The Suffix Tree

Giulia Bernardini
giulia.bernardini@units.it

Fundamentals of algorithms
a.y. 2021/2022

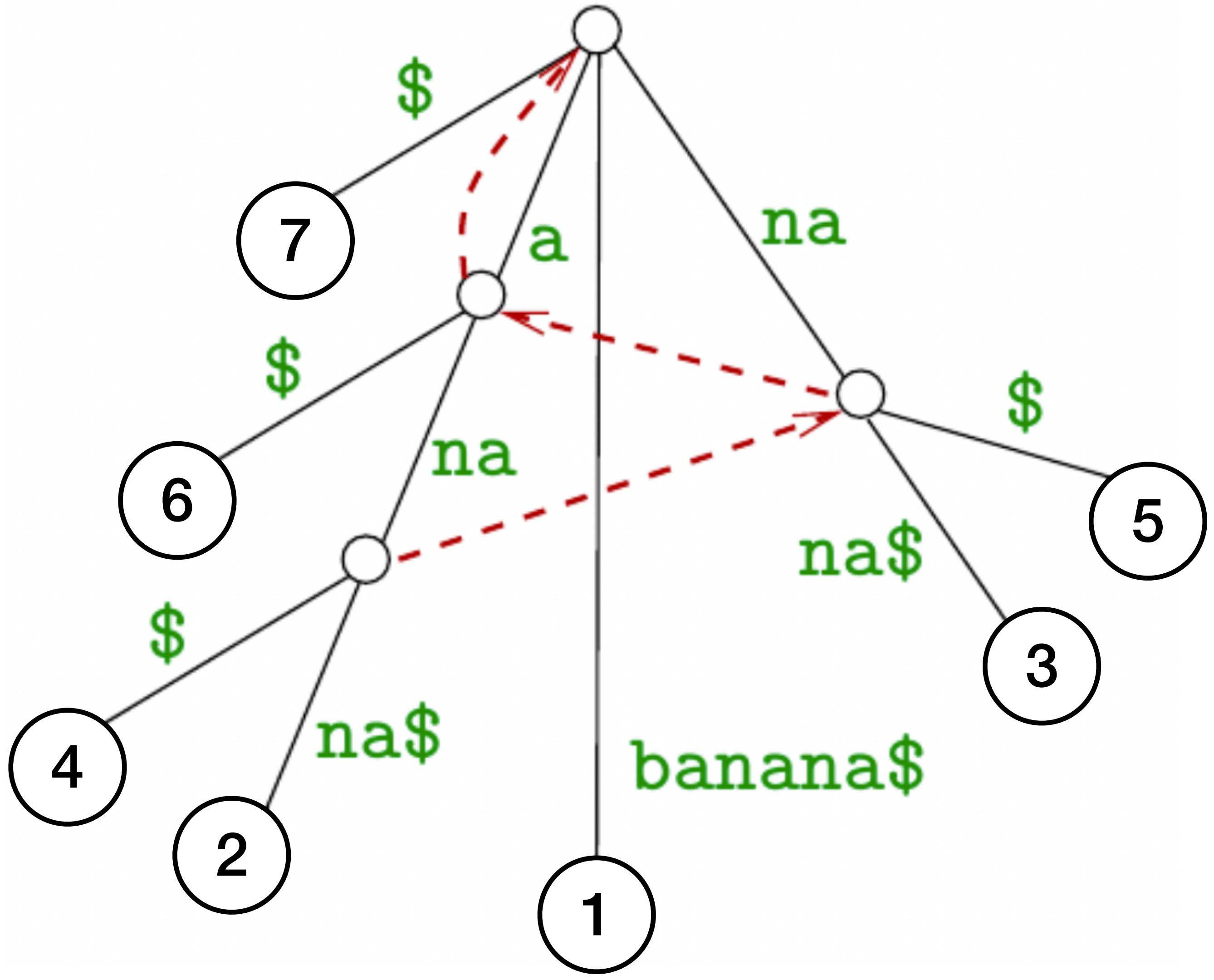
Suffix links

The key to efficient suffix tree construction are suffix links:

For an explicit node u, slink(u) is the node v such that S_v is the longest proper suffix of S_u, i.e., if $S_u = T[i..j]$ then $S_v = T[i+1..j]$.

For example, let T = banana$.

The suffix links are represented by the red arrows.



Suffix links

Suffix links are well defined for all nodes except the root. For the root, we define slink(root) = root.

Lemma 1. If the suffix tree of T has a node u representing T[i..j] for any $1 \leq i < j \leq n$, then it has a node v representing T[i+1..j]

Proof. If u is the leaf representing the suffix T_i , then v is the leaf representing the suffix T_{i+1}.

If u is an internal node, then it has two child edges with labels starting with different symbols, say *a* and *b*, which means that T[i..j]*a* and T[i..j]*b* are both substrings of T.

Then,T[i+1..j]*a* and T[i+1..j]*b* are substrings of T too, and thus there must be a branching node v representing T[i +1..j].

Suffix links

Suffix links are well defined for all nodes except the root. For the root, we define slink(root) = root.

Lemma 1. If the suffix tree of T has a node u representing T[i..j] for any $1 \leq i < j \leq n$, then it has a node v representing T[i+1..j]

Proof. If u is the leaf representing the suffix T_i, then v is the leaf representing the suffix T_{i+1}.

If u is an internal node, then it has two child edges with labels starting with different symbols, say *a* and *b*, which means that T[i..j]*a* and T[i..j]*b* are both substrings of T.

