ requires $N + O(n \log N / k)$ bits of space for the first level and additional $n \cdot \max \left\{ i \in \{0, k, 2k, \ldots\} \left\lceil \log k - 1 \sum_{j=1}^{X} |a[i+j]| \right\rceil \right\}$ bits of space for the second level.

Example on the board/chalkboard-teacher
Definition: Two-Level Sampling

In addition to sampling every $k$-th element, also sample the offset to the closest preceding sampled element for each non-sampled element.

Lemma: VLA with Two-Level Sampling

Using two-level sampling, storing $a$ requires $N + O(n \log N/k)$ bits of space for the first level and additional

$$n \cdot \max_{i \in \{0,k,2k,...\}} \left\lceil \log \sum_{j=1}^{k-1} |a[i + k]| \right\rceil$$

bits of space for the second level.
Definition: Two-Level Sampling
In addition to sampling every \( k \)-th element, also sample the offset to the closest preceding sampled element for each non-sampled element.

Lemma: VLA with Two-Level Sampling
Using two-level sampling, storing \( a \) requires \( N + O(n \log N/k) \) bits of space for the first level and additional

\[
n \cdot \max_{i \in \{0, k, 2k, \ldots \}} \left\lceil \log \sum_{j=1}^{k-1} |a[i + j]| \right\rceil
\]

bits of space for the second level.

For elements of polylogarithmic size, this means \( O(n \log \log n) \) additional bits of space.

- constant access time
- example on the board

\[
a = \begin{bmatrix} 011 & 11 & 11 & 10 & 010 & 010 & 011 & 0001 & 0011 \end{bmatrix}
\]

\[
s = \begin{bmatrix} 1 & 6 & 10 & 16 & 23 \end{bmatrix}
\]

\[
s' = \begin{bmatrix} 3 & 2 & 3 & 3 \end{bmatrix}
\]
Directly Addressable Codes (1/2) [BLN09]

- now, encode problem instead of indexing
- partition variable bit-length elements
- mark if not last partition
- similar to VByte encoding [WZ99]

Definition: Directly Addressable Codes

Each element is partitioned into length-$\ell$ slices. Every element's $k$-th (fixed-length) slice is stored in $a_k$. Use bit vector $v_k$ to mark elements that continue in $a_{k+1}$.

$$a = \begin{bmatrix} 011 & 11 & 11 & 10 & 010 & 010 & 011 & 0001 & 0011 \end{bmatrix}$$

$$a_1 = \begin{bmatrix} 00 & 11 & 11 & 10 & 00 & 00 & 00 & 00 \end{bmatrix}$$

$$bv_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$a_2 = \begin{bmatrix} 11 & 10 & 10 & 11 & 01 & 11 \end{bmatrix}$$

$$bv_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$
Lemma: VLA with Directly Addressable Codes

Using Directly Addressable codes, storing a requires at most $\ell n + N/\ell$ bits of space.

Proof (Sketch)
- at most $\ell - 1$ bits wasted in first slice
- one bit needed to mark each slice
- can be made more space-efficient
- choose different partition size for each level

\[
\begin{align*}
ad_0 &= \begin{array}{cccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \\
bv_0 &= \begin{array}{ccccccc}
1 & 0 & 0 & 0 & 1 & 1 & 1
\end{array} \\
ad_1 &= \begin{array}{cccccccc}
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \\
bv_1 &= \begin{array}{ccccccc}
1 & 0 & 0 & 0 & 1 & 1 & 1
\end{array} \\
ad_2 &= \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \\
bv_2 &= \begin{array}{ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]
An Efficient Representation of a Sparse Set [BT93]

- represent a sparse (dynamic) set $S \subseteq [1, u]
- using bit vector
  - $u$ bits of space
  - iterating, clearing, comparing requires $|S|$ select queries
  - inserting requires rebuilding select support
  - without select support $O(u)$ time operations
- use custom representation
An Efficient Representation of a Sparse Set [BT93]

