VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

KMP AND AHO-CORASICK

Võ Đình Cao Minh Hào - 24127035
Trịnh Duy Nhân - 24127094
Phan Thanh Trúc Quân - 24127227

7/4/2025

Vietnam National University

University of Science

Faculty of Information Technology

KMP and Aho-Corasick

Topic: String Algorithms

Course: Data Structures and Algorithms

Students:

Võ Đình Cao Minh Hào (24127035)

Trịnh Duy Nhân (24127094)

Phan Thanh Trúc Quân (24127227)

Professors:

Dr. Nguyễn Ngọc Thảo

BSc. Nguyễn Thanh Tình

April 6, 2025

Contents

1 History and Applications of the Knuth-Morris-Pratt (KMP) Algorithm

1.1 History

The Knuth-Morris-Pratt (KMP) algorithm was developed by Donald Knuth, Vaughan Pratt, and James H. Morris in 1970 and was officially published in 1977. The KMP algorithm provides an efficient method for locating a substring W within a larger string S. Instead of re-examining previously matched characters upon encountering a mismatch, the KMP algorithm leverages information inherent in the pattern W itself to determine the next position for comparison. One of the key advantages of the KMP algorithm is its high efficiency. It operates with a time complexity of $O(n+m)$, where n is the length of the text S and m is the length of the pattern W. Furthermore, by utilizing the Longest Prefix Suffix (LPS) table, the algorithm eliminates redundant comparisons, thereby optimizing search performance. The KMP algorithm represents a significant improvement over conventional brute-force search methods, making it particularly beneficial when processing large datasets or executing repeated search operations.

1.2 Applications

- Text searching: Implemented in text editors (such as Microsoft Word, Notepad++) to locate words or phrases within documents.
- Pattern matching in DNA and computational biology: Applied in genetic research to identify specific gene sequences within extensive DNA datasets.
- Spam filtering: Used to detect suspicious keywords in emails, aiding in the identification of spam messages.
- Search engines: Enhances search efficiency by facilitating substring searches within large-scale databases.
- Plagiarism detection: Compares textual documents to identify duplicated content, assisting in academic integrity verification.
- File processing and compilation: Employed in certain compilers to efficiently analyze and parse source code.

2 Knuth - Morris - Pratt Step by step

Note: The string will start at number 0

2.1 Problem

- Given string s: "ABXABABABAA".
- Given string t: "ABABAA".

Check if string t exists in string s?

2.2 Main idea

The idea behind KMP is to extend a suffix ending at position i to a suffix ending at position i+1 while still matching the corresponding prefix, which improves efficiency by eliminating expensive string comparisons. [1]

2.3 Definition

- **Proper Prefix**: A prefix that is not equal to the string itself.
- **Prefix Function**[2]: An array π of length m, where $\pi[i]$ is the length of the longest proper prefix of the substring $s[0..i]$, which is also a suffix of the substring. Mathematically, the definition of the prefix function can be written as follows:

$$\pi[i] = \max_{0 \le k \le i} \{ k : s[0 \dots k-1] = s[i-(k-1) \dots i] \}. \tag{1}$$

2.4 Preprocessing

Due to the definition of proper prefix, $\pi[0] = 0$.

Here is the example code:

```
vector <int> prefix_function(string t){
    int m = t.size()
    int j = 0;
    vector <int> pi(m, 0);
    for(int i = 1; i < m; i++){
        if(t[i] == t[j]){
            pi[i] = ++j;
        }
        else{
            while(t[i] != t[j] && j > 0) j = pi[j - 1];
            if(t[i] == t[j]) pi[i] = ++j;
        }
    }
    return pi;
}
```

The example code will run in $\Theta(m)$ where m is the length of the pattern.

Explain step by step

Step 1: $\pi[0] = 0$ so i starts at 1. We use j to trace the prefix, and it starts at 0:

pos	j	i				
t	A	B	A	B	A	A
π	0					

Step 2: *At* $i = 1$ *and* $j = 0$

$t[i]$ mismatches $t[j]$ ("A" $\neq$ "B"), so we set $\pi[i=1]=0$ then increases i by 1:

pos	j		i			
t	A	B	A	B	A	A
π	0	0				

Step 3: *At* $i=2$ *and* $j=0$

$t[i]$ matches $t[j]$ ("A" = "A"), so we set $\pi[i=2]=j+1=1$ then increases both i and j by 1:

pos		j		i		
t	A	B	A	B	A	A
π	0	0	1			

Note: $\pi[i]=j+1$ because $\pi[i]$ represents the length of the longest proper prefix of the substring $t[0\dots i]$, which is also a suffix of the substring.

Step 4: *At* $i=3$ *and* $j=1$

$t[i]$ matches $t[j]$ ("B" = "B"), so we set $\pi[i=3]=j+1=2$ then increases both i and j by 1:

pos			j		i	
t	A	B	A	B	A	A
π	0	0	1	2		

Step 5: *At* $i=4$ *and* $j=2$

$t[i]$ matches $t[j]$ ("A" = "A"), so we set $\pi[i=4]=j+1=3$ then increases both i and j by 1:

pos				j		i
t	A	B	A	B	A	A
π	0	0	1	2	3	

Step 6: *At* $i=5$ *and* $j=3$

$t[i]$ mismatches $t[j]$ ("B" $\neq$ "A") and j is greater than 0. This situation is a bit tricky because we must trace back to the $\pi[j-1]$ position. Since $\pi[i]$ represents the longest proper prefix of $s[0\dots i]$, $\pi[i-1]$ represents the second longest proper prefix and so on. Therefore, if there is a mismatch, we have to trace back to the second, third, etc., longest proper prefix. In this case, j moves to position 1 because $\pi[j-1]=1$:

pos		j				i
t	A	B	A	B	A	A
π	0	0	1	2	3	

Step 7: Now, $t[i]$ and $t[j]$ still do not match, and j is still greater than 0, so we repeat the backtracking step. As a result, j goes back to 0 because $\pi[j-1]=0$:

pos	j					i
t	A	B	A	B	A	A
π	0	0	1	2	3	

Step 8: Now, $t[i]$ and $t[j]$ match ("A" = "A") so we set $\pi[i] = j + 1 = 1$ and done the prefix function.

t	A	B	A	B	A	A
π	0	0	1	2	3	1

Important: The code always runs in linear time regardless of the string's distribution. Therefore, the time and space complexity of the code is $\Theta(m)$, where m is the length of the pattern string.

2.5 String matching

After computing the prefix function for the pattern t, we use it to efficiently search for t in the text s. Recall that for $t =$ `ABABAA`, the prefix function is:

$$\pi = [0, 0, 1, 2, 3, 1].$$

The string matching algorithm proceeds as follows:

1. Initialize two indices: i for the text s (starting at 0) and j for the pattern t (starting at 0).
2. Compare $s[i]$ with $t[j]$:
   - If $s[i] = t[j]$, increment both i and j.
   - If $s[i] \neq t[j]$ and $j > 0$, update j to $\pi[j-1]$ (i.e., backtrack j to the length of the next best candidate prefix) and compare again.
   - If $s[i] \neq t[j]$ and $j = 0$, increment i only.
3. If j reaches the length of t (i.e., $j = m$ where m is the length of t), a full match is found at index $i - j$ in s.

Example Walk-through

Consider the given strings:

- $s =$ `ABXABABABAA`
- $t =$ `ABABAA`

Step 1: *Start with $i = 0$ and $j = 0$*

pos	i										
s	A	B	X	A	B	A	B	A	B	A	A

pos	j					
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[0] =$ `A` and $t[0] =$ `A` $\longrightarrow$ match, so increment: $i = 1$, $j = 1$.

Step 2: *Now,* $i = 1$ *and* $j = 1$

pos		i									
s	A	B	X	A	B	A	B	A	B	A	A

pos		j				
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[1] =$ `B` and $t[1] =$ `B` $\longrightarrow$ match, so increment: $i = 2$, $j = 2$.

Step 3: *Now,* $i = 2$ *and* $j = 2$

pos			i								
s	A	B	X	A	B	A	B	A	B	A	A

pos			j			
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[2] =$ `X` and $t[2] =$ `A` $\longrightarrow$ mismatch.

