Compressed Suffix Trees in Practice

Simon Gog

Computing and Information Systems
The University of Melbourne

February 13th 2013
Outline

1. Introduction
   - Basic data structures
   - The suffix tree

2. CST design
   - NAV (tree topology and navigation)
   - CSA (lexicographic information)
   - LCP (longest common prefixes)

3. CST in practice
   - The sdsl library
Succinct data structures (1)

Data structure $D$

- representation of an object $X$
- + operations on $X$

Example: Rank-bit-vector

- bit vector $b$ of length $n$
  - $(0, 1, 0, 1, 1, 0, 1, 1)$
  - $(0, 0, 1, 1, 2, 3, 3, 4)$
  - in $n$ bits space
- access $b[i]$ in $O(1)$ time
- rank($i$) = $\sum_{j=0}^{i-1} b[j]$ in $O(n)$ time

Succinct data structure $D$

Space of $D$ is close the information theoretic lower bound to represent $X$, while operations can still be performed efficient.
Succinct data structures (1)

**Data structure** $D$

representation of an object $X$ + operations on $X$

**Example: Rank-bit-vector**

bit vector $b$ of length $n$

$\begin{align*}
(0, 1, 0, 1, 1, 0, 1, 1) \\
(0, 0, 1, 1, 2, 3, 3, 4)
\end{align*}$

in $n$ bits space

+ access $b[i]$ in $O(1)$ time

+ $rank(i) = \sum_{j=0}^{i-1} b[j]$ in $O(n)$ time

**Succinct data structure** $D$

Space of $D$ is close the information theoretic lower bound to represent $X$, while operations can still be performed efficient.
Succinct data structures (1)

**Data structure $D$**
representation of an object $X$ + operations on $X$

**Example: Rank-bit-vector**
bit vector $b$ of length $n$
$(0, 1, 0, 1, 1, 0, 1, 1)$
$(0, 0, 1, 1, 2, 3, 3, 4)$
in $n + n \log n$ bits space

access $b[i]$ in $O(1)$ time
$rank(i) = \sum_{j=0}^{i-1} b[j]$ in $O(1)$ time

Succinct data structure $D$
Space of $D$ is close the information theoretic lower bound to represent $X$, while operations can still be performed efficient.
Succinct data structures (1)

Data structure $D$

- representation of an object $X$ + operations on $X$

Example: Rank-bit-vector

- bit vector $b$ of length $n$
  - $(0, 1, 0, 1, 1, 0, 1, 1)$
  - $(0, 0, 1, 1, 2, 3, 3, 4)$
  - in $n + n \log n$ bits space

- access $b[i]$ in $O(1)$ time
- $rank(i) = \sum_{j=0}^{i-1} b[j]$ in $O(1)$ time

Succinct data structure $D$

- Space of $D$ is close to the information theoretic lower bound to represent $X$, while operations can still be performed efficiently.
Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures \textit{in practice}?

- Less memory $\Rightarrow$ fewer CPU cycles !?
- Less memory $\Rightarrow$ less costs !?

Problems:
- in theory
  - develop succinct data structures
- in practice
  - constants in $\mathcal{O}(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement
Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures in practice?

- **Less memory ⇒ fewer CPU cycles !?**

![Memory Hierarchy Diagram]

- **Less memory ⇒ less costs !?**

**Problems:**
- **in theory**
  - develop succinct data structures
- **in practice**
  - constants in $O(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement
Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures \textit{in practice}?

- Less memory $\Rightarrow$ fewer CPU cycles !?
- Less memory $\Rightarrow$ less costs !?

<table>
<thead>
<tr>
<th>Instance name</th>
<th>main memory</th>
<th>price per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro</td>
<td>613.0 MB</td>
<td>0.02 US$</td>
</tr>
<tr>
<td>High-Memory Quadruple Extra Large</td>
<td>68.4 GB</td>
<td>2.00 US$</td>
</tr>
</tbody>
</table>

Pricing of Amazon's Elastic Cloud Computing (EC2) service in July 2011.

Problems:

- in theory
  - develop succinct data structures
- in practice
  - constants in $O(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement
Succinct data structures (2)

Can succinct data structures replace classic uncompressed data structures \textit{in practice}?

- Less memory $\Rightarrow$ fewer CPU cycles !?
- Less memory $\Rightarrow$ less costs !?

Problems:

- in theory
  - develop succinct data structures
- in practice
  - constants in $O(1)$-time terms are large
  - $o(n)$-space term is not negligible
  - complex data structures are hard to implement
The classic index data structure: The suffix tree (ST)

Let $T$ be a text of length $n$ over alphabet $\Sigma$ of size $\sigma$.

