Succinct Data Structures for Data Mining

Rajeev Raman

University of Leicester

ALSIP 2014, Tainan
Overview

Introduction

Compressed Data Structuring

Data Structures

Applications

Libraries

End
Big Data vs. big data

- Big Data: 10s of TB+.
  - Must be processed in streaming / parallel manner.
- Data mining is often done on big data: 10s-100s of GBs.
  - Graphs with 100s of millions of nodes, protein databases 100s of millions of compounds, 100s of genomes etc.
- Often, we use Big Data techniques to mine big data.
  - Parallelization is hard to do well [Canny, Zhao, KDD'13].
  - Streaming is inherently limiting.
- Why not use (tried and trusted, optimized) “vanilla” algorithms for classical DM problems?
Mining big data

- Essential that data fits in main memory.
  - Complex memory access patterns: out-of-core ⇒ thrashing.
- Data stored accessed in a complex way is usually stored in a data structure that supports these access patterns.
  - Often data structure is MUCH LARGER than data!
  - Cannot mine big data if this is the case.
- Examples:
  - Suffix Tree (text pattern search).
  - Range Tree (geometric search).
  - FP-Tree (frequent pattern matching).
  - Multi-bit Tree (similarity search).
  - DOM Tree (XML processing).
Compressed Data Structures

Store data in memory in compact or compressed format and operate directly on it.

- (Usually) no need to decompress before operating.
- Better use of memory levels close to processor, processor-memory bandwidth.
  - Usually compensates for some overhead in CPU operations.

Programs = Algorithms + Data Structures

- If compressed data structure implements same/similar API to uncompressed data structure, can reuse existing code.
Compression vs. Data Structuring

Answering queries requires an index on the data that may be large:

- **Suffix tree**: data structure for indexing a text of $n$ bytes.
  - Supports many indexing and search operations.
  - Careful implementation: $10n$ bytes of index data [Kurtz, SPrEx '99]

- **Range Trees**: data structures for answering 2-D orthogonal range queries on $n$ points.
  - Good worst-case performance but $\Theta(n \log n)$ space.

**Succinct/Compressed Data Structures**

Space usage = “space for data” + “space for index”.

Redundancy should be smaller than space for data.
Memory Usage Models

**Indexing model**

- Preprocess input to get *index*.
- Queries read index and *access* input (may be expensive).
- Redundancy = index size.

**General model**

- No restriction on access to data structure memory.
- Redundancy = memory usage − “space for data”.

▷ Trade-off: redundancy vs. query time.
▷ Index or DS depends *crucially* on operations to be supported.
Space Measures

### Data Size
- Naive
- Information-theoretic Entropy ($H_0$)
- Entropy ($H_k$)

### Redundancy
- $O(1)$ (implicit/in-place)
- lower-order (succinct)
- “lower-order” (density-sensitive)
- “lower-order” (compressed)

- Best known example of implicit data structure: array-based representation of binary heap.
- “Information-theoretic” count total number of instances of a given size; take the log base 2 ($\lg$).
- “Entropy” is usually an *empirical* version of classical Shannon entropy.
Example: Sequences

$x \in \Sigma^n$ for some finite set of symbols $0, 1, \ldots, \sigma - 1$.

- **Naive:** $n \lceil \lg \sigma \rceil$ bits [$\sigma = 3, 2n$ bits].

- **Information-Theoretic:**
  \[ \lg \sigma^n = \lceil n \lg \sigma \rceil \text{ bits.} \] [$\sigma = 3, \sim 1.73n$ bits.]
  [Dodis, Patrascu, Thorup, *STOC’10*]

- **Entropy (0-th):** $H_0 \sim n_0 \lg(n/n_0) + n_1 \lg(n/n_1)$ ($\Sigma = \{0, 1\}$).
  - If $n_0 = n_1 = n/2$, $H_0 = n$. If $n_0 = 0.2n$, $H_0 \sim 0.72n$ bits.
  - Closely related to $\lg \binom{n}{n_1} = n_1 \lg \left( \frac{n}{n_1} \right) + O(n_1)$.
  - If $n_1 \ll n$, $H_0 \ll n$.