Since $j > 0$, update j to $\pi[j-1] = \pi[1] = 0$.

Step 4: *Now,* $i = 2$ *and* $j = 0$

pos			i								
s	A	B	X	A	B	A	B	A	B	A	A

pos	j					
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[2] =$ `X` and $t[0] =$ `A` $\longrightarrow$ mismatch.

But now $j = 0$, so increment: $i = 3$.

Step 5: *Now,* $i = 3$ *and* $j = 0$

pos				i							
s	A	B	X	A	B	A	B	A	B	A	A

pos	j					
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[3] =$ `A` and $t[0] =$ `A` $\longrightarrow$ match, so increment: $i = 4$, $j = 1$.

Step 6: *Now,* $i = 4$ *and* $j = 1$

pos					i						
s	A	B	X	A	B	A	B	A	B	A	A

pos		j				
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[4] =$ `B` and $t[1] =$ `B` $\longrightarrow$ match, so increment: $i = 5$, $j = 2$.

Step 7: *Now,* $i = 5$ *and* $j = 2$

pos						i					
s	A	B	X	A	B	A	B	A	B	A	A

pos			j			
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[5] =$ `A` and $t[2] =$ `A` $\longrightarrow$ match, so increment: $i = 6$, $j = 3$.

Step 8: *Now,* $i = 6$ *and* $j = 3$

pos							i				
s	A	B	X	A	B	A	B	A	B	A	A

pos				j		
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[6] =$ `B` and $t[3] =$ `B` $\longrightarrow$ match, so increment: $i = 7$, $j = 4$.

Step 9: *Now,* $i = 7$ *and* $j = 4$

pos								i			
s	A	B	X	A	B	A	B	A	B	A	A

pos					j	
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[7] =$ `A` and $t[4] =$ `A` $\longrightarrow$ match, so increment: $i = 8$, $j = 5$.

Step 10: *Now,* $i = 8$ *and* $j = 5$

pos									i		
s	A	B	X	A	B	A	B	A	B	A	A

pos						j
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[8] =$ `B` and $t[5] =$ `A` $\longrightarrow$ mismatch.

Since $j > 0$, update j to $\pi[j-1] = \pi[4] = 3$.

Step 11: *Now,* $i = 8$ *and* $j = 3$

pos									i		
s	A	B	X	A	B	A	B	A	B	A	A

pos				j		
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[8] =$ `B` and $t[3] =$ `B` $\longrightarrow$ match, so increment: $i = 9$, $j = 4$.

Step 12: *Now,* $i = 9$ *and* $j = 4$

pos										i	
s	A	B	X	A	B	A	B	A	B	A	A

pos					j	
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[9] =$ `A` and $t[4] =$ `A` $\longrightarrow$ match, so increment: $i = 10$, $j = 5$.

Step 13: *Now,* $i = 10$ *and* $j = 5$

pos											i
s	A	B	X	A	B	A	B	A	B	A	A

pos						j
t	A	B	A	B	A	A
π	0	0	1	2	3	1

$s[10] =$ `A` and $t[5] =$ `A` $\longrightarrow$ match.

Since $j = m$ where m is the length of t, a full match is found at index $i - j = 5^{th}$ in s.

Here is the full code of String Matching:

```
bool StringMatching(string s, string t){
    int m = t.size();
    int n = s.size();
    int j = 0;
    vector <int> pi = prefix_function(t);
    for(int i = 0; i < n; i++){
        while(j > 0 && s[i] != t[j]) j = pi[j - 1];
        if(s[i] == t[j]){
            j++;
        }
```

```
        if(j == m){
            return true;
        }
    }
    return false;
}
```

The overall complexity of this string matching process is $\mathcal{O}(n+m)$, where n is the length of the text s and m is the length of the pattern t.

2.6 Complexity Analysis

- The total **Space Complexity** of the algorithm is $\Theta(m)$, due to the contruction of π length m.
- However, the **Time Complexity** of the algorithm depends on the kind of problem:
  - Find all the position that string pattern t occurs in string text s.
  - Return the first string pattern t occurs in string text s or Check if string pattern t occurs in string text or not.

Find all the position

- In this problem, we must agree that the **Preprocessing** step always runs in $\Theta(m)$ because it always iterates through pattern t.
- In **String Matching** step, we also iterate through string s to find all the position that appears pattern t. So the time complexity is $\Theta(n)$.

$\Longrightarrow$ The total **Time Complexity** is $\Theta(n+m)$

Check if occurs or not

- In this problem, we must agree that the **Preprocessing** step always runs in $\Theta(m)$ because it always iterates through pattern t.
- In **String Matching** step, it happens 2 cases:
  - **Best Case:** We instantly find the pattern t at the prefix of the string s.
  - **Worst Case:** We find the pattern t at the suffix of the string s.

$\Longrightarrow$ The **Time Complexity** of the algorithm is:

- **Best Case:** $\mathcal{O}(m)$.
- **Worst Case:** $\mathcal{O}(n+m)$.

3 History and Applications of the Aho-Corasick Algorithm

3.1 History

The Aho-Corasick algorithm was developed by Alfred V. Aho and Margaret J. Corasick in 1975. It is a multi-pattern string-searching algorithm that operates based on the Trie data structure, augmented with a failure function to enable efficient identification of multiple pattern occurrences within a given text. By constructing a finite-state automaton, the algorithm is capable of detecting multiple patterns concurrently in a single pass over the input text.
One of the primary advantages of the Aho-Corasick algorithm is its ability to search for multiple patterns simultaneously, eliminating the need for separate executions as required by algorithms such as KMP or Boyer-Moore. It achieves a time complexity of $O(n + m + z)$, where n is the length of the input text, m is the total length of all patterns, and z is the number of pattern matches found. Additionally, the algorithm does not require redundant character comparisons, as the failure function enables efficient transitions upon mismatches. The Aho-Corasick algorithm finds extensive applications in network security, natural language processing, and computational biology. Due to its capability of handling large-scale multi-pattern searches, it significantly reduces processing time and resource consumption.

3.2 Applications

- Network security and malware detection: Deployed in Intrusion Detection Systems (IDS) to identify attack signatures embedded within network traffic. Implemented in antivirus software to detect malware patterns within system files and memory.
- Text processing and keyword searching: Efficiently searches for multiple keywords simultaneously, making it ideal for content filtering in document processing, social media platforms, and email monitoring systems.
- Computational biology (bioinformatics): Utilized in DNA sequence analysis to locate multiple genetic markers or protein sequences within large genomic datasets, facilitating research in genetics and medicine.
- Spell checking and word suggestions: Applied in Natural Language Processing (NLP) to enhance spell checking and autocomplete functionalities, improving text input efficiency in word processing applications.

4 Aho - Corasick Step by step

4.1 Problem

Given the text: "DIDUDUADI".
Given the 6 patterns: "DI", "DIDU", "DIDI", "DU", "DUDUA", "DUADI".
Count how many patterns are there occur in the text?

4.2 Main idea

Aho-Corasick uses trie structure to contain multiple strings.

In this trie structure, every node has a suffix link. At node i, the suffix link points to a node satisfying that the string ending at that node is the longest suffix of string ending at node i. If there are no nodes satisfying that condition, the suffix link will point to the root.

4.3 Preprocessing

There are two substeps in this process:

Step 1: Initialize a trie with suffix link pointing to the root to contain patterns.