**Suffix tree**
- Index data structure for $T$ (construction $O(n)$)
- Can be used to solve many problems in optimal time complexity
  - Bioinformatics
  - Data compression
- Uses $O(n \log n)$ bits!
  - In practice (ASCII-alphabet) $\geq 17$ times the size of $T$

**Can not handle „The Attack of Massive Data”**
- DNA sequencing data (NGS)
- ...

Bioinformatics
Data compression
Example: ST of $T=\text{umulmundumulmum}$$

$n = 16$

$\Sigma =\{\$, d, l, m, n, u\}$

$\sigma = 6$

Classic implementation uses pointers each of size 4 or 8 bytes!
Example: ST of $T = \text{umulmumundumulmum}$

Operations

- $\text{root}()$
- $\text{is\_leaf}(v)$
- $\text{parent}(v)$
- $\text{degree}(v)$
- $\text{child}(v, c)$
- $\text{select\_child}(v, i)$
- $\text{depth}(v)$
- $\text{edge}(v, d)$
- $\text{lca}(v, w)$
- $\text{sl}(v)$
- $\text{wl}(v, c)$
## CSTs

### Goal of a CST implementation

Replace fastest uncompressed ST implementations in different scenarios

(a) both fit in RAM and we measure time
(b) both fit in RAM and we measure resource costs
(c) only CST fits in RAM and we measure time

### Proposals

- Sadakane’s CST `cst_sada`
- Fully Compressed Suffix Tree (Russo et al.)
- CSTs based on interval representation of nodes (Fischer et al. `cstY`, Ohlebusch et al. `cst_sct3`)
CSTs

Goal of a CST implementation

Replace fastest uncompressed ST implementations in different scenarios

(a) both fit in RAM and we measure time
(b) both fit in RAM and we measure resource costs
(c) only CST fits in RAM and we measure time

Proposals which might work for (a) and (b)

- Sadakane’s CST cst_sada
- Fully Compressed Suffix Tree (Russo et al.)
- CSTs based on interval representation of nodes (Fischer et al. cstY, Ohlebusch et al. cst_sct3)
Outline

1. Introduction
   - Basic data structures
   - The suffix tree

2. CST design
   - NAV (tree topology and navigation)
   - CSA (lexicographic information)
   - LCP (longest common prefixes)

3. CST in practice
   - The sdsl library
Big picture of CST design

- NAV
- CSA
- LCP

Pioneers

- Balanced Parentheses Sequence
- Wavelet Tree
- Huffman
- Min-Max-Tree
- Min-PLCP
- First child
Big picture of CST design
Example: Compressing NAV

$BPS_{dfs} = (())((()))((()))(())(())((()))((()))((()))(())(())((()))(())(())(())$}

Tree uncompressed
$\mathcal{O}(n \log n)$ bits

Compressed
$4n$ bits
Example: Compressing NAV

BPS\textsubscript{dfs} = (0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)

Tree uncompressed \(\mathcal{O}(n \log n)\) bits

Compressed \(4n\) bits
Example: Compressing NAV

$\text{BPS}_{dfs} = ((())())()(((()))())())(((())(())())())$

Tree uncompressed $O(n \log n)$ bits

Compressed $4n$ bits
Example: Compressing NAV

$$\text{BPS}_{dfs}=(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)$$

Tree uncompressed: $O(n \log n)$ bits

Compressed: $4n$ bits
Example: Compressing NAV

$BPS_{dfs} = (0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)$

Tree uncompressed: $\mathcal{O}(n \log n)$ bits

Compressed: $4n$ bits
Example: Compressing NAV

\[ BPS_{dfs} = (\text{tree uncompressed}) \equiv O(n \log n) \text{ bits} \]

\[ \text{compressed} \approx 4n \text{ bits} \]
Example: Compressing NAV

BPS\textsubscript{dfs} = (())(()(()(()(())(())())()))((())(()())))

Tree uncompressed
\(\mathcal{O}(n \log n)\) bits

Compressed
\(4n\) bits
**Example: Compressing NAV**

BPS_{dfs} = ( ) ( ( ( ) ( ( ( ) ) ) ) ) ( ( ( ) ) ( ( ( ) ) ) )

- Tree uncompressed: \( O(n \log n) \) bits
- Compressed: 4n bits
Example: Compressing NAV

 TREE UNCOMPRESSED

$$O(n \log n) \text{ bits}$$

$$BPS_{dfs} = (())(())(())(())(())(())(())(())(())(())(())(())$$

COMPRESSED

$$4n \text{ bits}$$
Example: Compressing NAV

\[ \text{tree uncompressed} \quad \Theta(n \log n) \text{ bits} \]

\[ \text{compressed} \quad 4n \text{ bits} \]

\[ \text{BPS}_{dfs} = (())(()(()())(()(()())())())(()())(()())(()) \]
Example: Compressing NAV

$BPS_{dfs} = (0)(0)(0)(0((0))(0)((0)(0))(0)(0)(0))(0)(0)(0))$

Tree uncompressed $\mathcal{O}(n \log n)$ bits

Compressed $4n$ bits
Example: Compressing NAV

BPS\textsubscript{dfs} = (())(()())()(()())(())()()()((())())((())())(())())

