Advanced Data Structures

Lecture 06: Suffix Arrays and String B-Trees

Florian Kurpicz
External Memory Model [AV88]

Definition: External Memory Model

- internal memory of $M$ words
- instances of size $N \gg M$
- unlimited external memory
- transfer blocks of size $B$ between memories

- measure number of blocks I/Os
- scanning $N$ elements: $\Theta(N/B)$
- sorting $N$ elements: $\Theta\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$
External Memory Model \[AV88\]

**Definition: External Memory Model**
- internal memory of \( M \) words
- instances of size \( N \gg M \)
- unlimited external memory
- transfer blocks of size \( B \) between memories
- measure number of blocks I/Os
- scanning \( N \) elements: \( \Theta(N/B) \)
- sorting \( N \) elements: \( \Theta(\frac{N}{B} \log_{\frac{M}{B}} N) \)

**Set of Strings**
- alphabet \( \Sigma \) of size \( \sigma \)
- \( k \) strings \( \{s_1, \ldots, s_k\} \) over the alphabet \( \Sigma \)
- total size of strings is \( N = \sum_{i=1}^{k} |s_i| \)
- queries ask for pattern \( P \) of length \( m \)
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:
- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:

- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$

**Definition: Trie**

Given a set $S = \{S_1, \ldots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. $k$ leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is $S_i$
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:
- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$

**Definition: Trie**
Given a set $S = \{S_1, \ldots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. $k$ leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is $S_i$
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique

$S = \{\text{bear, bee, cab, car}\}$
## Theoretical Comparison

<table>
<thead>
<tr>
<th>Representation</th>
<th>Query Time (Contains)</th>
<th>Space in Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays of variable size</td>
<td>$O(m \cdot \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>arrays of fixed size</td>
<td>$O(m)$</td>
<td>$O(N \cdot \sigma)$</td>
</tr>
<tr>
<td>hash tables</td>
<td>$O(m)$ w.h.p.</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>balanced search trees</td>
<td>$O(m \cdot \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>weight-balanced search trees</td>
<td>$O(m + \lg k)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>two-levels with weight-balanced search trees</td>
<td>$O(m + \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
## Theoretical Comparison

<table>
<thead>
<tr>
<th>Representation</th>
<th>Query Time (Contains)</th>
<th>Space in Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays of variable size</td>
<td>$O(m \cdot \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>arrays of fixed size</td>
<td>$O(m)$</td>
<td>$O(N \cdot \sigma)$</td>
</tr>
<tr>
<td>hash tables</td>
<td>$O(m)$ w.h.p.</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>balanced search trees</td>
<td>$O(m \cdot \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>weight-balanced search trees</td>
<td>$O(m + \lg k)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>two-levels with weight-balanced search trees</td>
<td>$O(m + \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

- more details in lecture Text Indexing
Compact Trie

- Tries have unnecessary nodes
- Branchless paths can be removed
- Edge labels can consist of multiple characters
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters

**Definition: Compact Trie**

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges’ labels.
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters

**Definition: Compact Trie**

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges’ labels.
Suffix Array and LCP-Array

**Definition: Suffix Array** [GBS92; MM93]

Given a text $T$ of length $n$, the **suffix array** (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$
Suffix Array and LCP-Array

**Definition: Suffix Array** [GBS92; MM93]

Given a text $T$ of length $n$, the **suffix array** (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

**Definition: Longest Common Prefix Array**

Given a text $T$ of length $n$ and its SA, the **LCP-array** is defined as

$$LCP[i] = \begin{cases} 0 & i = 1 \\ \max\{\ell : T[SA[i]..SA[i] + \ell) = T[SA[i - 1]..SA[i - 1] + \ell)\} & i \neq 1 \end{cases}$$
Suffix Array and LCP-Array

Definition: Suffix Array [GBS92; MM93]
Given a text $T$ of length $n$, the suffix array (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

Definition: Longest Common Prefix Array
Given a text $T$ of length $n$ and its SA, the LCP-array is defined as

$$LCP[i] = \begin{cases} 0 & i = 1 \\ \max\{\ell : T[SA[i]..SA[i] + \ell) = T[SA[i - 1]..SA[i - 1] + \ell]} & i \neq 1 \end{cases}$$
Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Prefix Doubling
- 1990: [MM] original
- 1999: [LS] qsufsort