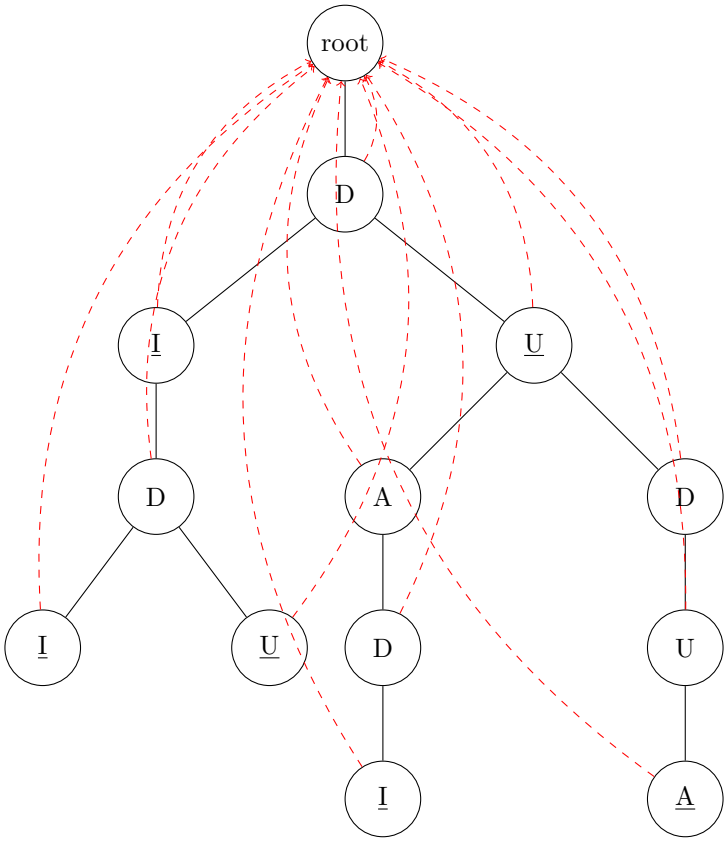
Step 2: Linking the suffix link of each node to the node that satisfies the condition.

Initialize the trie with suffix link points to the root

Source code

```
struct trie
{
    struct Node
    {
        bool check = 0;
        int end = 0;
        Node *node[26];
        Node *link;
        Node()
        {
            for(int i = 0; i < 26; i++)
                node[i] = NULL;
        }
    };
    Node *root = new Node;
    void add(std::string s)
    {
        Node *temp = root;
        for(int i = 0; i < s.size(); i++)
        {
            int idx = s[i] - 'a';
            if(temp->node[idx] == NULL)
                temp->node[idx] = new Node;
            temp = temp->node[idx];
            temp->link = root;
        }
        temp->end++;
    }
};
```

Result



Linking the suffix link

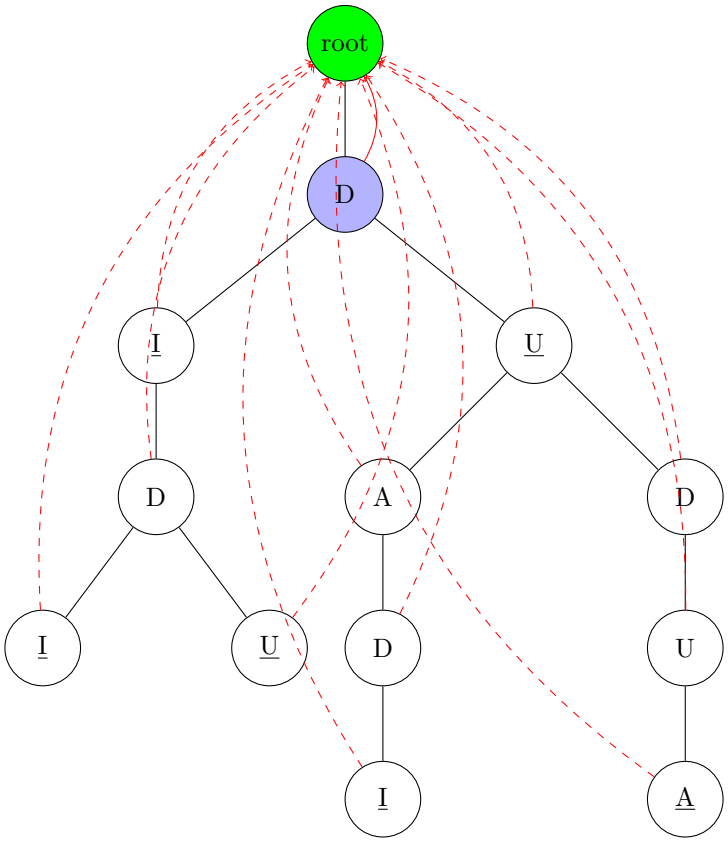
Source code

```
void link(Node *&p)
{
    for(int i = 0; i < 26; i++)
    {
        if(p->node[i] == NULL) continue;
        Node* temp = p->link;
        while(temp != root && temp->node[i] == NULL)
            temp = temp->link;
        if(temp->node[i] != NULL) (p->node[i])->link = temp->node[i];
        link(p->node[i]);
    }
}

void preprocessing()
{
    for(int i = 0; i < 26; i++)
        if(root->node[i]) link(root->node[i]);
}
```

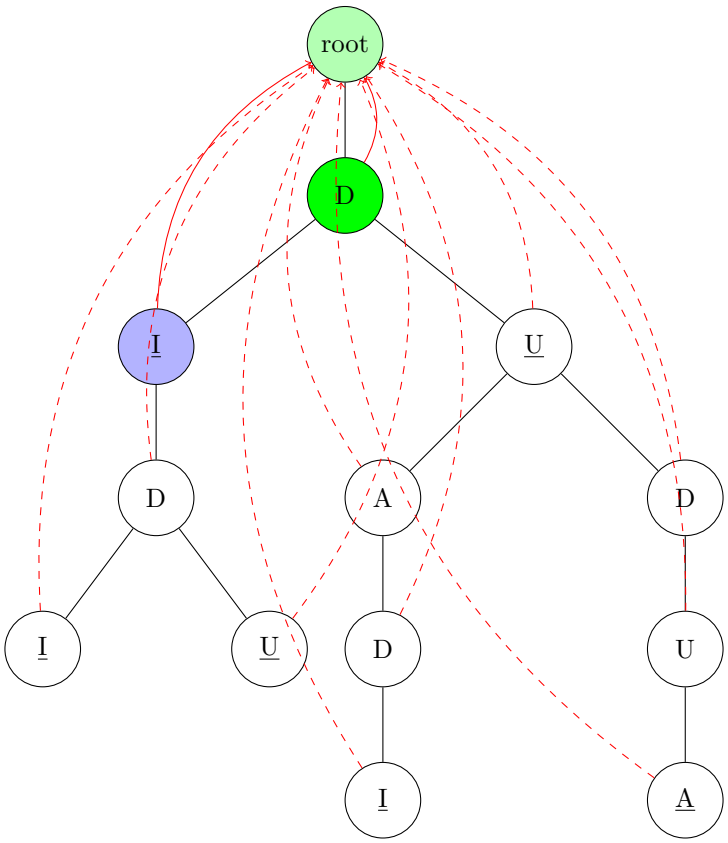
Step by step

Step 1:



Travel from root to node D, the parent is the root so we do nothing.

Step 2:

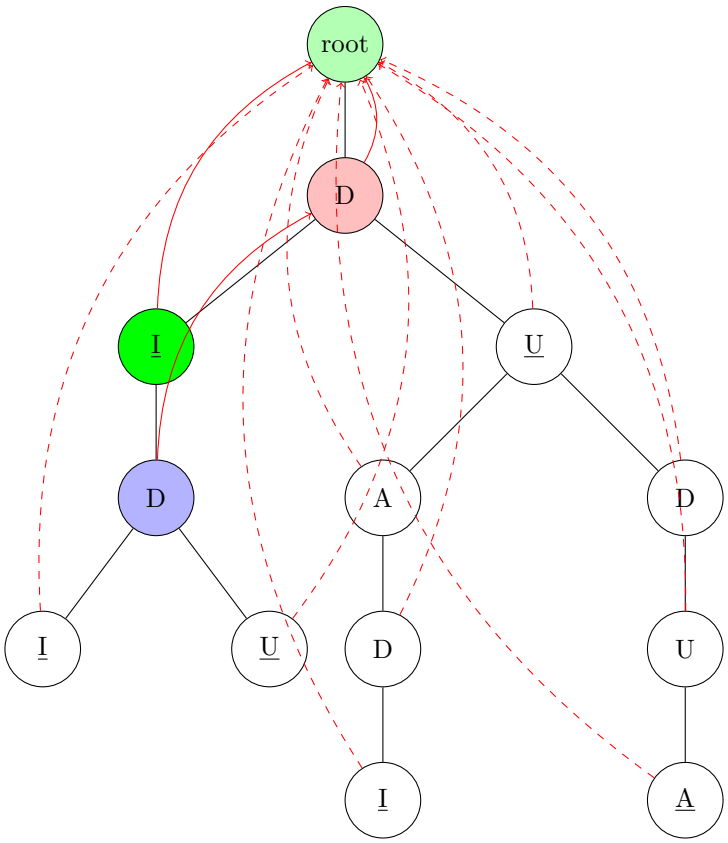


Travel to node I. The parent of this node is node D.