Tree uncompressed \( \mathcal{O}(n \log n) \) bits

Compressed \( 4n \) bits
NAV data structures (1)

$4n$ bits

$2n$ bits

$+o(n)$ bits to answer $\text{find\_open}(i)$, $\text{find\_close}(i)$, $\text{enclose}(i)$, $\text{double\_enclose}(i, j)$, $\text{rank}(i, c)$, $\text{select}(i, c)$, ... in constant time
Comparison of different NAV structures

<table>
<thead>
<tr>
<th>space in bits</th>
<th>cst_sada</th>
<th>cst_sct</th>
<th>cst_sct3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4n + o(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>2n + o(n)</td>
<td>O(t_{LCP} log σ)</td>
<td>O(t_{LCP})</td>
<td>O(t_{LCP})</td>
</tr>
<tr>
<td>3n + o(n)</td>
<td>O(1)</td>
<td>O(t_{LCP} log σ)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

- root(): O(1)
- degree(v): O(σ)
- depth(v): O(t_{LCP})
- parent(v): O(1)
- select_child(v, i): O(i)
- sibling(v): O(1)
- sl(v), lca(v, w): O(1)
- child(v, c): O(t_{SA} log σ)
Example operations: \texttt{select\_leaf}(i) and \texttt{lca}(v, w) on NAV

\[
\text{BPS}_{dfs} = (())()()(()(()))(()(()))(()())()(()())()
\]

\[
\begin{align*}
\text{select\_leaf}(4) &= \text{select}(4,\ '10') \\
&= 8 \\
\text{select\_leaf}(5) &= \text{select}(5,\ '10') \\
&= 12 \\
\text{lca}(8,12) &= \text{double\_enclose}(8,12) \\
&= 0
\end{align*}
\]
Virtues of a CSA based on BWT

- Small size: $|\text{CSA}| = |\text{BWT}| + \frac{n \log n}{s_{\text{SA}}}$ bits
  where $|\text{BWT}|$ can be chosen to be
  - $n \log \sigma$ bits
  - $nH_0(T)$ bits
  - $nH_k(T) + O(\sigma^k)$ bits
- Pattern matching in time $O(|P| \log \sigma)$ (even $O(|P|)$ for $\sigma \in \text{polylog}(n)$) by backward search (Ferragina & Manzini)
### $H_k$ of the Pizza&Chili 200MB test cases

<table>
<thead>
<tr>
<th>$k$</th>
<th>$H_k$</th>
<th>$CT/n$</th>
<th>$H_k$</th>
<th>$CT/n$</th>
<th>$H_k$</th>
<th>$CT/n$</th>
<th>$H_k$</th>
<th>$CT/n$</th>
<th>$H_k$</th>
<th>$CT/n$</th>
<th>$H_k$</th>
<th>$CT/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.257</td>
<td>0.000</td>
<td>1.974</td>
<td>0.000</td>
<td>4.525</td>
<td>0.000</td>
<td>4.201</td>
<td>0.000</td>
<td>7.000</td>
<td>0.000</td>
<td>5.465</td>
<td>0.000</td>
</tr>
<tr>
<td>1</td>
<td>3.479</td>
<td>0.000</td>
<td>1.930</td>
<td>0.000</td>
<td>3.620</td>
<td>0.000</td>
<td>4.178</td>
<td>0.000</td>
<td>7.000</td>
<td>0.000</td>
<td>4.077</td>
<td>0.000</td>
</tr>
<tr>
<td>2</td>
<td>2.170</td>
<td>0.000</td>
<td>1.920</td>
<td>0.000</td>
<td>2.948</td>
<td>0.000</td>
<td>4.156</td>
<td>0.000</td>
<td>6.993</td>
<td>0.000</td>
<td>3.102</td>
<td>0.000</td>
</tr>
<tr>
<td>3</td>
<td>1.434</td>
<td>0.0007</td>
<td>1.916</td>
<td>0.000</td>
<td>2.422</td>
<td>0.0005</td>
<td>4.066</td>
<td>0.0001</td>
<td>5.979</td>
<td>0.010</td>
<td>2.337</td>
<td>0.0012</td>
</tr>
<tr>
<td>4</td>
<td>1.045</td>
<td>0.0043</td>
<td>1.910</td>
<td>0.000</td>
<td>2.063</td>
<td>0.0028</td>
<td>3.826</td>
<td>0.0011</td>
<td>0.666</td>
<td>0.6939</td>
<td>1.852</td>
<td>0.0082</td>
</tr>
<tr>
<td>5</td>
<td>0.817</td>
<td>0.0130</td>
<td>1.901</td>
<td>0.000</td>
<td>1.839</td>
<td>0.0103</td>
<td>3.162</td>
<td>0.0173</td>
<td>0.006</td>
<td>0.9969</td>
<td>1.518</td>
<td>0.0250</td>
</tr>
<tr>
<td>6</td>
<td>0.705</td>
<td>0.0265</td>
<td>1.884</td>
<td>0.0001</td>
<td>1.672</td>
<td>0.0265</td>
<td>1.502</td>
<td>0.1742</td>
<td>0.000</td>
<td>1.000</td>
<td>1.259</td>
<td>0.0509</td>
</tr>
<tr>
<td>7</td>
<td>0.634</td>
<td>0.0427</td>
<td>1.862</td>
<td>0.0001</td>
<td>1.510</td>
<td>0.0553</td>
<td>0.340</td>
<td>0.4506</td>
<td>0.000</td>
<td>1.000</td>
<td>1.045</td>
<td>0.0850</td>
</tr>
<tr>
<td>8</td>
<td>0.574</td>
<td>0.0598</td>
<td>1.834</td>
<td>0.0004</td>
<td>1.336</td>
<td>0.0991</td>
<td>0.109</td>
<td>0.5383</td>
<td>0.000</td>
<td>1.000</td>
<td>0.867</td>
<td>0.1255</td>
</tr>
<tr>
<td>9</td>
<td>0.537</td>
<td>0.0773</td>
<td>1.802</td>
<td>0.0013</td>
<td>1.151</td>
<td>0.1580</td>
<td>0.074</td>
<td>0.5588</td>
<td>0.000</td>
<td>1.000</td>
<td>0.721</td>
<td>0.1701</td>
</tr>
<tr>
<td>10</td>
<td>0.508</td>
<td>0.0955</td>
<td>1.760</td>
<td>0.0051</td>
<td>0.963</td>
<td>0.2292</td>
<td>0.061</td>
<td>0.5699</td>
<td>0.000</td>
<td>1.000</td>
<td>0.602</td>
<td>0.2163</td>
</tr>
</tbody>
</table>
# Backward search