Induced Copying
- 2000: [Sew] 1/2 copy
- 2002: [MF] deep-shallow
- 2003: [Sew] BWT
- 2004: [MF] A/B copy
- 2005: [SS] bpr
- 2006: [MP] cache aware
- 2007: [SS] DvSufSort
- 2008: [MP] cache aware
- 2009: [SS] bpr
- 2011: [MP] cache aware
- 2016: [Bai] GSACA
- 2017: [LLH] O(1) space
- 2021: [Got] O(1) space

Recursion
- 1990: [BW] BWT
- 1999: [IT] A/B copy
- 2000: [MF] deep-shallow
- 2002: [BK] diffcover
- 2004: [KA] L/S split
- 2005: [Man] chains
- 2006: [SS] bpr
- 2007: [MP] cache aware
- 2008: [SS] bpr
- 2009: [MP] cache aware
- 2011: [SS] bpr
- 2016: [Bai] GSACA
- 2017: [LLH] O(1) space
- 2021: [Got] O(1) space

Timeline Sequential Suffix Sorting based on [Bah+19; Bin18; Kur20; PST07] with darker grey indicating linear running time and brown indicating available implementation.
Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Special Mentions

- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Special Mentions

- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
- until 2021: DivSufSort fastest in practice with $O(n \lg n)$ running time
Timeline Sequential Suffix Sorting
- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Special Mentions
- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
- until 2021: DivSufSort fastest in practice with $O(n \lg n)$ running time
- since 2021: libSAIS fastest in practice with $O(n)$ running time
Suffix Sorting in External Memory


- using induced copying
- $O(N/B) \log^2_{M/B}(N/B)$ I/Os
Pattern Matching with the Suffix Array (1/2)

Function SeachSA(T, SA[1..n], P[1..m]):

1. \( \ell = 1, r = n + 1 \)
2. while \( \ell < r \) do
3.   \( i = \lfloor (\ell + r) / 2 \rfloor \)
4.   if \( P > T[SA[i]..SA[i] + m] \) then
5.     \( \ell = i + 1 \)
6.   else \( r = i \)
7. s = \( \ell \), \( \ell = \ell - 1 \), \( r = n \)
8. while \( \ell < r \) do
9.   \( i = \lceil (\ell + r) / 2 \rceil \)
10. if \( P = T[SA[i]..SA[i] + m] \) then \( \ell = i \)
11. else \( r = i - 1 \)
12. return \([s, r]\)

pattern \( P = abc \)
Pattern Matching with the Suffix Array (1/2)

Function SeachSA($T$, $SA[1..n]$, $P[1..m]$):

1. $\ell = 1, r = n + 1$
2. while $\ell < r$ do
   3. $i = \lfloor (\ell + r)/2 \rfloor$
   4. if $P > T[SA[i]..SA[i] + m]$ then
      5. $\ell = i + 1$
   6. else $r = i$
7. $s = \ell, \ell = \ell - 1, r = n$
8. while $\ell < r$ do
   9. $i = \lceil (\ell + r)/2 \rceil$
  10. if $P = T[SA[i]..SA[i] + m]$ then $\ell = i$
  11. else $r = i - 1$
12. return $[s, r]$

pattern $P = abc$
Pattern Matching with the Suffix Array (1/2)

Function `SearchSA(T, SA[1..n], P[1..m])`:

1. \( \ell = 1, r = n + 1 \)
2. while \( \ell < r \) do
   3.   \( i = \lfloor (\ell + r)/2 \rfloor \)
   4.   if \( P > T[SA[i]..SA[i] + m] \) then
      5.     \( \ell = i + 1 \)
   6.   else \( r = i \)
7. \( s = \ell, \ell = \ell - 1, r = n \)
8. while \( \ell < r \) do
   9.   \( i = \lceil (\ell + r)/2 \rceil \)
  10.  if \( P = T[SA[i]..SA[i] + m] \) then \( \ell = i \)
  11.  else \( r = i - 1 \)
12. return \([s, r]\)

pattern \( P = abc \)
Pattern Matching with the Suffix Array (2/2)