The suffix link of node D points to the root.

The root doesn't have node I as its child so we do nothing

Step 3:

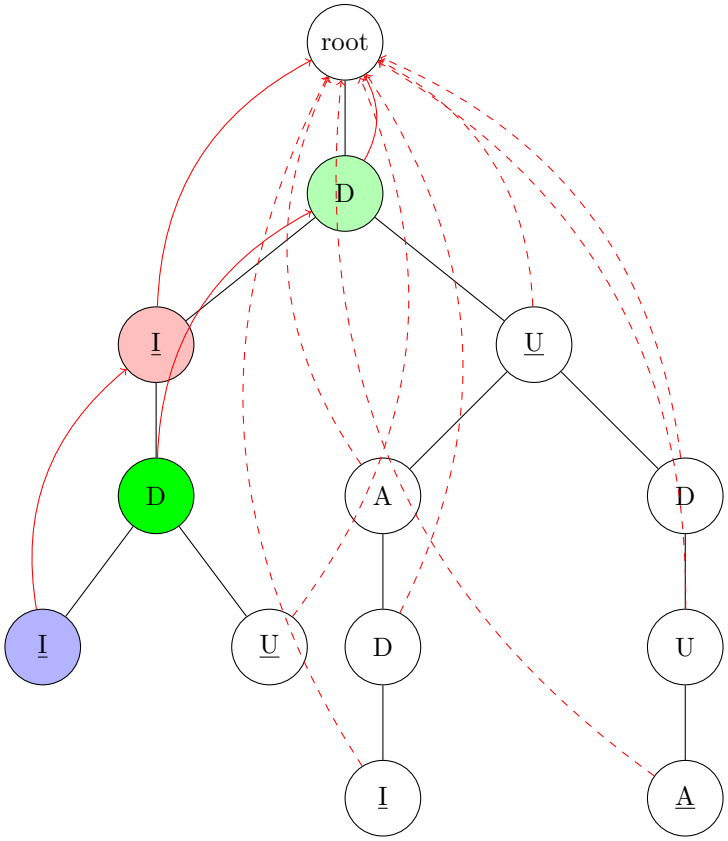


Travel to node D. The parent of this node is node I.

The suffix link of node I points to the root.

The root has node D as its child so the suffix link is now pointing to that node.

Step 4:

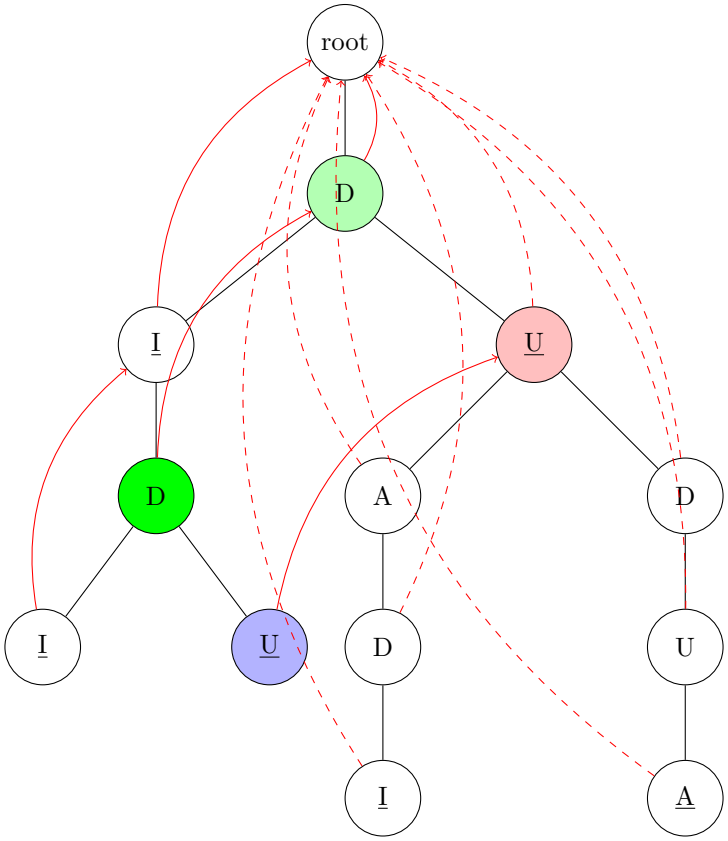


Travel to node I. The parent of this node is node D.

The suffix link of node D points to the other node D.

The other node D has node U as its child so the suffix link is now pointing to that node.

Step 5:

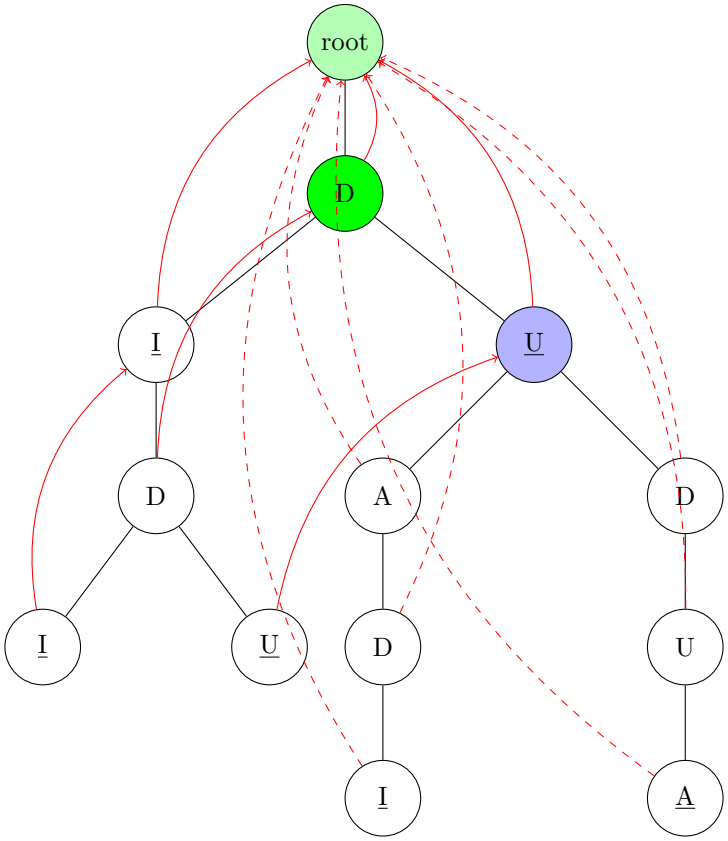


Travel to node U. The parent of this node is node D.

The suffix link of node D points to the other node D.

The other node D has node U as its child so the suffix link is now pointing to that node.

