Suffix trees

Data structures for string pattern matching: Suffix trees

- Linear algorithms for exact string matching
  - KMP
  - Z-value algorithm

- What is suffix tree?
  - A tree-like **data structure** for solving problems involving strings.
  - Related data structures: Trie (re**trie**val) & PATRICIA (radix tree)
  - Allow the storage of **all substrings** of a given string in linear space
  - Simple algorithm to solve string pattern matching problem in linear time

Better than hash tables?

- Hash tables are certainly easier to understand. And, one can produce a hash table of all length *k* strings in O(*m*) time and look up a *k*-length string *x* in O(*k*) time, finding all *p* places where string *x* is found in O(*p*) time. This is the same as the bound for suffix trees.
- What if you don't know how long the string *x* is going to be?
- And most other string matching tricks don't work for it either.

Suffix Tree: definition

- A suffix tree *ST* for an m-character string *S* is a rooted directed tree with exactly *m* leaves numbered 1 to *m*.
- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of *S*.

Suffix tree: definition

- No two edges out of a node can have edge-labels beginning with the same character.
- The key feature of the suffix tree is that for any leaf *i*, the concatenation of the edge-labels on the path from the root to the leaf *i* exactly spells out the suffix of S that starts at position *i*.

Suffix Trees: Example

- Suffixes of ‘papua’
  - ‘papua’
  - ‘apua’
  - ‘pua’
  - ‘ua’
  - ‘a’
  - ‘’

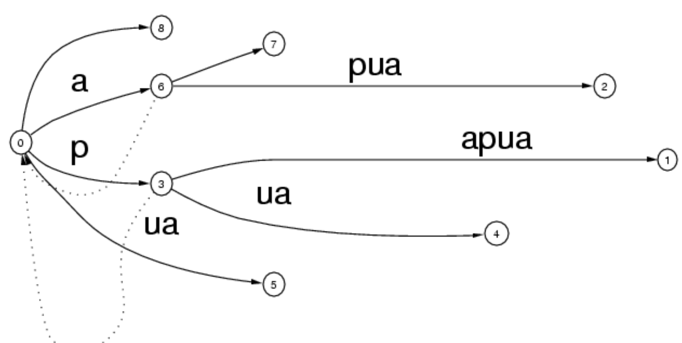
Suffix Trees: Example

- Suffixes of 'papua'
  - 'papua'
  - 'apua'
  - 'pua'
  - 'ua'
  - 'a'
  - ''

NOTE: Assume the string terminates with some character found nowhere else in the string. (eg. '\0')

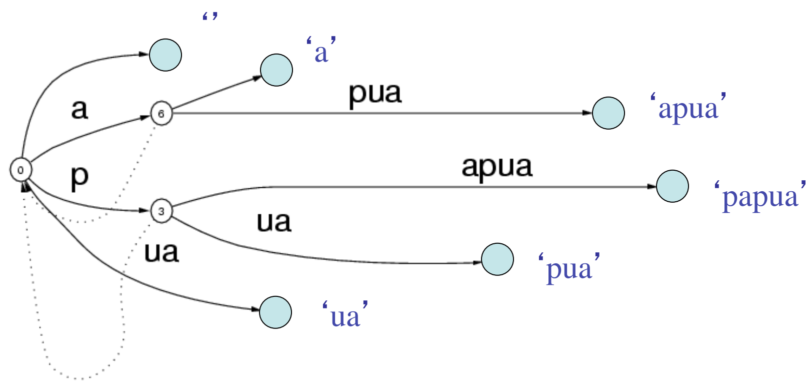
Suffix Trees: Example

- Suffix tree for 'papua'