**Function** `SearchSA(T, SA[1..n], P[1..m])`:

1. \( \ell = 1, r = n + 1 \)
2. while \( \ell < r \) do
   3. \( i = \ceil{(\ell + r)/2} \)
   4. if \( P > T[SA[i]..SA[i] + m] \) then
      5. \( \ell = i + 1 \)
   6. else \( r = i \)
   7. \( s = \ell, \ell = \ell - 1, r = n \)
3. while \( \ell < r \) do
   4. \( i = \lfloor (\ell + r)/2 \rfloor \)
   5. if \( P = T[SA[i]..SA[i] + m] \) then \( \ell = i \)
   6. else \( r = i - 1 \)
5. return \([s, r]\)

**Lemma: Running Time `SearchSA`**

The `SearchSA` answers counting queries in \( O(m \lg n) \) time and reporting queries in \( O(m \lg n + \text{occ}) \) time.

**Proof (Sketch)**

- two binary searches on the `SA` in \( O(\lg n) \) time
Pattern Matching with the Suffix Array (2/2)

**Function** `SeachSA(T, SA[1..n], P[1..m])`:

1. \( \ell = 1, r = n + 1 \)
2. while \( \ell < r \) do
   3. \( i = \lfloor (\ell + r)/2 \rfloor \)
   4. if \( P > T[SA[i]..SA[i] + m] \) then
      5. \( \ell = i + 1 \)
   6. else \( r = i \)
   7. \( s = \ell, \ell = \ell - 1, r = n \)
3. while \( \ell < r \) do
   4. \( i = \lceil \ell + r/2 \rceil \)
   5. if \( P = T[SA[i]..SA[i] + m] \) then \( \ell = i \)
   6. else \( r = i - 1 \)
5. return \([s, r]\)

**Lemma: Running Time SeachSA**

The `SeachSA` answers counting queries in \( O(m \log n) \) time and reporting queries in \( O(m \log n + \text{occ}) \) time.

**Proof (Sketch)**

two binary searches on the `SA` in \( O(\log n) \) time
Pattern Matching with the Suffix Array (2/2)

Function `SeachSA(T, SA[1..n], P[1..m])`:

1. \( \ell = 1, r = n + 1 \)
2. **while** \( \ell < r \) **do**
   3. \( i = \lfloor (\ell + r)/2 \rfloor \)
   4. if \( P > T[SA[i]..SA[i] + m] \) then
      5. \( \ell = i + 1 \)
   6. else \( r = i \)
   7. \( s = \ell, \ell = \ell - 1, r = n \)
3. **while** \( \ell < r \) **do**
   4. \( i = \lceil \ell + r/2 \rceil \)
   5. if \( P = T[SA[i]..SA[i] + m] \) then \( \ell = i \)
   6. else \( r = i - 1 \)
5. return \([s, r] \)

**Lemma: Running Time SeachSA**

The `SeachSA` answers counting queries in \( O(m \lg n) \) time and reporting queries in \( O(m \lg n + \text{occ}) \) time.

**Proof (Sketch)**

- Two binary searches on the `SA` in \( O(\lg n) \) time
- Each comparison requires \( O(m) \) time

PINGO 11/23 2022-05-30 Florian Kurpicz | Advanced Data Structures | 06 Suffix Arrays & String B-Trees Institute of Theoretical Informatics, Algorithm Engineering
Function: `SeachSA(T, SA[1..n], P[1..m]):`

1. \( \ell = 1, r = n + 1 \)
2. while \( \ell < r \) do
   3. \( i = \left\lfloor \frac{\ell + r}{2} \right\rfloor \)
   4. if \( P > T[SA[i..SA[i] + m)] \) then
      5. \( \ell = i + 1 \)
   6. else \( r = i \)
   7. \( s = \ell, \ell = \ell - 1, r = n \)
3. while \( \ell < r \) do
   4. \( i = \left\lceil \frac{\ell + r}{2} \right\rceil \)
   5. if \( P = T[SA[i..SA[i] + m)] \) then \( \ell = i \)
   6. else \( r = i - 1 \)
5. return \([s, r] \)

Lemma: Running Time SeachSA

The SeachSA answers counting queries in \( O(m \lg n) \) time and reporting queries in \( O(m \lg n + \text{occ}) \) time.

