Linear Work Suffix Array Construction
Juha Karkkainen, Peter Sanders, Stefan Burkhardt

Presented by Roshni Sahoo

March 7, 2019
Outline

1. Introduction
2. Related Work
3. DC3 Algorithm Example
4. Generalized Difference Cover Algorithm
5. Advanced Models of Computation
6. Final Remarks
Outline

1. Introduction
2. Related Work
3. DC3 Algorithm Example
4. Generalized Difference Cover Algorithm
5. Advanced Models of Computation
6. Final Remarks
A suffix tree of a string $S$ is compacted trie of all the suffixes of $S$.

Suffix trees have explicit structure and a direct linear-time construction algorithm (Farach’s algorithm).

Applications: Locating a substring $P$ in $S$ in $O(|P|)$ time.
A suffix array is the lexicographically sorted array of the suffixes of a string.

Suffix arrays have a more implicit form and are simpler and more compact than suffix trees. In practice, they use three to five times less space.

Applications: Locating a substring $P$ in a string $S$ in $O(|P| + \log |S|)$ time.
Querying for a Substring in a Suffix Array

Assume that we have constructed a suffix array for a string $S$. We are searching for a substring $P$ in $S$.

- Naive algorithm: Binary search the suffix array for the substring. Each comparison between the substring and an element of the array takes $O(|P|)$ time and the binary search takes $O(\log |S|)$ time to complete. \( \rightarrow O(|P| \log |S|) \).

- When we construct a suffix array, we can also construct a *longest common prefixes* (LCP) array. We can use the LCP array to augment a classic binary search yielding $O(|P| + \log |S|)$-time algorithm.
Contributions

- Goal: Find a direct linear-time suffix array construction algorithm.
- Bridge theory and practice by finding a linear-time construction algorithm for a data structure that practitioners prefer.
1. Introduction

2. Related Work

3. DC3 Algorithm Example

4. Generalized Difference Cover Algorithm

5. Advanced Models of Computation

6. Final Remarks
Farach’s Algorithm

- A linear-time suffix tree construction algorithm for integer alphabet.
- Algorithm:
  1. Recursively compute the suffix tree of the suffixes starting at odd positions.
  2. Next, compute the suffix tree of the suffixes starting at even positions based on the results of the first step.
  3. Finally, merge the even and odd suffix trees together.
Outline

1. Introduction
2. Related Work
3. DC3 Algorithm Example
4. Generalized Difference Cover Algorithm
5. Advanced Models of Computation
6. Final Remarks
1. Construct the suffix array of a sample of the suffixes. In the sample, we include the suffixes starting at positions $i \mod 3 \neq 0$. We recursively find the suffix array of a string of two-thirds length of the original string.

2. Construct the suffix array of the remaining suffixes using the result of the first step.

3. Merge the two suffix arrays into one using comparison-based merging.
Given a string $T = yabbadabbado$, we construct suffix array $SA = [12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0]$. 

For $k = 0, 1, 2$, we can define sets of indices $B_k = \{i \in [0, n] \mid i \mod 3 = k\}$

$B_0 = \{0, 3, 6, 9, 12\}$,
$B_1 = \{1, 4, 7, 10\}$,
$B_2 = \{2, 5, 8, 11\}$.

Let $S_i$ denote a suffix starting at index $i$ in $T$. Let $C = B_1 \cup B_2$ be the set of sample start indices and $S_C$ is the set of sample suffixes.

$C = \{1, 4, 7, 10, 2, 5, 8, 11\}$

$S_C = \{S_1, S_4, S_7, ..., S_8, S_{11}\}$. 

Presented by Roshni Sahoo
DC3 Step 1: Construct the sample.

Given a string $T = yabbadabbado$, we construct suffix array $SA = [12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0]$.

1. For $k = 0, 1, 2$, we can define sets of indices

   $$B_k = \{ i \in [0, n] | i \mod 3 = k \}$$

Which indices do $B_0$, $B_1$, and $B_2$ contain?
Given a string $T = yabbadabbado$, we construct suffix array $SA = [12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0]$.