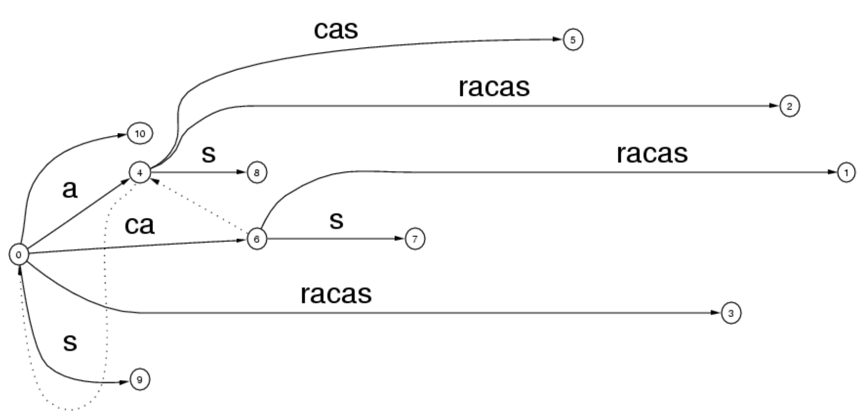


Suffix Trees: Example

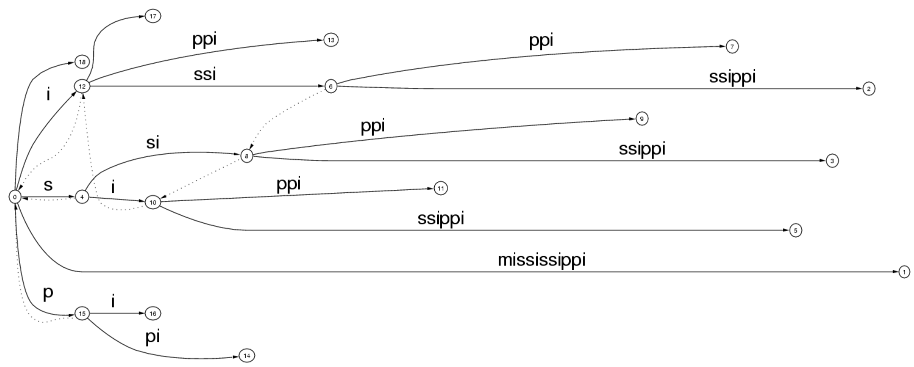
- Suffix tree for ‘papua’



caracas



mississippi



Suffix Trees

- Exact matching in linear time
- Many others
- “We know of no other single data structure that allows efficient solutions to such a wide range of complex string problems.” - Dan Gusfield

Exact string matching problem

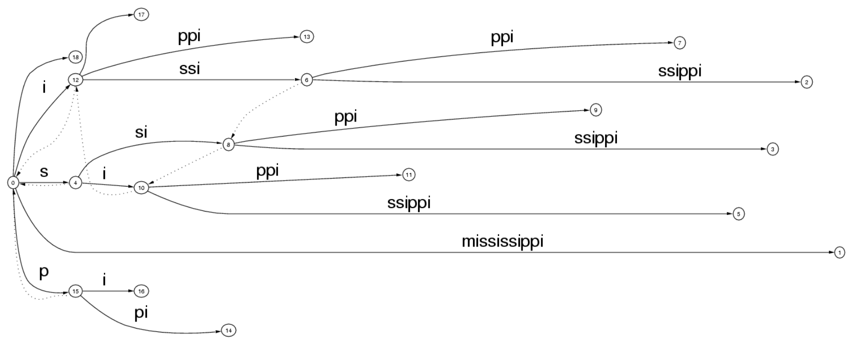
- Given a pattern *P* of length *m*, find all occurrences of *P* in text *T*
  - *O(n+m)* algorithm
- Solution: Build a suffix tree *ST* for text *T* in *O(m)* time. Then, match the characters of *P* along the unique path in *ST* until either *P* is exhausted or no more matches are possible.

