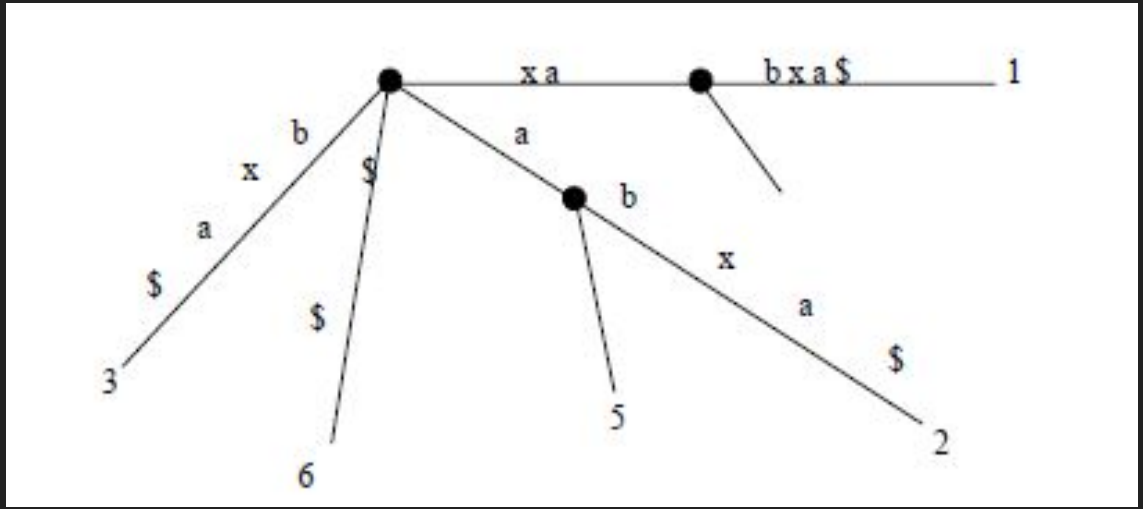
Suffix Tree Construction

Using Ukkonen’s Algorithm



What is a suffix tree?

Naive approach to build a suffix tree

$O(m^3)$

Ukkonen’s Algorithm

3 Extension Rules:

#Rule1

#Rule2

#Rule3

Rule 1 In the current tree, path b ends at a leaf. That is, the path from the root labeled b extends to the end of some leaf edge. To update the tree, character S(i + 1) is added to the end of the label on that leaf edge.

Rule 2 No path from the end of string b starts with character S(i + 1), but at least one labeled path continues from the end of b.
In this case, a new leaf edge starting from the end of b must be created and labeled with character S(i + 1). A new node will also have to be created their if, b ends inside an edge. The leaf at the end of the new leaf edge is given the number j.

Rule 3 Some path from the end of string b starts with character S(i +1). In this case the string bS(i + 1) is already in the current tree, so (remembering that in an implicit suffix tree the end of a suffix need not be explicitly marked) we do nothing.

Ukkonen’s Algorithm

Implementation and Speed-ups

Suffix Links

First implementation speed-up

Definition

Let XA denote an arbitrary string, where X denotes a single character and A denotes a (possibly empty) substring. For an internal node V with path-label, XA if there is another node S(V) with path-label A, then a pointer from V to S(V) is called a **suffix link**.

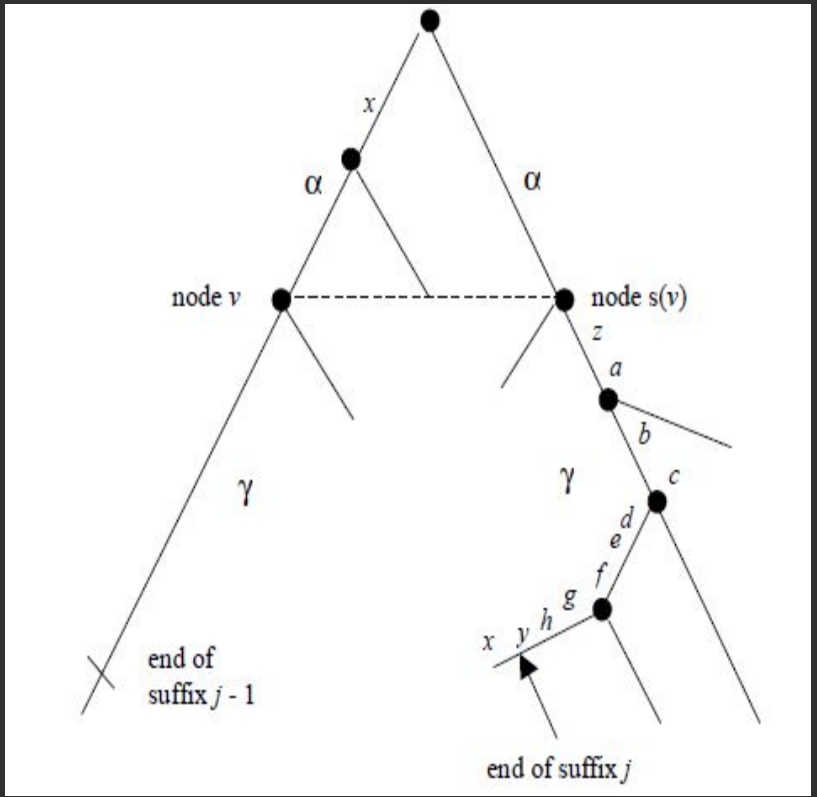
What has been achieved so far?

The use of suffix links is clearly a practical improvement over walking from the root in each extension, as done in the naive algorithm. But does their use improve the worst case running time?