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{T}^{\text{BWT}}$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>m</td>
<td>$$$</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>dumulmum$</td>
</tr>
<tr>
<td>2</td>
<td>u</td>
<td>l mum$</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>l mumundumulmum$</td>
</tr>
<tr>
<td>4</td>
<td>u</td>
<td>m$</td>
</tr>
<tr>
<td>5</td>
<td>u</td>
<td>mulmum$</td>
</tr>
<tr>
<td>6</td>
<td>u</td>
<td>mulmumundumulmum$</td>
</tr>
<tr>
<td>7</td>
<td>l</td>
<td>mum$</td>
</tr>
<tr>
<td>8</td>
<td>l</td>
<td>mumundumulmum$</td>
</tr>
<tr>
<td>9</td>
<td>u</td>
<td>ndumulmum$</td>
</tr>
<tr>
<td>10</td>
<td>m</td>
<td>ulmum$</td>
</tr>
<tr>
<td>11</td>
<td>m</td>
<td>ulmumundumulmum$</td>
</tr>
<tr>
<td>12</td>
<td>m</td>
<td>um$</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>umulmum$</td>
</tr>
<tr>
<td>14</td>
<td>$$$</td>
<td>umulmumundumulmum$</td>
</tr>
<tr>
<td>15</td>
<td>m</td>
<td>undumulmum$</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{T}^{\text{BWT}}$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>m</td>
<td>$$$</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>dumulmum$</td>
</tr>
<tr>
<td>2</td>
<td>u</td>
<td>l mum$</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>l mumundumulmum$</td>
</tr>
<tr>
<td>4</td>
<td>u</td>
<td>m$</td>
</tr>
<tr>
<td>5</td>
<td>u</td>
<td>mulmum$</td>
</tr>
<tr>
<td>6</td>
<td>u</td>
<td>mulmumundumulmum$</td>
</tr>
<tr>
<td>7</td>
<td>l</td>
<td>mum$</td>
</tr>
<tr>
<td>8</td>
<td>l</td>
<td>mumundumulmum$</td>
</tr>
<tr>
<td>9</td>
<td>u</td>
<td>ndumulmum$</td>
</tr>
<tr>
<td>10</td>
<td>m</td>
<td>ulmum$</td>
</tr>
<tr>
<td>11</td>
<td>m</td>
<td>ulmumundumulmum$</td>
</tr>
<tr>
<td>12</td>
<td>m</td>
<td>um$</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>umulmum$</td>
</tr>
<tr>
<td>14</td>
<td>$$$</td>
<td>umulmumundumulmum$</td>
</tr>
<tr>
<td>15</td>
<td>m</td>
<td>undumulmum$</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{T}^{\text{BWT}}$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>m</td>
<td>$$$</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>dumulmum$</td>
</tr>
<tr>
<td>2</td>
<td>u</td>
<td>l mum$</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>l mumundumulmum$</td>
</tr>
<tr>
<td>4</td>
<td>u</td>
<td>m$</td>
</tr>
<tr>
<td>5</td>
<td>u</td>
<td>mulmum$</td>
</tr>
<tr>
<td>6</td>
<td>u</td>
<td>mulmumundumulmum$</td>
</tr>
<tr>
<td>7</td>
<td>l</td>
<td>mum$</td>
</tr>
<tr>
<td>8</td>
<td>l</td>
<td>mumundumulmum$</td>
</tr>
<tr>
<td>9</td>
<td>u</td>
<td>ndumulmum$</td>
</tr>
<tr>
<td>10</td>
<td>m</td>
<td>ulmum$</td>
</tr>
<tr>
<td>11</td>
<td>m</td>
<td>ulmumundumulmum$</td>
</tr>
<tr>
<td>12</td>
<td>m</td>
<td>um$</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>umulmum$</td>
</tr>
<tr>
<td>14</td>
<td>$$$</td>
<td>umulmumundumulmum$</td>
</tr>
<tr>
<td>15</td>
<td>m</td>
<td>undumulmum$</td>
</tr>
</tbody>
</table>