1. For $k = 0, 1, 2$, we can define sets of indices

$$B_k = \{i \in [0, n]| i \mod 3 = k\}$$

Which indices do $B_0$, $B_1$, and $B_2$ contain?

- $B_0 = \{0, 3, 6, 9, 12\}$, $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$.
DC3 Step 1: Construct the sample.

Given a string \( T = yabbadabbado \), we construct suffix array \( SA = [12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0] \).

1. For \( k = 0, 1, 2 \), we can define sets of indices

\[
B_k = \{ i \in [0, n] | i \mod 3 = k \}
\]

Which indices do \( B_0 \), \( B_1 \), and \( B_2 \) contain?

- \( B_0 = \{0, 3, 6, 9, 12\} \), \( B_1 = \{1, 4, 7, 10\} \), \( B_2 = \{2, 5, 8, 11\} \).

Let \( S_i \) denote a suffix starting at index \( i \) in \( T \). Let \( C = B_1 \cup B_2 \) be the set of sample start indices and \( S_C \) is the set of sample suffixes.

\[
C = \{1, 4, 7, 10, 2, 5, 8, 11\}
\]

\[
S_C = \{S_1, S_4, S_7...S_8, S_{11}\}.
\]
Recall that $T = yabbadabbado$

2. Construct a new string $R$ to sort the sample suffixes. Let $t_i$ be the $i$-th element of $T$. For $k = 1, 2$, we can construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\text{max}B_k} t_{\text{max}B_k+1} t_{\text{max}B_k+2}].$$

What do $R_1$ and $R_2$ look like?

$$R_1 = [abb][ada][bba][do0] \text{ and } R_2 = [bba][dab][bad][o00].$$
Recall that $T = yabbadabbado$

2. Construct a new string $R$ to sort the sample suffixes.

Let $t_i$ be the $i$-th element of $T$. For $k = 1, 2$, we can construct the strings

$$R_k = [t_k \ t_{k+1} \ t_{k+2}] [t_{k+3} \ t_{k+4} \ t_{k+5}] \cdots [t_{\max B_k} \ t_{\max B_k+1} \ t_{\max B_k+2}] .$$

What do $R_1$ and $R_2$ look like?

$$R_1 = [abb][ada][bba][do0] \text{ and } R_2 = [bba][dab][bad][o00].$$
Recall that \( T = yabbadabbado \)

2. Construct a new string \( R \) to sort the sample suffixes.
   Let \( t_i \) be the \( i \)-th element of \( T \). For \( k = 1, 2 \), we can construct the strings
   \[
   R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\text{max}B_k} t_{\text{max}B_k+1} t_{\text{max}B_k+2}].
   \]
   What do \( R_1 \) and \( R_2 \) look like?

   \[
   R_1 = [abb][ada][bba][do0] \text{ and } R_2 = [bba][dab][bad][o00].
   \]
DC3 Step 2: Construct $R$ to sort sample suffixes.

Recall that $T = yabbadabbado$

2. Construct a new string $R$ to sort the sample suffixes.
   Let $t_i$ be the $i$-th element of $T$. For $k = 1, 2$, we can construct the strings
   \[ R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\max B_k} t_{\max B_k+1} t_{\max B_k+2}] \]

   What do $R_1$ and $R_2$ look like?

   \[ R_1 = [abb][ada][bba][do0] \text{ and } R_2 = [bba][dab][bad][o00]. \]

   We can concatenate $R_1$ and $R_2$ into a string $R$.
   \[ R = [abb][ada][bba][do0][bba][dab][bad][o00] \]
DC3 Step 2: Construct $R$ to sort sample suffixes.

