Multiple String Matching

Giulia Bernardini
giulia.bernardini@units.it

Fundamentals of algorithms
a.y. 2021/2022

From pattern to text preprocessing

Both the Knuth-Morris-Pratt and the Boyer-Moore-Galil algorithms have a worst-case optimal running time, and Boyer-Moore-Galil in particular has even a sublinear expected running time.

Both these methods work by preprocessing the pattern. Many real-world applications, though, require to search for multiple patterns in the same text.

From pattern to text preprocessing

Both the Knuth-Morris-Pratt and the Boyer-Moore-Galil algorithms have a worst-case optimal running time, and Boyer-Moore-Galil in particular has even a sublinear expected running time.

Both these methods work by preprocessing the pattern, thus their running time can be split Many real-world applications require to search for multiple patterns in the same text. This is known as the multiple pattern matching problem.

Input: a text T and a set of k patterns $P_1,\ldots,P_k$

Output: all occurrences of each of the k patterns in T

From pattern to text preprocessing

Solving the multiple pattern matching problem by applying KMP or Boyer-Moore-Galil to each pattern individually would require preprocessing each of them and then scanning the text once for each of them: thus $O(\sum_{i=1}^{k} |P_k| + k|T|)$ time overall.

The text is the same for all pattern, so why should we read all its characters k times?

From pattern to text preprocessing

The general idea is to preprocess the text instead of the patterns.

The problem of preprocessing a text to allow efficient multiple pattern matching is called text indexing.

The fundamental data structure for text indexing is the suffix tree.

Suffix Trees

Reference: Chapter 5 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*

Tries

A simple but powerful data structure for a set of strings is the trie. It is a rooted tree with the following properties:

- Edges are labelled with symbols from an alphabet Σ
- For every node v, the edges from v to its children have different labels

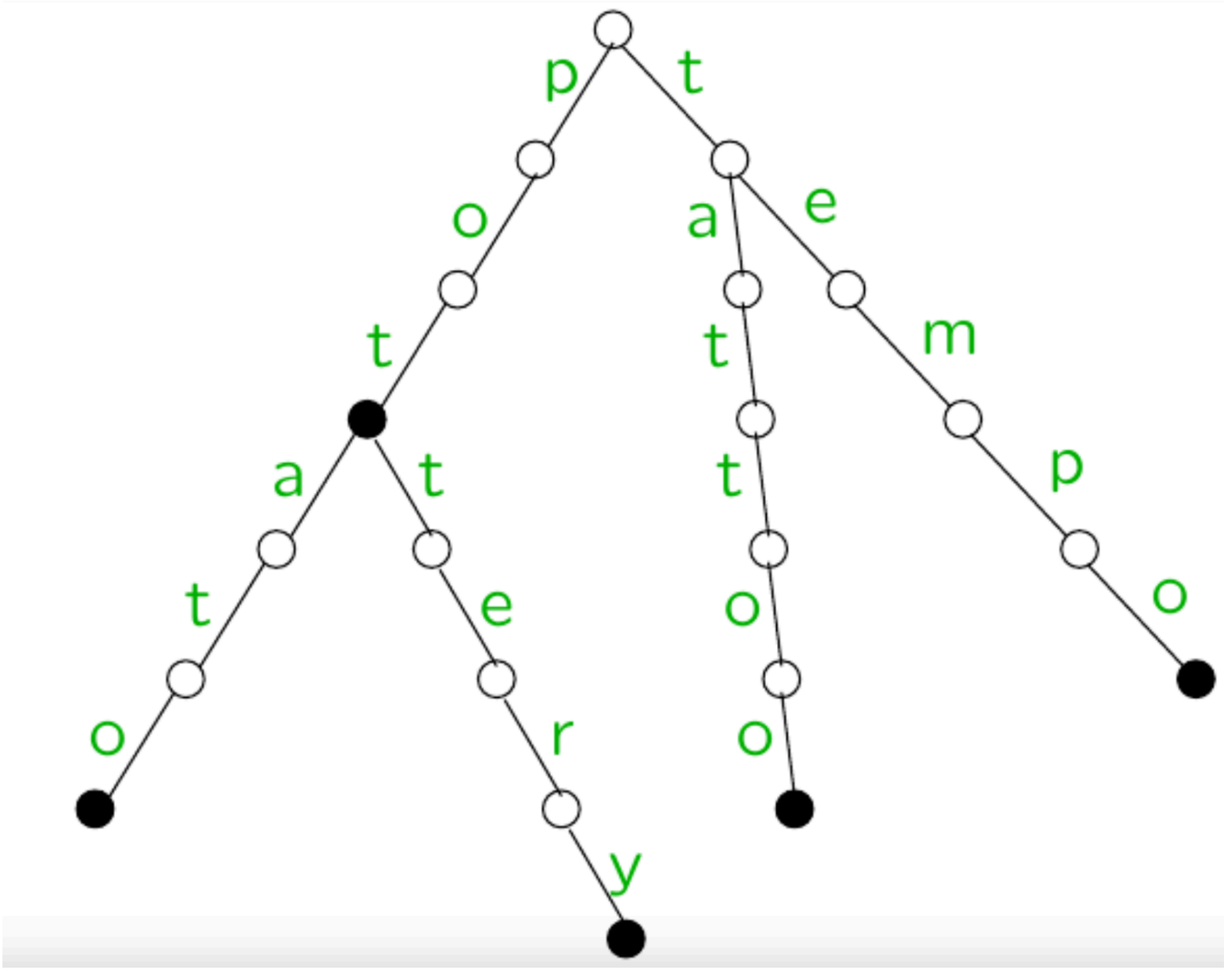
Each node represents the string obtained by concatenating the symbols on the path from the root to that node.