Exact string matching problem

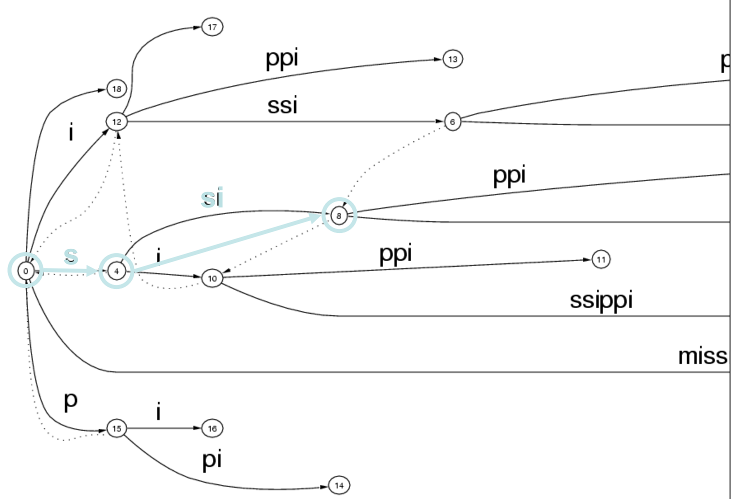
- Find 'ssi' in 'mississippi'

Exact string matching problem

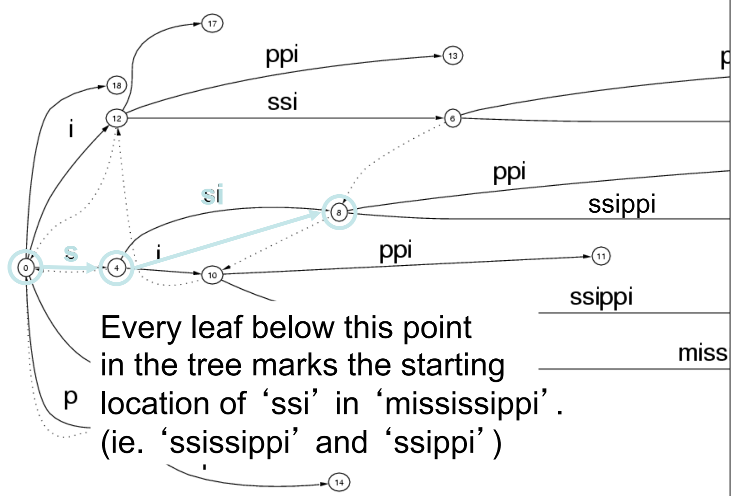
- Find ‘ssi’ in ‘mississippi’



Exact string matching problem



Exact string matching problem

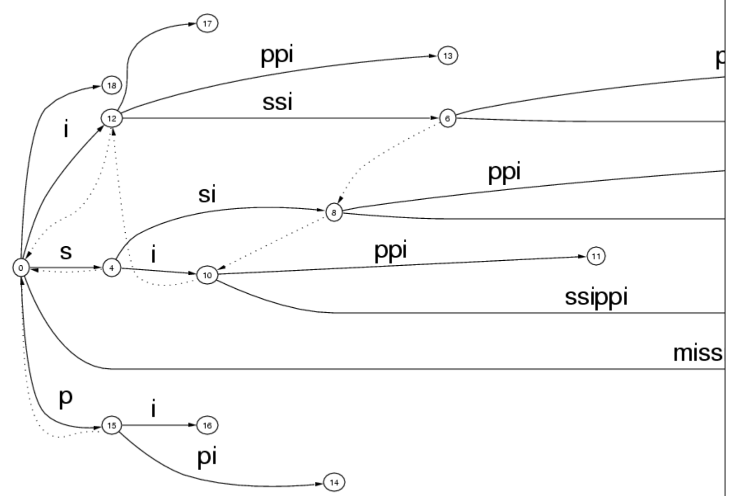


Every leaf below this point in the tree marks the starting location of 'ssi' in 'mississippi'. (ie. 'ssissippi' and 'ssippi')