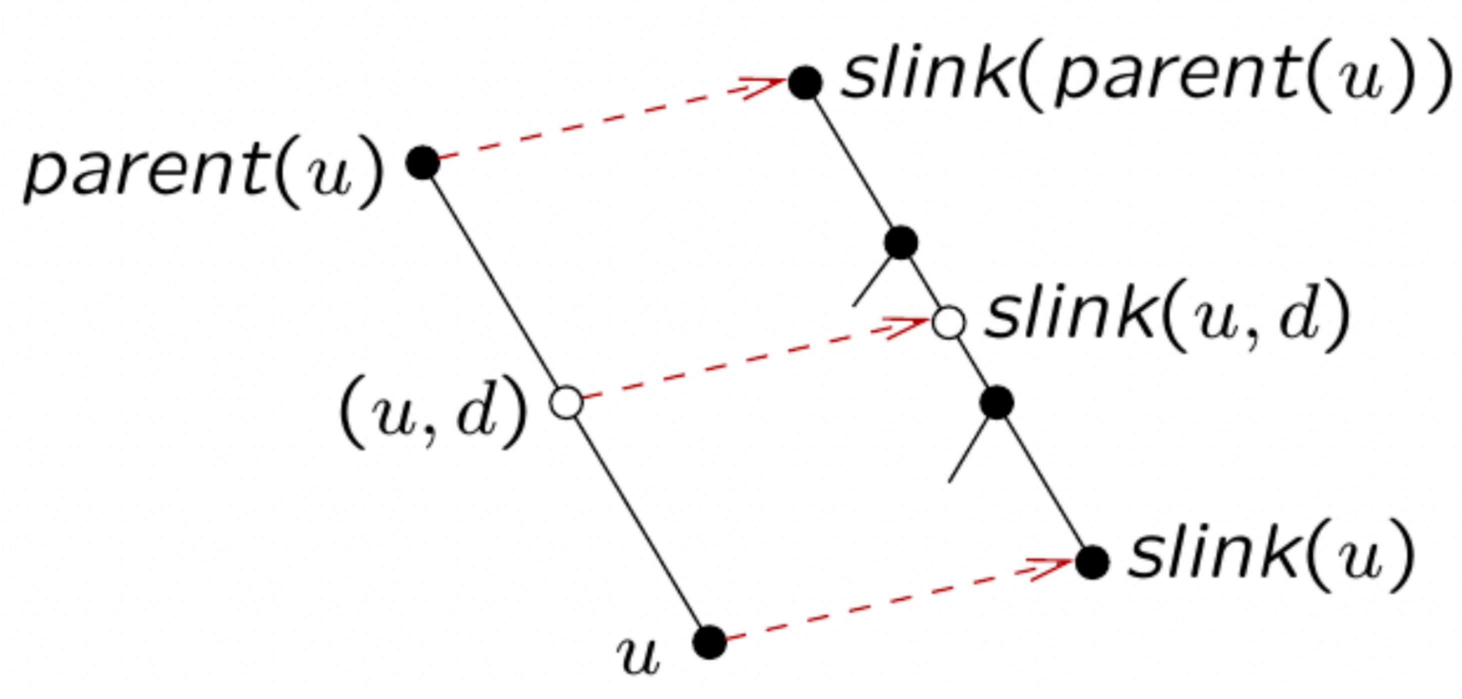
Then,T[i+1..j]*a* and T[i+1..j]*b* are substrings of T too, and thus there must be a branching node v representing T[i +1..j].

This lemma ensures that depth(slink(u)) = depth(u)-1.

Suffix links

Suffix links are stored for branching nodes only, but we can define and compute them for any locus (u,d):

```
SuffixLink(u, d)
    1 v←slink(parent(u));
    2 **while** depth(v) < d-1 **do**
        3 v←child(v,T[start(u)+depth(v)+1]);
    4 **return**(v,d-1);
```



Suffix links

The same idea can be used for computing the suffix links during or after the brute force construction:

```
ComputeSuffixLink(u)
    1 d←depth(u);
    2 v←slink(parent(u));
    3 while depth(v) < d-1 do
        4 v←child(v,T[start(u)+depth(v)+1]);
    5 if depth(v)>d-1 then
        6 v←CreateNode(v,d-1);
    7 slink(u)←v;
```

Suffix links

The same idea can be used for computing the suffix links during or after the brute force construction:

```
ComputeSuffixLink(u)
    1 d←depth(u);
    2 v←slink(parent(u));
    3 **while** depth(v) < d-1 **do**
        4 v←child(v,T[start(u)+depth(v)+1]);
    5 **if** depth(v)>d-1 **then**
        6 v←CreateNode(v,d-1);
    7 slink(u)←v;
```

Lines (5)-(6) execute if there is no node at (v,d-1). The procedure CreateNode(v, d-1) sets slink(v) = ε. The algorithm uses the suffix link of the parent of u, which must have been computed before.

Suffix links

The creation of a new node on line (6) never happens in a fully constructed suffix tree, but during the brute force algorithm the necessary node may not exist yet.

If a new internal node u_i was created during the insertion of the suffix T_i, there exists an earlier suffix T_j, with $j < i$, that branches at u_i into a different direction than T_i .

Then slink(u_i) represents a prefix of T_{j+1} and thus it exists at least as a locus on the path labelled T_{j+1}. However, it may happen that it does not become a branching node until the insertion of T_{i+1}.

In such a case, ComputeSuffixLink(u_i) creates the slink(u_i) node a moment before it would otherwise be created by the brute force construction.

Linear-Time Construction of the Suffix Tree

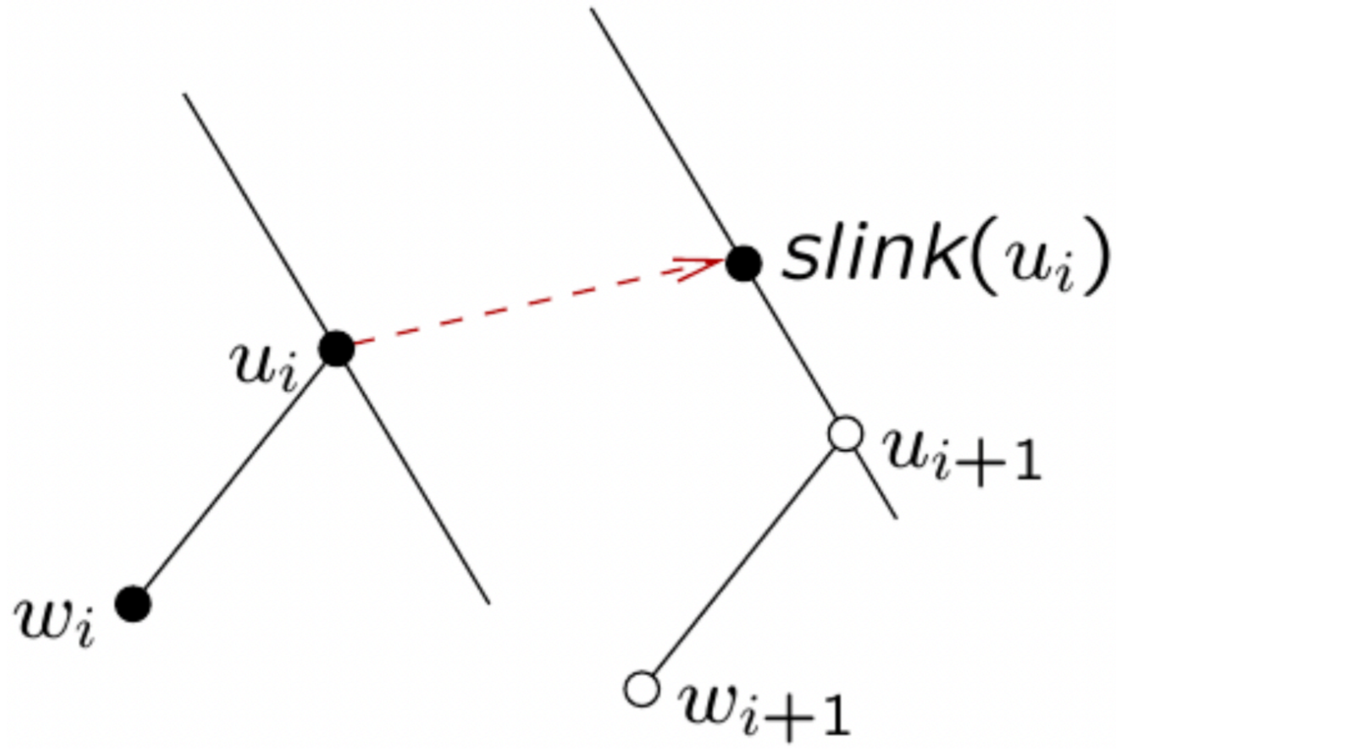
Reference1: Chapters 6.3 and 6.5 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*
Reference2: Original paper by McCreight (in the material of the course)