The trie for a string set R, denoted by trie(R), is the smallest trie that has nodes representing all the strings in R. The nodes representing strings in R may be marked.

Tries: an example

Let R={pot, potato, pottery, tattoo, tempo}

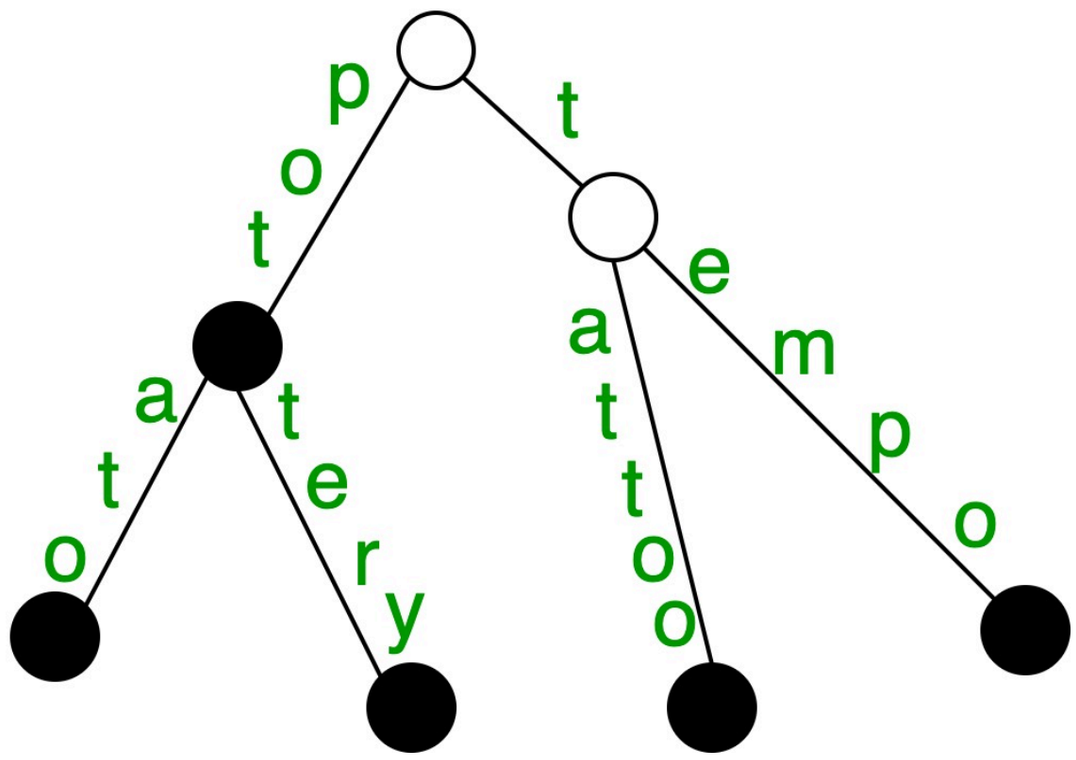
Trie(R) is represented below. Black nodes mark the end of the strings in R.



Tries: an example

Let R={pot, potato, pottery, tattoo, tempo}

Trie(R) is represented below. Black nodes mark the end of the strings in R. A compacted trie has edges labelled by strings instead of letters, and no nodes with just one child.



Definition of Suffix Tree

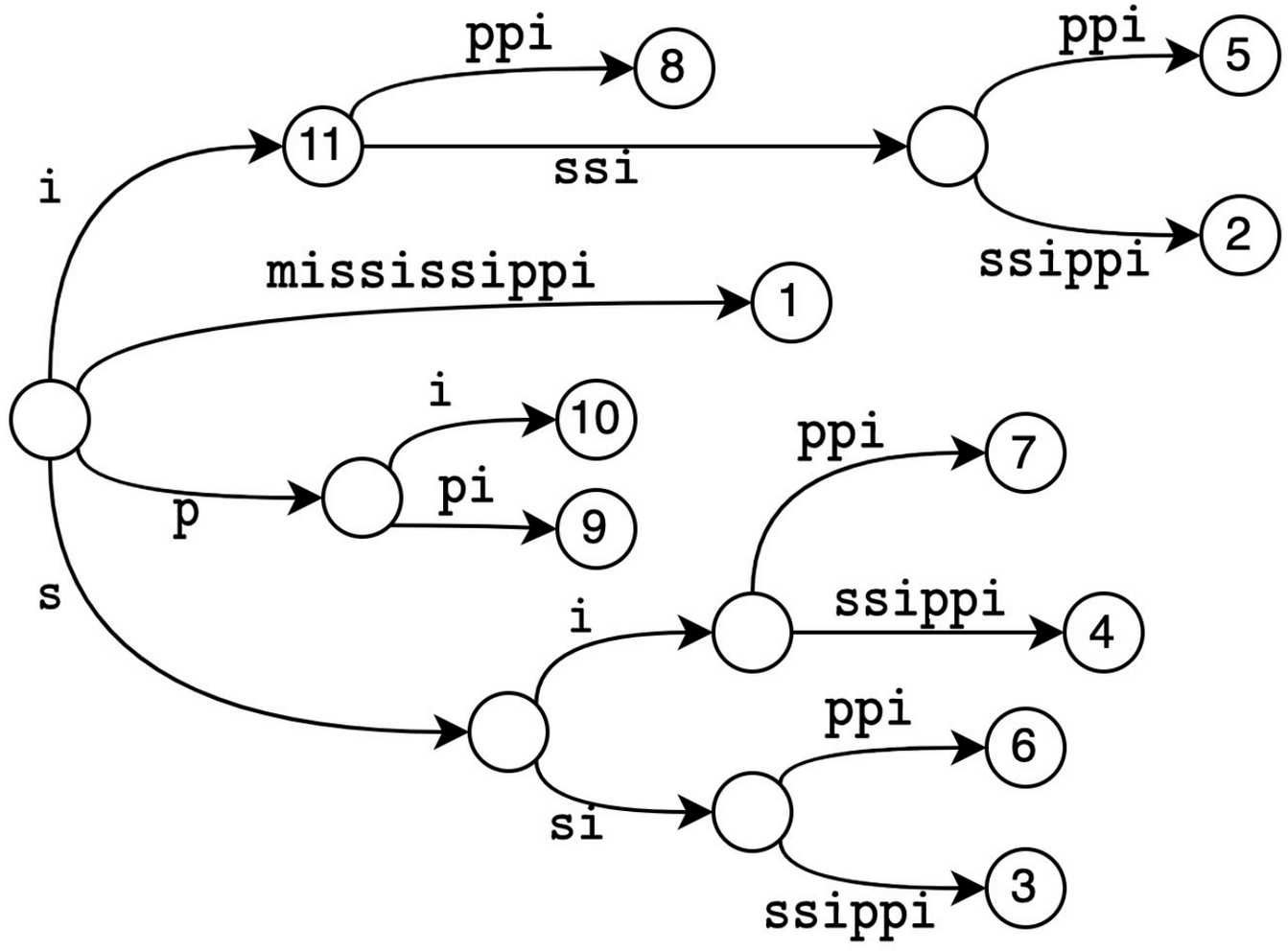
Consider a text $T=T[1..n]$ and denote by T_i the suffix $T[i..n]$.