(c)
Wavelet tree: rank for character sequences

(a)

(b)
Example: Compressing CSA (practical approach)

<table>
<thead>
<tr>
<th>i</th>
<th>SA</th>
<th>LF</th>
<th>T^{BWT}</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>4</td>
<td>m</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>9</td>
<td>n</td>
<td>dumulmum$</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>10</td>
<td>u</td>
<td>l mum$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>11</td>
<td>u</td>
<td>l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>12</td>
<td>u</td>
<td>m$</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>13</td>
<td>u</td>
<td>mul m um</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>14</td>
<td>u</td>
<td>mul m um l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>2</td>
<td>l</td>
<td>l mum$</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
<td>l</td>
<td>l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>15</td>
<td>u</td>
<td>n d um l mum $</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>5</td>
<td>m</td>
<td>ul m um$</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>6</td>
<td>m</td>
<td>ul m um l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>7</td>
<td>m</td>
<td>u m$</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>1</td>
<td>d</td>
<td>u mul m um</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>$</td>
<td>u mul m um l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum l mum</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>8</td>
<td>m</td>
<td>und um l mum</td>
</tr>
</tbody>
</table>

The diagram illustrates the process of compressing a string using the Burrows-Wheeler Transform (BWT) and the LCP array. The SA (sorted array) is used to index into the BWT, and the LF (left-to-right) is used to construct the final compressed string.
Example: Compressing CSA (practical approach)

SA uncompressed
\[ n \log n \text{ bits} \]

compressed

\[ \text{CSA} = \text{T}^{\text{BWT}} + \text{SA samples:} \]
\[ n \log \sigma + \mathcal{O}(n \log \sigma) \text{ bits} \]
\[ + \frac{n \log n}{s_{\text{SA}}} \text{ bits} \]

\[ s_{\text{SA}} = 3 \]

access LF[i] in time \( \mathcal{O}(\log \sigma) \)
access CSA[i] in time \( \mathcal{O}(s_{\text{SA}} \log \sigma) \)
Example: Compressing CSA (practical approach)

SA[13] =

\[ \text{SA uncompressed} \quad n \log n \text{ bits} \]

\[ \text{compressed} \]

\[ \text{CSA} = T^\text{BWT} + \text{SA samples:} \]

\[ n \log \sigma + o(n \log \sigma) \text{ bits} + \frac{n \log n}{s_{SA}} \text{ bits} \]

\[ s_{SA} = 3 \]

access LF[i] in time \( \mathcal{O}(\log \sigma) \)

access CSA[i] in time \( \mathcal{O}(s_{SA} \log \sigma) \)
Example: Compressing CSA (practical approach)


SA uncompressed
\(n \log n\) bits

compressed

CSA = \(T^{BWT} + \text{SA samples:}\)
\(n \log \sigma + o(n \log \sigma)\) bits
\(+ \frac{n \log n}{s_{SA}}\) bits

\(s_{SA} = 3\)
access LF[i] in time \(O(\log \sigma)\)
access CSA[i] in time \(O(s_{SA} \log \sigma)\)
Example: Compressing CSA (practical approach)


SA uncompressed
\( n \log n \) bits

Compressed

\[
\text{CSA} = \text{T}^{\text{BWT}} + \text{SA samples}: \\
\quad n \log \sigma + o(n \log \sigma) \text{ bits} \\
\quad + \frac{n \log n}{s_{\text{SA}}} \text{ bits}
\]

\( s_{\text{SA}} = 3 \)

Access LF[i] in time \( \mathcal{O}(\log \sigma) \)

Access CSA[i] in time \( \mathcal{O}(s_{\text{SA}} \log \sigma) \)
Example: Compressing CSA (practical approach)