Step 6:

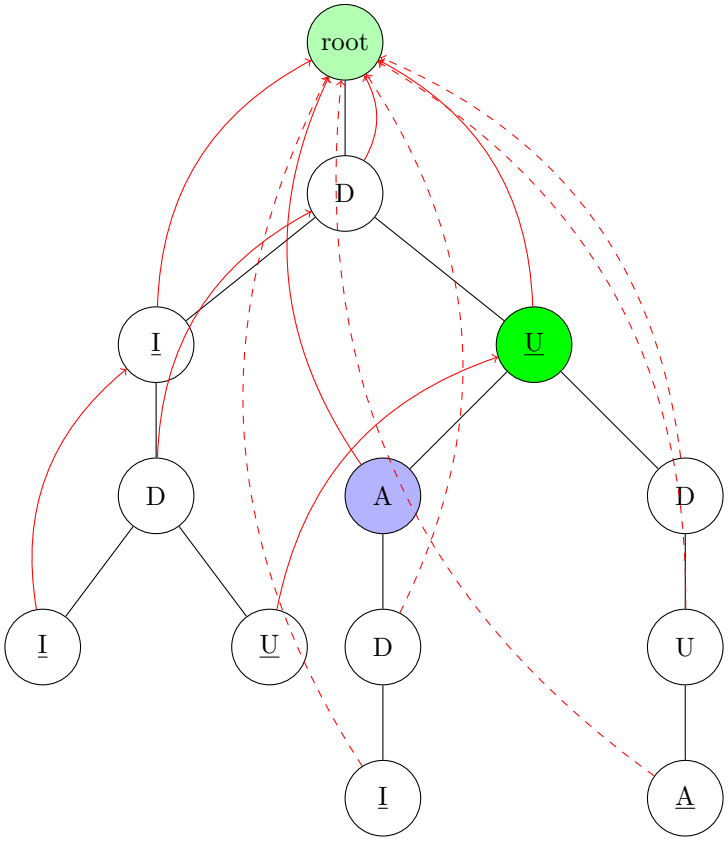


Travel to node U. The parent of this node is node D.

The suffix link of node D points to the root.

The root doesn't have node U as its child node so we do nothing.

Step 7:

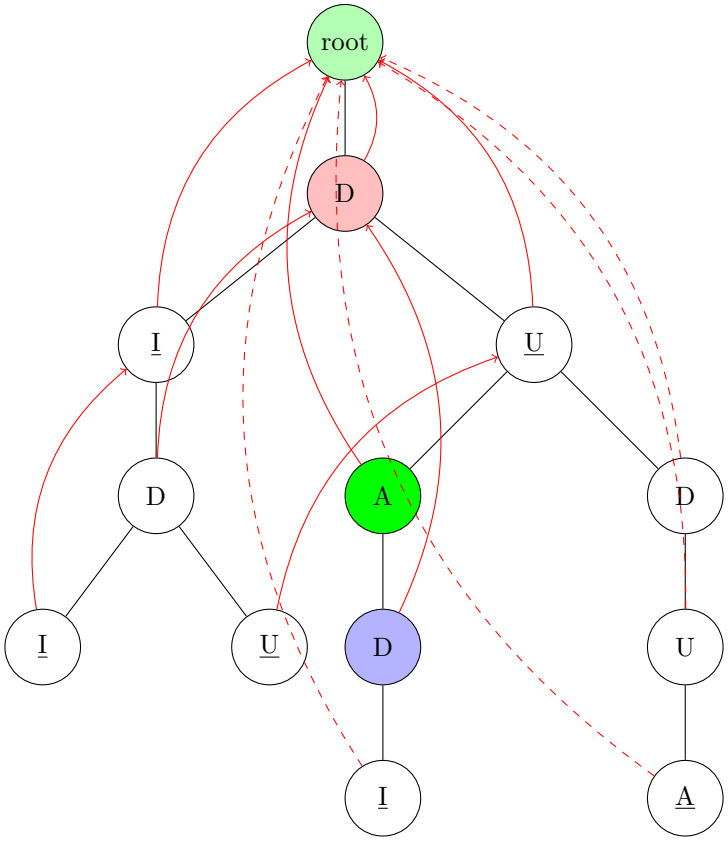


Travel to node A. The parent of this node is node U.

The suffix link of node U points to the root.

The root doesn't have node A as its child node so we do nothing.

Step 8:

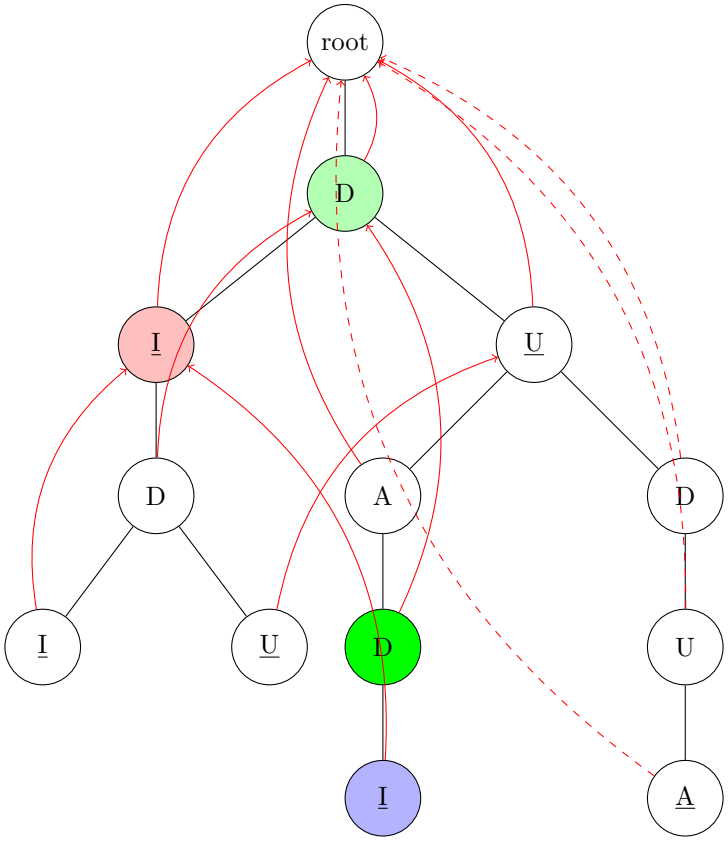


Travel to node D. The parent of this node is node A.

The suffix link of node A points to the root.

The root has node D as its child node so the suffix link is now pointing to that node.

Step 9:

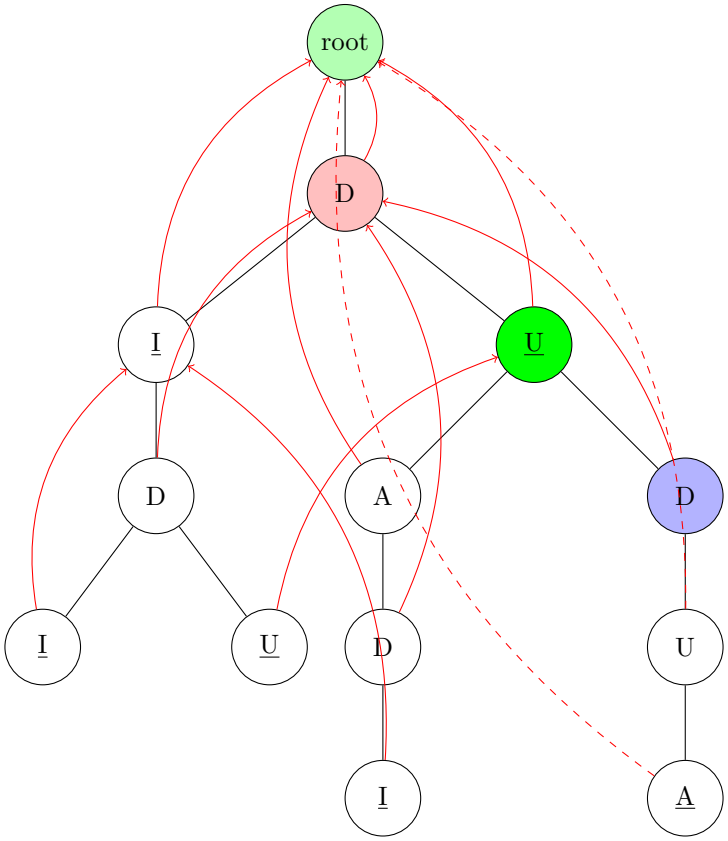


Travel to node I. The parent of this node is node I.

The suffix link of node D points to the other node D.

The other node D has node I as its child node so the suffix link is now pointing to that node.