Furthermore, for any set $C \subseteq [1,\ldots,n]$, we write $T_C=\{T_i : i \in C\}$.

The suffix tree of T is the compact trie of $T_{[1,\ldots,n]}$, that is, the compact trie of all its suffixes.

Definition of Suffix Tree

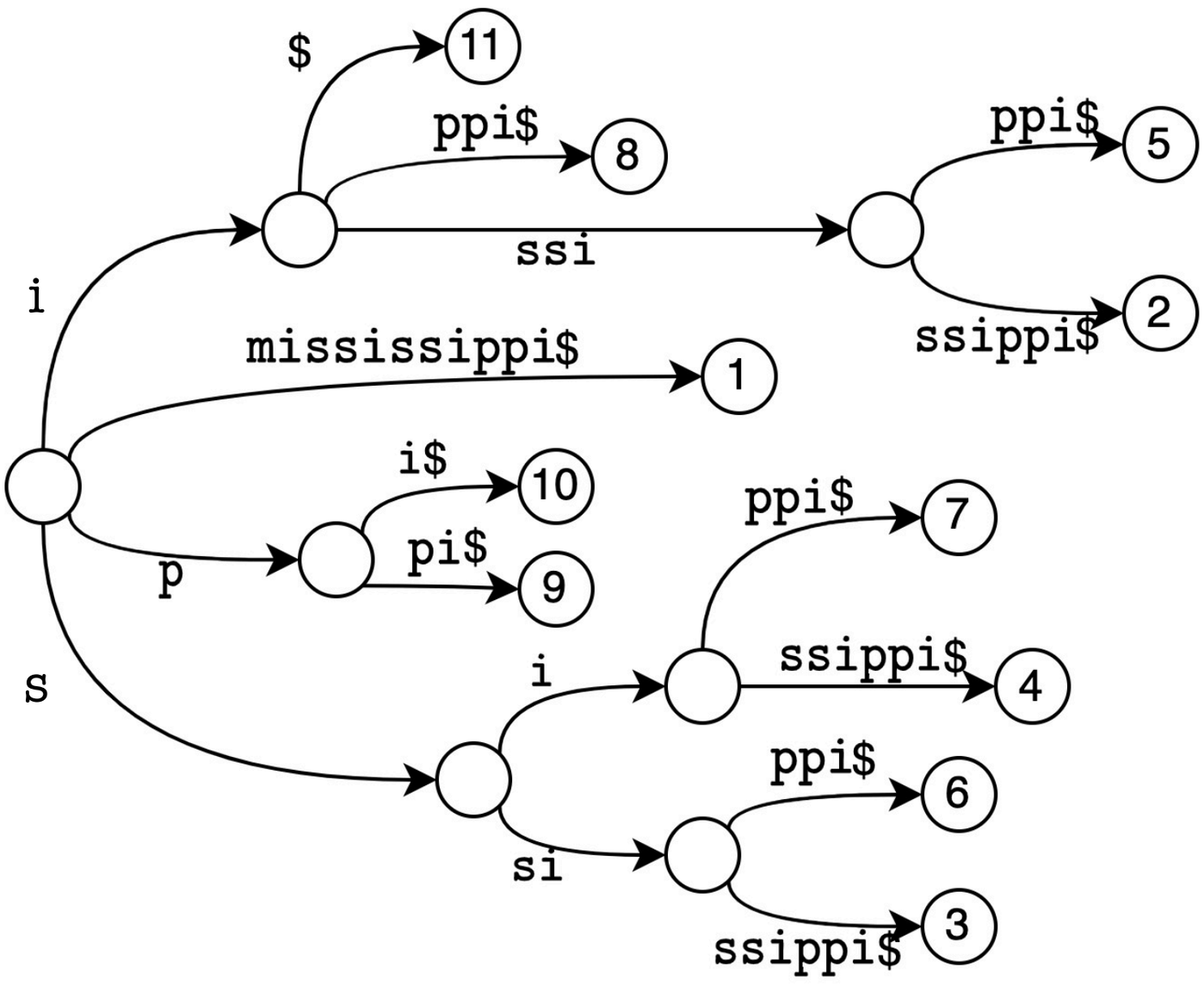
Let T=mississippi. Its suffix tree is pictured below. The number in a node represents the starting position in T of the suffix represented by the node.



