Suffix Trees

Katharina Pentenrieder
Introduction

Usage

- Solving many string problems in linear time

History

- String algorithms
  - Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick

- Suffix tree algorithms
  - 1973 P. Weiner: first linear construction algorithm
  - 1976 E. M. McCreight: more space-efficient algorithm
  - 1993 E. Ukkonen: conceptually different approach
Outline

1. Data Structures
   - Suffix tries and trees

2. Construction Algorithms
   - Naïve algorithm
   - Algorithms of Weiner, McCreight & Ukkonen

3. Examples of Use
   - Exact string matching problems
   - Longest common substring
   - Assembly of Strings

4. Conclusion
## Preliminaries

### Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>finite, non empty alphabet</td>
<td>$S=s_1s_2...s_n$</td>
</tr>
<tr>
<td>$\alpha$, $\beta$, $\gamma$</td>
<td>possibly empty strings</td>
<td>$</td>
</tr>
</tbody>
</table>

### Definitions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S[i..j] = s_is_{i+1}...s_j$</td>
<td>substring of $S$</td>
</tr>
<tr>
<td>$S[1..i]$</td>
<td>prefix of $S$ that ends at position $i$</td>
</tr>
<tr>
<td>$S_i = S[i..n], 1 \leq i \leq n+1$</td>
<td>suffix of $S$ that starts at position $i$</td>
</tr>
<tr>
<td>$S_{n+1}$</td>
<td>empty string $\varepsilon$</td>
</tr>
<tr>
<td>$S(i)$</td>
<td>character at position $i$ in $S$</td>
</tr>
<tr>
<td>$\sigma(S) = {S_i</td>
<td>1 \leq i \leq</td>
</tr>
</tbody>
</table>
Suffix Trie

Definition

A suffix trie for a string $S \subseteq \Sigma^n$ is a directed tree with edge labels $\in \Sigma$ where

- The concatenation of the labels of all paths from the root to a leaf just give $\sigma(S)$.
- The labels of sibling edges from one node start with different characters.
- Atomic tree

Termination Symbol $\$$

No suffix must be prefix of another suffix.
Suffix Tree

Definition

A suffix tree $T$ for a string $S \subseteq \Sigma^n$ is a compact suffix trie. This means

- $T$ has exactly $n$ leaves numbered 1 to $n$.
- The concatenation of the labels from the root to a leaf $i$ spells out $S_i$. ($\rightarrow$ all paths give $\sigma(S)$)
- The labels of sibling edges from one node start with different characters.
- Every node except the root has at least two children. ($\rightarrow$ compact)
More Definitions

Paths and Labels
- A path is a downwards connected sequence of edges.
- The label of a path is the concatenation of the edge labels on the path.
- The path-label of node n is the concatenation of the edge labels on the path to node n. (→ path-label of leaf i is $S_i$)

Reference pair $(n, \alpha)$ of s
- n: node on the path to s
- $\alpha$: concatenation of edge-labels from n to s
- s not necessarily a node

Canonical reference pair
- n: last node on the path to s
A naïve algorithm

Suffix tree for S
- Start: single edge $S[1..n] = S_1$ → Tree $T_1$
- Successive adding of $S[i..n] = S_i$, i from 2 to n+1
  $T_{i-1} \rightarrow T_i$

- Find longest path from root in $T_{i-1}$ matching a prefix of $S_i$.
- Matching ends at node n (eventually new created).
- Add new edge (n,i) labelled with unmatched suffix of $S_i$.

Time analysis
Inserting $S_i$ takes $O(|S_i|)$ time → Complexity: $O(n^2)$
Ukkonen’s Algorithm I

Proceeding

- Construction of suffix tree for string S in $O(n)$ via implicit suffix trees $\mathcal{I}_1..\mathcal{I}_n$
  $\rightarrow$ true suffix tree $\mathcal{T}$
- Start: $O(n^3)$ method to build $\mathcal{T}$
  $\rightarrow$ optimization to linear time

Implicit suffix trees

- Remove every occurrence of $\$$.  
- Re-establish suffix-tree conditions.
- $\mathcal{I}_i$ implicit suffix tree for S[1..i]
  $(\mathcal{I}_n$ encodes all suffixes of S!)}
Ukkonen’s Algorithm II

Algorithm at a high level
Construct implicit suffix tree $I_1$.