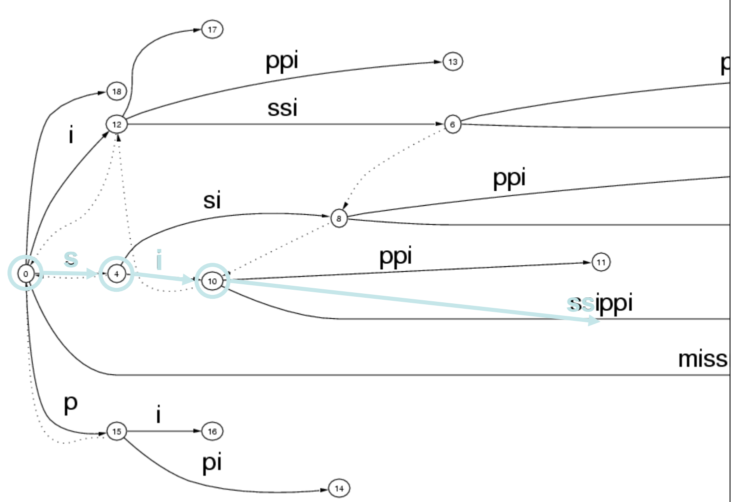
Exact string matching problem

- Find 'sissy' in 'mississippi'

Exact string matching problem



Exact string matching problem



Comparing to the other algorithms

- KMP and Boyer-Moore both achieve this worst case bound.
  - *O(m+n)* when the text and pattern are presented together.
- Suffix trees are <u>much</u> faster when the text is fixed and known first while the patterns vary.
  - *O(m)* for single time processing the text, then only *O(n)* for each new pattern.
- Based on suffix trees, is faster for searching a number of patterns at one time against a single text (**exact set matching problem**)
  - Aho-Corasick algorithm: preprocessing P instead of T.

Building the Suffix Tree

- How do we build a suffix tree?

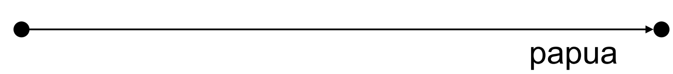
```
while suffixes remain:
    add next shortest suffix to the tree
```