Step 10:

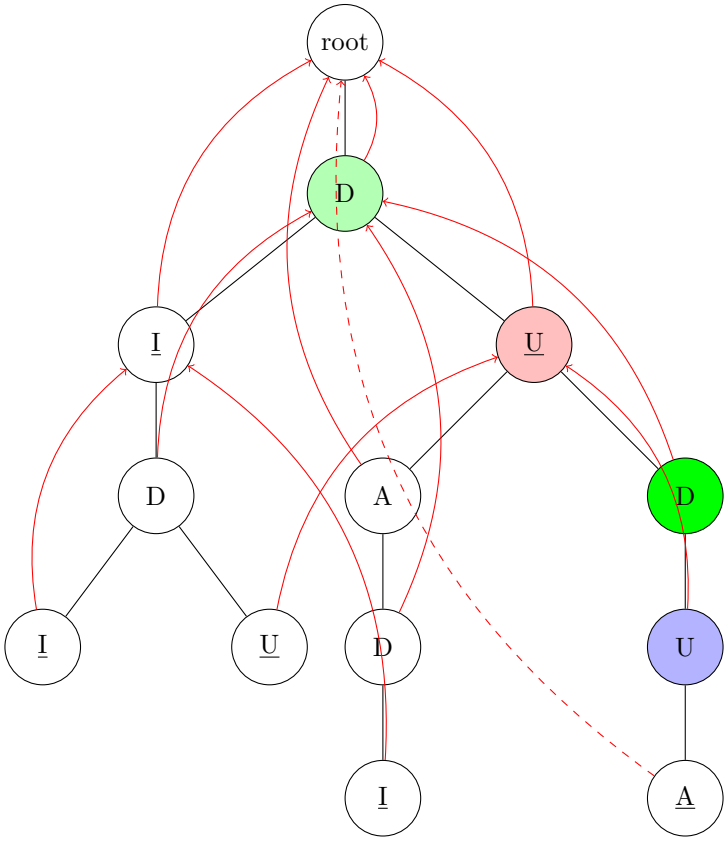


Travel to node D. The parent of this node is node U.

The suffix link of node U points to the root.

The root has node D as its child node so the suffix link is now pointing to that node.

Step 11:

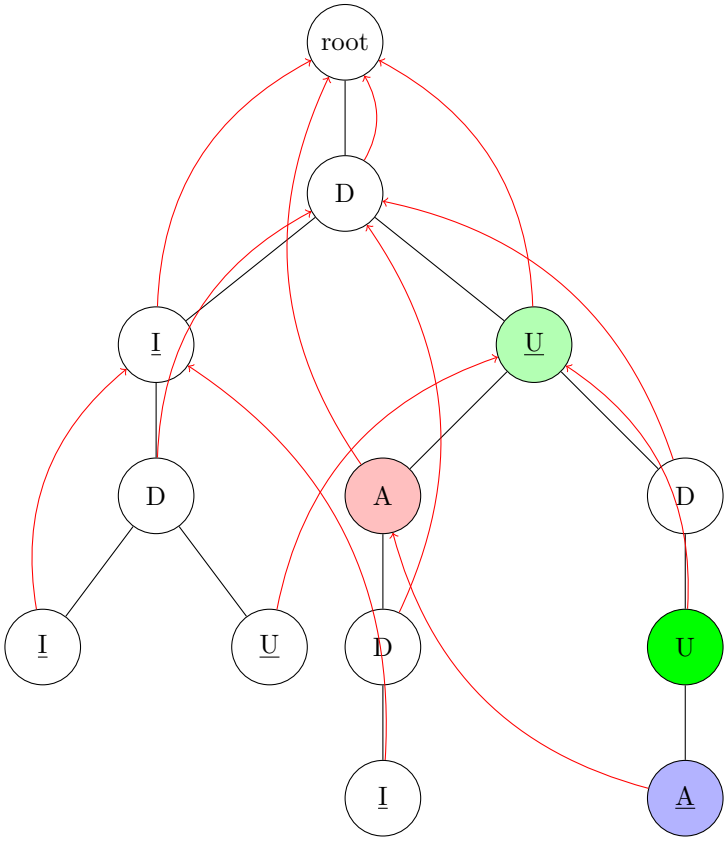


Travel to node U. The parent of this node is node D.

The suffix link of node D points to the other node D.

The other node D has node U as its child node so the suffix link is now pointing to that node.

Step 12:

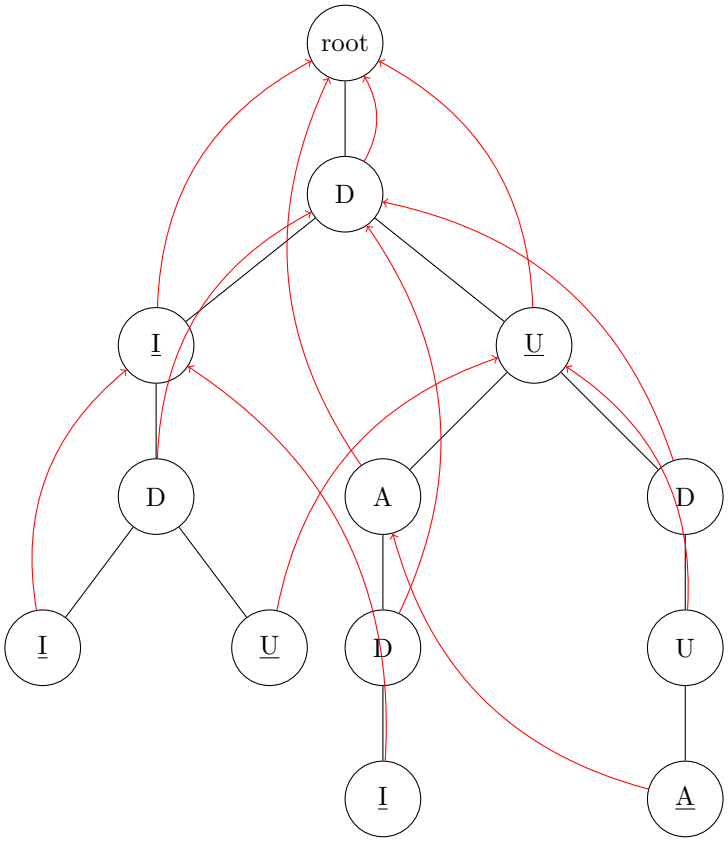


Travel to node A. The parent of this node is node U.

The suffix link of node U points to the other node U.

The other node U has node A as its child node so the suffix link is now pointing to that node.

Result



4.4 Matching/ Searching process

At this process, we use the character at the position i of the text to travel through trie respectively.

At position i, if the current has the child text[i], we will travel to that child node. Otherwise, we use the suffix link to trace back to the other node to check whether that node has that child text[i]. This procedure repeats until there is a child node text[i] or the node is the root. If the node is the root, the searching process will stop. But there is a child node text[i], we will travel to that node and continue the procedure.

After travel to that node, if the node is the end of a pattern and we have not counted that pattern, so we count it. Then we trace back using the suffix link procedure until the node is the root to check that all the prefixes are the end of any pattern or not. If there is the end of the pattern and it has not been counted, we count it.