Note that node 11 is not a leaf. This happens because suffix “i” is also the prefix of some other suffix (for example “ippi”)

Definition of Suffix Tree

For constructing the suffix tree, it is desirable that all the terminal nodes are leaves. That's why it is standard to add an extra letter $ $\notin \Sigma$ at the end of the string, and to construct the suffix tree of this extended string. The suffix tree of T=mississippi$ is



Definition of Suffix Tree

The suffix tree of T supports fast exact string matching on T, because a pattern P has an occurrence in T starting at position i if and only if P is a prefix of T_i.

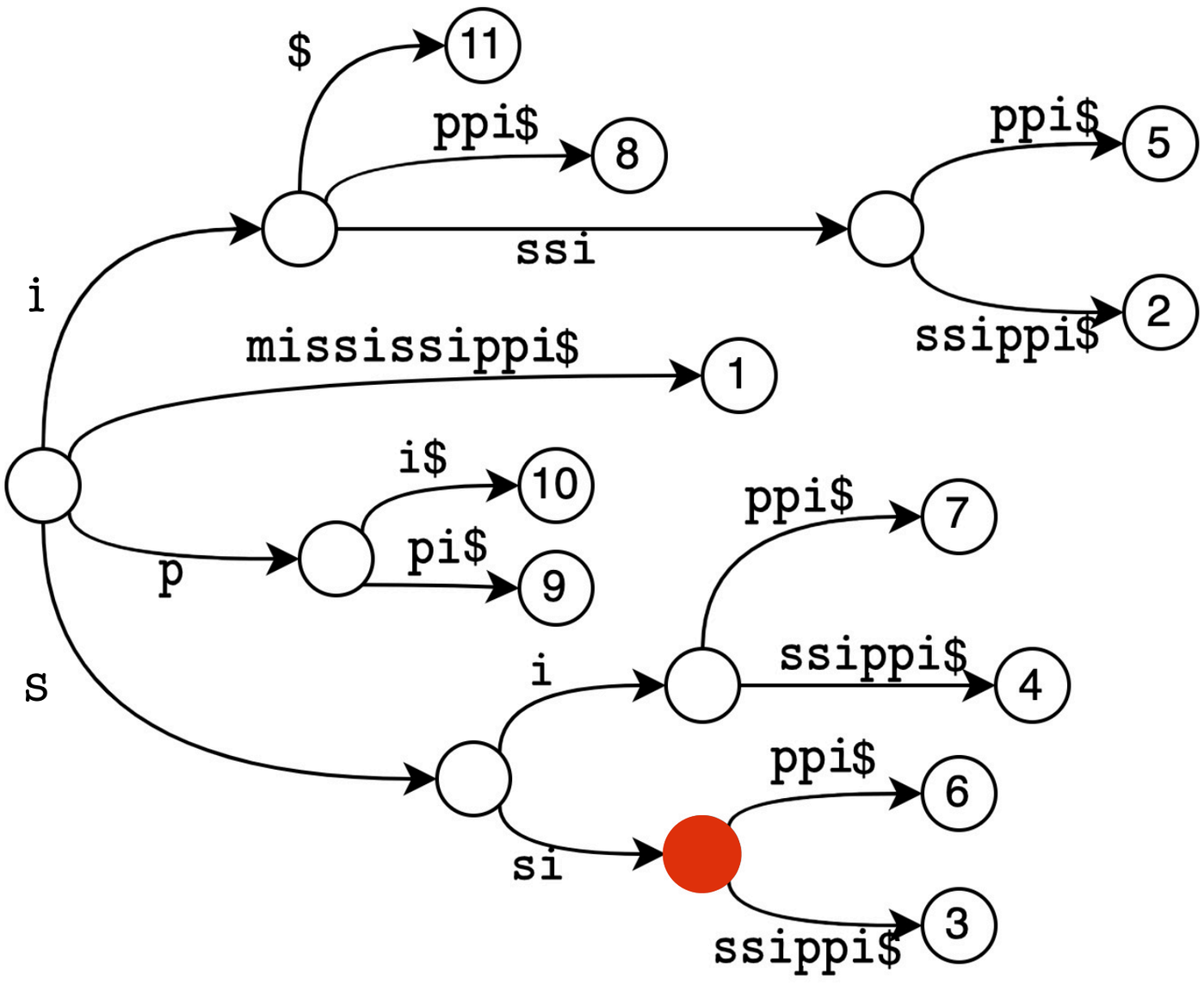
For example, pattern P=sissi occurs in T=mississippi at position 4, and it is also a prefix of the suffix T_4=T[4..11].

The label of a path from the root to any point in the suffix tree is the concatenation, in order, of the strings labelling the edges in the path. The path label of a node u of the suffix tree is the label of the path from the root to u.

The key observation is that the paths starting from the root represent all the prefixes of the suffixes of T.

Definition of Suffix Tree

In the suffix tree of T=mississippi$, for example, follow the path from the root to the red node, with path label “ssi”. Pattern “ssi” occurs in T at positions 3 and 6, as a prefix of suffixes T_3 and T_6. Note that these are exactly the labels of the leaves below the path.



Use of the Suffix Tree

Given a text T, a pattern P to be searched in T and the suffix tree of T, to find all the occurrences of P in T it suffices to match the characters of P starting from the root of T, until either:

1. P is exhausted. Then all the occurrences of P in T are identified by the leaves descending from the point where the last character of P was matched.
2. No more matches are possible and P is not exhausted. Then P does not occur anywhere in T.

In both cases, the problem is solved by matching $O(|P|)$ characters and by visiting a subtree with a number of leaves equal to the number of occurrences of P in T, denoted by occ, which can be done in $O(occ)$ time. Thus matching P in T requires $O(|P|+occ)$ time.