- represent a sparse (dynamic) set $S \subseteq [1, u]$
- using bit vector
  - $u$ bits of space
  - iterating, clearing, comparing requires $|S|$ select queries
  - inserting requires rebuilding select support
  - without select support $O(u)$ time operations
- use custom representation

Definition: Sparse Set Representation

The sparse set consists of a dense set $d$ and a sparse set $s$. Let $S$ contain $n$ elements. To insert $i \notin S$ in $S$, set $d[n] = i$ and $s[i] = n$. 
An Efficient Representation of a Sparse Set [BT93]

- represent a sparse (dynamic) set $S \subseteq [1, u]$
- using bit vector
  - $u$ bits of space
  - iterating, clearing, comparing requires $|S|$ select queries
  - inserting requires rebuilding select support
  - without select support $O(u)$ time operations
- use custom representation

**Definition: Sparse Set Representation**

The sparse set consists of a dense set $d$ and a sparse set $s$. Let $S$ contain $n$ elements. To insert $i \notin S$ in $S$, set $d[n] = i$ and $s[i] = n$. 

![Sparse Set Example](image-url)
Operations on the Sparse Set

**insert(i)**

- \( d[n] = i \)
- \( s[i] = n \)
- \( n++ \)
Operations on the Sparse Set

<table>
<thead>
<tr>
<th>insert$(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d[n] = i$</td>
</tr>
<tr>
<td>$s[i] = n$</td>
</tr>
<tr>
<td>$n++$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>is_in_set$(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>return $s[i] &lt; n$ and $d[s[i]] == i$</td>
</tr>
</tbody>
</table>
Operations on the Sparse Set

**insert(i)**
- \(d[n] = i\)
- \(s[i] = n\)
- \(n++\)

**is_in_set(i)**
- return \(s[i] < n\) and \(d[s[i]] == i\)

**iterate**
- for \(i\) in 1..\(n\)
  - yield \(d[i]\)
Operations on the Sparse Set

**insert(i)**
- \( d[n] = i \)
- \( s[i] = n \)
- \( n++ \)

**clear**
- \( n = 0 \)

**is_in_set(i)**
- return \( s[i] < n \) and \( d[s[i]] == i \)

**iterate**
- for \( i \) in 1..n
  - yield \( d[i] \)
Operations on the Sparse Set

**insert(i)**
- \( d[n] = i \)
- \( s[i] = n \)
- \( n++ \)

**is_in_set(i)**
- return \( s[i] < n \) and \( d[s[i]] == i \)

**iterate**
- for \( i \) in 1..n
  - yield \( d[i] \)

**clear**
- \( n = 0 \)

**remove_from_set(i)**
- if not is_in_set(i)
  - return
- \( tmp = d[n - 1] \)
- \( d[s[i]] = tmp \)
- \( s[tmp] = s[i] \)
- \( n-- \)
Recap: Advanced Data Structures

- bit vectors with rank and select support
- succinct trees \( \text{ LOUDS, BP, DUFUDS } \)
- succinct planar graphs
- predecessor data structures \( \text{ Elias-Fano, y-fast trie } \)
- range minimum queries \( \text{ three solutions } \)
- persistent data structures \( \text{ partial and full persistence } \)
- orthogonal range search \( \text{ kd-trees, range trees, layered range trees } \)
- binary space partition \( \text{ BSP-tree } \)
- PaCHash
- compressed suffix array \( \text{ Elias-Fano with quotenting and recursive } \)
- String B-trees
- retroactive data structures \( \text{ decomposable search problems, partial retroactive PQs } \)
- minimal perfect hashing \( \text{ BDZ, CHD, RecSplit } \)
- learned data structures \( \text{ encoding and indexing } \)
- sparse sets and variable bit-length arrays

11/12
2024-07-22
Florian Kurpicz | Advanced Data Structures | 12 Sparse Sets and Variable Bit-Length Arrays
Institute of Theoretical Informatics, Algorithm Engineering
everybody can choose first topic
everybody can choose first topic

Now, some examples
Bibliography I