This set of slides was partially adapted from the slides of Juha Kärkkäinen [https://www.cs.helsinki.fi/u/tpkarkka/teach/15-16/SPA/lecture08.pdf]

McCreight's Construction Algorithm

McCreight's suffix tree construction is a simple modification of the brute force algorithm that computes the suffix links during the construction and uses them as shortcuts.

Say that we have just added a leaf w_i representing the suffix T_i as a child to a node u_i . The next step is to add w_{i+1} as a child to a node u_{i+1}. The brute force algorithm finds u_{i+1} by traversing the partially constructed suffix tree from the root; McCreight's algorithm takes a shortcut to slink(u_i). This is safe because slink(u_i) represents a prefix of T_{i+1}!



McCreight’s Construction Algorithm

```
McCreight-Construction(T[1 . . n])
  1 root←EmptyNode();
  2 depth(root)←0;
  3 (u,d)←(root,0);
  4 slink(root)←root;
  5 for i=1,…,n do
      6 while d=depth(u) and child(u,T[i+d]) ≠ ε do
          7 (u,d)←( child(u,T[i +d]) , d+1 );
          8 while d < depth(u) and T[start(u) + d] = T[i + d]) do
              9 d←d+1;
      10 if d < depth(u) then
          11 u←CreateNode(u, d);
      12 w←CreateLeaf(i,u,d);
      13 if slink(u) is empty then
          14 slink(u)←ComputeSuffixLink(u);
      15 (u,d)←(slink(u),max{d-1,0});
```

McCreight's Construction Algorithm

Theorem. Let T be a string of length n over an alphabet of constant size. McCreight's algorithm computes the suffix tree of T in O(n) time.

Proof. The insertion of a suffix T_i takes *O*(1) time except in two points:

1. The while loops on lines (6)-(9), that spell all characters on the path from the node slink(u_{i-1}) to u_i. Every round increments the string depth d. The only place where d decreases is on line (15) and even then by one (because of Lemma 1). Since d can never exceed n, the total time on lines (6)-(9) is O(n) (with an aggregate analysis).

McCreight’s Construction Algorithm

2. The subroutine ComputeSuffixLink(u). The while loop at lines (3)-(4) checks exactly one character for each explicit node on the path from slink(parent(u)) to slink(u), thus its time complexity is proportional to the total number of explicit nodes it visits.

Let NN(u) denote the number of explicit nodes on the path from the root to a node u. It clearly holds that NN(p(u))=NN(u)-1; and NN(slink(u))≥NN(u)-1.

In ComputeSuffixLink(u) we start traversing from slink(p(u)), so it holds that NN(slink(p(u))≥NN(u)-2. Other than that, NN only increases at each call of ComputeSuffixLink. The total number of explicit nodes visited over all calls of ComputeSuffixLink is bounded by the total decrease plus the total increase in NN.

There are n steps, so the total decrease is at most 2n; and the total increase is at most 3n, because NN(u)≤n for all nodes u. Thus all the calls to ComputeSuffixLink require O(n) time in total.

Implementation Issues

The main design issue of any suffix tree construction algorithm is how to represent and search the branches out of the nodes. We must balance the space required and the need for speed in searching, both in building the tree and in using it afterwards. There are four basic choices possible to represent branches.

- An array of size $\Theta(|\Sigma|)$ at each nonleaf branching node v. The array at v is indexed by the characters of Σ: the entry of a character x stores a pointer to the child of v whose edge label starts with x (if it exists), and the two indices of T representing the edge label. With this representation, searching an edge requires O(1) time; but it requires $\Theta(|\Sigma|)$ space per node, which is impractical for large alphabets/long texts.

Implementation Issues

- A linked list for each nonleaf branching node v, containing the first character of the edge label of each branch out of v. In the construction, whenever a new edge from v is added to the tree, a new character is added to the list. The search of a character is done sequentially from the beginning of the list. The characters in these lists can be maintained in lexicographic order, which speeds up the search in practice, but it still requires $O(|\Sigma|)$ time per every node operation in the worst case. On the other hand, there is no waste of space.

- Some sort of balanced tree for each nonleaf branching node v. If v has k children, then, adding a new edge and searching cost $O(\log k)$ time, and the space required is $O(k)$. The drawback is that balanced trees have space and programming overhead, and thus in practice they are only convenient when k is very large.

Implementation Issues

- Some sort of (perfect) hashing techniques for each nonleaf branching node.

In practice, for large alphabets and long texts, a mixture of the above options is the best choice, e.g., using arrays for the nodes close to the root (which typically have many children) and sorted lists for the deeper nodes.