For $i$ from 1 to $n-1$ { for j from 1 to $i+1$ {

1. find end of path from root labelled $S[j..i]$ in $I_i$
2. apply appropriate extension rule ($S[j..i+1]$ in tree)
   1. $S[j..i]$ ends at leaf $\rightarrow$ add $S(i+1)$ to edge label
   2. No path from end of $S[j..i]$ starts with $S(i+1)$ $\rightarrow$ add new leaf edge labelled $S(i+1)$ and leaf node $j$
   3. $\exists$ path from $S[j..i]$ beginning with $S(i+1)$ $\rightarrow$ do nothing

} \ extension
} \ phase

Time $O(n^3)$
Ukkonen’s Algorithm III

Suffix Links
Definition
Suffix link \((u, s(u))\) is a pointer from internal node \(u\) labelled \(x\alpha\) to node \(s(u)\) labelled \(\alpha\).

Single Extension Algorithm
1. Find first node \(u\) up from \(S[j-1..i]\) that has suffix link or is root (at most one edge up!)
2. If \(u \neq\) root: walk down from \(s(u)\) following path for \(\alpha\).
   If \(u =\) root: walk down from root following path for \(S[j..i]\).
3. Apply appropriate extension rule \(\rightarrow S[j..i]S(i+1)\) in the tree
4. If new internal node \(w\) was created, create suffix link \((w, s(w))\)

Time complexity
Worst case not yet improved: \(O(n^3)\)
Ukkonen’s Algorithm IV

Problem:
Down-walking along path labelled $\alpha$ costs $\mathcal{O}(|\alpha|)$ time

Trick 1: Skip/Count
- Skip edge if |unmatched part of $\alpha$| > |edge label|
- Time complexity
  - Traversing of edge $\mathcal{O}(1) \rightarrow$ down-walk in $\mathcal{O}(\#\text{nodes})$
  - $\rightarrow$ 1 phase in $\mathcal{O}(n) \rightarrow$ algorithm in $\mathcal{O}(n^2)$

Edge-label compression
- Time for algorithm $\geq$ size of its output ($\Theta(n^2)$)
  - $\rightarrow$ different representation scheme for edge labels
- Pair of indices (i,j)
  - i beginning position of substring in S
  - j ending position of substring in S
Ukkonen’s Algorithm V

**Trick 2:** Rule 3 is a show stopper
- If rule 3 applies for $S[j..i]$ it also applies for any $S[k..i]$, $k>j$ (Implicit extensions)
- End phase after first extension $j^*$ where rule 3 applies

**Trick 3:** Once a leaf, always a leaf
- $j_i = \# \text{ initial extensions in phase } i \text{ where rule 1 or 2 applies} \rightarrow j_i \leq j_{i+1}$
- In phase $i+1$ do
  - Label new created leaf-edges $(n,e) \rightarrow S[n..i+1]$ (e global symbol denoting current current end)
  - In extensions 1 to $j_i$ only increment $e \rightarrow$ rule 1 for leaf-edges

**Combination**
In phase $i+1$ explicit extensions only from
- Extension $j_i + 1 = \text{ active point}$ to
- Extension $j^* = \text{ end point}$
Ukkonen’s Algorithm VI

Single phase algorithm
1. Increment index $e$ to $i+1$
2. Explicitly compute successive extensions (using SEA) starting at $j_i+1$ until first extension $j^*$ where rule 3 applies (or until all extensions are done)
3. Set $j_{i+1}$ to $j^*-1$ to prepare for next phase.

Time complexity
Suffix Links + Edge Compression + Trick 1-3 allows construction of suffix tree for String $S$ in $O(|S|)$.

Creating the true suffix tree
Conversion in $O(|S|)$.
1. Add termination symbol $\$$ to end of $S$.
2. Let Ukkonen’s algorithm continue with extended string.
3. Replace each index $e$ on every leaf edge with $n$. 
McCreight’s Algorithm I

Definitions

- $T_i$: intermediate suffix tree encoding suffixes $S_1$ to $S_{i-1}$
- McHead(i): longest prefix of $S_i$ that is also prefix of $S_j$, $j < i$
- McTail(i): $S_i$-McHead(i)

Proceeding

The “Algorithm M” inserts suffixes in order from $S_1$ to $S_n$.