Recall that $T = \textit{yabbadabbado}$

2. Construct a new string $R$ to sort the sample suffixes.

Let $t_i$ be the $i$-th element of $T$. For $k = 1, 2$, we can construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\max B_k} t_{\max B_k+1} t_{\max B_k+2}] .$$

What do $R_1$ and $R_2$ look like?

$$R_1 = [\textit{abb}] [\textit{ada}] [\textit{bba}] [\textit{do0}]$$
$$R_2 = [\textit{bba}] [\textit{dab}] [\textit{bad}] [\textit{o00}] .$$

We can concatenate $R_1$ and $R_2$ into a string $R$.

$$R = [\textit{abb}] [\textit{ada}] [\textit{bba}] [\textit{do0}] [\textit{bba}] [\textit{dab}] [\textit{bad}] [\textit{o00}]$$

The nonempty suffixes of $R$ correspond to $S_C$ of sample suffixes. By sorting the suffixes of $R$, we get the order of the sample suffixes $S_C$. 

Presented by Roshni Sahoo
DC3 Step 3: Sort the characters of $R$

Recall that $T = \text{yabbadabbado}$.

3. Sort the suffixes of $R$. First, radix sort the characters of $R$ (the triples $[t_it_{i+1}t_{i+2}]$) and rename them with their ranks to obtain a new string $R'$.

$$R = [abb][ada][bba][do0][bba][dab][bad][o00]$$

<table>
<thead>
<tr>
<th>Rank</th>
<th>Index in $R$</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>abb</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>ada</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>bad</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>bba</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>bba</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>dab</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>do0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>o00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index in $R$</th>
<th>$R'$ (Rank)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
Recall that $T = yabbadabbado$, and $R' = [1, 2, 4, 6, 4, 5, 3, 7]$. 

4. If any of the characters of $R$ are the same, recursively sort the suffixes of $R'$. 

<table>
<thead>
<tr>
<th>Rank</th>
<th>Index in R'</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>$$$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>12464537$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2464537$</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>37$</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4537$</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>464537$</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>537$</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>64537$</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>7$</td>
</tr>
</tbody>
</table>
DC3 Step 4: Sort the suffixes of $R'$ (if needed).

Recall that $T =$ \textit{yabbadabbado}, and $R' = [1, 2, 4, 6, 4, 5, 3, 7]$.

4. If any of the characters of $R$ are the same, recursively sort the suffixes of $R'$.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Index in $R'$</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>$$$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>12464537$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2464537$</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>37$</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4537$</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>464537$</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>537$</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>64537$</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>7$</td>
</tr>
</tbody>
</table>

But how does this relate to the suffixes of the original string $T$?
DC3 Step 4: Sort the suffixes of $R'$ (if needed).

We can write the correspondence between start indices of the suffixes $R'$ to the start indices of $T$.

\[ T = y \ a \ b \ b \ a \ d \ a \ b \ b \ a \ d \ o \]
\[ R = [a \ b \ b \] \ [a \ d \ a \] \ [b \ b \ a \] \ [d \ o \ 0] \ldots \]

<table>
<thead>
<tr>
<th>Start Index of Suffix in $R'$</th>
<th>Start Index of Suffix in $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>
5. Combining the results in the last two tables, we see that we can assign a rank to each suffix in $S_C$. 

<table>
<thead>
<tr>
<th>Suffix Start Index in $T$</th>
<th>Rank of Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>x</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
</tbody>
</table>
DC3 Step 6: Sort non-sample suffixes

6. The non-sample suffixes are the suffixes with start indices in $B_0$. We represent each of these suffixes $S_i$ by a tuple, $(t_i, \text{rank}(S_{i+1}))$.

<table>
<thead>
<tr>
<th>Start Index of Suffix in T</th>
<th>Tuple Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(y, 1)</td>
</tr>
<tr>
<td>3</td>
<td>(b, 2)</td>
</tr>
<tr>
<td>6</td>
<td>(a, 5)</td>
</tr>
<tr>
<td>9</td>
<td>(a, 7)</td>
</tr>
<tr>
<td>12</td>
<td>(0, 0)</td>
</tr>
</tbody>
</table>

We can compare these suffixes as follows

$$S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).$$

Radix-sorting the tuples gives us an ordering of the non-sample suffixes. What is the sorted order of these suffixes?
DC3 Step 6: Sort non-sample suffixes

6. The non-sample suffixes are the suffixes with start indices in \( B_0 \). We represent each of these suffixes \( S_i \) by a tuple, \((t_i, \text{rank}(S_{i+1}))\).