Use of the Suffix Tree

Using the Suffix Tree: Longest Repeating Factor

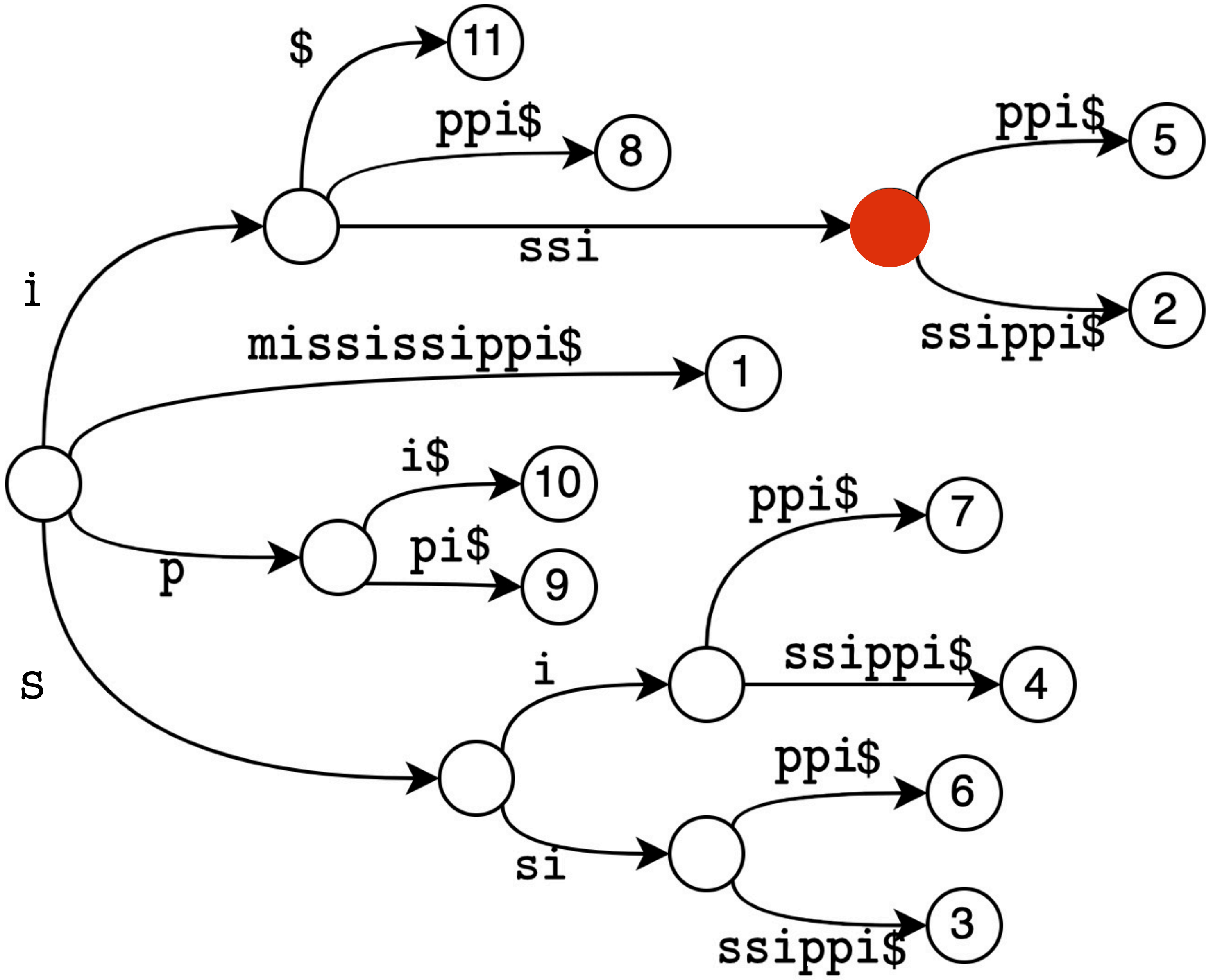
The longest repeating factor of a text T is the longest substring that occurs at least twice in T. It is represented by the deepest branching node in the suffix tree.

Using the Suffix Tree: Longest Repeating Factor

The longest repeating factor of a text T is the longest substring that occurs at least twice in T. It is represented by the deepest branching node in the suffix tree.

The longest repeating factor of T=mississippi$ is “issi”.

Exercise. Write pseudocode for a solution to this problem, and analyse its time complexity.



Using the Suffix Tree: Number of distinct substrings

Input: a text T

Output: the number of distinct substrings of T

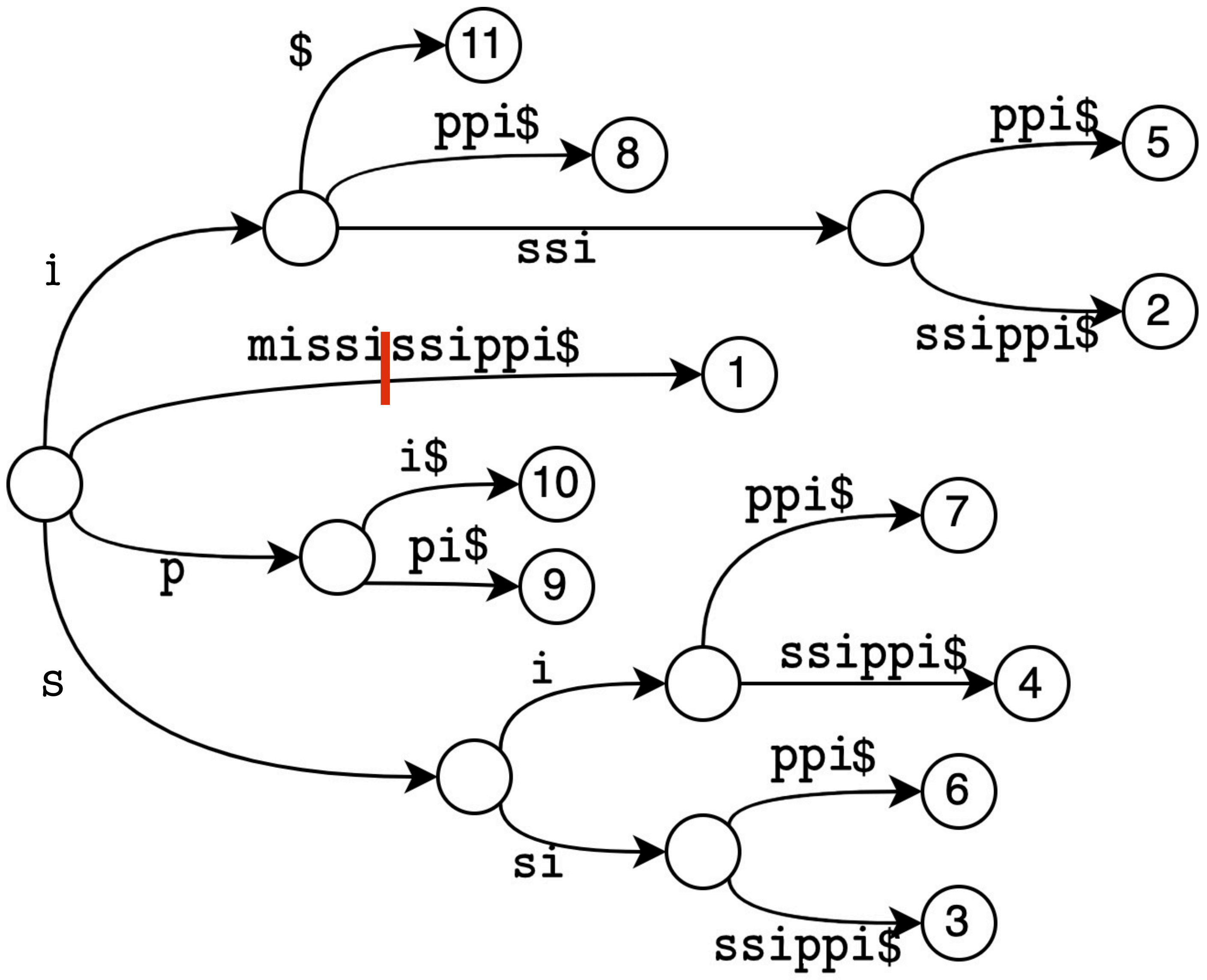
Using the Suffix Tree: Number of distinct substrings

