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
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
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
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
Efficient Bit Vectors in Practice (1/3)

\[\text{std::vector<char/int/...>}\]
- easy access
- very big: 1, 4, \ldots\) bytes per bit

\[\text{std::vector<bool>}\]
- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

\[\text{std::vector<uint64_t>}\]
- requires 8 bytes per bit(?)
- store 64 bits in 8 bytes
- how to access bits

\[i/64\text{ is position of 64-bit word}\]
\[i\%64\text{ is position in 64-bit word}\]
// 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;
// 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;

shift bits right

0 1 2 3 4 5 ... 62 63
1 1 1 0 1 0 ... 1 0
// 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;

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

>> 60

| 0 1 2 3 4 5 ... 62 63 |
| 0 0 0 0 0 0 ... 1 0 |
// 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;

<table>
<thead>
<tr>
<th>Shift bits right</th>
<th># bits</th>
<th>Logical and 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 ... 62 63</td>
<td>0 1 2 3 4 5 ... 62 63</td>
<td>0 1 2 3 4 5 ... 62 63</td>
</tr>
<tr>
<td>1 1 1 0 1 0 ... 1 0</td>
<td>0 0 0 0 0 0 ... 1 0</td>
<td>0 0 0 0 0 0 ... 1 0</td>
</tr>
</tbody>
</table>

>>> 60
Efficient Bit Vectors in Practice (3/3)

\[(\text{block} \gg (63-(\text{i}\%64))) \& \text{1ULL};\]
- 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 0</td>
<td></td>
</tr>
</tbody>
</table>

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

\[(\text{block} \gg (\text{i}\%64)) \& \text{1ULL};\]
- 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 1 ...</td>
<td>0</td>
<td>1 0</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

6/12 2024-04-16 Florian Kurpicz | Advanced Data Structures | 01 Bit Vectors

Institute of Theoretical Informatics, Algorithm Engineering
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  |

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

- fill bit vector from left to right

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

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

Efficient Bit Vectors in Practice (3/3)

\[(\text{block} >> (63-(i\%64))) \& 1\text{ULL};\]

- fill bit vector from left to right

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & \cdots & 62 & 63 \\
1 & 1 & 1 & 0 & 1 & 0 & \cdots & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 \\
\end{array}
\]

- assembler code:  
  \[
  \text{mov ecx, edi} \\
  \text{not ecx} \\
  \text{shr rsi, cl} \\
  \text{mov eax, esi} \\
  \text{and eax, 1}
  \]

\[(\text{block} >> (i\%64)) \& 1\text{ULL};\]

- fill blocks in bit vector right to left

\[
\begin{array}{cccccccc}
63 & 62 & \cdots & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 1 & \cdots & 0 & 1 & 0 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & \cdots & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

- assembler code:  
  \[
  \text{mov ecx, edi} \\
  \text{shr rsi, cl} \\
  \text{mov eax, esi} \\
  \text{and eax, 1}
  \]
Rank Queries on Bit Vectors (1/2)

\[
\text{rank}_\alpha(i) \quad \# \text{ of } \alpha \text{s before } i \\
\text{select}_\alpha(j) \quad \text{position of } j\text{-th } \alpha
\]

---

0 1 2 3 4 5 6 7 8 9

0 1 1 0 1 1 0 1 0 0
Rank Queries on Bit Vectors (1/2)

\( \text{rank}_\alpha(i) \) # of \( \alpha \)s before \( i \)
\( \text{select}_\alpha(j) \) position of \( j \)-th \( \alpha \)

\( \text{rank}_0(5) \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Rank Queries on Bit Vectors (1/2)

\[ \text{rank}_\alpha(i) \] \# of \(\alpha\)s before \(i\)

\[ \text{select}_\alpha(j) \] position of \(j\)-th \(\alpha\)

rank\(_0\)(5)
Rank Queries on Bit Vectors (1/2)

\( \text{rank}_{\alpha}(i) \) # of \( \alpha \)s before \( i \)

\( \text{select}_{\alpha}(j) \) position of \( j \)-th \( \alpha \)

\( \text{rank}_0(5) \)

\[
\begin{array}{cccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Rank Queries on Bit Vectors (1/2)

\[
\text{rank}_\alpha(i) \quad \# \text{ of } \alpha \text{s before } i \\
\text{select}_\alpha(j) \quad \text{position of } j\text{-th } \alpha \\
\text{rank}_0(5) \\
\text{select}_1(5)
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Rank Queries on Bit Vectors (1/2)

\[ \text{rank}_\alpha(i) \# \text{ of } \alpha \text{s before } i \]

\[ \text{select}_\alpha(j) \text{ position of } j\text{-th } \alpha \]

- \[ \text{rank}_0(5) \]
- \[ \text{select}_1(5) \]
Rank Queries on Bit Vectors (1/2)

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

**Example:**

- \( \text{rank}_0(5) \):
  - **Super-block # of 0s w.r.t. BV**
  - **Super-block # of 0s w.r.t. BV**

**Diagram:**

- **PINGO-Frage**

```plaintext
0 1 1 0 1 1 0 1 0 0
```

- **2**

- **5**
Rank Queries on Bit Vectors (1/2)

\[ \text{rank}_\alpha (i) \] \# of \( \alpha \)s before \( i \)

\[ \text{select}_\alpha (j) \] position of \( j \)-th \( \alpha \)

\[ \text{rank}_0 (5) \]