- **Entropy (k-th):** Count frequencies in all contexts of length $k$:
  \[ \ldots 1011 \ldots 1010 \ldots 1011 \ldots \]
  - For $0^{n/2}1^{n/2}$, $H_0 = n$ bits, $H_1 = O(\lg n)$ bits.
Example: Binary Trees

Given object $x$ is an binary tree with $n$ nodes.

- **Naive:** $\geq 2n$ pointers; $\Omega(n \lg n)$ bits.
- **Information-Theoretic:** $\lg \left( \frac{1}{n+1} \binom{2n}{n} \right) = 2n - O(\lg n)$ bits.
- **Entropy (0-th, k-th):**
  - Maybe define $H_0$ as $\sum_{i\in\{0,L,R,LR\}} n_i \lg(n/n_i)$, where $n_0 = \#$ leaves, $n_{LR} = \#$ nodes with both L and R child etc. [Davoodi et al., *PTRS-A '14*]
Bit Vectors

**Data:** Sequence $X$ of $n$ bits, $x_1, \ldots, x_n$. $m = n_1$. **Operations:**

- $\text{rank}_1(i)$: number of 1s in $x_1, \ldots, x_i$.
- $\text{select}_1(i)$: position of $i$th 1.

Also $\text{rank}_0$, $\text{select}_0$.

**Example:** $X = 01101001$, $\text{rank}_1(4) = 2$, $\text{select}_0(4) = 7$.

Want $\text{rank}$ and $\text{select}$ to take $O(1)$ time.

Operations introduced in [Elias, J. ACM ’75], [Tarjan and Yao, C. ACM ’78], [Chazelle, SIAM J. Comput ’85], [Jacobson, FOCS ’89].
Bit Vectors: Results

All operations \( \text{rank}_1, \text{select}_0, \text{select}_1 \) in \( O(1) \) time and:

- space \( n + o(n) \) bits [Clark and Munro, *SODA ’96*].
- space \( H_0(X) + o(n) \) bits [RRR, *SODA ’02*].
  - \( H_0(X) + o(n) = \lg \left( \binom{n}{m} \right) + o(n) = m \lg(n/m) + O(m) + o(n) \).
  - \( n/m \) is roughly the average gap between 1s.
  - \( m \ll n \Rightarrow H_0(X) \ll n \) bits.
  - \( o(n) \) term is not necessarily a “lower-order” term.

Only \( \text{select}_1 \) in \( O(1) \) time and:

- space \( H_0(X) + O(m) \) bits [Elias, *J. ACM ’75*].

Called “Elias-Fano“ representation.
Bit Vector Uses (1): Array of Strings

**Data:** $m$ variable-length strings, total size $n$. Access $i$-th string.

- Naive: $8n$ bits (raw) + $64m$ bits
- Bitvector: $8n$ bits (raw) + $n$ bits.
- Elias-Fano: $8n$ bits (raw) + $m \log(n/m) + O(m)$ bits.

XML textual data [Delpratt et al., *EDBT’08*], $n/m \sim 11$.

Use to random-access any variable-length data encoding.
Bit Vector Uses (2): Binary Tree

**Data:** $n$-node binary tree. [Jacobson, *FOCS ’89*]

- Add $n + 1$ external nodes.
- Visit each node in level-order and write bitmap ($2n + 1$ bits).

```
Binary tree:
```

```
Level-order bitmap (● = 1, □ = 0):
1 1 1 1 1 0 1 0 0 1 0 1 0 0 0 0 0
```

- Number internal nodes by position of 1 in bit-string.
- Store bitmap as bit vector. Left child = $2 \times \text{rank}_1(i)$, Right child = $2 \times \text{rank}_1(i) + 1$, parent = \text{select}_1(\lceil i/2 \rceil)$.
Bit Vector Uses (3): Elias-Fano representation

**Aim:** Using represent bit-string $X$ of lengusing $H_0(X) + O(m)$ bits support *only select*1 in $O(1)$ time.

[Elías, *J. ACM’75*], [Grossi/Vitter, *SICOMP’06*], [Raman et al., *TALG’07*].

