Advanced Data Structures

Lecture 01: Bit Vectors

Florian Kurpicz
https://pingo.scc.kit.edu/498901
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
- 11011101 should require only a single byte
std::vector<char/int/...>

- easy access
- very big: 1, 4, ... bytes per bit
### Efficient Bit Vectors in Practice (1/3)

<table>
<thead>
<tr>
<th>std::vector&lt;char/int/...&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>- easy access</td>
</tr>
<tr>
<td>- very big: 1, 4, ... bytes per bit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>std::vector&lt;bool&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>- bit vector in C++ (1 bit per byte)</td>
</tr>
<tr>
<td>- easy access</td>
</tr>
<tr>
<td>- layout depending on implementation</td>
</tr>
</tbody>
</table>
### Efficient Bit Vectors in Practice (1/3)

<table>
<thead>
<tr>
<th>std::vector&lt;char/int/...&gt;</th>
<th>std::vector&lt;uint64_t&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>- easy access</td>
<td>- requires 8 bytes per bit(?)</td>
</tr>
<tr>
<td>- very big: 1, 4, ... bytes per bit</td>
<td>- store 64 bits in 8 bytes</td>
</tr>
<tr>
<td>std::vector&lt;bool&gt;</td>
<td>- how to access bits</td>
</tr>
<tr>
<td>- bit vector in C++ (1 bit per byte)</td>
<td></td>
</tr>
<tr>
<td>- easy access</td>
<td></td>
</tr>
<tr>
<td>- layout depending on implementation</td>
<td></td>
</tr>
</tbody>
</table>
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 in 64-bit word
- \( i \% 64 \) is position in word
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 in 64-bit word
- \( i \% 64 \) is position in word

![Diagram of bit vector access](image-url)
// 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**

<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>
// 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
# bits
logical and 1

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

>> 60

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

and 1

1 0
Efficient Bit Vectors in Practice (3/3)

(block >> (63-(i%64))) & 1ULL;
- fill bit vector from left to right

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

0 0 0 0 0 0 ... 1 0

(block >> (i%64)) & 1ULL;
- fill blocks in bit vector right to left

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

0 0 ... 1 1 0 0 1 0

assembler code:
mov ecx, edi
not ecx
shr rsi, cl
mov eax, esi
and eax, 1
Efficient Bit Vectors in Practice (3/3)

- Fill bit vector from left to right
  - (block >> (63-(i%64))) & 1ULL;

- Fill blocks in bit vector right to left
  - (block >> (i%64)) & 1ULL;
Efficient Bit Vectors in Practice (3/3)

\[
(block \gg (63-(i\%64))) \& 1\text{ULL};
\]
- fill bit vector from left to right

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

\[
0 & 0 & 0 & 0 & 0 & 0 & \ldots & 1 & 0 \\
\]

- assembler code:
  \begin{verbatim}
  mov ecx, edi 
  not ecx 
  shr rsi, cl 
  mov eax, esi 
  and eax, 1 
  \end{verbatim}
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 | 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)

<table>
<thead>
<tr>
<th>Rank Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{rank}_\alpha(i) ) # of ( \alpha )s before ( i )</td>
</tr>
<tr>
<td>( \text{select}_\alpha(j) ) position of ( j )-th ( \alpha )</td>
</tr>
</tbody>
</table>

<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 \( \alpha \)

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

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

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

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

```
2
0 1 2 3 4 5 6 7 8 9
0 1 1 0 1 1 0 1 0 0
```

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

(rank_α(i)) \# of αs before i
(select_α(j)) position of j-th α

rank_0(5) = 2
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 \( \alpha \)

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

\[ \text{select}_1(5) \]
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 $\alpha$

```
  0 1 2 3 4 5 6 7 8 9
0 1 1 0 1 1 0 1 0 0
```

- $\text{rank}_0(5) = 2$
Rank Queries on Bit Vectors (1/2)

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

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

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

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

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

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

---

`PINGO-Frage` (5)

`rank_0(5)`

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

## # of 0s w.r.t. BV

---

**block**

**super-block**

---

`2`

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

---

2022-04-25 Florian Kurpicz | Advanced Data Structures | 01 Bit Vectors

Institute of Theoretical Informatics, Algorithm Engineering
Rank Queries on Bit Vectors (1/2)

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

\[ \text{rank}_{0}(5) \]

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

# of 0s w.r.t. BV

\begin{array}{cccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
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 $s' = s^2 = \Theta(\lg^2 n)$
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
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ 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 \( \left\lfloor \frac{n}{s'} \right\rfloor \) 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 \( \left\lfloor \frac{n}{s'} \right\rfloor \) 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
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 $i$
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O\left(\sqrt{n} \lg n \lg \lg n\right) = o(n)$ bits of space

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

query in $O(1)$ time
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
  - $n/s' \cdot \lg s' = O(\frac{n \lg n}{\lg n}) = o(n)$ 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

- 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 $\alpha$

$\text{rank}_0(5)$

$\text{select}_1(5)$

<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
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
- \text{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
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

- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor} \cdot j - (\lfloor i/b \rfloor b))$
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 \( \text{select}_0(i) = S[i] \) 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

\begin{align*}
\text{select}_0(i) &= \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + \text{select}_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b)) \\
&+ \text{storing all possible results for the (prefix) sum}
\end{align*}

\( O((k \lg n)/b) = o(n) \) bits of space
Select in \( o(n) \) Space and \( O(1) \) Time

- **select** \( b \) 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 \( select(b) = S[i] \) if \( k \in O(n/lgn) \) this suffice

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

- 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

- $\text{select}_0$ 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 $k \in O(n/\log n)$ this suffice

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

$\text{select}_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + \text{select}_0(B_{\lfloor i/b \rfloor} \cdot j - (\lfloor i/b \rfloor b))$

- 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

  $|B_{\lfloor i/b \rfloor}| \geq \log^4 n$: 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

  $|B_{\lfloor i/b \rfloor}| < \log^4 n$: 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.
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
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