<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>

2

super-block

# of 0s w.r.t. BV
Rank Queries on Bit Vectors (1/2)

\( \text{rank}_\alpha(i) \) \# of \( \alpha \)s before \( i \)

\( \text{select}_\alpha(j) \) position of \( j \)-th \( \alpha \)

---

```
0 1 1 0 1 1 0 1 0 0
```

---

2

\( \text{rank}_0(5) \)

---

# of 0s w.r.t. super-block

---

# of 0s w.r.t. BV

---

block

---

super-block
Rank Queries on Bit Vectors (1/2)

\( \text{rank}_\alpha(i) \) \# of \( \alpha \)s before \( i \)

\( \text{select}_\alpha(j) \) position of \( j\)-th \( \alpha \)

## Diagram Description

- **Block**: Represents a group of elements.
- **Super-block**: A larger grouping, typically used for optimization.
- **# of 0s w.r.t. super-block**: Indicates the number of 0s within the super-block.
- **# of 0s w.r.t. BV**: Gives the number of 0s within the entire bit vector.

**Example**:

- For \( \text{rank}_0(5) \), the diagram shows that there are 2 0s before the 5th position.

---

PINGO-Frage: 7/12

2024-04-16 Florian Kurpicz | Advanced Data Structures | 01 Bit Vectors

Institute of Theoretical Informatics, Algorithm Engineering
for a bit vector of size $n$

blocks of size $s = \left\lfloor \frac{\lg n}{2} \right\rfloor$

super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
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 bit vector to end of super-block
- $n/s' \cdot \lg n = O\left(\frac{n}{\lg n}\right) = o(n)$ bits of space
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 bit vector to end of super-block
- $n/s' \cdot \lg s' = O\left(\frac{n \lg \lg n}{\lg n}\right) = o(n)$ bits of space

for all $\lfloor \frac{n}{s} \rfloor$ 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

query in $O(1)$ time
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
$n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ 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
Rank Queries on Bit Vectors (2/2)

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

- for all $\left\lfloor \frac{n}{s'} \right\rfloor$ 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 $i$
  - $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O\left(\sqrt{n} \lg n \lg \lg n\right) = o(n)$ bits of space

- query in $O(1)$ time 📅
  - $\text{rank}_0(i) = i - \text{rank}_1(i)$
Rank Queries on Bit Vectors (1/2)

\( \text{rank}_\alpha(i) \) # of \( \alpha \)s before \( i \)

\( \text{select}_\alpha(j) \) position of \( j \)-th \( \alpha \)

\( \text{rank}_0(5) \)

\( \text{select}_1(5) \)

0 1 1 0 1 1 0 1 0 0

0 1 2 3 4 5 6 7 8 9
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- PINGO-Frage
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
    
    $select_0(i) = S[i]$ \(\text{if } k \in O(n/\lg n) \text{ this suffice}

if size $\geq \lg n$ store all answers
if size $< \lg n$ store lookup table
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 $k \in O(n/\lg n)$ this suffice

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

- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, i - (\lfloor i/b \rfloor b))$
Select in \( o(n) \) Space and \( O(1) \) Time

- \( \text{select}_0 \) 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
    \[ \text{select}_0(i) = S[i] \quad \text{if } k \in O(n/\lg n) \] this suffice

- better: \( k/b \) variable-sized super-blocks \( B_i \), such that super-block contains \( b = \lg^2 n \) zeros
- \[ \text{select}_0(i) = \sum_{j=0}^{[i/b]-1} |B_j| + \text{select}_0(B_{[i/b]}, i - ([i/b]b)) \]

- storing all possible results for the (prefix) sum
- \( O((k \lg n)/b) = o(n) \) bits of space
Select in \( o(n) \) Space and \( O(1) \) Time

- \( \text{select}_0 \) 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
    \( \text{select}_0(i) = S[i] \) if \( k \in O(n / \lg n) \) this suffice

- better: \( k / b \) variable-sized super-blocks \( B_i \), such that super-block contains \( b = \lg^2 n \) zeros

- \( \text{select}_0(i) = \sum_{j=0}^{[i/b]-1} |B_j| + \text{select}_0(B_{[i/b]}, i - ([i/b]b)) \)

- storing all possible results for the (prefix) sum
  - \( O((k \lg n) / b) = o(n) \) bits of space

- select on block depends on size of block

- \( |B_{[i/b]}| \geq \lg^4 n \): store answers naively
  - requires \( O(b \lg n) = O(\lg^3 n) \) bits of space
  - there are at most \( O(n / \lg^4 n) \) such blocks
  - total \( O(n / \lg n) = o(n) \) bits of space
Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size $n$ that contains $k$ zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store $k$ solutions in $S[1..k]$ and
    $select_0(i) = S[i] \uparrow$ 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
  - $select_0(i) = \sum_{j=0}^{[i/b]-1} |B_j| + select_0(B_{[i/b]}; i - ([i/b]b))$
- 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
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.
Conclusion and Outlook

This Lecture
- bit vectors
- rank and select on bit vectors

Advanced Data Structures
BV
Conclusion and Outlook

This Lecture
- bit vectors
- rank and select on bit vectors
- efficient bit vectors in practice

Advanced Data Structures
- BV
# Conclusion and Outlook

## 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