Building the Suffix Tree

- papua

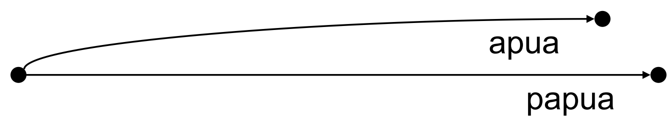
Building the Suffix Tree

- papua



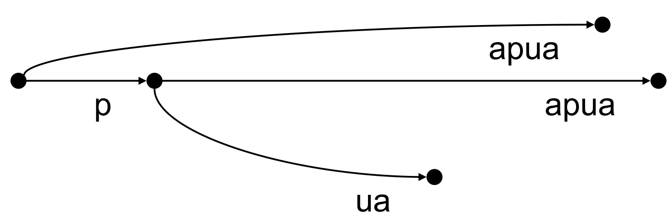
Building the Suffix Tree

- papua



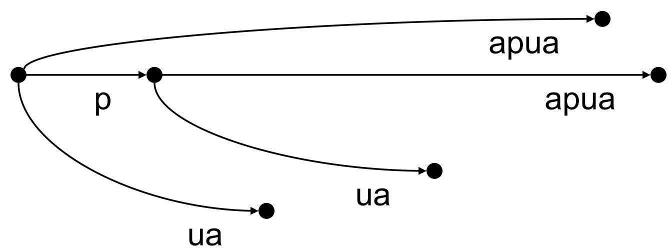
Building the Suffix Tree

- papua



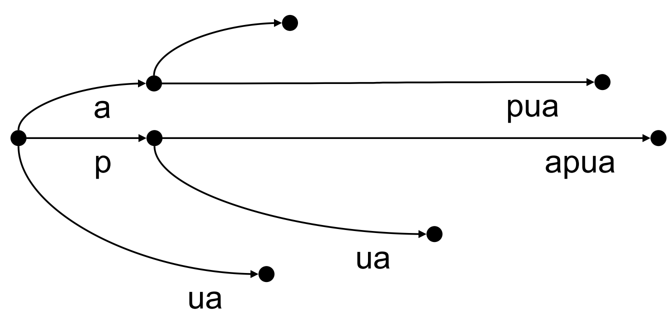
Building the Suffix Tree

- papua



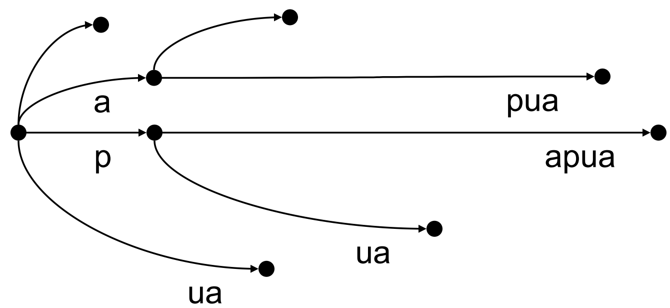
Building the Suffix Tree

- papua



Building the Suffix Tree

- papua



Building the Suffix Tree

- How do we build a suffix tree?

```
while suffices remain:
  add next shortest suffix to the tree
```

Naïve method - *O(m^2)* (m = text size)

Building the Suffix Tree in O(m) time

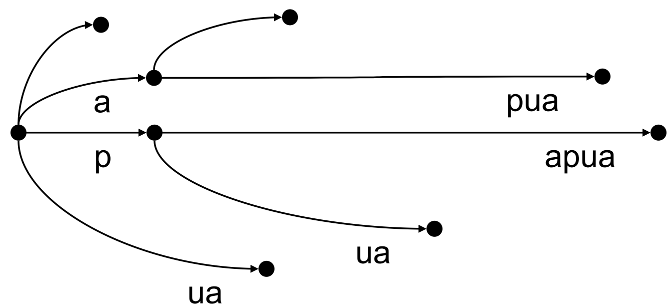
- In the previous example, we assumed that the tree can be built in *O(m)* time.
- Weiner showed original *O(m)* algorithm (Knuth is claimed to have called it "the algorithm of 1973")
- More space efficient algorithm by McCreight in 1976
- Simpler 'on-line' algorithm by Ukkonen in 1995

Ukkonen's Algorithm

- Build suffix tree *T* for string *S[1..m]*
  - Build the tree in *m* phases, one for each character. At the end of phase *i*, we will have tree T_i, which is the tree representing the prefix *S[1..i]* → online construction
    - In each phase *i*, we have *i* extensions, one for each character in the current prefix. At the end of extension *j*, we will have ensured that *S[j..i]* is in the tree T_i.

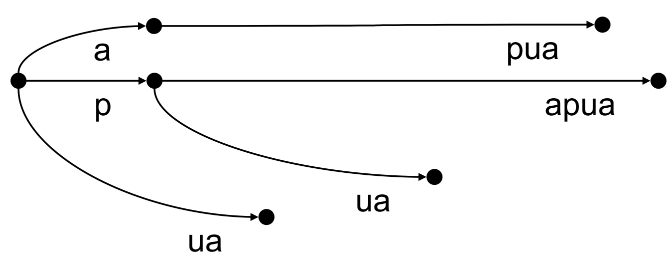
Implicit suffix tree

- papua



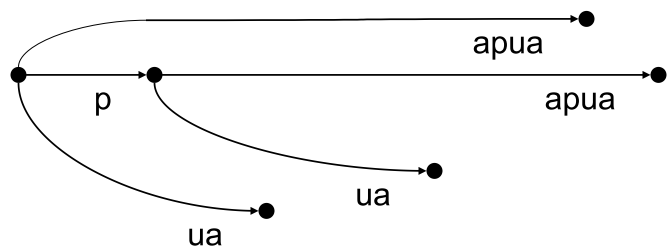
Implicit suffix tree

- papua



Implicit suffix tree

- papua



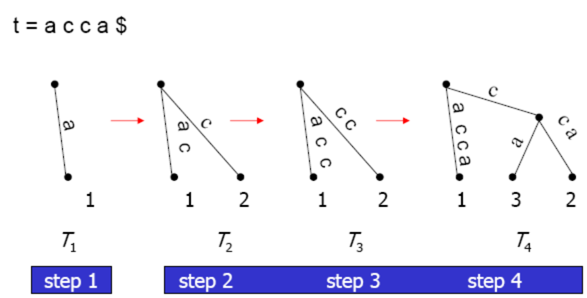
Implicit suffix tree can be transformed from/into regular suffix tree in O(n) time.