SA[13]=6+2=8

\[ SA \text{ uncompressed} \]
\[ n \log n \text{ bits} \]

\[ CSA = T^{BW^T} + \text{SA samples:} \]
\[ n \log \sigma + o(n \log \sigma) \text{ bits} \]
\[ + \frac{n \log n}{s_{SA}} \text{ bits} \]

\[ s_{SA} = 3 \]
access LF[i] in time \( O(\log \sigma) \)
access CSA[i] in time \( O(s_{SA} \log \sigma) \)
## Overview of LCP data structures

<table>
<thead>
<tr>
<th>data structure</th>
<th>uses</th>
<th>access</th>
<th>memory in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>lcp_uncompressed</td>
<td>-</td>
<td>$O(1)$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>lcp_support_sada</td>
<td>CSA</td>
<td>$O(t_{SA})$</td>
<td>$2n + o(n)$</td>
</tr>
<tr>
<td>lcp_kurtz</td>
<td>-</td>
<td>$O(\log n)$ or $O(1)$</td>
<td>$8n_1 + 2n_2 \log n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>lcp_support_tree</td>
<td>NAV</td>
<td>$O(\log \sigma')$</td>
<td>$H_0 q_1 + q_2 \log n$</td>
</tr>
<tr>
<td>lcp_support_tree2</td>
<td>NAV &amp; LF</td>
<td>$O(s_{LCP} \log \sigma')$</td>
<td>$H_0 q_1 + (q_2 \log n)/s_{LCP}$</td>
</tr>
</tbody>
</table>

with $n_1 + n_2 = n$

and $q_1 + q_2 = q < n$

$q$ number of inner nodes of the ST
Runtime for random access to the LCP array (1)

The graphs show the runtime (in nanoseconds per operation) for accessing the LCP array for two datasets: `dblp.xml.200MB` and `proteins.200MB`. The x-axis represents memory in bits per character, while the y-axis represents time in nanoseconds per operation. The graphs compare the performance of different LCP representations:

- `lcp_uncompressed`
- `lcp_wt`
- `lcp_dac`
- `lcp_kurtz`
- `lcp_support_sada`
- `lcp_support_tree`
- `lcp_support_tree2`
Runtime for random access to the LCP array (2)

- DNA 200MB
  - Memory in bits per character
  - Time in nanoseconds per operation

- Rand_k128 200MB
  - Memory in bits per character
  - Time in nanoseconds per operation
Runtime for random access to the LCP array (3)

```
<table>
<thead>
<tr>
<th>Memory in bits per character</th>
<th>Time in nanoseconds per operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>english.200MB</td>
<td></td>
</tr>
<tr>
<td>sources.200MB</td>
<td></td>
</tr>
</tbody>
</table>

- lcp_uncompressed
- lcp_wt
- lcp_dac
- lcp_kurtz
- lcp_support_sada
- lcp_support_tree
- lcp_support_tree2
```

**Graphs:**

- english.200MB
- sources.200MB

**Legend:**

- lcp_uncompressed
- lcp_wt
- lcp_dac
- lcp_kurtz
- lcp_support_sada
- lcp_support_tree
- lcp_support_tree2
Outline

1. Introduction
   - Basic data structures
   - The suffix tree

2. CST design
   - NAV (tree topology and navigation)
   - CSA (lexicographic information)
   - LCP (longest common prefixes)

3. CST in practice
   - The sdsl library
The succinct data structure library *sdsl*

- Provides basic and advanced succinct data structures
- Easy to use (very similar to C++ STL)
- Fast and space-efficient construction of data structures
- 64-bit implementation
- Well-optimized implementation (e.g. now using hardware POPCOUNT operation, ..)
- Easy configuration of myriads of CSAs, CSTs with many time-space trade-offs
- Fast prototyping of other complex succinct data structures
CST space in practice (for English.200MB)

CST: 540.2 MB

- CSA: 180.8 MB
  - wt: 148 MB
    - data: 114 MB
    - rank: 7.1 MB
    - select 1: 14.4 MB
    - select 0: 12.5 MB
  - isa_sample: 10.9 MB
- lcp: 215 MB
  - lcp values: 170.4 MB
  - overflow mark: 41.9 MB
  - rank: 2.6 MB
- nav: 144.4 MB
  - BPS$_{dfs}$ (bit_vector): 80.9 MB
  - bp_support: 37.1 MB
    - small block: 5.7 MB
    - medium block: 1.6 MB
    - bp rank: 20.2 MB
    - bp select: 9.6 MB
  - rank_support10: 20.2 MB
  - select_support10: 6.1 MB

- cst_sada<csa_wt<wt_huff<> >,lcp_dac<> >
CST space in practice (for English.200MB)