Properties of the Suffix Tree

Now the natural question is: what is the time complexity of building the suffix tree of a text? And how much space does its representation take?

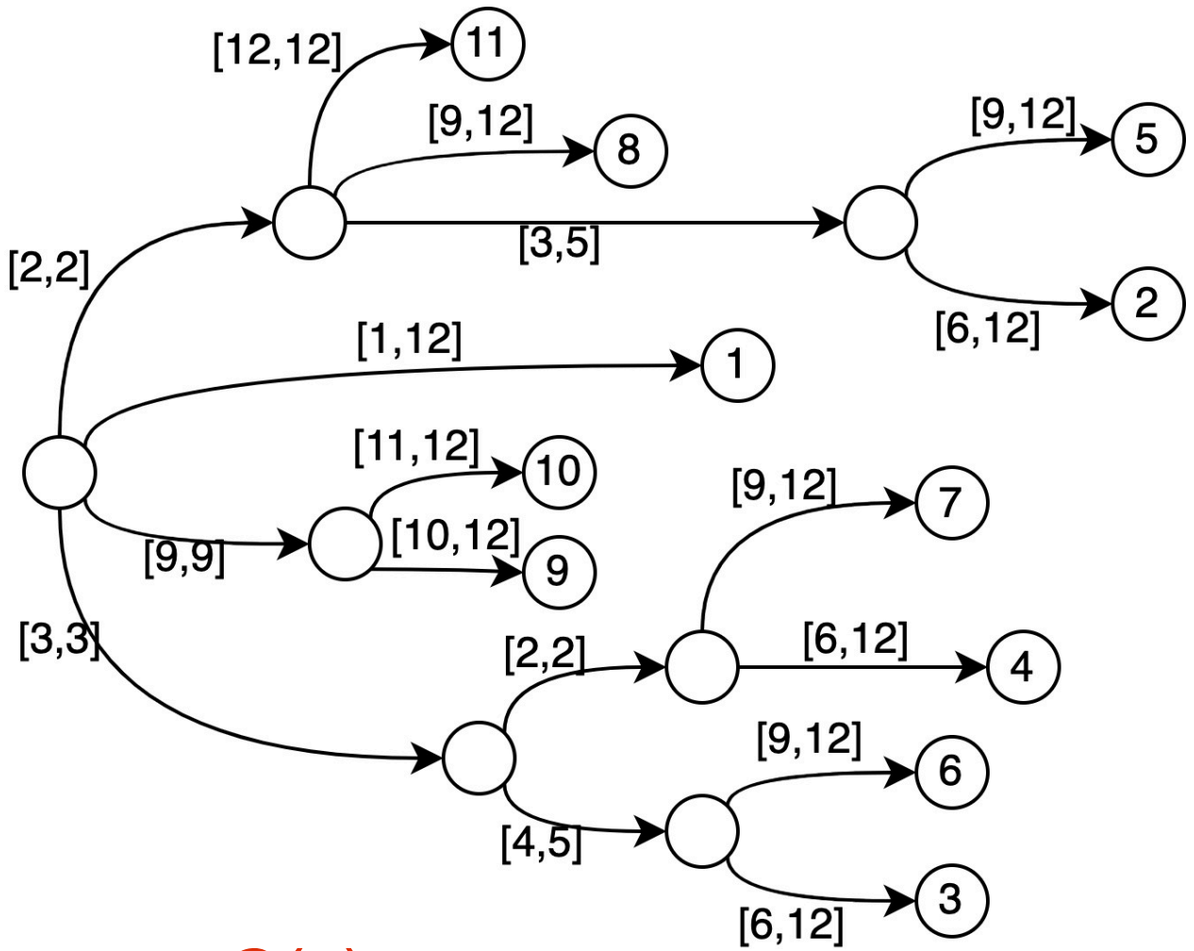
Let us first focus on the space. We need $O(1)$ space to represent a node or a character.

A string of length n has n suffixes (one for each starting position). There is one leaf for each suffix and the suffix tree has only branching nodes, thus there are $O(n)$ nodes in total, requiring $O(n)$ space to be represented.