Ukkonen's Algorithm

Pseudo code for Ukk:

Construct tree T_1.
for i = 1 **to** m–1 **do**
begin {phase i+1}
  for j = 1 **to** i +1 **do**
    begin {extension j}
      In the current tree find the end of the path from the root labeled $t[j \ldots i]$. If necessary, extend that path by adding character $t[i+1]$, thus ensuring that string $t[j \ldots i+1]$ is in the tree.
    end;
end;

An Exmaple



Ukkonen’s Algorithm

- This is an *O(m³)* time, *O(m²)* space algorithm.
- We need a few implementation speed-ups to achieve the *O(m)* time and *O(m)* space bounds.

Suffix extension rules

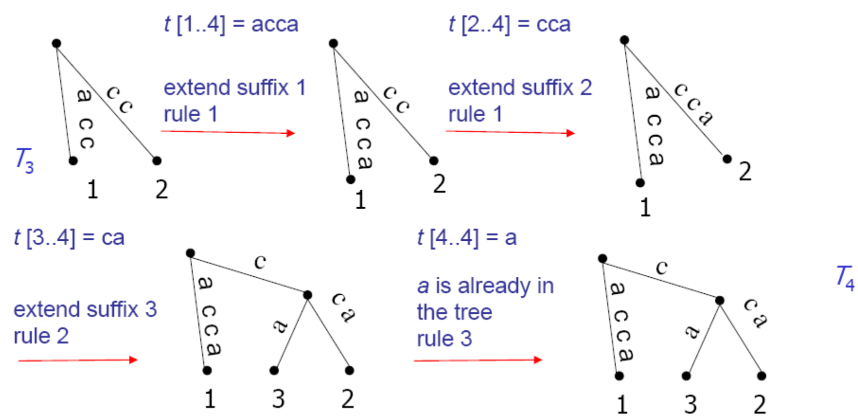
- 3 possible ways to extend *S[j..i]* with character *i +1*.
  1. *S[j..i]* ends at a leaf. Add the character *i+1* to the end of the leaf edge.
  2. No path from the end of *S[j..i]* starts with the *i+1* character. Split the edge and create a new node if necessary, then add a new leaf with character *i+1*.
     (This is the only extension that increases the number of leaves! The new leaf represents the suffix starting at position *j*.)
  3. There is already a path from the end of *S[j..i]* starts with the *i+1* character, or S[j..i+1] correspond to a path. Do nothing.