Proof (Sketch)
- Two binary searches on the SA in \( O(\lg n) \) time
- Each comparison requires \( O(m) \) time
Pattern Matching with the Suffix Array (2/2)

```
Function SearchSA(T, SA[1..n], P[1..m]):
    \ell = 1, r = n + 1
    while \ell < r do
        i = \lfloor (\ell + r)/2 \rfloor
        if P > T[SA[i]..SA[i] + m] then
            \ell = i + 1
        else
            r = i
    s = \ell, \ell = \ell - 1, r = n
    while \ell < r do
        i = \lceil \ell + r/2 \rceil
        if P = T[SA[i]..SA[i] + m] then \ell = i
        else
            r = i - 1
    return [s, r]
```

Lemma: Running Time SearchSA

The SearchSA answers counting queries in $O(m \log n)$ time and reporting queries in $O(m \log n + occ)$ time.

Proof (Sketch)

- two binary searches on the SA in $O(\log n)$ time
- each comparison requires $O(m)$ time
- counting in $O(1)$ additional time
Function `SeachSA(T, SA[1..n], P[1..m])`:

1) \( \ell = 1, r = n + 1 \)

2) while \( \ell < r \)

   3) \( i = \lfloor (\ell + r)/2 \rfloor \)

   4) if \( P > T[SA[i]..SA[i] + m] \)

      \( \ell = i + 1 \)

   else \( r = i \)

5) \( s = \ell, \ell = \ell - 1, r = n \)

6) while \( \ell < r \)

   7) \( i = \lceil \ell + r/2 \rceil \)

   8) if \( P = T[SA[i]..SA[i] + m] \)

      \( \ell = i \)

   else \( r = i - 1 \)

9) return \([s, r]\)

Lemma: Running Time `SeachSA`

The `SeachSA` answers counting queries in \( O(m \log n) \) time and reporting queries in \( O(m \log n + \text{occ}) \) time.

Proof (Sketch)

- Two binary searches on the `SA` in \( O(\log n) \) time
- Each comparison requires \( O(m) \) time
- Counting in \( O(1) \) additional time

Pattern Matching with the Suffix Array (2/2)
Pattern Matching with the Suffix Array (2/2)

Function SeachSA(T, SA[1..n], P[1..m]):

1. \( \ell = 1, r = n + 1 \)
2. while \( \ell < r \) do
   3. \( i = \lceil (\ell + r)/2 \rceil \)
   4. if \( P > T[SA[i]..SA[i] + m] \) then
      5. \( \ell = i + 1 \)
   6. else \( r = i \)

7. \( s = \ell, \ell = \ell - 1, r = n \)
8. while \( \ell < r \) do
   9. \( i = \lceil \ell + r/2 \rceil \)
   10. if \( P = T[SA[i]..SA[i] + m] \) then \( \ell = i \)
   11. else \( r = i - 1 \)

12. return \([s, r]\)

Lemma: Running Time SeachSA

The SeachSA answers counting queries in \( O(m \lg n) \) time and reporting queries in \( O(m \lg n + \text{occ}) \) time.

Proof (Sketch)

- two binary searches on the SA in \( O(\lg n) \) time
- each comparison requires \( O(m) \) time
- counting in \( O(1) \) additional time
- reporting in \( O(\text{occ}) \) additional time

---

11/23 2022-05-30 Florian Kurpicz | Advanced Data Structures | 06 Suffix Arrays & String B-Trees

Institute of Theoretical Informatics, Algorithm Engineering
Function SeachSA(T, SA[1..n], P[1..m]):
\[
\begin{align*}
\ell &= 1, r = n + 1 \\
\text{while } \ell < r \text{ do} & \\
& \quad i = \lfloor (\ell + r)/2 \rfloor \\
& \quad \text{if } P > T[SA[i]..SA[i] + m) \text{ then} \\
& \quad \quad \ell = i + 1 \\
& \quad \text{else } r = i \\
& \quad s = \ell, \ell = \ell - 1, r = n \\
\text{while } \ell < r \text{ do} & \\
& \quad i = \lceil \ell + r/2 \rceil \\
& \quad \text{if } P = T[SA[i]..SA[i] + m) \text{ then} \\
& \quad \quad \ell = i \\
& \quad \text{else } r = i - 1 \\
\text{return } [s, r]
\end{align*}
\]

Lemma: Running Time SeachSA

The SeachSA answers counting queries in \(O(m \lg n)\) time and reporting queries in \(O(m \lg n + \text{occ})\) time.