Input: a text T

Output: the number of distinct substrings of T

Every locus (node, depth) in the suffix tree represents a substring of the text; and every substring is represented by some locus.

E.g., locus (1,5) represents the substring “missi”



Using the Suffix Tree: Number of distinct substrings

Input: a text T

Output: the number of distinct substrings of T

It suffices to count the number of distinct loci using any traversal (e.g., DFS or BFS) of the suffix tree.

Exercise. Write pseudocode for the solution described above. What is its time complexity?

Using the Suffix Tree: Longest Common Prefix

Problem: preprocess a text T of length n so that the following queries can be answered efficiently.

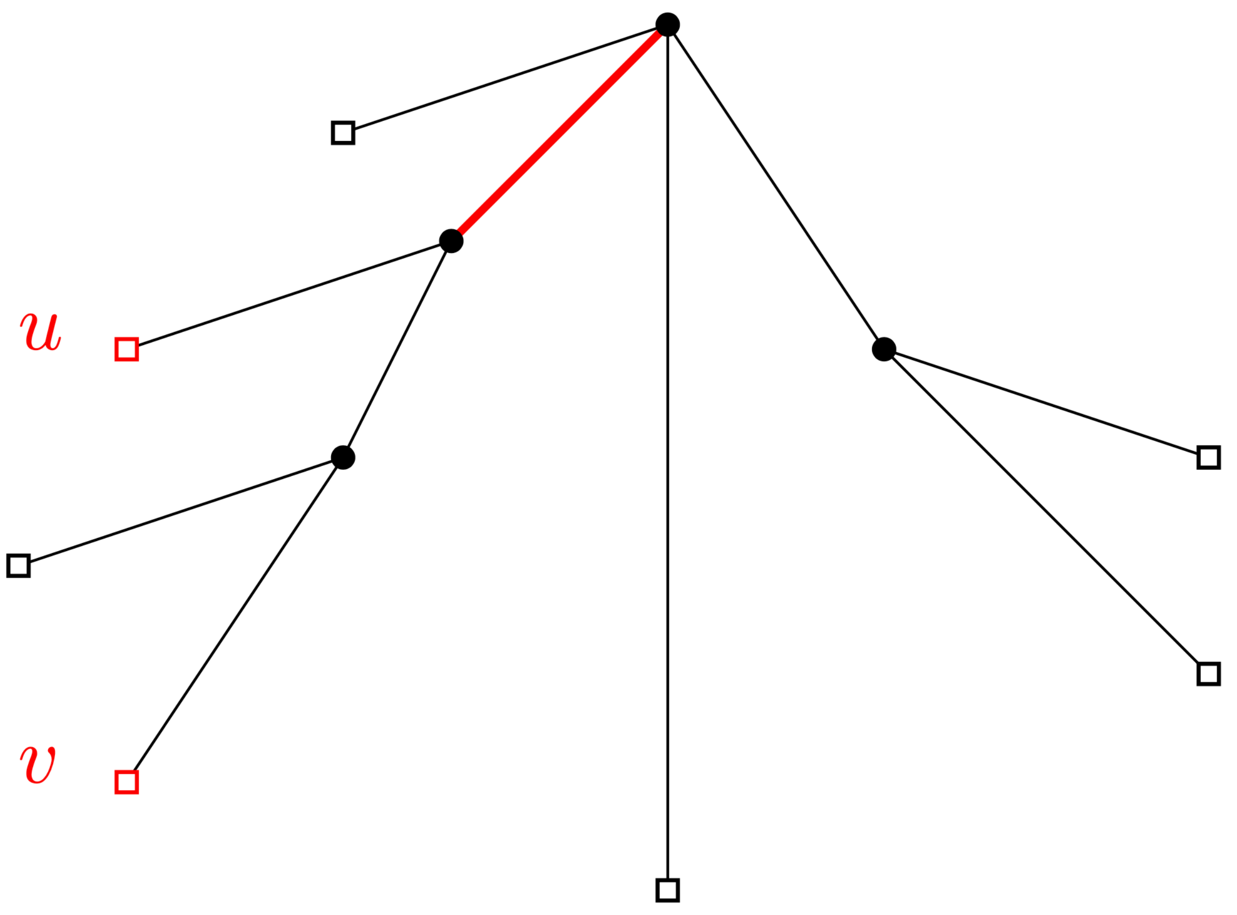
Query: given a pair (i,j), return the longest common prefix of T[i..n] and T[j..n]

Using the Suffix Tree: Longest Common Prefix

Problem: preprocess a text T of length n so that the following queries can be answered efficiently.

Query: given a pair (i,j), return the longest common prefix of T[i..n] and T[j..n]

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.



Using the Suffix Tree: Longest Common Prefix

Problem: preprocess a text T of length n so that the following queries can be answered efficiently.

Query: given a pair (i,j), return the longest common prefix of T[i..n] and T[j..n]

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.

Theorem (Bender and Farach-Colton). Any tree of size O(N) can be preprocessed in O(N) time so that the LCA of any two nodes can be computed in O(1) time.

Using the Suffix Tree: Longest Common Prefix

Problem: preprocess a text T of length n so that the following queries can be answered efficiently.

Query: given a pair (i,j), return the longest common prefix of T[i..n] and T[j..n]

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.

Theorem (Bender and Farach-Colton). Any tree of size O(N) can be preprocessed in O(N) time so that the LCA of any two nodes can be computed in O(1) time.