- Space is always $O(nH_0)$.
- Consider bit-vector $X$ as *subset* of $\{0, \ldots, n-1\}$, i.e. as characteristic vector of set $X = \{x_1, \ldots, x_m\}$.
- $select1(i) = x_i$. 
Elias-Fano: MSB Bucketing

Bucket according to most significant $b$ bits: $x \rightarrow \lfloor x/2^{\lceil \lg n \rceil - b} \rfloor$.

Example. $b = 3$, $\lceil \lg n \rceil = 5$, $m = 7$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$x_3$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$x_4$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x_5$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$x_6$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$x_7$</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$-$</td>
</tr>
<tr>
<td>001</td>
<td>$-$</td>
</tr>
<tr>
<td>010</td>
<td>$x_1, x_2, x_3$</td>
</tr>
<tr>
<td>011</td>
<td>$x_4$</td>
</tr>
<tr>
<td>100</td>
<td>$x_5, x_6$</td>
</tr>
<tr>
<td>101</td>
<td>$x_7$</td>
</tr>
<tr>
<td>110</td>
<td>$-$</td>
</tr>
<tr>
<td>111</td>
<td>$-$</td>
</tr>
</tbody>
</table>
Elias-Fano: Bucketing saves space

- Store only low-order bits.
- (Keep cumulative bucket sizes.)

**Example**

`select(6)`

<table>
<thead>
<tr>
<th>bkt</th>
<th>sz</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00, 01, 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x1</code>, <code>x2</code>, <code>x3</code></td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x4</code></td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>00, 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x5</code>, <code>x6</code></td>
</tr>
<tr>
<td>101</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x7</code></td>
</tr>
</tbody>
</table>
Elias-Fano: Wrap up

- Choose $b = \lfloor \lg m \rfloor$ bits. In bucket: $\lfloor \lg n \rfloor - \lfloor \lg m \rfloor$-bit keys.
- $m \lg n - m \lg m + O(m) + \text{space for cumulative bucket sizes.}$

<table>
<thead>
<tr>
<th>Bucket no</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucket size</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Unary encoding: 0, 0, 3, 1, 2, 1, 0, 0 $\rightarrow$ 110001010010111.

$z$ buckets, total size $m \Rightarrow m + z$ bits ($z = 2^{\lfloor \lg m \rfloor}$).

- Total space $= m \log(n/m) + O(m) = nH_0 + O(m)$ bits.
- In which bucket is the 6th key? $\implies \text{Index of 6th 0} - 6$. 
**Bit Vector Uses (4): Wavelet Tree**

**Data:** $n$-symbol string, alphabet size $\sigma$. [Grossi and Vitter, *SJC ’05*]

Convert string into $\lg \sigma$ bit-vectors: $n \log \sigma + o(n \log \sigma)$ bits.

*raw size*

\[ \text{alabar}_a\_la\_alabarda} \]

\[ \text{aaba}_a\_a\_aabaa} \]

\[ \text{aaa}_a\_a\_aaa} \]

\[ \text{bb} \]

\[ \text{llld} \]

\[ \text{rr} \]

\[ \text{[Navarro, *CPM’12*]} \]

*rank and select* operations on any character in $O(\log \sigma)$ time.
Bit Vector Uses (4): Wavelet Tree (cont’d)

**Data:** Permutation over \(\{1, \ldots, n\}\) (string over alphabet of size \(n\)). View as 2-D point set on \(n \times n\) integer grid [Chazelle, *SJC '88*]:

Represent as wavelet tree: using \(n + o(n)\) bits, in \(O(1)\) time reduce to \(n/2 \times n/2\) integer grid.

