Perspectives in Informatics 3
Succinct Data Structures for Data Mining

Rajeev Raman

University of Leicester

Kyōto University, Fall 2015
Overview

Introduction

Compressed Data Structuring

Data Structures
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 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).
Overview

Introduction

Compressed Data Structuring

Data Structures
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.
Compressed Data Structuring

Compressing vs. Data Structuring

Answering queries on data requires an index in addition to the data. Index may be larger than the data.

- **Suffix tree**: data structure for indexing a text of \( n \) bytes.
  - Supports many indexing and search operations.
  - Careful implementation: \( 20n \) bytes of index data in worst case
    \([Kurtz, \text{SPrEx '99}]\)

- **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.
Space Measures

### Data Size
- Naive
- Information-theoretic Entropy ($H_0$)
- Information-theoretic 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 ($\log_2$).
- “Entropy” is usually an empirical version of classical Shannon entropy.
Compressibility: Empirical Entropy

Definition (zero-th order Empirical Entropy)

A string $S$ of length $n$ with alphabet $\{\sigma_1, \ldots, \sigma_n\}$, which contains $n_i$ occurrences of symbol $i$. Let $p_i = n_i/n$. The zeroth-order empirical entropy of $S$ is given by:

$$H_0(S) = p_1 \log_2(1/p_1) + p_2 \log_2(1/p_2) + \ldots + p_n \log_2(1/p_n)$$

Entropy is a measure of compressibility.

1. Suppose that the alphabet is $\{0, 1\}$ and $p_0 = 0.7$, $p_1 = 0.3$.
   - $H_0(S) = -(0.7 \times \log_2 0.7 + 0.3 \times \log_2 0.3) \sim 0.88$ bits/symbol.
   - This string is compressible.

2. $p_0 = 0.5$, $p_1 = 0.5$
   - $H_0(S) = 0.5 \times 1 + 0.5 \times 1 = 1$ bits/symbol.
   - This string is not compressible according to $H_0$ entropy.

3. Algorithms such as arithmetic coding and Huffman coding will compress a string to close to $nH_0(S)$ bits.
Compressibility: Empirical Entropy

$H_0$ is a basic measure of compressibility.

- $0^{n/2}1^{n/2}$ is very compressible, but $p_0 = p_1 = 1/2$ and $H_0 = 1$.

$H_k$ for $k > 1$ ($k$-th order empirical entropy) is better.

- Given $k$ previous symbols, how compressible is the next one?
- Definition of $H_1(S)$ for alphabet $\{0, 1\}$:
  - $n_{00}, n_{01}, n_{10}, n_{11}$: # times 00, 01, 10, 11 appear in $S$. E.g. if $S = 001011$ then $n_{00} = 1$, $n_{01} = 2$, $n_{10} = 1$ and $n_{11} = 1$.
  - Let $p_{0,0} = n_{00}/n_0, p_{0,1} = n_{01}/n_0$ and so on.
  - $n_0, n_1, p_0$ and $p_1$ be as before.
  - $H_1(S) = p_0 \cdot (p_{0,0} \log_2(1/p_{0,0}) + p_{0,1} \log_2(1/p_{0,1})) + p_1 \cdot (p_{1,0} \log_2(1/p_{1,0}) + p_{1,1} \log_2(1/p_{1,1}))$
  - If $S = 0^{n/2}1^{n/2}$ then $p_{0,1} \sim 2/n$, $p_{0,0} \sim (1 - 2/n)$ and $nH_0(S) = O(\log n)$ bits.

- Algorithm such as bzip compress $S$ to $nH_k(S)$. 
Summary of Compressibility for Binary Strings

\( x \in \{0, 1\}^n. \)

- **Naive**: \( n \) bits.
- **Information-Theoretic**: \( \lg 2^n = n \) bits.
- **Entropy (0-th)**: \( H_0. \)
  - If \( n_0 = n_1 = n/2 \), \( nH_0(S) = n. \)
  - If \( n_0 = 0.7n \), \( nH_0(S) \sim 0.88n \) bits.
  - Closely related to \( \lg \binom{n}{n_1} = n_1 \lg \left( \frac{n}{n_1} \right) + O(n_1). \)
  - If \( n_1 \ll n \), \( nH_0(S) \ll n. \)
- **Entropy (k-th)**: As before.

Even \( H_k \) doesn’t capture everything. Take a randomly generated bit string \( S \). Then \( H_k = 1 \) bit/symbol. Take \( T = SS \). \( H_k = 1 \) bit/symbol, but \( T \) is compressible.
Given object $x$ is a binary tree with $n$ nodes (excluding external nodes, which are shown as blue squares above).

- **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*]
Overview

Introduction

Compressed Data Structuring

Data Structures
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.

Bit Vectors: Results

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

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

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

- space \(H_0(X) + O(m)\) bits [Elias, \textit{J. ACM '75}].

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

**Aim:** Represent bit-string $X$ of length $n$ using $H_0(X) + O(m)$ bits, and support only $select_1$ in $O(1)$ time.

[Eliaś, *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 $X = \{x_1, \ldots, x_m\}$.
- $select_1(i) = x_i$. 
Elias-Fano: MSB Bucketing

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

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

<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>

| $x_1$ | 0 1 0 | 0 0 |
| $x_2$ | 0 1 0 | 0 1 |
| $x_3$ | 0 1 0 | 1 1 |
| $x_4$ | 0 1 1 | 0 1 |
| $x_5$ | 1 0 0 | 0 0 |
| $x_6$ | 1 0 0 | 1 0 |
| $x_7$ | 1 0 1 | 1 1 |
Elias-Fano: Bucketing saves space

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

**Example**

$select(6)$
Elias-Fano: Wrap up

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

**Bucket no:** 000 001 010 011 100 101 110 111

**Bucket size:** 0 0 3 1 2 1 0 0

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? ▶ “Index of 6th 0” $- 6$.  

Bit Vector Uses (2): Binary Tree

**Data:** $n$-node binary tree.

**Operations:** Navigation (left child, right child, parent).

- Encode internal node by 1 and external by 0. Visit nodes in level-order and output the bit in that node ($2n + 1$ bits).
  
  [Jacobson, *FOCS '89*] Store sequence of bits as bit vector.

- Number internal nodes by position of 1 in bit-string
  
  - *Non-consecutive* numbering (1, 2, 3, 4, 6, 7, 9, 12 above).

- Left child $= 2 \times \text{rank}_1(i)$. E.g. Left child of node 7 $= 6 \times 2 = 12$. Right child $= 2 \times \text{rank}_1(i) + 1$. parent $= \text{select}_1(\lfloor i/2 \rfloor)$.
Bit Vector Uses (3): Wavelet Tree

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

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

\[
\text{rank and select operations on any character in } O(\log \sigma) \text{ time.}
\]
Bit Vector Uses (3): 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 (3): 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.