Aho-Corasick Algorithm

Let there be a set of strings with a total length of m (sum of the length of strings in the set). Aho-Corasick Algorithm constructs a trie-like data structure with some additional links and constructs a finite state automaton about which we are going to study.
The algorithm was proposed by Alfred Aho and Margaret Corasick in 1975.
Let's take an example problem to understand the algorithm:

Given an input string S and an array A of k words. Find all the occurrences of all the words in the given input string S.
Let n be the total number of words in the array and m be the total length of words i.e.
m = length[A[0]] + length[A[1]] + length[A[2]] + ... + length[A[k-1]], k is the number of words in array A.

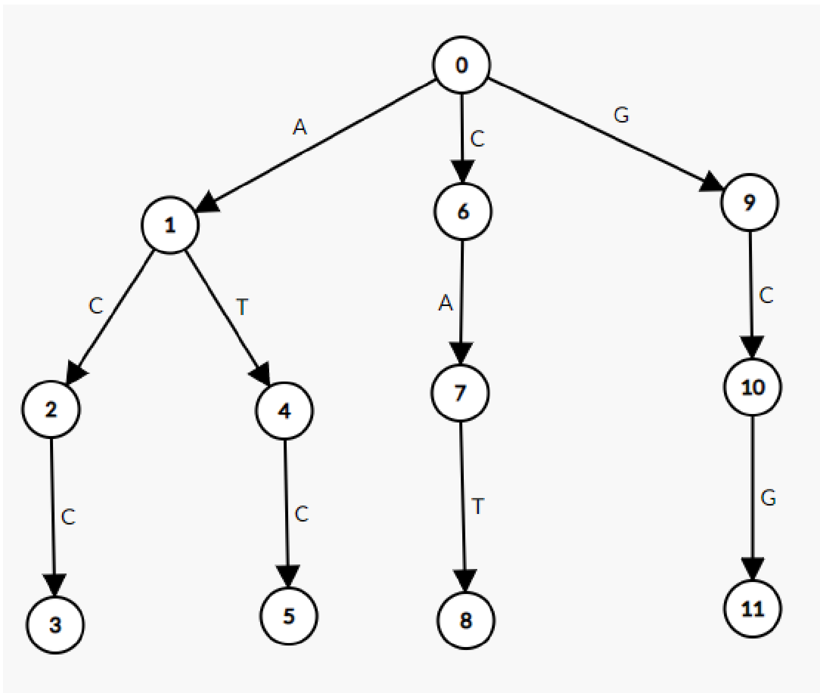
Let array A be {ACC, ATC, CAT, GCG} string S be "GCATCG".

If we use a linear time searching algorithm like KMP, we will have to search each word one by one in the string and the time complexity will be O(n*k + m) whereas the **Aho-Corasick Algorithm** finds all words in ***O(n + m + z)*** time where **z** is a total number of occurrences of words in the string.

Steps for Aho-Corasick Algorithm are as follows:

- Pre-processing phase(creating the automaton)
  It further consists of two steps, creating a trie and adding failure links(will discuss about this).