▷ Range tree in disguise, but takes only \(O(n)\) words of space, not \(O(n \lg n)\) words. Orthogonal range queries in \(O(\lg n)\) time.
## Bit Vector Uses (4): Wavelet Tree (cont’d)

A wavelet tree can represent any array of (integer) values from an alphabet of size $\sigma$. A huge number of complex queries can be supported in $O(\log \sigma)$ time:

- range queries
- positional inverted indexes
- graph representation
- permutations

- numeric sequences
- document retrieval
- binary relations
- similarity search

▷ Very flexible and space-efficient data representation.
▷ Adequately fast for most applications (a few memory accesses).
▷ Well supported by libraries.
Graph Similarity Search [Tabei and Tsuda, SDM 2011]

- View chemical compound as a feature vector / bag of words.
- Database of compounds: $W_1, W_2, \ldots$
- Given query $Q$, find all compounds at small Jaccard distance
- Filtering criterion: find all $W_i$ s.t. $|Q \cap W_i| \geq k$.
- Like text search, but $|Q|$ is large.

Inverted list as wavelet tree.
Fairly fast, fairly memory-efficient, much better scalability.
• View chemical compound as a feature vector / bag of words.
• Given query $Q$, find all compounds at small Jaccard distance.
• Filtering criterion: find all $W_i$ s.t. $|Q \cap W_i| \geq k$.
• Based on multi-bit tree [Kristensen et al., WABI ’09].

• Fast, but memory-hungry.
• Represent tree structure succinctly.
• Represent original fingerprints with VLE and bit-vectors.
• 10x reduction in space, scaled to 30 million compounds.
Compound-Protein Pair Similarity [Tabei et al., *KDD ’13*]

- View chemical compound as a feature vector / bag of words.
- Given query $Q$, find all compounds at small Jaccard distance
- Database: $W_1, W_2, \ldots$
- New constraint: each $W_i$ has a real number $F(W_i)$ giving its *functionality*.
  -Filtering criterion: find all $W_i$ s.t. $|Q \cap W_i| \geq k$.
  -In addition, we only want those $W_i$ s.t. $|F(W_i) - F(Q)| \leq \delta$.
- Idea is similar to the wavelet-tree based similarity search, but we have an additional array containing $F(W_i)$.
- This array is also represented as a wavelet tree, except this time viewed as a range tree.
- Extra constraint is just orthogonal range reporting.
- Highly scalable, very accurate results.
A number of good implementations of succinct data structures in C++ are available.

Different platforms, coding styles:

- **sdsl-lite** (Gog et al. U. Melbourne).
- **succinct** (Grossi and Ottaviano, U. Pisa).
- **Sux4J** (Vigna, U. Milan, Java).
- **LIBCDS** (Claude and Navarro, Akori and U. Chile).

All open-source and available as Git repositories.
#include <sdsl/wavelet_trees.hpp>
#include <iostream>

using namespace sdsl;
using namespace std;

int main(int argc, char* argv[]) {
    wt_huff<rrr_vector<63>> wt;
    construct(wt, argv[1], 1);

    cout << "wt.size()=" << wt.size() << endl;
    cout << "wt.sigma =" << wt.sigma << endl;
    if (wt.size() > 0) {
        cout << "wt[0]=" << wt[0] << endl;
        uint64_t r = wt.rank(wt.size(), wt[0]);
        cout << "wt.rank(wt.size(), wt[0])=" << r << endl;
        cout << "wt.select(r, wt[0]) = " << wt.select(r, wt[0]) << endl;
    }
}

- Probably the most extensive, best-documented and maintained library.
- Easy to use, flexible, very efficient (may not be the absolute best in efficiency).
Other sdsl-lite features:

- Structured to facilitate flexible prototyping of new high-level structures (building upon bases such as bit vectors).
- Robust in terms of scale, handling input sequences of arbitrary length over arbitrary alphabets.
- Serialization to disk and loading, memory usage visualization.
Conclusions

- Use of succinct data structures can allow scalable mining of big data *using existing algorithms*.
  - With machines with 100s of GB RAM, maybe even Big Data can be mined using compressed data structures.
- Succinct data structures need to be chosen and used appropriately: much room for application-specific tweaks and heuristics.
- Many of the basic theoretical foundations have been laid, and succinct data structures have never been easier to use.