Ukkonen's Algorithms

t = a c c a $

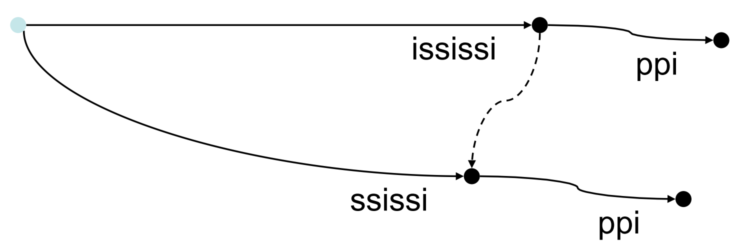
t [1...3] = acc

t [1...4] = acca



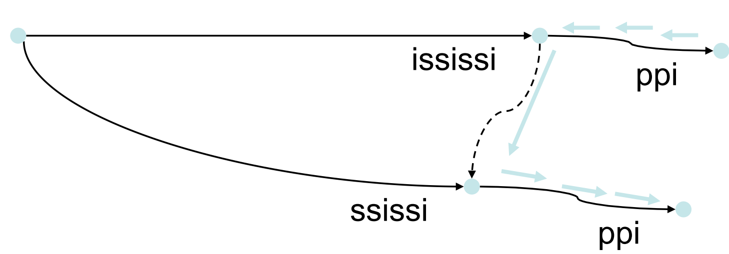
Ukkonen's Algorithm: Speed-up 1

- Suffix Links
  - speed up navigation to the next extension point in the tree



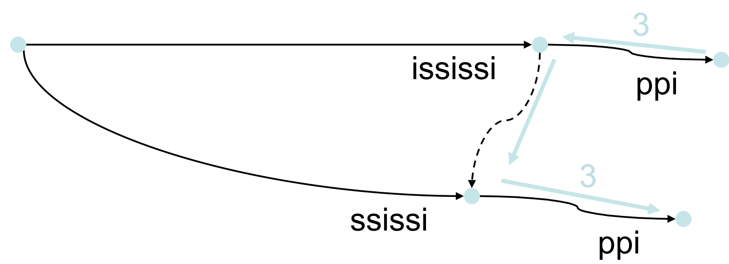
Ukkonen's Algorithm: Speed-up 2

- Skip/Count Trick
  - instead of stepping through each character, we know that we can just jump, as long as we're the right distance



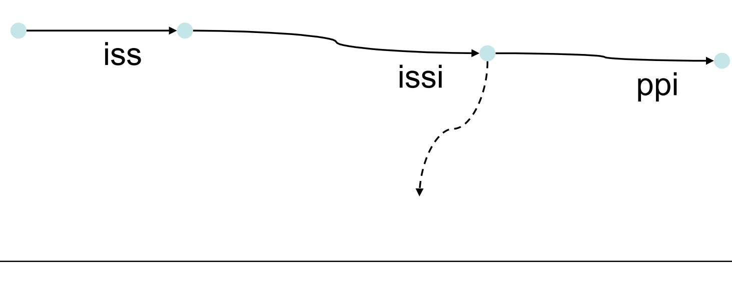
Ukkonen's Algorithm: Speed-up 2

- Skip/Count Trick
  - instead of stepping through each character, we know that we can just jump, as long as we're the right distance



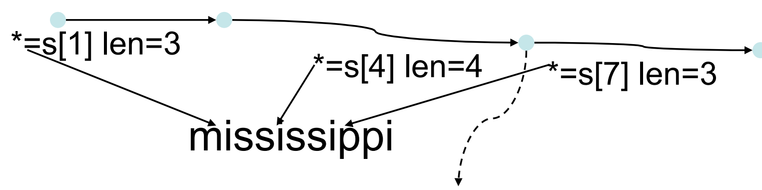
Ukkonen's Algorithm: Speed-up 3

- Edge-Label Compression
  - since we have a copy of the string, we don't need to store copies of the substrings for each edge



Ukkonen's Algorithm: Speed-up 3

- Edge-Label Compression
  - since we have a copy of the string, we don't need to store copies of the substrings for each edge
  - *O(m²)* space becomes *O(m)* space



Ukkonen's Algorithm: Speed-up 4

- A match is a show stopper.
  - If we find a match to our next character (rule 3 applies), we're done this phase.

Ukkonen's Algorithm: Speed-up 5

- Once a leaf, always a leaf (implicitly implement rule 1).
  - We don't need to update each leaf, since it will always be the end of the current string. We can get these updates for free.
  - Either 1)maintain a global *end-of-string* index or 2) insert the whole string for every leaf

Ukkonen's Algorithm: Speed-ups

- Because of speed-ups 4 and 5, we can pick up the next phase right where we ended the last one!