  The trie for above array of words will look like

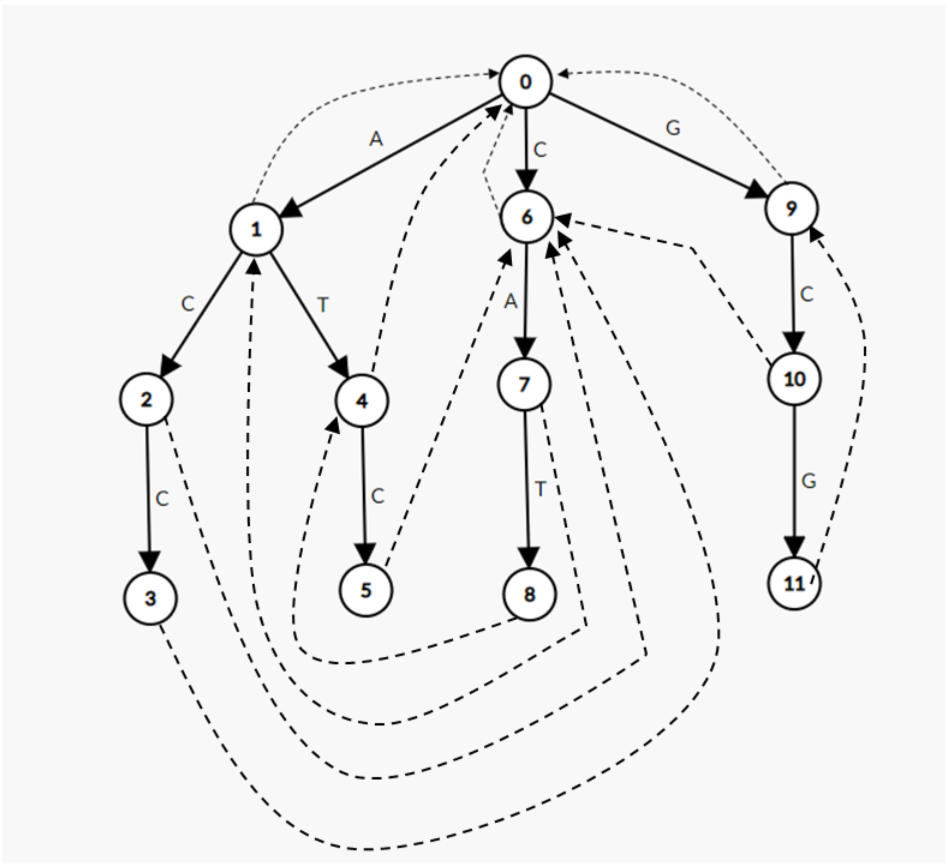


Pseudocode:

```
 HashMap map<Vertex, String> // to store the last vertex of a string versus string
//   create a class/structure of Vertex/node of a trie as defined below
class Vertex{

                //   an array of type vertex of size 26
                Vertex child[26];

                //   pointer to the suffix link(failure links)
                Vertex suffixLink;

                //   to store if the vertex is the end of the string
                boolean isLeaf;

                //   to store the current character of the string
                char data;

                public Vertex(char ch){
                        child = new Vertex[26];
                        suffixLink = null;
                        isLeaf= false;
                        this.data = ch;
                 }

}

//   addString method: to add new string S in the trie
function addString(String S){
```

```
                Vertex v = root;
                for(all character ch of string s) {

                        int idx = ch-'a';

                        if(v.child[idx] == null) {
                                v.child[idx] = new Vertex(ch);
                        }

                        v = v.child[idx];
                }

                v.isLeaf = true;
               //   add string s corresponding to vertex v in map
                map.put(v, s);
}
```

- The next step is to extend the trie to an automaton by adding failure links.



For every node in the trie, consider the longest suffix from the root that exists as the prefix in the trie and add the failure link from the node to the prefix node.
Example:

- ➔ Consider node number 2, the string from the root to node 2 is “AC”, the suffixes of this string are “AC”, “C” and “”, longest of them is “AC” but “AC” doesn’t exist as a prefix in any other string so check for “C”. Yes, “C” exists as a prefix in string “CAT”. So add a failure link from node 2 to node 6, where the prefix of the string exists.
- ➔ Now consider node 11, suffixes are “GCG”, “CG”, “G”, and “”. Out of the three suffixes, the longest suffix that exists as a prefix is “G” at node 9. So add a failure link from node 11 to node 9.
- ➔ Consider node 8, the suffixes are “CAT”, ”AT”, ”T” and “”. The longest suffix that exists as a prefix is “AT” which ends at node 4 in the trie. So add a failure link from 8 to node 4.
- ➔ Now consider node 7, suffixes are “CA”, “A”, and “”. The longest suffix that exists as a prefix is “A” at node 1. Draw a failure link from node 7 to node 1.
- ➔ Consider node 1, suffixes are “A” and “”. The longest suffix that exists is “”, so add a failure link from node 1 to node 0.

- Last step of automata building is adding output links.

```
function addSuffixLinks() {

        root.suffixLink = root;
        //   create a queue and add all children of root
        Queue<Vertex> q = new LinkedList<>();

        for(all childVertex ch of root.child) {
            if(ch != null) {
                q.add(ch);
                ch.suffixLink = root;
            }
        }

        while(q is not empty) {

            Vertex cur_state = q.getFirst()
            q.remove()

            //   iterating over all child of current Vertex

            for(int i = 0 ; i < cur_state.child.length ; i++){
                    if(cur_state.child[i] == null) {
                    continue;
                }
                char c = (char)(i + 'a')
```

```
                    Vertex temp = cur_state.suffixLink

                    //   finding longest proper suffix
                    while(temp.child[i] == null && temp != root  )
                            temp = temp.suffixLink

                    //   if proper suffix is found
                    if(temp.child[i] != null)
                        cur_state.child[i].suffixLink = temp.child[i]

                    //   if proper suffix not found
                    else
                        cur_state.child[i].suffixLink = root;

                    q.add(cur_state.child[i]);

                }

            }
```

- The final step is reading the input string and finding matches.
  For this, we iterate over the input string and see if the current character is present as a child of some current vertex then we move forward to that child vertex and if not we move to the suffix link.
  If the suffix link points to the root, then we break our iteration.
  If in this iteration we reach the leaf node, we increase the result by 1, where the result stores the number of matches.

```
function patternMatching(String s) {

        Vertex v = root;
        i=0
        while(i is less than the length of s) {
            char ch = s.charAt(i);

            int idx = ch - 'a';

            if(v.child[idx] != null) {
                v = v.child[idx];
            }else {
                while(v != root && v.child[idx] == null) {
                //   follow suffix links till matching suffix or root is found
```

```
                                v = v.suffixLink;

                        }

                        if(v.child[idx] != null) {
                                i--;
                        }
                }

                if(v == root)
                        continue;

                if(v.isLeaf) {
                        print(map.get(v));
                }
                i++

        }
```

Time Complexity:
The time complexity of this algorithm is: **O(N + L + Z)**, where N is the length of the text, L is the length of keywords and the Z is the number of matches.

Applications:
- Detecting plagiarism
- Text mining
- Bioinformatics
- Intrusion Detection