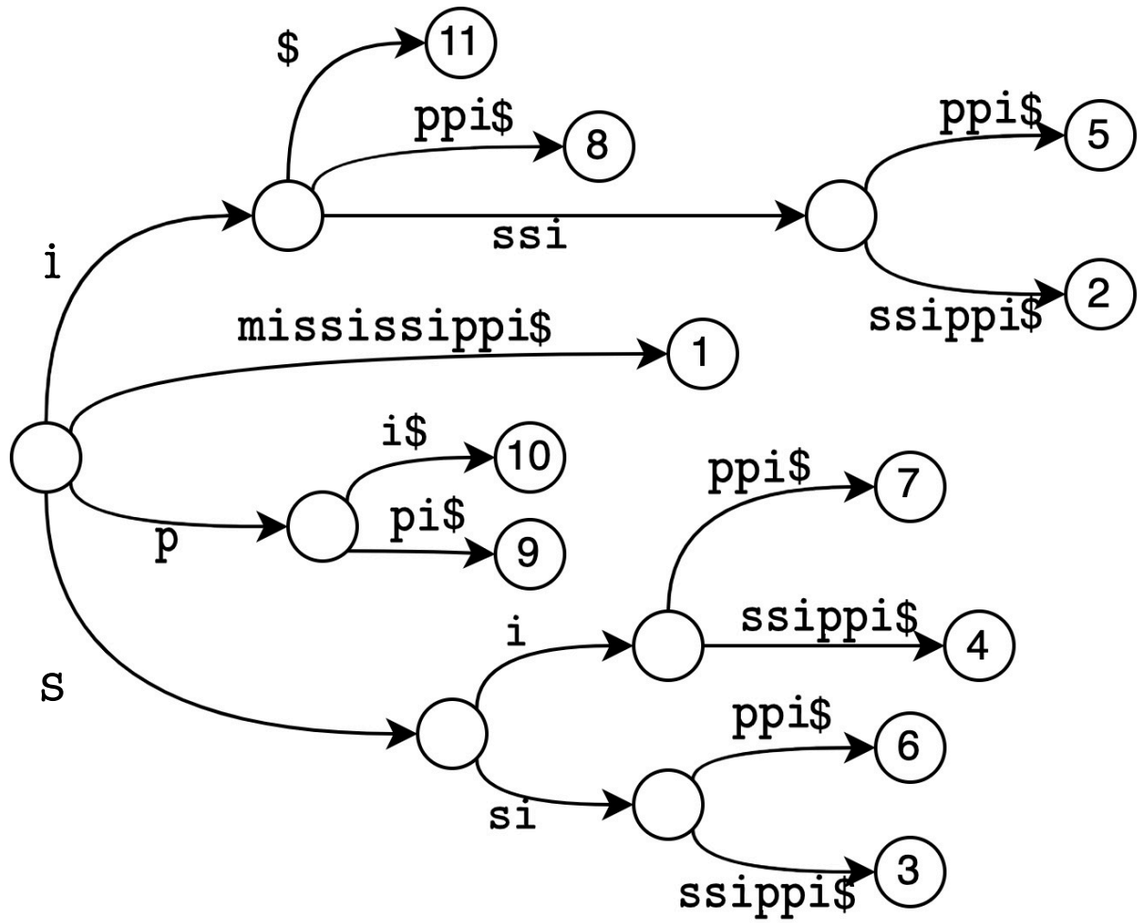
What about representing the edge labels? Every suffix appear in the suffix tree, and their total length is $1+2+3+\ldots+n-1+n=O(n^2)$. Thus it looks like we need $O(n^2)$ total space in the worst case, but…

Properties of the Suffix Tree

…but all the strings labelling the edges of the suffix tree of T are substrings of T. Thus each of them can be represented by an interval of positions over T. Representing one such interval requires *O*(1) space, and since the suffix tree has *O*(n) edges (because there are *O*(n) nodes) the whole representation requires *O*(n) space!



O(n) space



O(n^2) space

Properties of the Suffix Tree

For the construction, there are several linear-time algorithms, thus constructing the suffix tree of T in $O(|T|)$ time. We will see one such algorithms in details. We have obtained the following result:

Theorem. The problem of matching multiple patterns $P_1, P_2, \ldots, P_k$ in a text T can be solved in $O(\sum_{i=1}^{k} |P_k| + |T|)$ time using the suffix tree of T.

Constructing the Suffix Tree

We start with a brute force algorithm with time complexity $O(n^2)$, and later we will modify this algorithm to obtain a linear-time complexity.

Constructing the Suffix Tree

We start with a brute force algorithm with time complexity $O(n^2)$, and later we will modify this algorithm to obtain a linear-time complexity.

The idea is to add suffixes to the trie one at a time starting from the longest to the shortest suffix.

Constructing the Suffix Tree

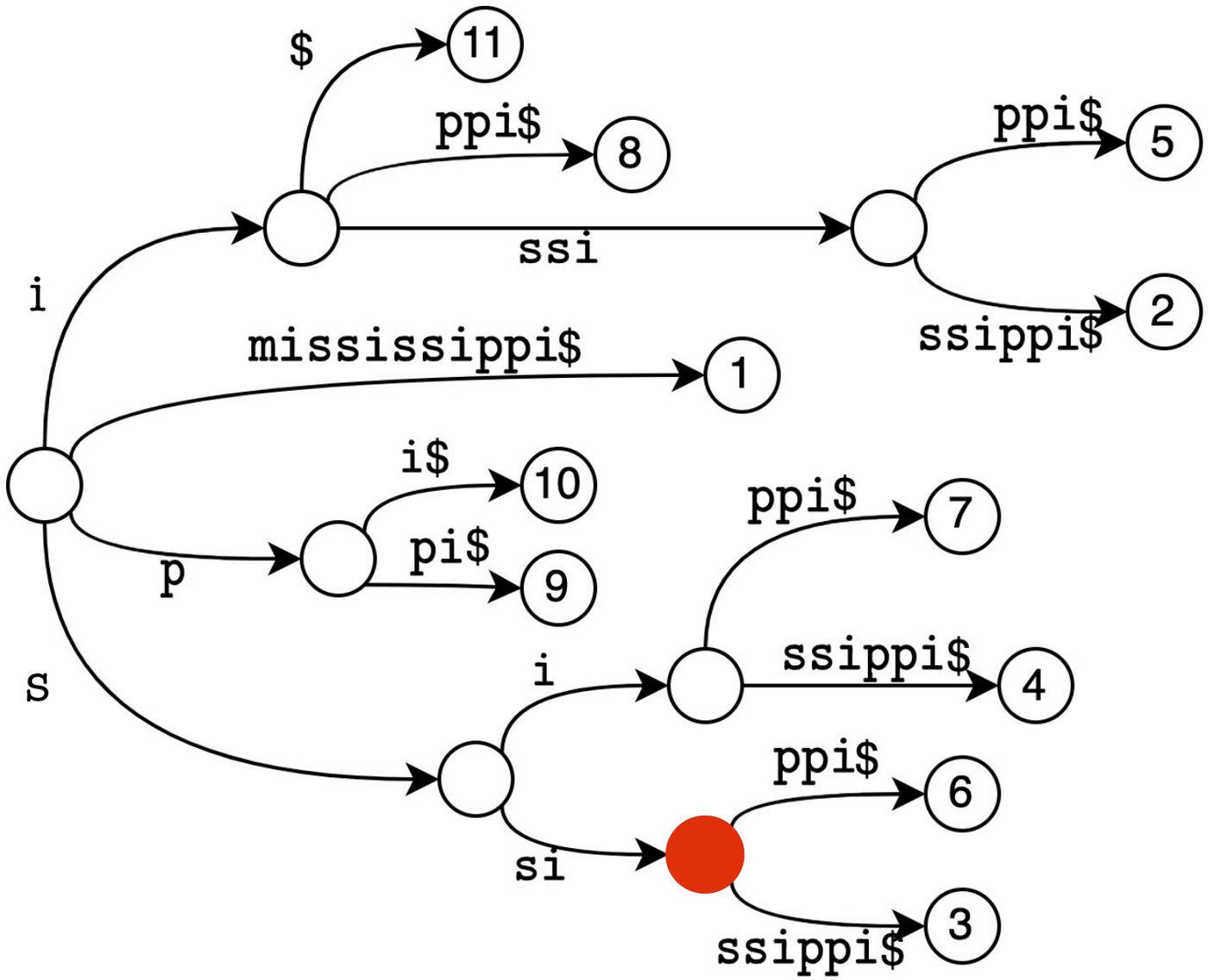
While constructing the suffix tree we will use the following notation.