Ukkonen's Algorithm – mississippi with Speed-ups

```
void SuffixTree::update(char* s, int len) {
  …
  int i;
  int j;
  …
  for (i = 0, j = 0; i < len; i++) {
    while (j <= i) {
      … all the work …
    }
  }
}
```

Possible execution

Extension Case distribution

j=	1	2	3	4	5	6	7	8	9=m
i=1:	1	3							
i=2:	1	1	2						
i=3:	1	1	2	2					
i=4:	1	1	3	2	2				
i=5:	1	1	1	2	2	2			
i=6:	1	1	1	3	3	2	2		
i=7:	1	1	1	1	1	3	2	2	
i=8:	1	1	1	1	1	1	3	2	2

Need to compute:

At most one Case 2 per i

At most one Case 1 or 3 per j

Creating a true suffix tree

- Run another iteration of Ukkonen algorithm on S$
- No suffix is now a prefix of any other suffix.
- As a result, each suffix will end at a leaf.
- Replace each index on every leaf edge with the number m.

Total Algorithm time O(m)

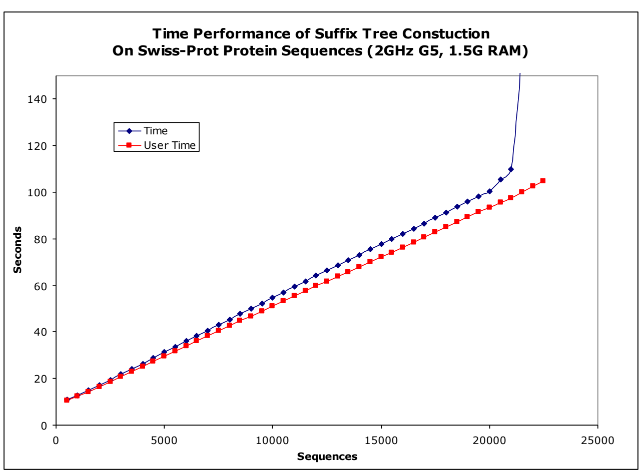
Ukkonen's Algorithm – The Punch Line

- By combining all of the speed-ups, we can now construct a suffix tree T_m representing the string *S[1..m]* in
  - *O(m)* time and in
  - *O(m)* space!

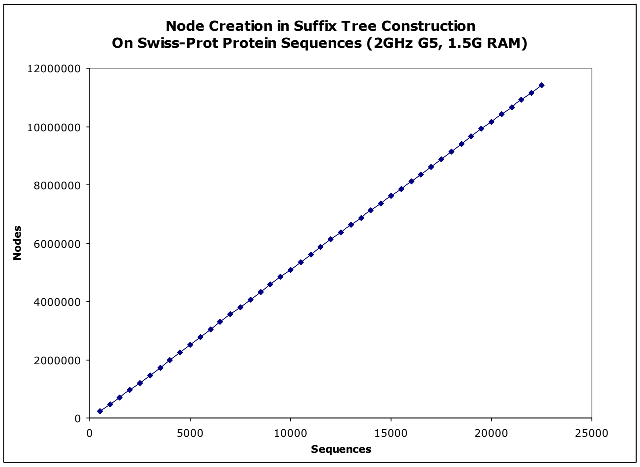
Exact string matching

- Both P (|P|=n) and T (|T|=m) are known:
  - Suffix tree method achieves same worst-case bound O(n+m) as KMP.
- T is fixed and build suffix tree, then P is input, k is the number of occurrences of P
  - Using suffix tree: O(n+k)
  - In contrast (KMP, preprocess P): O(n+m) for any single P
- P is fixed, then T is input
  - Selecting KMP rather than suffix tree
  - or Aho-Corasick algorithm (exact set matching problem)

Ukkonen’s Algorithm – Time Performance



Ukkonen's Algorithm – Memory Usage



Applications

- Problems
  - linear-time longest common substring
  - constant-time least common ancestor
  - maximally repetitive structures
  - all-pairs suffix-prefix matching
  - compression
  - inexact matching
  - conversion to suffix arrays

Bioinformatics applications

- Applications
  - Sequence comparison
  - motif discovery
  - PST – probabilistic suffix trees
  - SVM string kernels
  - chromosome-level similarities and rearrangements