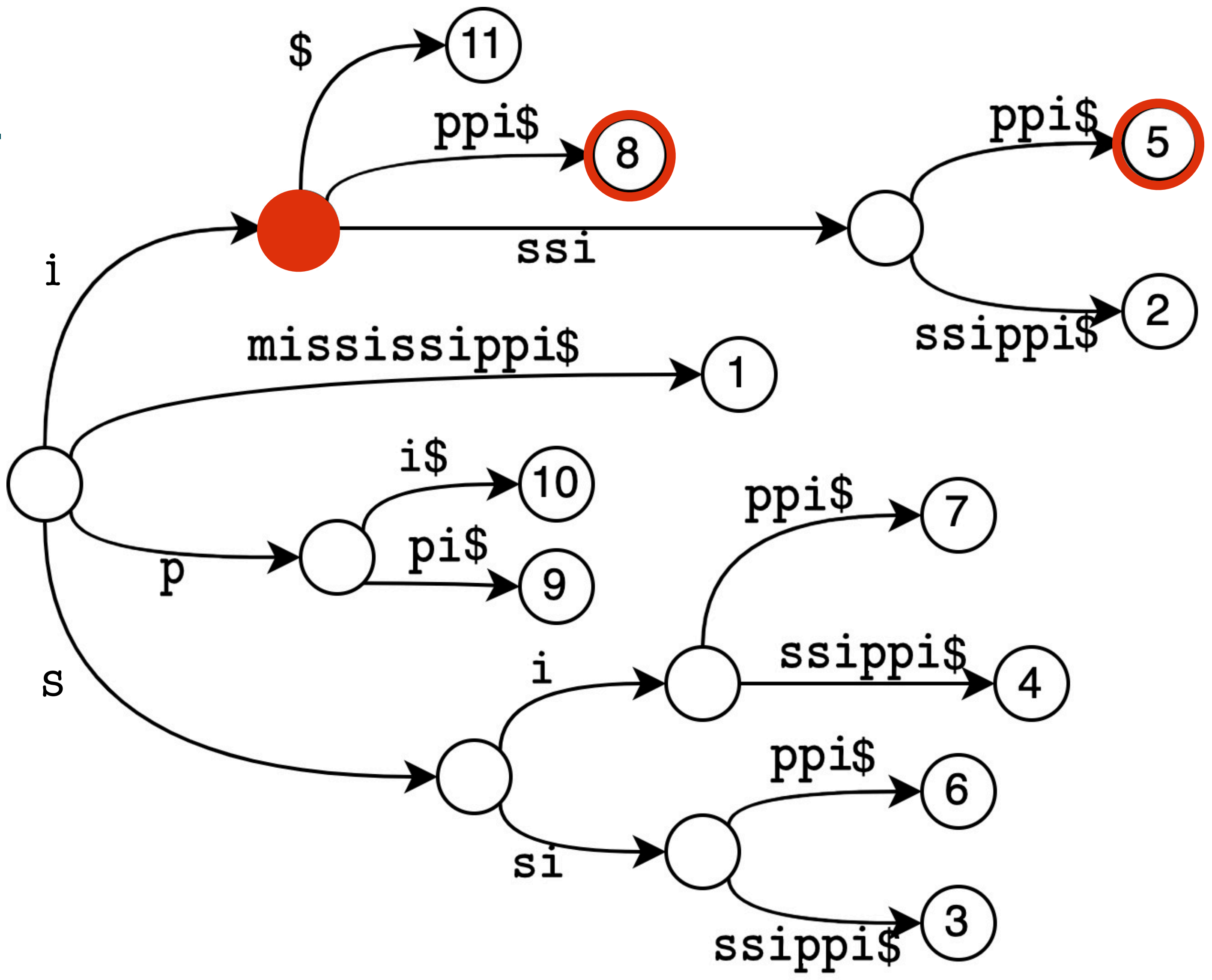
Theorem. Longest Common Prefix queries in T can be answered in O(1) time after O(n) time preprocessing of the suffix tree of T.

Using the Suffix Tree: Longest Common Prefix

Problem: preprocess a text T of length n so that the following queries can be answered efficiently.

Query: given a pair (i,j), return the longest common prefix of T[i..n] and T[j..n]

For T=mississippi$, let (5,8) be the query. The answer is “i”, which is the path label of the LCA of leaves 5 and 8.



The Generalised Suffix Tree

Reference: Chapter 6.4 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*

Generalised Suffix Tree for a Set of Strings

The concept of suffix tree of a string can be easily extended to a set of strings.

The generalised suffix tree of a set of strings $S_1,S_2,\ldots,S_k$ is the compacted trie of all the suffixes of all the strings in the set.

To build it, it suffices to build the suffix tree of their concatenation $S_1\$_1S_2\$_2\ldots S_k\$_k$, where $\$_1,\$_2,\ldots,\$_k$ are distinct terminal symbols.

Generalised Suffix Tree for a Set of Strings

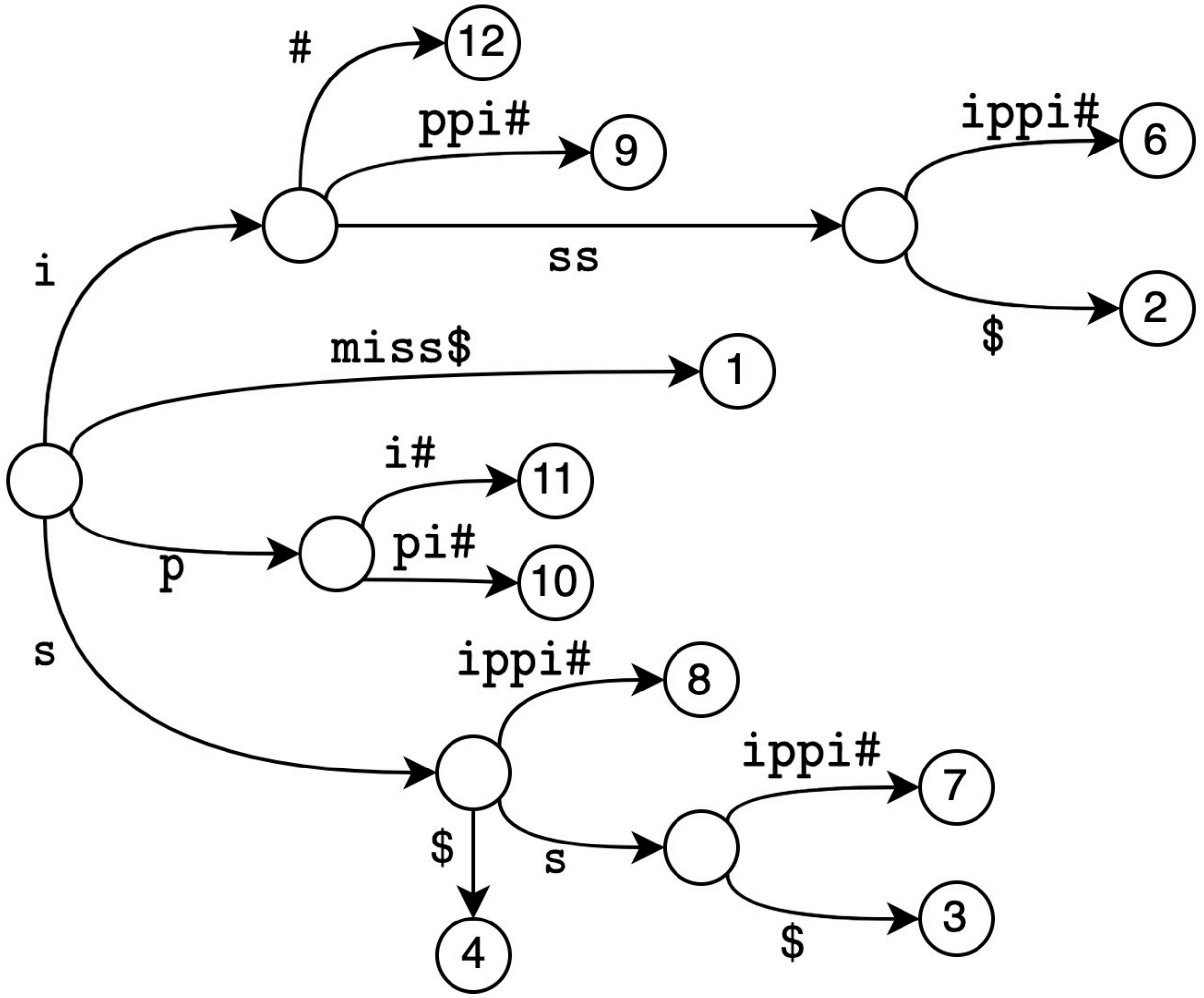
The concept of suffix tree of a string can be easily extended to a set of strings.