Proof (Sketch)

- two binary searches on the SA in \(O(\lg n)\) time
- each comparison requires \(O(m)\) time
- counting in \(O(1)\) additional time
- reporting in \(O(\text{occ})\) additional time
Function SeachSA(T, SA[1..n], P[1..m]):
1 \( \ell = 1, r = n + 1 \)
2 while \( \ell < r \) do
3 \( i = \lceil (\ell + r) / 2 \rceil \)
4 if \( P > T[SA[i]..SA[i] + m] \) then
5 \( \ell = i + 1 \)
6 else \( r = i \)
7 \( s = \ell, \ell = \ell - 1, r = n \)
8 while \( \ell < r \) do
9 \( i = \lfloor (\ell + r) / 2 \rfloor \)
10 if \( P = T[SA[i]..SA[i] + m] \) then \( \ell = i \)
11 else \( r = i - 1 \)
12 return \([s, r]\)

Lemma: Running Time SeachSA
The SeachSA answers counting queries in \( O(m \lg n) \) time and reporting queries in \( O(m \lg n + \text{occ}) \) time.

Proof (Sketch)
- two binary searches on the SA in \( O(\lg n) \) time
- each comparison requires \( O(m) \) time
- counting in \( O(1) \) additional time
- reporting in \( O(\text{occ}) \) additional time

how can this be improved?
Speeding Up Pattern Matching with the LCP-Array (1/4)

- remember how many characters of the pattern and suffix match
- identify how long the prefix of the old and next suffix is
- do so using the LCP-array and range minimum queries

Definition: Range Minimum Queries
Given an array $A[1..m]$, a **range minimum query** for a range $\ell \leq r \in [1, n)$ returns

$$RMQ_A(\ell, r) = \arg \min \{A[k] : k \in [\ell, r]\}$$

- $lcp(i, j) = \max\{k : T[i..i + k)\}$
- $lcp(i, j) = T[j..j + k) = LCP[RMQ_{LCP}(i + 1, j)]$
- RMQs can be answered in $O(1)$ time and require $O(n)$ space
during binary search matched
- $\lambda$ characters with left border $\ell$ and $\rho$ characters with right border $r$
- w.l.o.g. let $\lambda \geq \rho$

middle position $i$
- decide if continue in $[\ell, i]$ or $[i, r]$

let $\xi = lcp(SA[\ell], SA[i]) \in O(1)$ time with RMQs
Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = \text{lcp}(SA[\ell], SA[i])$

\[
\begin{array}{|c|c|c|}
\hline
\ell & i & r \\
\hline
\vdots & \vdots & \vdots \\
\lambda & P[3] & \rho \\
\vdots & \vdots & \vdots \\
\perp & P[\lambda] & \perp \\
\hline
\end{array}
\]
Speeding Up Pattern Matching with the LCP-Array (3/4)

- Let $\xi = \text{lcp}(SA[\ell], SA[i])$

  - $\xi > \lambda$
    - $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
    - $\ell = i$ without character comparison

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$i$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P[2]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P[3]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P[\rho]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\lambda$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\perp$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
let $\xi = \text{lcp}(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison
- Let $\xi = \text{lcp}(SA[\ell], SA[i])$

$\xi > \lambda$

- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$i$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[1]$</td>
<td>$P[3]$</td>
<td>$\rho$</td>
</tr>
<tr>
<td>$P[2]$</td>
<td>$P[\rho]$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$\lambda$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\perp$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P[\lambda]$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\xi < \lambda$

$\xi \geq \rho$ and $P[\xi + 1] < T[SA[i] + \xi]$

$r = i$ and $\rho = \xi$ without character comparison
Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

**$\xi > \lambda$**
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

**$\xi = \lambda$**
- compare as before
Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$
- compare as before
Speeding Up Pattern Matching with the LCP-Array (3/4)

- Let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$
- compare as before
Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = \text{lcp}(SA[\ell], SA[i])$

$\xi > \lambda$
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$
- compare as before