The answer is that as described, the use of suffix links does not yet improve the time bound. However, here we introduce a trick that will reduce the worst-case time for the algorithm to **0(m2).** This trick will also be central in other algorithms to build and use suffix trees

Trick number 1

skip/count trick



Using the skip/count trick any phase of Ukkonen's algorithm takes 0(m) time.

All the operations other than the downwalking take constant time per extension, so we only need to analyze the time for the

down-walks. We do this by examining how the current node-depth can change over the

phase.

The up-walk in any extension decreases the current node-depth by at most one (since it moves up at most one node), each suffix link traversal decreases the node-depth by at most another one (by Lemma 6.1.2) and each edge traversed in a down-walk moves to anode of greater node-depth.

Thus over the entire phase the current node-depth is

decremented at most 2m times, and since no node can have depth greater than m, the

total possible increment to current node-depth is bounded by 3m over the entire phase.

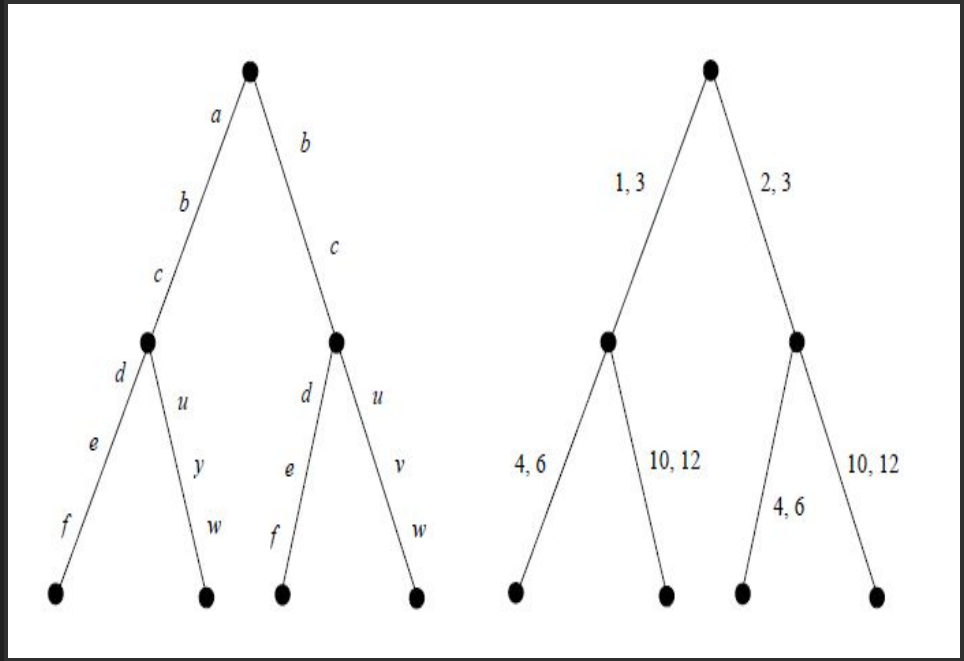
It follows that over the entire phase, the total number of edge traversals during downwalks

is bounded by 3m. Using the skip/count tick, the time per down-edge traversal is

constant so the total time in a phase for all the down-walking is 0(m).

Edge Label Compression

Alternate scheme for edge labeling. Instead of explicitly writing a substring on an edge of the tree, only write a pair of indices on the edge, specifying beginning and end positions of that substring in S.



Why did we compress the edges

By using an index pair to specify an edge-label, only two numbers are written on any edge, and since the number of edges is at most 2m - 1, the suffix tree uses only 0(m) symbols and requires only 0(m) space. This makes it more plausible that the tree can actually be built in 0(m) time.

Trick 2:

End any phrase i + 1 the first time that extension rule 3 applies If this happens in extension, then there is no need to explicitly find the end of any string S[k…i] for k > j.

The extensions in phase i + 1 that are ‘done’ after the first execution of rule 3 are said to be done implicitly. This is in contrast to any extension j where the end of S[j…i] is explicitly found, An extension of that kind is called an explicit extension.

Trick 2 is clearly a good heuristic to reduce work, but it’s not clear if it leads to a better worst-case time bound. For that we need one more observation and trick.

Observation: Once a leaf, Always a leaf

If at some point in Ukkonen’s algorithm a leaf is created, then that leaf will remain a leaf in all successive trees created during the algorithm. This is true because the algorithm has no mechanism for extending a leaf edge beyond its current leaf. In more detail, once there is a leaf labeled j, extension rule 1 will always apply to extension j in any successive phase. So once a leaf, always a leaf.

Trick 3:

In phase i +1, when a leaf edge is first created and would normally be labeled with substring S[p..i + 1], instead of writing indices (p, i + 1) on the edge, write (p, e), where e is a symbol denoting “**the current end**”. Symbol e is a global index that is set to **i + 1** once in each phase. In phase i + 1, since the algorithm knows that rule 1 will apply in extensions i through j, at least, it need do no additional explicit work to implement those ji extensions.

Instead, it only **does constant work** to increment variable e, and then does explicit work for (some) extensions starting with extension ji + 1.

Conclusion

- Using suffix links and implementation tricks 1,2, and 3, Ukkonen‘s algorithm builds implicit suffix trees in 0(m) total time.
- Hence, it’s an efficient algorithm to build a suffix tree.
- Therefore, it is used in many algorithms which will be discussed in next slide

Applications for Ukkonen’s Algorithm

- Substring matching
- Longest repeated substring
- Longest palindromic substring
- Longest common substring
- Build suffix array in O(n)