CSA: 180.8 MB
wt: 148 MB
data: 114 MB
rank: 7.1 MB
select 1: 14.4 MB
select 0: 12.5 MB
sa_sample: 21.9 MB
isa_sample: 10.9 MB
lcp: 96.1 MB
small lcp: 72.1 MB
data: 67.9 MB
rank: 4.2 MB
big lcp: 24 MB
nav: 129.2 MB
bp: 80.9 MB
bp_support: 21.9 MB
small block: 5.7 MB
medium block: 1.6 MB
bp rank: 5.1 MB
bp select: 9.6 MB
rank_support10: 20.2 MB
select_support10: 6.1 MB

cst_sada<csa_wt<wt_huff<> >,lcp_support_tree2<> >
CST space in practice (for english.200MB)

CST: 108.4 MB
wt: 75.6 MB
data: 75.6 MB
bt: 30.4 MB
btnr: 31.4 MB
btnrp: 6.7 MB
rank samples: 6.9 MB
invert: 0.2 MB
rank: 0 MB
select 1: 0 MB
select 0: 0 MB
sa_sample: 21.9 MB
isa_sample: 10.9 MB
lcp: 96.1 MB
small lcp: 72.1 MB
data: 67.9 MB
rank: 4.2 MB
big lcp: 24 MB
nav: 129.2 MB
BPSdfs(bit_vector): 80.9 MB
bp_support: 21.9 MB
small block: 5.7 MB
medium block: 1.6 MB
bp rank: 5.1 MB
bp select: 9.6 MB
rank_support10: 20.2 MB
select_support10: 6.1 MB

cst_sada<csa_wt<wt_huff<rrr_vector<> > >,
lcp_support_tree2<> >
CST space in practice (for English 200MB)

```
cst_sct3<csa_wt<wt_huff<rrr_vector<> >,
lcp_support_tree2<> >
```
Experimental setup

\[0 \triangleq \text{cst}_sada<\text{csa}_sada>, \text{lcp}_d\text{ac}> \]
\[1 \triangleq \text{cst}_sada<\text{csa}_sada>, \text{lcp}_\text{support}_\text{tree}2> \]
\[2 \triangleq \text{cst}_sada<\text{csa}_\text{wt}>, \text{lcp}_d\text{ac}> \]
\[3 \triangleq \text{cst}_sada<\text{csa}_\text{wt}>, \text{lcp}_\text{support}_\text{tree}2> \]
\[4 \triangleq \text{cst}_\text{sct}3<\text{csa}_sada>, \text{lcp}_d\text{ac}> \]
\[5 \triangleq \text{cst}_\text{sct}3<\text{csa}_sada>, \text{lcp}_\text{support}_\text{tree}2> \]
\[6 \triangleq \text{cst}_\text{sct}3<\text{csa}_\text{wt}>, \text{lcp}_d\text{ac}> \]
\[7 \triangleq \text{cst}_\text{sct}3<\text{csa}_\text{wt}>, \text{lcp}_\text{support}_\text{tree}2> \]

The same basic data structures are used, i.e. its a very fair comparison
Runtime of operations of \texttt{cst\_sada}

- \texttt{csa[i]}:  
- \texttt{psi[i]}:  
- \texttt{lcp[i]}:  
- \texttt{id(v)}:  
- \texttt{depth(v)}:  
- \texttt{depth(v)^*}:  
- \texttt{parent(v)}:  
- \texttt{Sibling(v)}:  
- \texttt{sl(v)}:  
- \texttt{child(v, c)}:  
- \texttt{select\_child(v, 1)}:  
- \texttt{lca(v, w)}:  
- \texttt{dfs and id(v)}:  
- \texttt{dfs and depth(v)}:  
- \texttt{mstats}:  

- \texttt{dfs and depth(v)}:

- \texttt{mstats}:

- \texttt{dfs and depth(v)}:
Runtime of operations of \texttt{cst\_sct3}
Time-space trade-off for \texttt{select\_child}(v, 1)

```
<table>
<thead>
<tr>
<th>Memory in bits per character</th>
<th>Time in nanoseconds per operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>200</td>
</tr>
</tbody>
</table>
```

\texttt{english.200MB}

\texttt{sources.200MB}
Time-space trade-off for \textit{child}(v, c)

- **english.200MB**
  - Time in nanoseconds per operation
  - Memory in bits per character