- $T_i \rightarrow T_{i+1}$
- Find end of path labelled McHead(i)
- $n =$ node labelled McHead(i) (eventually new created)
- Add new leaf i and new edge $(n,i)$ labelled McTail(i)

More efficiency

- Edge compression, suffix links
- Lemma: McHead(i-1) = $x\delta \Rightarrow \delta$ is a prefix of McHead(i)
**McCreight’s Algorithm II**

**Step i of “Algorithm M”**

1. Starting from McHead(i-1) = $\xi \alpha \beta$ walk upwards till first node a (labelled $\xi \alpha$); if a = root go to 3.
2. Follow suffix link to node c (labelled $\alpha$)
3. “Rescanning”: walk downwards along path labelled $\beta$ using skip/count trick $\rightarrow$ node d
4. Add suffix link (a,d)
5. “Scanning”: search downwards along path labelled $\gamma$ (unknown length!) $\rightarrow$ node e
6. Add leaf i and edge (e,i)

**Time complexity**

Rescanning and scanning in $O(1) \rightarrow O(n)$
Weiner’s Algorithm I

Definitions
- $\mathcal{W}_i$: suffix tree for $S_i = S[i..n]$
- $\text{WHead}(i)$: longest prefix of $S_i$ that is also prefix of $S_{j, j>i}$

Proceeding
Build $\mathcal{W}_{n+1} = \text{edge (root, n+1) labelled }$
For $i$ from $n$ to $1$ do
- Find $\text{WHead}(j)$ in $\mathcal{W}_{j+1}$
- $w =$ node labelled $\text{WHead}(j)$ (eventually new created)
- Create new leaf $j$ and edge $(w,j)$ labelled $S[j..n]-\text{WHead}(j)$

More efficiency
- Edge compression
- 2 vectors: Indicator Vector $I_u(x)$ and Link Vector $L_u(x)$
Weiner’s Algorithm II

The Vectors

- $\mathcal{I}_u(x) = 1 \iff u$ labelled $\alpha$ & $\exists$ partial path in $\mathcal{W}$ labelled $x\alpha$
- $\mathcal{L}_u(x) = \uparrow \hat{u} \iff \hat{u}$ labelled $x\alpha$ & $u$ labelled $\alpha$; otherwise $\mathcal{L}_u(x) = \text{null}$

Vector Usage

$\mathcal{W}_{i+1} \rightarrow \mathcal{W}_i$:

- Start at leaf $i+1$, find first $u$ with $\mathcal{I}_u(S(i)) = 1$ ($u$ labelled $\alpha$)
- Continue till first $u'$ with $\mathcal{L}_{u'}(S(i)) \neq \text{null}$ ($l_i = |u-u'|$)
  - $u$, $u'$ don’t exist $\rightarrow \mathcal{W}\text{Head}(i) = \epsilon$
  - $u$, $u'$ exist $\rightarrow \mathcal{W}\text{Head}(i) = S(i)\alpha$ & $\mathcal{W}\text{Head}(i)$ ends $l_i$ chars below $\hat{u}$
  - $u$ exists, $u'$ doesn’t $\rightarrow \mathcal{W}\text{Head}(i)$ ends $l_i$ chars below root

Time complexity

Head $(i)$ found in $O(1) \rightarrow$ Complexity of algorithm $O(n)$
Exact string matching

Find all occurrences for pattern \( P \) in text \( T \):
- Build suffix tree in \( \mathcal{O}(|T|) \) and match \( P \) along unique path \( \mathcal{O}(|P|) \).
- \( P \) exhausted: numbers of leaves below are starting points for \( P \)
- Mismatch: \( P \) does not occur

Comparison with KMP and BM algorithms
- \( P \) and \( T \) fix; \( P \) fix \( \rightarrow \) same time and space bound
- Fixed \( T \) and varying \( P s \) \( \rightarrow \mathcal{O}(|T|) + \sum_{P} \mathcal{O}(|P|+|T|+\#occurrences of P|) \)
  \( \rightarrow \) vastly better performance