- S_u denotes the path label of the node u, that is, the string represented by u in the suffix tree
- child(u,c) is the child v of node u such that the label of the edge (u,v) starts with the letter c, and ε if u has no such child
- parent(u) is the parent of u
- depth(u) is the length of S_u
- start(u) is the starting position of some occurrence of S_u in T

Constructing the Suffix Tree

In the suffix tree of mississippi$, if we call u the red node, we have

S_u = ssi; child(u,p)=6; child(u,m)=ε; depth(u)=3; start(u)=3 or (equivalently) start(u)=6.



Constructing the Suffix Tree

While constructing the suffix tree we will use the following notation.

- S_u denotes the path label of the node u, that is, the string represented by u in the suffix tree
- child(u,c) is the child v of node u such that the label of the edge (u,v) starts with the letter c, and ε if u has no such child
- parent(u) is the parent of u
- depth(u) is the length of S_u
- start(u) is the starting position of some occurrence of S_u in T

Then we have that $S_u = T[start(u)..start(u)+depth(u)-1]$, and the label of an edge (parent(u),u) will be

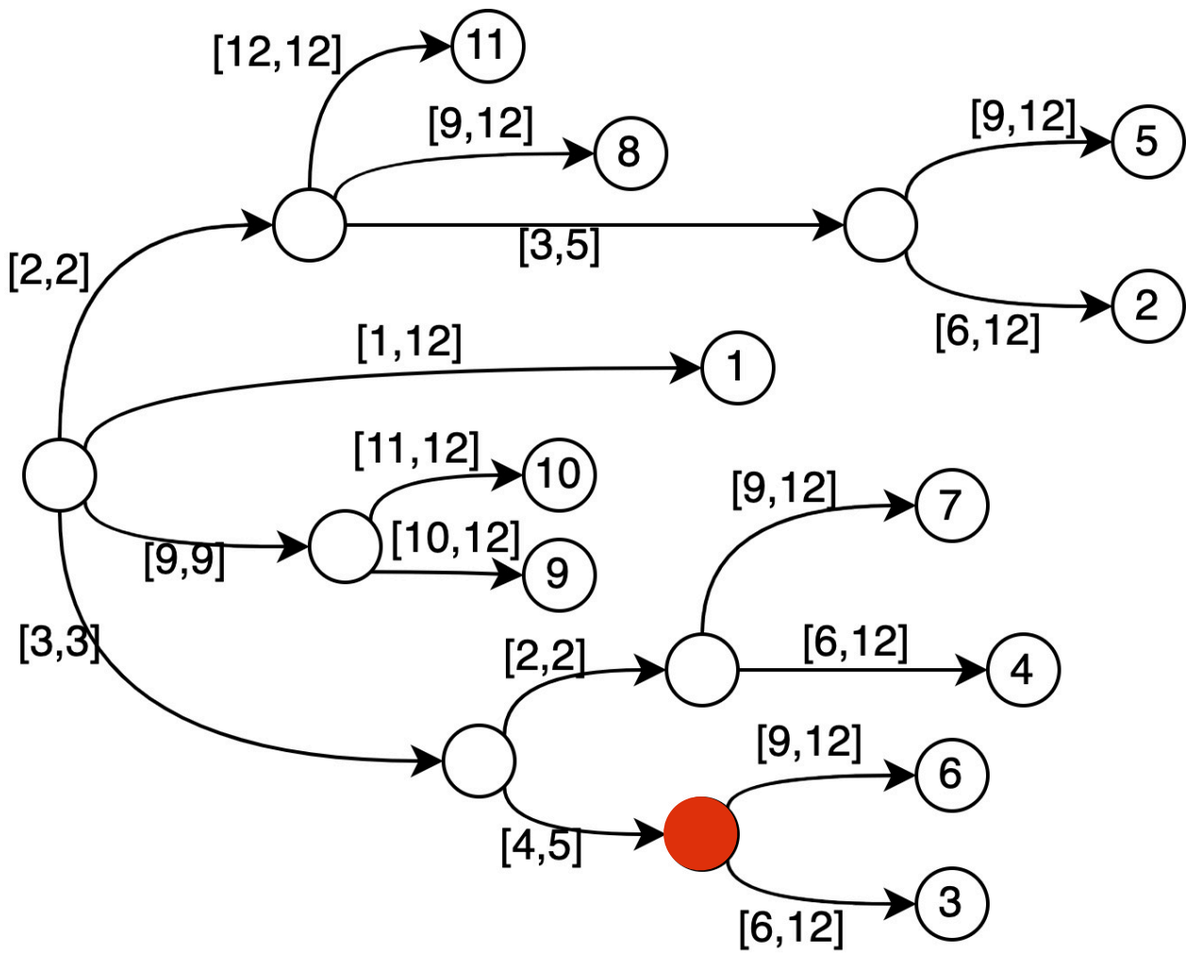
T[start(u)+depth(parent(u))..start(u)+depth(u)-1], which can be represented in $O(1)$ space by the interval