<table>
<thead>
<tr>
<th>Start Index of Suffix in ( T )</th>
<th>Tuple Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((y, 1))</td>
</tr>
<tr>
<td>3</td>
<td>((b, 2))</td>
</tr>
<tr>
<td>6</td>
<td>((a, 5))</td>
</tr>
<tr>
<td>9</td>
<td>((a, 7))</td>
</tr>
<tr>
<td>12</td>
<td>((0, 0))</td>
</tr>
</tbody>
</table>

We can compare these suffixes as follows

\[
S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).
\]

Radix-sorting the tuples gives us an ordering of the non-sample suffixes. What is the sorted order of these suffixes? \( S_{12}, S_6, S_9, S_3, S_0 \).
7. We can merge the two sorted sets of suffixes using standard comparison-based merging.

To compare a suffix $S_i \in B_1$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).$$

To compare a suffix $S_i \in B_2$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2})).$$
7. We can merge the two sorted sets of suffixes using standard comparison-based merging.

To compare a suffix $S_i \in B_1$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).$$

To compare a suffix $S_i \in B_2$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2})).$$

Final Suffix Array: [12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0].
7. We can merge the two sorted sets of suffixes using standard comparison-based merging.
To compare a suffix $S_i \in B_1$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).$$

To compare a suffix $S_i \in B_2$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2})).$$

Final Suffix Array: [12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0].
What’s the recurrence for this algorithm?
7. We can merge the two sorted sets of suffixes using standard comparison-based merging.
To compare a suffix $S_i \in B_1$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).$$

To compare a suffix $S_i \in B_2$ with $S_j \in B_0$,

$$S_i \leq S_j \iff (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2})).$$

Final Suffix Array: [12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0].
What’s the recurrence for this algorithm?

$$T(n) = T\left(\frac{2n}{3}\right) + O(n).$$
A difference cover $D_v \mod v$ is a subset of $[0, v)$ such that all values in $[0, v)$ can be expressed as a difference of two elements in $D_v \mod v$. In other words,

$$[0, v) = \{i - j \mod v | i, j \in D_v\}.$$

Example: Show that $1, 2, 4 = D_7$.

In general, we want the smallest possible difference cover for a given $v$. For any $v$, there exist a difference cover $D_v$ of size $O(\sqrt{v})$. 

---

Definition
Generalized Algorithm and Lightweight Algorithm

- **Generalized:** Instead of using a difference cover mod 3, we can use any difference cover \( D \mod v \).
- **Merge step is different in the generalized version:** we sort the suffixes by the first \( v \) characters, then use a comparison based merge.
- **Lightweight:** The generalized DC algorithm can be implemented in \( O(n/\sqrt{v}) \) space in addition to the input and output and takes \( O(vn) \)-time.
Outline

1 Introduction
2 Related Work
3 DC3 Algorithm Example
4 Generalized Difference Cover Algorithm
5 Advanced Models of Computation
6 Final Remarks
External Memory: The complexity is governed by the complexity of the integer sort. $O\left(\frac{n}{DB} \log \frac{M}{B} \frac{n}{B}\right)$.

Cache-Oblivious: The number of cache faults, $O\left(\frac{n}{B} \log \frac{M}{B} \frac{n}{B}\right)$, is a corollary of the optimal comparison based sorting algorithm.
Final Remarks

- DC3 Algorithm was very well-explained; it was very useful to have an example to understand the intricacies of the algorithm.
- The authors provided their source code at the end of the article, which is useful so that readers can replicate their results.
- The authors mention that there are already experiments with an external memory implementation and a parallel implementation, which show excellent performance. However, it would have been useful to have more empirical data in the article.
- The paper lacked a detailed explanation of the lightweight algorithm. It would have been useful if the authors provided more justification for each step of the algorithm.