The generalised suffix tree of a set of strings $S_1, S_2, \ldots, S_k$ is the compacted trie of all the suffixes of all the strings in the set.

To build it, it suffices to build the suffix tree of their concatenation $S_1\$_1S_2\$_2\ldots S_k\$_k$, where $\$_1, \$_2, \ldots, \$_k$ are distinct terminal symbols.

S_1=miss$

S_2=issippi#



Generalised Suffix Tree for a Set of Strings

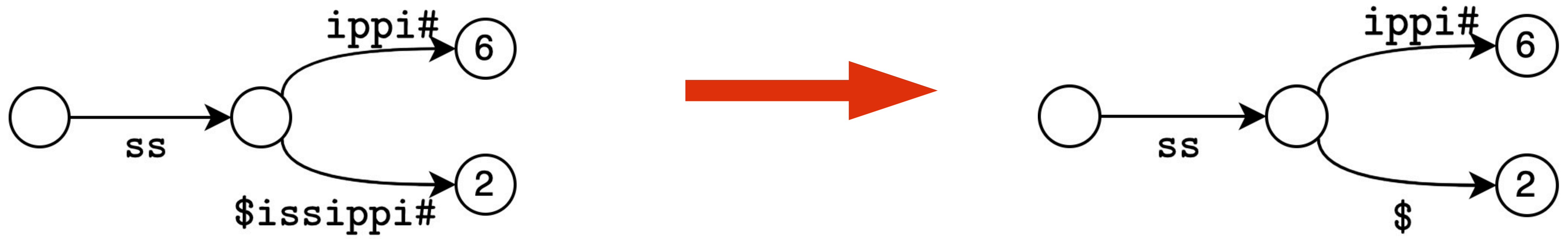
Building the suffix tree of $S_1\$_1S_2\$_2\ldots S_k\$_k$, requires time linear in the sum of the lengths of the strings in the set.

The suffix tree built in this way, though, contains also spurious substrings that span more than one input string.

For example, the concatenation miss$issippi# contains the substring ss$issippi#.

However, because each terminal symbol is distinct and is not in any of the original strings, the label on any path from the root to a branching node must be a substring of one of the original strings.

To remove these spurious substrings it suffices to truncate the labels of the branches ending at the leaves to the first terminal symbol.



Use of the Generalised Suffix Tree

The generalised suffix tree of a set of strings can obviously be used to search for one or more patterns in all strings of the set, i.e., to solve the exact pattern matching problem in a database.

To do so, we spell the pattern(s) from the root, much like we do with the suffix tree of one string.

Use of the GST: Longest Common Substring

The Longest Common Substring (LCS) of two strings S and T is the longest substring that occurs both in S and in T.

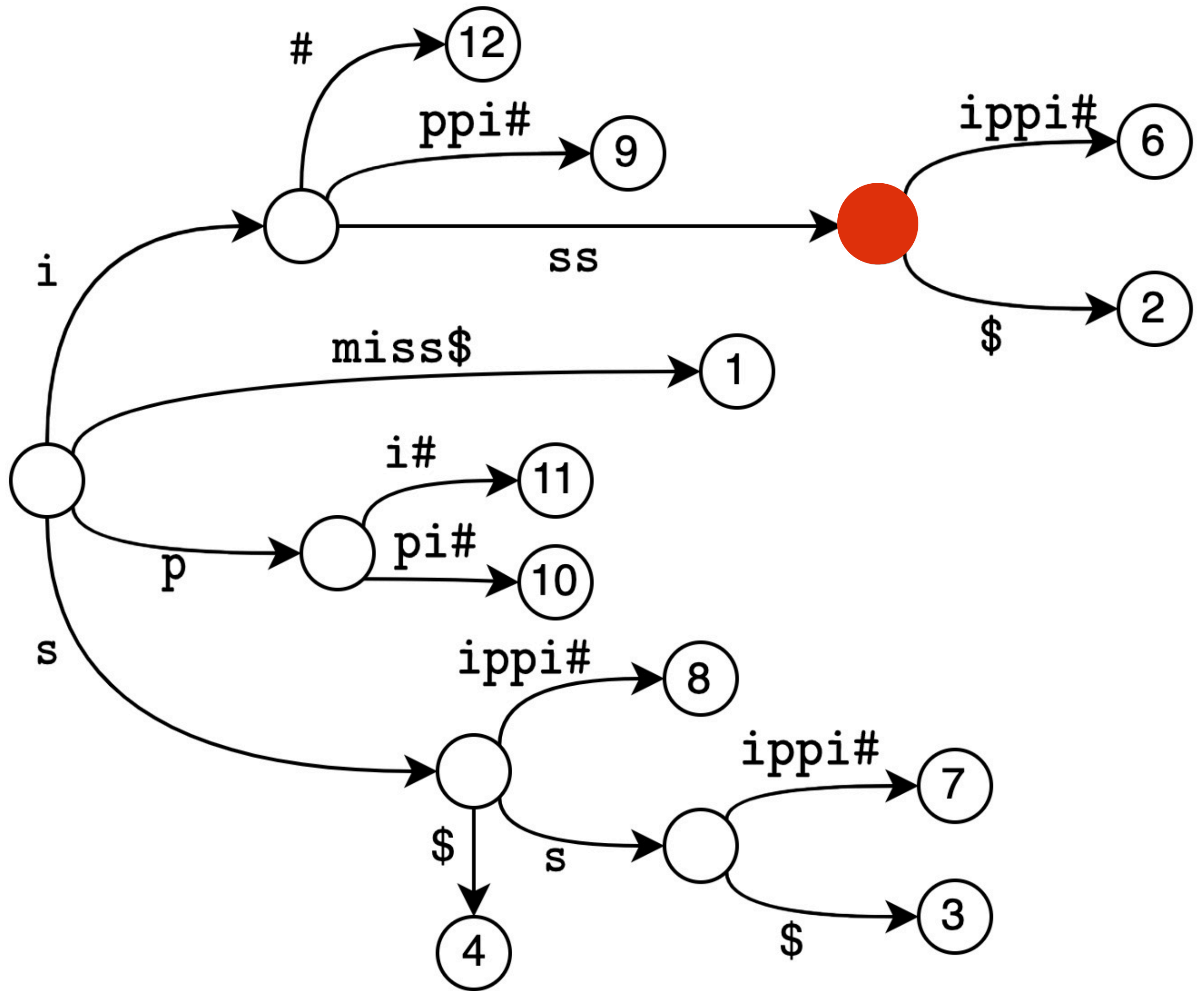
It is represented by the deepest branching node in the suffix tree that have at least a descending leaf corresponding to S and at least a descending leaf corresponding to T.