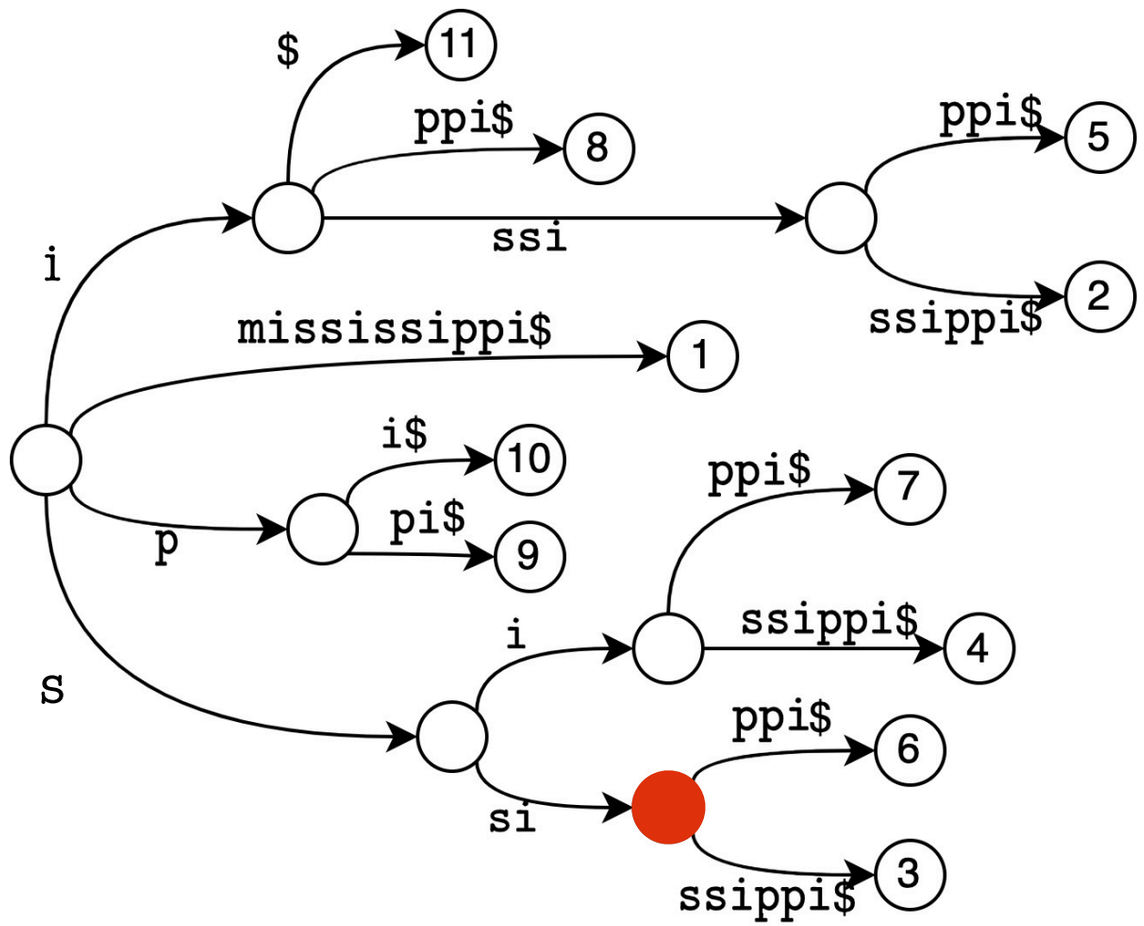
[start(u)+depth(parent(u))..start(u)+depth(u)-1]

Constructing the Suffix Tree

mississippi$

[start(u)+depth(parent(u))..start(u)+depth(u)-1]





Constructing the Suffix Tree

A locus in the suffix tree is a pair (u,d) where depth(parent(u)) < d ≤depth(u). It represents:

- The uncompact trie node that would be at depth d along the edge (parent(u), u), and
- The corresponding string $S_{(u,d)}$ =T[start(u)..start(u)+d-1].

Recall that the nodes of the uncompact trie represent the set of all factors of T. Thus every factor of T has its own locus.

During the construction, we need to create nodes at an existing locus in the middle of an edge, splitting the edge into two edges.

Constructing the Suffix Tree

```
CreateNode(u,d)   \\splits the edge above u at string depth d
 1  i←start(u);
 2  p←parent(u);
 3  v←EmptyNode();
 4  (start(v),depth(v))←(i,d);
 5  child(v,T[i+d])←u;
 6  parent(u)←v;
 7  child(p,T[i +depth(p)])←v;
 8  parent(v)←p;
 9  return v;
```

Constructing the Suffix Tree

```
CreateLeaf(i,u,d)                    \\attaches a new leaf at depth d under u
  1  w←EmptyLeafNode();
  2  (start(w),depth(w))←(i,|T|-i+1);   \\depth=length of suffix T[i..|T|]
  3  child(u,T[start(w)+d])←w;
  4  parent(w)←u;
  5  return w;
```

Brute Force Suffix Tree Construction

```
BF-Construction(T[0 . . n])
    1 root←EmptyNode();
    2 depth(root)←0;
    3 (u,d)←(root,0);
    4 for i=0,…,n do
        5 while d=depth(u) and child(u,T[i+d]) ≠ ε do
            6 u←child(u,T[i +d]);
            7 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 (u,d)←(root,0);
```