$\xi < \lambda$
- $\xi \geq \rho$ and $P[\xi + 1] < T[SA[i] + \xi]$
- $r = i$ and $\rho = \xi$ without character comparison
Speeding Up Pattern Matching with the LCP-Array (3/4)

- let $\xi = \text{lcp}(SA[\ell], SA[i])$

$\xi > \lambda$
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$
- compare as before

$\xi < \lambda$
- $\xi \geq \rho$ and $P[\xi + 1] < T[SA[i] + \xi]$
- $r = i$ and $\rho = \xi$ without character comparison
Lemma:

Using RMQs, SeachSA answers counting queries in $O(m + \lg n)$ time and reporting queries in $O(m + \lg n + \text{occ})$ time.
Lemma:

Using RMQs, SeachSA answers counting queries in $O(m + \lg n)$ time and reporting queries in $O(m + \lg n + \text{occ})$ time.

Proof (Sketch):
- either halve the range in the suffix array ($\xi \neq \lambda$)
- or
- compare characters of the pattern (at most $m$)
(Recap) B-Trees

- search tree with out-degree in \([b, 2b]\)
- works well in external memory
- uses separators to find subtree
- can be dynamic
- who knows B-trees?

Example on the board

From Atomic Values to Strings

- strings take more time to compare
- load as few strings from disk as possible
String B-Tree [FG99]

- Strings are stored in EM
- Strings are identified by starting positions
- B-tree layout for sorted suffixes identified by position
- At least $b = \Theta(B)$ children
- Tree height $O(\log_B N)$

- Given node $v$ with children $v_0, \ldots, v_k$ with $k \in [b, 2b)$
- Inner: store separators $L(v_0), R(v_0), \ldots, L(v_k), R(v_k)$
- Leaf: store strings and link leaves

- Given node $v$
  - $L(v)$ is lexicographically smallest string at $v$
  - $R(v)$ is lexicographically largest string at $v$
Search in String B-Tree

- task: find all occurrences of pattern $P$
- two traversals of String B-Tree
- identify leftmost/rightmost occurrence
- output all strings in $O(\text{occ}/B)$

Lemma: String B-Tree

Using a String B-tree, a pattern $P$ can be found in a set of strings with total length $N$ in $O(|P|/B \log N)$ I/Os

Proof (Sketch)

- String B-Tree has height $\log_B N$
- load separators of node: $O(1)$ I/O
- load strings for binary search: $O(|P|/B)$ I/Os
- total: $O(\log_B N \cdot \log B \cdot |P|/B) = O(|P|/B \log N)$ I/Os
Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie

- for strings $S = \{S_0, \ldots, S_{k-1}\}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for $k$ strings
Patricia Trie

- for strings $S = \{ S_0, \ldots, S_{k-1} \}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for $k$ strings

Search requires two steps:
- first blind search using only trie
- blind search can result in false matches
- second a comparison with resulting string

How do Patricia tries help?
Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie

- for strings $S = \{S_0, \ldots, S_{k-1}\}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for $k$ strings

- search requires two steps
  - first **blind search** using only trie
  - blind search can result in false matches
  - second a comparison with resulting string
  - use any leaf after matching pattern
Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie
- for strings $S = \{ S_0, \ldots, S_{k-1} \}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for $k$ strings

- search requires two steps
  - first blind search using only trie
  - blind search can result in false matches
  - second a comparison with resulting string
  - use any leaf after matching pattern

How do Patricia tries help?

PINGO
in each inner node build Patricia trie for separators
if blind search finds leaf \(w\)
compute \(L = \text{lcp}(P, w)\)
let \(u\) be first node on root-to-\(w\) path with \(d \geq L\)
Improving String B-Tree with Patricia Tries (2/2)

- in each inner node build Patricia trie for separators
- if blind search finds leaf $w$
- compute $L = lcp(P, w)$
- let $u$ be first node on root-to-$w$ path with $d \geq L$

$d = L$

- find matching children $v_i$ and $v_{i+1}$ of $w$ with
- branching characters $c_i < P[L + 1] < c_{i+1}$
- example on the board 📚
in each inner node build Patricia trie for separators
if blind search finds leaf $w$
compute $L = lcp(P, w)$
let $u$ be first node on root-to-$w$ path with $d \geq L$