Source code

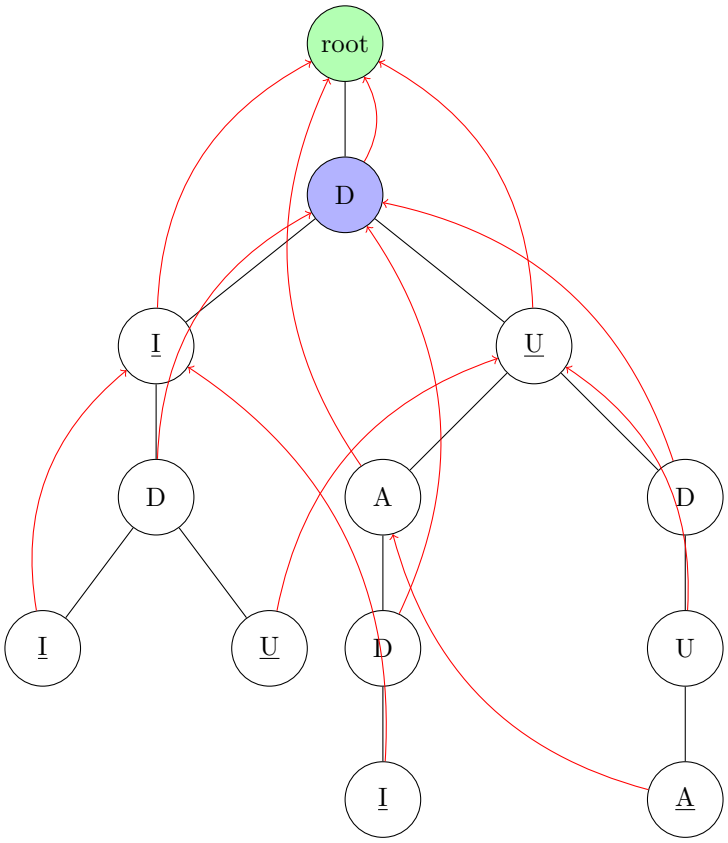
```
int checkPrefix(Node *r)
{
        if(r == root) return 0;
        int count = 0;
        if(r->end && !r->check)
        {
            r->check = 1;
            count++;
        }
        return count + checkPrefix(r->link);
}

int travel(std::string s, int idx, Node *r)
{
    if(idx == s.size()) return 0;
    int count = 0;
    if(r->end)
    {
        r->check = 1;
        count++;
    }
    count += checkPrefix(r->link);
    int indx = s[idx] - 'a';
    if(r->node[indx]) count += travel(s, idx + 1, r->node[indx]);
    else if(r != root) count += travel(s, idx, r->link);
    return count;
}
```

Step by step

Step 1:

pos	i								
text	D	I	D	U	D	U	A	D	I

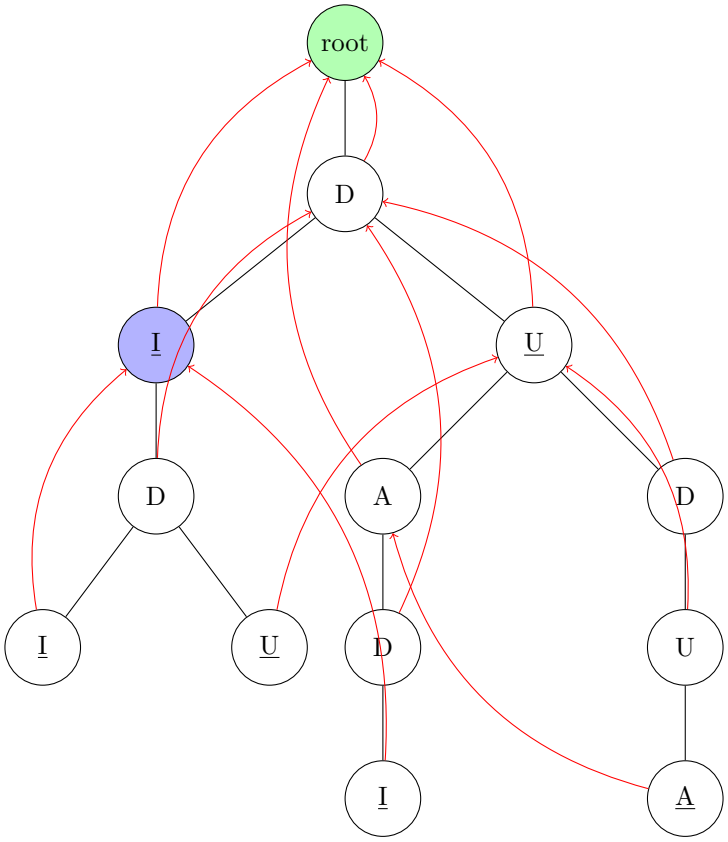


The number of patterns that occur in the text: 0.

Now text[i] = D, we travel to node D from the root. Because node D is not the end of any patterns, we do nothing in this node. We trace back to the node the suffix link of this node points at, which is the root. Because this is the root, we stop the procedure.

Step 2:

pos		i							
text	D	I	D	U	D	U	A	D	I

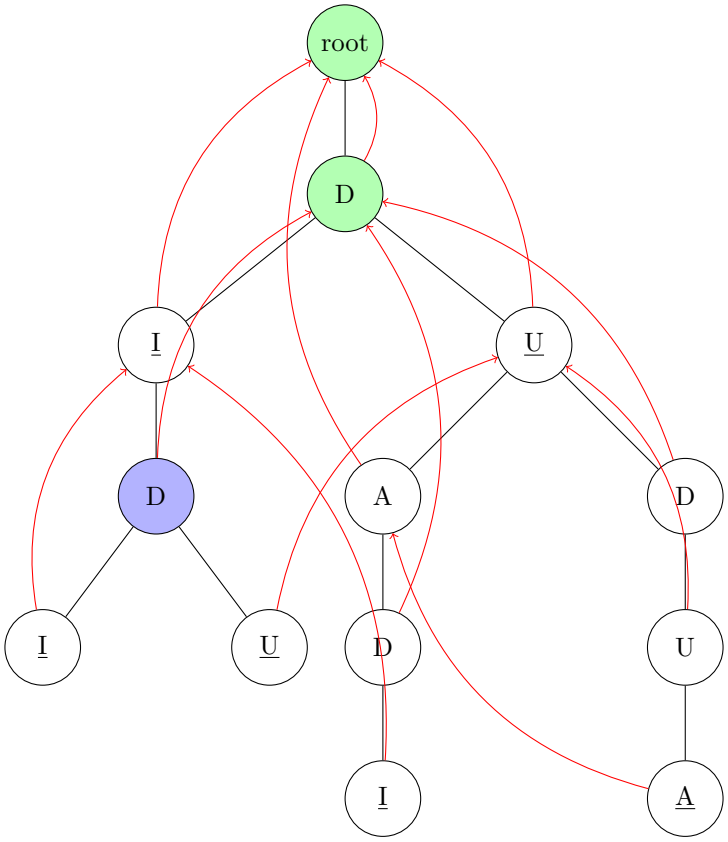


Number of patterns that occur in the text: 1 (DI).

Now text[i] = I, we travel to node I from node D. Because node I is the end of a pattern DI and pattern have not been counted, the number of pattern that occur in the text is increased by 1. We trace back to the node the suffix of this node points at, which is the root, Because this is the root, we stop the procedure.

Step 3:

pos			i						
text	D	I	D	U	D	U	A	D	I

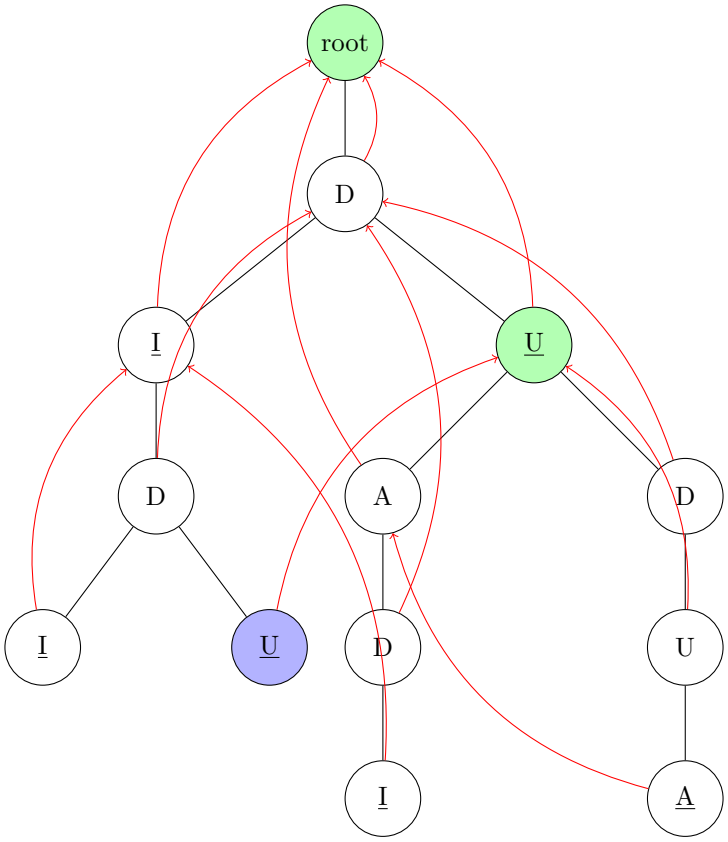


Number of patterns that occur in the text: 1 (DI).