Exact set matching

Task: Find all \( k \) occurrences of a set of strings \( \mathcal{P} \) in text \( T \)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Tree</th>
<th>Search</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aho-Corasick</td>
<td>( \mathcal{O}(\Sigma</td>
<td>P</td>
<td>) )</td>
</tr>
<tr>
<td>Suffix Trees</td>
<td>( \mathcal{O}(</td>
<td>T</td>
<td>) )</td>
</tr>
</tbody>
</table>
Longest common substring

Generalized suffix tree

- **Definition:** Tree which represents the suffixes of a set \( \{S_1, S_2, \ldots, S_n\} \)
- **Construction:** Variation of Ukkonen’s algorithm
  1. Build tree for \( S_1 \)$
  2. Match \( S_2 \)$ against path in tree, first mismatch \( S[i+1] \)
     \( \rightarrow \) tree encodes \( \sigma(S_1) \) and implicitly \( \sigma(S_2[1..i]) \)
  3. Resume Ukkonen’s algorithm on \( S_2 \) in phase \( i+1 \)
  4. Repeat for each string

Longest common substring (lcs)

- **Proceeding:**
  - Build generalized suffix tree for \( S_1 \) and \( S_2 \)
  - Mark internal nodes \( v \) with 1(2) if leaf in subtree of \( v \) represents suffix of \( S_1(S_2) \)
  - Search node marked 1 and 2 with longest path-label (= lcs)

- **Time complexity:** \( O(\Sigma |S_i|) \)
Introduction

- **Application:** DNA Analysis
- **Definition:** Superstring Problem
  For a given set of strings \( \{S_1, S_2, \ldots, S_n\} \) find superstring \( S \) which contains every \( S_i \) as substring.
- **Solution:** Blending
  Assembling of two strings \( S_i, S_j \) as follows:
  Find longest suffix \( \alpha \) of \( S_i \) which is prefix of \( S_j \) and create new string \( \text{blend} \ (S_i, S_j) = S_i - \alpha + S_j = S_i - \text{ov}(S_i, S_j) + S_j \)

- GREEDY-Heuristic with Suffix Trees (Kosaraju/Delcher)
  \( \rightarrow \) approximate solution for smallest \( S \)
String Assembly II

GREEDY-Heuristic with Suffix Trees

- Generalized suffix tree $\mathcal{T}$:
  - leaf numbers: $(i, p) \rightarrow$ suffix $S_1[p..|S_1|]$
  - implicit $\rightarrow$ $\$$ omitted, internal nodes can be leaves
  - substrings of other strings and copies of identical strings removed

- Arrays & Sets:
  - chain $\rightarrow$ already blended strings
  - wrap $\rightarrow$ unavailable suffixes and prefixes
  - $S_u$ $\rightarrow$ suffixes available at node $u$
    initially $S_u = \{i|u$ has leaf number $(i,1)\}$
  - $P_u$ $\rightarrow$ prefixes available at node $u$
    initially $P_u = \{i|u$ has leaf number $(i,d), d>1\}$
Proceeding

1. Find node $u$ with largest string-depth
2. Find pair $(i, j)$ with
   - $i \in S_u$, $j \in P_u$
   - $\text{chain}(i) = 0$, $\text{wrap}(i) \neq j$
3. Discard all $i$ from $S_u$ with $\text{chain}(i) \neq 0$
4. Remove $i$ from $S_u$ and $j$ from $P_u$ and set
   - $\text{chain}(i) = j$
   - $\text{wrap}(\text{wrap}(i)) = \text{wrap}(j)$, $\text{wrap}(\text{wrap}(j)) = \text{wrap}(i)$
5. Repeat 2. – 4. until no further blends are feasible
6. Union remaining $P_u$ to set $P$ of $u$’s parent
7. Discard $S_u$ and remove $u$ from string-depth-order
9. Generate superstring from chain array
Conclusion

Suffix Trees

Implementation Details

Comparison of the algorithms

- Time
- Space
- Comprehensibility

Applications