$d > L$
- consider next branching character $c$ on path
- if $P[L + 1] < c$ continue in leftmost leaf
- if $P[L + 1] > c$ continue in rightmost leaf

d = L
- find matching children $v_i$ and $v_{i+1}$ of $w$ with
- branching characters $c_i < P[L + 1] < c_{i+1}$
- example on the board 📚
Searching in Improved String B-Tree

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for $w$ (result of blind search)
- load one string and compare with $P$
- identify child and continue
Searching in Improved String B-Tree

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for $w$ result of blind search
- load one string and compare with $P$
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern $P$ can be found in a set of strings with total length $N$ with $O(|P|/B \log_B N)$ I/Os
Searching in Improved String B-Tree

- at every node with children $v_0$, $v_1$, ..., $v_k$
- load Patricia trie for $L(v_0)$, $L(v_1)$, ..., $L(v_k)$
- search Patricia trie for $w$ (result of blind search)
- load one string and compare with $P$
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern $P$ can be found in a set of strings with total length $N$ with $O(|P|/B \log_B N)$ I/Os

Proof (Sketch)

- loading PT: $O(1)$ I/Os
- blind search: no I/Os
- loading one string: $O(|P|/B)$ I/Os
- identify child: no I/Os
- total $O(|P|/B \log_B N)$ I/Os
Searching in Improved String B-Tree

- at every node with children \(v_0, \ldots, v_k\)
- load Patricia trie for \(L(v_0), \ldots, R(v_k)\)
- search Patricia trie for \(w\) result of blind search
- load one string and compare with \(P\)
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern \(P\) can be found in a set of strings with total length \(N\) with \(O(|P|/B \log_B N)\) I/Os

Proof (Sketch)
- loading PT: \(O(1)\) I/Os
- blind search: no I/Os
- loading one string: \(O(|P|/B)\) I/Os
- identify child: no I/Os
- total \(O(|P|/B \log_B N)\) I/Os

How can this be improved even further?

PINGO
Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree \( p_0, p_1, p_2, \ldots \)
- in Patricia tries \( PT_{p_i} \), compute \( L = lcp(P, w) \)
- all strings in \( p_i \) have prefix \( P[0..L) \)
- do not compare previously matched characters
- load only \( |P| - L \) characters at next node
- pass \( L \) down the String B-tree
Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree \( p_0, p_1, p_2, \ldots \)
- in Patricia tries \( PT_{p_i} \), compute \( L = lcp(P, w) \)
- all strings in \( p_i \) have prefix \( P[0..L) \)
- do not compare previously matched characters
- load only \( |P| - L \) characters at next node
- pass \( L \) down the String B-tree

Lemma: String B-Tree with PTs and LCP

Using a String B-tree with Patricia tries and passing down the LCP-value, a pattern \( P \) can be found in a set of strings with total length \( N \) in \( O(|P|/B + \log_B N) \) I/Os.
Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree \( p_0, p_1, p_2, \ldots \)
- in Patricia tries \( PT_p \), compute \( L = lcp(P, w) \)
- all strings in \( p_i \) have prefix \( P[0..L] \)
- do not compare previously matched characters
- load only \( |P| - L \) characters at next node
- pass \( L \) down the String B-tree

Lemma: String B-Tree with PTs and LCP

Using a String B-tree with Patricia tries and passing down the LCP-value, a pattern \( P \) can be found in a set of strings with total length \( N \) in 
\[ O(|P|/B + \log_B N) \] I/Os

Proof (Sketch)

- passing down LCP-value: no I/Os
- telescoping sum \( \sum_{i \leq h} \frac{L_i - L_{i-1}}{B} \)
- \( h = \log_B N \) \( \circ \) height of String B-tree
- \( L_i \) is LCP-value on Level \( i \)
- \( L_0 = 0 \) and \( L_h \leq |P| \)
- total: \( O(|P|/B + \log_B N) \) I/Os
Conclusion and Outlook

This Lecture
- suffix array and LCP array
- String B-tree

Advanced Data Structures

String B-tree
SA & LCP
Successor
RMQ
static/dynamic
BV
static/dynamic
succ. trees
range min-max tree
succ. graphs
Bibliography I


Bibliography II


