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 \( i \in [1, n] \).
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

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

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 📚

![Diagram](image_url)
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 elements \(k\)-th (fixed-length) slice is stored in \(a_k\). Use bit vector \(v_k\) to mark elements that continue in \(a_{k+1}\).
Lemma: VLA with Directly Addressable Codes

Using Directly Addressable codes, storing an element \( a \) requires at most \( \ell n + \frac{N}{\ell} \) 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

\[
\begin{align*}
    a &= \begin{array}{cccccccccccc}
        0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1
    \end{array} \\
    a_1 &= \begin{array}{cccccccccccc}
        0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
        1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
        1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
        0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
    \end{array}
\end{align*}
\]
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 Representation Diagram](image-url)
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: LOUDS, BP, DUFUDS
- Succinct planar graphs
- Predecessor data structures: Elias-Fano, y-fast trie
- Range minimum queries: three solutions
- Persistent data structures: partial and full persistence
- Orthogonal range search: kd-trees, range trees, layered range trees
- Binary space partition: BSP-tree
- PaCHash
- Compressed suffix array: Elias-Fano with quotenting and recursive
- String B-trees
- Retroactive data structures: decomposable search problems, partial retroactive PQs
- Minimal perfect hashing: BDZ, CHD, RecSplit
- Learned data structures: encoding and indexing
- Sparse sets and variable bit-length arrays
everybody can choose first topic

Now, some examples
Bibliography I


