Ukkonen’s suffix tree algorithm

• Recall McCreight’s approach:
  – For $i = 1 \ldots n+1$, build compressed trie of
    \{$x[j..n]$ | $j \leq i$\}

• Ukkonen’s approach:
  – For $i = 1 \ldots n+1$, build compressed trie of
    \{$x[j..i]$ | $j \leq i$\}
  – Compressed trie of all suffixes of prefix
    $x[1..i]$ of $x$
  – A suffix tree except for “leaf” property
McCreight’s algorithm, \( x = aba \)

\[
T_1 : \{ x[j..n] \mid j \leq 1 \}
\]

\[
T_2 : \{ x[j..n] \mid j \leq 2 \}
\]

\[
T_3 : \{ x[j..n] \mid j \leq 3 \}
\]

\[
T_4 : \{ x[j..n] \mid j \leq 4 \}
\]
Ukkonen’s algorithm, $x=aba$

$T_1 : \{x[j..1]\}$

$T_2 : \{x[j..2]\}$

$T_3 : \{x[j..3]\}$

$T_4 : \{x[j..n]\}$

Note: no node for $x[3..3] = \text{"a"}$
Tasks in iteration $i$

- In iteration $i$ we must
  - Update each $x[j..i]$ to $x[j..i+1]$ 
  - Add string $x[i+1]$ (special case of above)
• “Obvious” algorithm:

For $i=1,...,n+1$:
  for $j=1,...,i$:
    find $x[j..i]$
    append $x[i+1]$

- Running time $O(n^3)$
- Need lots of tricks to get $O(n)$!
“Free” operations – leaves

• If we label leaves with \((k, \infty)\) – denoting “\(k\) to the current \(i\)”, updating a leaf is automatic
"Free" operations – existing strings

- If $x[j..i+1]$ is already in the tree, the update is automatic
“Real” operations

• If we can recognize the free operations, we need only deal with the remaining
Lemma 5.2.4

Let j denote suffix $x[j..i]$ of $x[1..i]$

a) If $j>1$ is a leaf node in $T_i$, then so is $j-1$

b) If, from $j<i$, there is a path in $T_i$ that begins with $a$, then there is a path in $T_i$ from $j+1$ beginning with $a$
Proof of lemma 5.2.4 (a)

- If $j > 1$ is a leaf node in $T_i$, then so is $j-1$

Assume $j-1$ is not a leaf. Then there exists $k < j-1$ such that:

Then

thus:

\[
\begin{align*}
&x_{j-1..i} \quad x_{k..i} \\
&j-1 \\
k \\
k+1
\end{align*}
\]

\[
\begin{align*}
&x_{j+1..i} \\
j \\
k+1
\end{align*}
\]
Proof of lemma 5.2.4 (b)

- If, from \( j < i \), there is a path in \( T_i \) that begins with a, then there is a path in \( T_i \) from \( j + 1 \) beginning with a.

Assume \( j \) is followed by “a” there exists \( k < j \) such that:

\[
\begin{array}{c}
  j \\
  k \\
\end{array}
\]

thus:

\[
\begin{array}{c}
  j + 1 \\
  k + 1 \\
\end{array}
\]

Hence \( j + 1 \) is followed by “a”.
Corollary of lemma 5.2.4

- In iteration $i$, there exist indices $j_L$ and $j_R$ such that:
  - All suffixes $j \leq j_L$ are leaves
  - All suffixes $j \geq j_R$ are already in the trie
Corollary of lemma 5.2.4

- I and III are free operations
Updated algorithm

• Implicitly handling “free” operations:

\[
\text{For } i=1,\ldots,n+1: \\
\text{for } j=j_L,\ldots,j_R: \\
\text{find } x[j..i] \\
\text{append } x[i+1]
\]

- \(j_L\) in iteration \(i\) is the last leaf inserted in iteration \(i-1\) (“once a leaf, always a leaf”)
- \(j_R\) in iteration \(i\) is the first index where \(x[j..i+1]\) is already in the trie
Handling index $j$ in $II$

- Whenever $j_L < j < j_R$, $j$ is made a leaf:

- Once $j$ is a leaf, it will be in $I$ and never in $II$ again
Handling index $j$ in II

- We handle $j$ in II or implicitly in III $2^n$ times:
• Running time is $2n \times T(\text{find } x[j..i])$
  – We just have to deal with $T(\text{find } x[j..i])$ in $O(1)$
  – No worries!
Using **fastscan** and s(-)

- When searching for $x[j..i]$, it is already in the trie
  - We can use **fastscan** for the search
  - $T(\text{find } x[j..i])$ in $O(d)$ where $d$ is the (node-) depth of $x[j..i]$
- If we keep suffix links, $s(-)$, in the tree we can use these as shortcuts
• **Invariant:** All inner nodes have suffix links

• Ensuring the invariant:
  – We only insert inner nodes $x[j..i]$ when adding leaves $j$
  – Whenever we insert a new node, $x[j..i]$ for some $j<i$, we also find or insert $x[j+1..i]$, and can update $s(x[j..i]) := x[j+1..i]$
  – If we insert $x[i..i]$, then $s(x[i..i]) := \varepsilon$
Finding $x[j+1..i]$ from $x[j..i]$  

Starting from here (initial $j$ is $j_L$ and we can keep a pointer to that node between iterations)  

Using **fastscan** here
Bound on \texttt{fastscan}

- Time usage by \texttt{fastscan} is bounded by
  - $n$ – for the maximal (node-)depth in the trie
  - $+$ total decrease of (node-)depth

- Decrease in depth:
  - Moving to parent($j$): 1
  - Moving to $s(\text{parent}(j))$: max 1
  - “Restarting” at $j_L$: ?
Hacking the suffix links

- When searching for \(x[j_L+1..i]\), update \(s(x[j_L..i])\) to point to the nearest ancestor of \(x[j_L+1..i]\)

```
parent(j_L)
```

```
s(parent(j_L))
```

```
x[j_L+1..i]
```

```
Nearest ancestor of x[j_L+1..i]
```

- “Restarting” becomes essentially free
Running time

- Vertical steps are paid for by the previous horizontal step (free restarting)
- Horizontal steps are total **fastscan** bounded by $O(n)$
- **Runtime** $O(n)$
Why Ukkonen?

• Ukkonen’s algorithm is an “online” algorithm:
  – As long as no suffix is a prefix of another, the intermediate trees are suffix trees
  – Generalized suffix trees can be built one string at a time