- **sources.200MB**
  - Time in nanoseconds per operation
  - Memory in bits per character
Construction of a CST

```cpp
#include <sdsl/suffixtrees.hpp>
#include <sdsl/util.hpp>

using namespace sdsl;

typedef cst_sct3<> tCST;

int main(int argc, char* argv[]){
    tCST cst;
    construct_cst( argv[1], cst );
}```
Runtime for construction (prefixes of English text)

- cstV
- ST
- cst_sada (0)
- cst_sct3 (4)

Text size in MB

Time in seconds

0 100 200 300 400 500

0 10 100 1000 10000
CST construction – resources comparison

**Text**: `english.200MB`

**σ**: 226

**CST type**: cst_sada

**CST construction**

- **First CST implementation (2007)**
- **sds1 (2010)**

**Memory in bytes per character**

- **Time in seconds**
- **LCP type**: lcp_sada
- **CSA type**: csa_wt
Detailed resources for the construction of CSTs

- CST type: cst_sada
- CSA type: csa_wt

- CST type: cst_sada
- CSA type: csa_wt

- CST type: cst_sct3
- CSA type: csa_wt

Memory in bytes per input character:
- SA: 57.4
- CSA: 108.3
- LCP: 67.2
- NAV: 9.9
- ISA: 33.9
- BWT: 29.6
- wt: 9.9

Time in seconds:
- 0 4 8 0 4 8 4 0 8
- 200
- 300
- 400

bp_support: 24.3
NAV: 141.5
ISA: 33.9
T_BWT: 29.5
wt: 9.9
LCP: 98.8
CSA: 106.8
SA: 56.7
T_BWT: 29.4
SA: 57.4
bp_support: 23.5
NAV: 144.6
ISA: 33.9
T_BWT: 29.6
wt: 9.9
LCP: 65.9
CSA: 108.5
SA: 56.5
T_BWT: 29.4
SA: 57.4
Depth first search traversal in a CST

```cpp
template<class Cst>
void test_cst_dfs_iterator_and_depth(Cst &cst) {
    typedef typename Cst::const_iterator iterator;
    long long cnt = 0;
    for (iterator it = cst.begin(); it != cst.end(); ++it) {
        if (!cst.is_leaf(*it))
            cnt += cst.depth(*it);
    }
    cout << cnt << endl;
}
```
Runtime of dfs on CST (prefixes of English text)
Conclusion

CSTs ...

- can be build fast and space-efficient
- provide a rich set of functionality
  - fast operations: basic navigation, access LF or $\psi$
  - slow operations: $child(v, c)$, access to SA

You can use the sdsi library to configure a CSTs which fits your needs
Thank you!
Runtime of \texttt{int\_vector} access

Time in nanoseconds per operation

- \texttt{int\_vector<32>}
- \texttt{bit\_vector}
- \texttt{int\_vector<> v(\ldots, 32)}
- \texttt{int\_vector<> v(\ldots, 27)}

- \textcolor{green}{random access read}
- \textcolor{green}{random access write}
- \textcolor{blue}{sequential write}
Operation runtime of basic data structures

- `int_vector<32>`
- `rank_support_v`
- `rank_support_v5`
- `select_support_mcl`

(random access)
Runtime of CSA access (1)

---

<table>
<thead>
<tr>
<th>Memory in bits per character</th>
<th>Time in nanoseconds per operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10^6</td>
</tr>
<tr>
<td>6</td>
<td>10^5</td>
</tr>
<tr>
<td>16</td>
<td>10^4</td>
</tr>
<tr>
<td>24</td>
<td>10^3</td>
</tr>
<tr>
<td>32</td>
<td>10^2</td>
</tr>
</tbody>
</table>

**dblpxml.200MB**

**proteins.200MB**
Runtime of CSA access (2)

---

**dna.200MB**

- Time in nanoseconds per operation

**rand_k128.200MB**

- Memory in bits per character

---

<table>
<thead>
<tr>
<th>Memory in bits per character</th>
<th>Time in nanoseconds per operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10^6</td>
</tr>
<tr>
<td>6</td>
<td>10^5</td>
</tr>
<tr>
<td>16</td>
<td>10^4</td>
</tr>
<tr>
<td>24</td>
<td>10^3</td>
</tr>
<tr>
<td>32</td>
<td>10^2</td>
</tr>
</tbody>
</table>

---

Operational modes:
- csa_wt<wt<>>
- csa_wt<wt_huff<>>
- csa_wt<wt_rlnm<>>
- csa_wt<wt_rlg<8>>
- csa_sada<δ>
- csa_sada<Φ>
Runtime of CSA access (3)

- **english.200MB**
- **sources.200MB**

<table>
<thead>
<tr>
<th>Memory in bits per character</th>
<th>Time in nanoseconds per operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10^6</td>
</tr>
<tr>
<td>6</td>
<td>10^5</td>
</tr>
<tr>
<td>16</td>
<td>10^4</td>
</tr>
<tr>
<td>24</td>
<td>10^3</td>
</tr>
<tr>
<td>32</td>
<td>10^2</td>
</tr>
</tbody>
</table>

- `csa_wt<wt<>>`
- `csa_wt<wt_huff<>>`
- `csa_wt<wt_rlmn<>>`
- `csa_wt<wt_rlg<8>>`
- `csa_sada<δ>`
- `csa_sada<Φ>`
Runtime of CSA operations (1)

\[ \circ = \psi[i], \quad \triangle = \psi(i) = LF[i], \quad + = bwt[i] \]
Runtime of CSA operations (2)

\[ o = \psi[i], \ \triangle = \psi(i) = LF[i], \ + = bwt[i] \]
Runtime of CSA operations (3)

\[ \circ = \psi[i], \quad \triangle = \psi(i) = LF[i], \quad + = bwt[i] \]