Use of the GST: Longest Common Substring

The Longest Common Substring (LCS) of two strings S and T is the longest substring that occurs both in S and in T.

It is represented by the deepest branching node in the suffix tree that have at least a descending leaf corresponding to S and at least a descending leaf corresponding to T.

The LCS of “miss” and “issipi” is “iss”



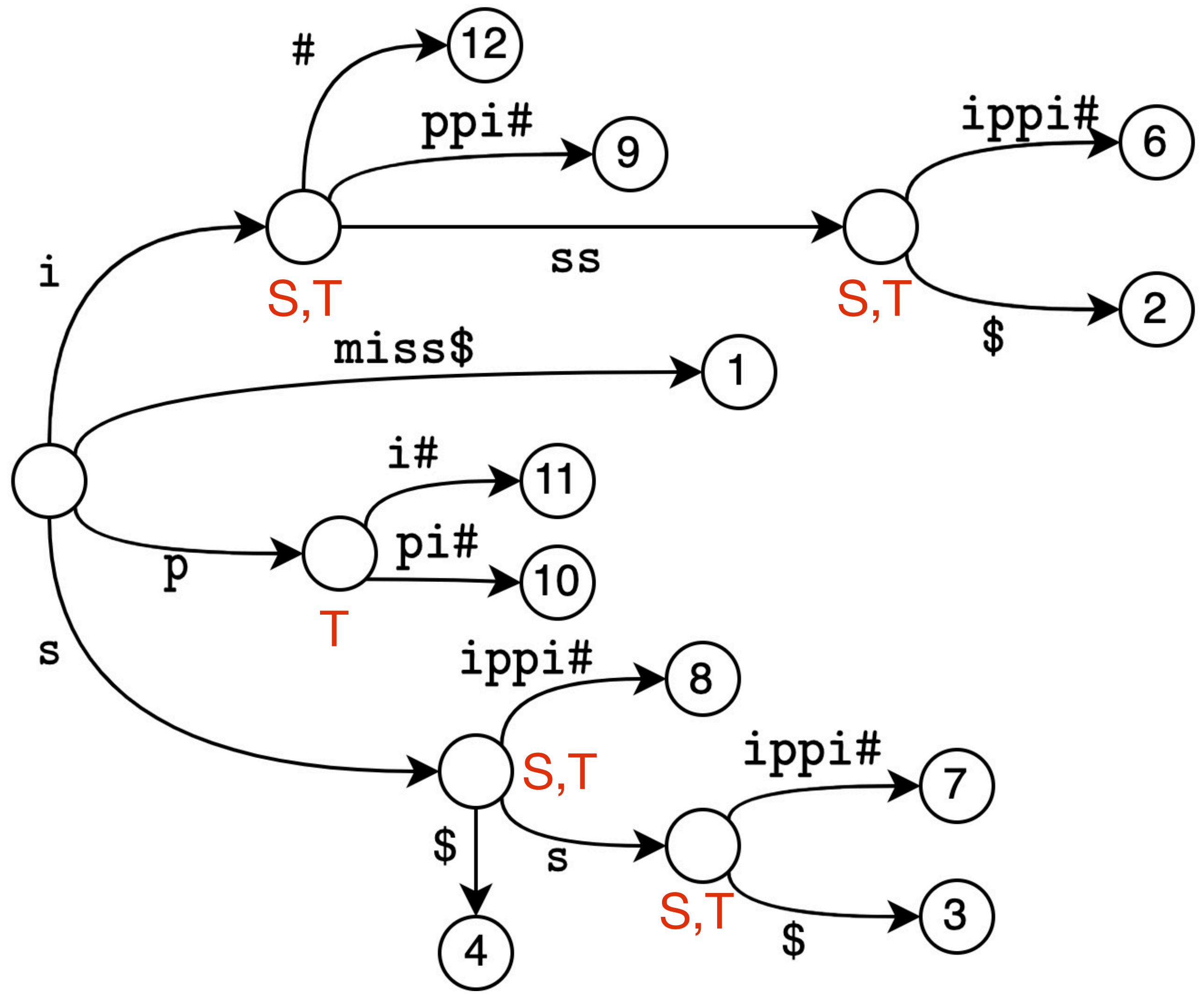
Use of the GST: Longest Common Substring

The LCS of S and T can be found in O(|S|+|T|) time by:

- preprocessing the GST of S and T to mark each branching node with the strings corresponding to the leaves descending from there. This can be done traversing the GST bottom-up.
- Picking the deepest node marked with both S and T. This can be done with a DFS.

S=miss$

T=issippi#



#
12
ppi#
9
ippi#
6
i
ss
S,T
S,T
2
$
miss$
1
i#
11
pi#
10
p
T
s
ippi#
8
S,T
ippi#
7
$
s
S,T
3
$
4