Advanced Data Structures

Lecture 01: Bit Vectors

Florian Kurpicz
Bit Vectors

Succinct Data Structures
- represent data structures space efficient
- close to their information theoretical minimum
- using every bit becomes necessary

Succinct Trees
- represent a tree with $n$ nodes using only $2n$ bits
- navigation is possible with additional $o(n)$ bits

- storing a bit vector in practice is tricky
- 1101101 should require only a single byte
Efficient Bit Vectors in Practice (1/3)

**std::vector<char/int/...>**
- easy access
- very big: 1, 4, ... bytes per bit

**std::vector<bool>**
- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

**std::vector<uint64_t>**
- requires 8 bytes per bit(?)
- store 64 bits in 8 bytes
- how to access bits

- $i/64$ is position of 64-bit word
- $i \% 64$ is position in 64-bit word

![Diagram showing bit vector layout]
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> (63 - (i % 64)) & 1ULL;
Efficient Bit Vectors in Practice (3/3)

(block >> (63-(i%64))) & 1ULL;

- fill bit vector from left to right

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>62</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| 0 | 0 | 0 | 0 | 0 | 0 |      | 1  | 0 |

- assembler code:  
  `mov ecx, edi`
  `not ecx`
  `shr rsi, cl`
  `mov eax, esi`
  `and eax, 1`

(block >> (i%64)) & 1ULL;

- fill blocks in bit vector right to left

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>...</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| 0  | 0  | ... | 1  | 1  | 0  | 0  | 1  | 0  |

- assembler code:  
  `mov ecx, edi`
  `shr rsi, cl`
  `mov eax, esi`
  `and eax, 1`
**Rank Queries on Bit Vectors (1/2)**

- \( \text{rank}_\alpha(i) \): Number of \( \alpha \)s before \( i \)
- \( \text{select}_\alpha(j) \): Position of \( j \)-th \( \alpha \)

### Diagram

- \( \text{rank}_0(5) \) and \( \text{select}_1(5) \)
- Number of 0s w.r.t. super-block
- Number of 0s w.r.t. BV

### Example Bit Vector

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

**PINGO-Frage**

- \( \text{select}_1(5) \): Position of the 5-th \( \alpha \)

---

**Rank Queries on Bit Vectors (2/2)**
Rank Queries on Bit Vectors (2/2)

- for a bit vector of size \( n \)
- blocks of size \( s = \lfloor \frac{\lg n}{2} \rfloor \)
- super blocks of size \( s' = s^2 = \Theta(\lg^2 n) \)

- for all \( \lfloor \frac{n}{s'} \rfloor \) super blocks, store number of 0s from beginning of super block to end of block
- \( n/s' \cdot \lg s' = O\left(\frac{n \lg \lg n}{\lg n}\right) = o(n) \) bits of space

- for all length-\( s \) bit vectors, for every position \( i \)
- store number of 0s up to \( i \)
- \( 2^{\frac{\lg n}{2}} \cdot s' \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n) \) bits of space

- query in \( O(1) \) time
- \( rank_0(i) = i - rank_1(i) \)
rank_\(\alpha\)(i) \# of \(\alpha\)s before \(i\)
select_\(\alpha\)(j) position of \(j\)-th \(\alpha\)

# of 0s w.r.t. super-block
# of 0s w.r.t. BV
PINGO-Frage

rank_0(5)
select_1(5)
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- PINGO-Frage
- Naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and $select_0(i) = S[i]$ if $k \in O(n/\log n)$ this suffice

Better: $k/b$ variable-sized super-blocks $B_i$, such that super-block contains $b = \log^2 n$ zeros

- $select_0(i) = \sum_{j=0}^{[i/b]-1} |B_j| + select_0(B_{[i/b]} \cdot j - ([i/b] \cdot b))$
- Storing all possible results for the (prefix) sum
  - $O((k \log n)/b) = o(n)$ bits of space
- Select on block depends on size of block
  - $|B_{[i/b]}| \geq \log^4 n$: store answers naively
    - requires $O(b \log n) = O(\log^3 n)$ bits of space
    - there are at most $O(n/\log^4 n)$ such blocks
    - total $O(n/\log n) = o(n)$ bits of space
  - $|B_{[i/b]}| < \log^4 n$: divide super-block into blocks
    - same idea: variable-sized blocks containing $b' = \sqrt{\log n}$ zeros
    - (prefix) sum $O((k \log n)/b') = o(n)$ bits
    - if size $\geq \log n$ store all answers
    - if size $< \log n$ store lookup table
Rank- and Select-Queries on Bit Vectors

Lemma: Binary Rank- and Select-Queries

Given a bit vector of size $n$, there exist data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time.
## This Lecture
- bit vectors
- rank and select on bit vectors
- efficient bit vectors in practice

## Next Lecture
- succinct trees using bit vectors
- navigation in succinct trees

### Advanced Data Structures
- BV