Now text[i] = D, we travel to node D from node I. Because node D is not the end of nay patterns, we do nothing. We trace back to the node the suffix link of this node points at, which is the other node D. Because node D is not the end of any pattern, we do nothing at this node. We trace back to the node the suffix link of this node points at, which is the root. Because this is the root, we stop the procedure.

Step 4:

pos				i					
text	D	I	D	U	D	U	A	D	I

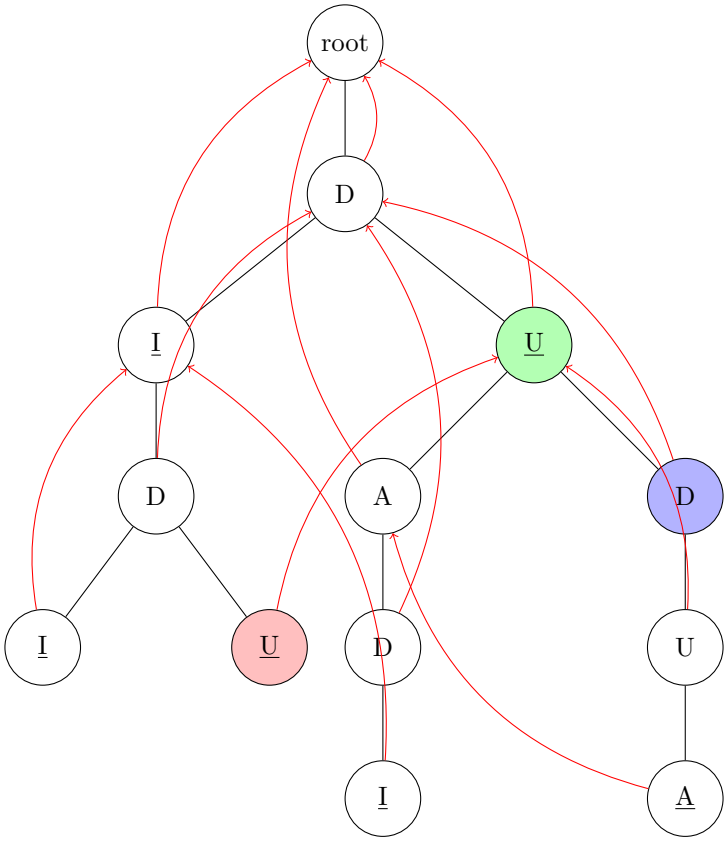


Number of patterns that occur in the text: 3 (DI, DIDU, DU).

Now text[i] = I, we travel to node U from node D. Because node U is the end of a pattern DIDU and this pattern have not been counted, the number of patterns that occur in the text in increased by 1. We trace back to the node the suffix link of this node points at, which is the other node U. Because node U is the end of pattern DU and this pattern have not been counted, the number of patterns that occur in the text is increased by 1. We trace back to the node the suffix link of this node points at, which is the root. Because this is the root, we stop the procedure.

Step 5:

pos					i				
text	D	I	D	U	D	U	A	D	I

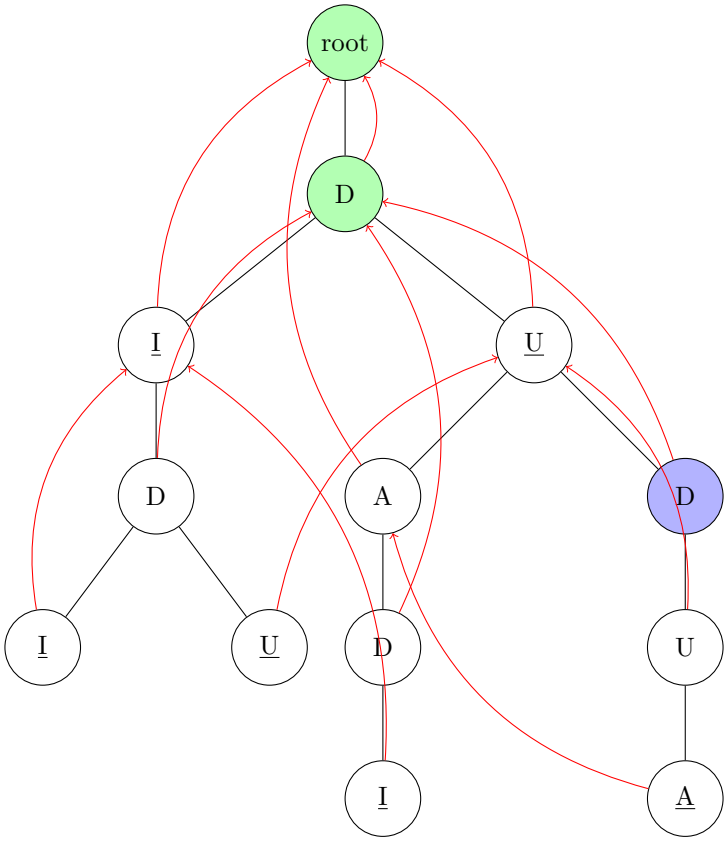


Number of patterns that occur in the text: 3 (DI, DIDU, DU).

Now text[i] = D, but node U doesn't have node child D we trace back to the node the suffix link of node U points at, which points to other node U. That node U has node child D so we travel to that node D.

Step 6:

pos					i				
text	D	I	D	U	D	U	A	D	I

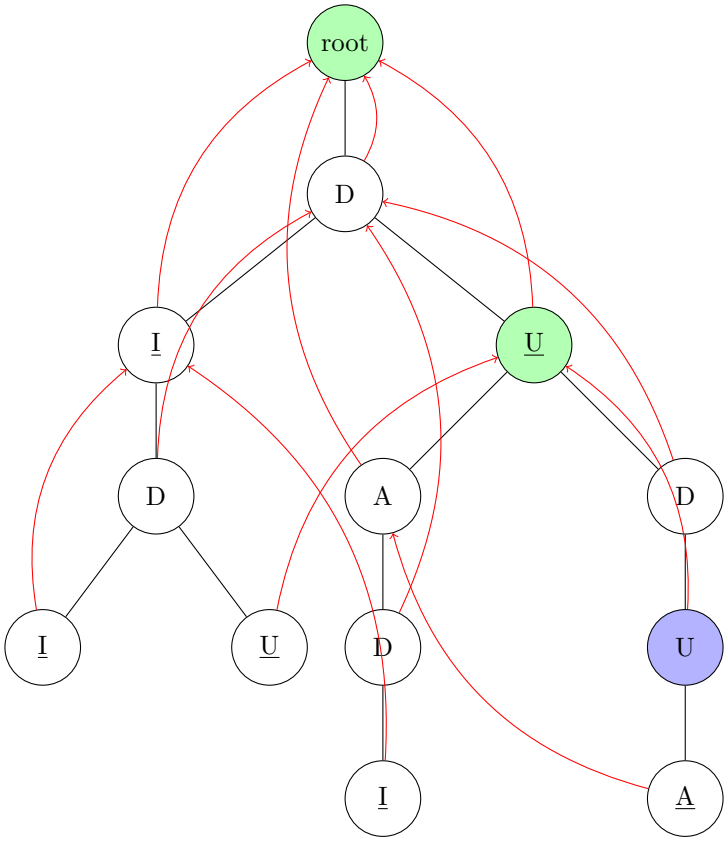


Number of patterns that occur in the text: 3 (DI, DIDU, DU).

After travel to node D, node D is not the end of any pattern so we do nothing at this node. We trace back to the node the suffix link of this node points at, which is other node D. Because this node is not the end of any pattern, we do nothing on this node. We continue to trace back to the node the suffix link of this node points at, which is the root. Because this is the root, we stop the procedure.

Step 7:

pos						i			
text	D	I	D	U	D	U	A	D	I

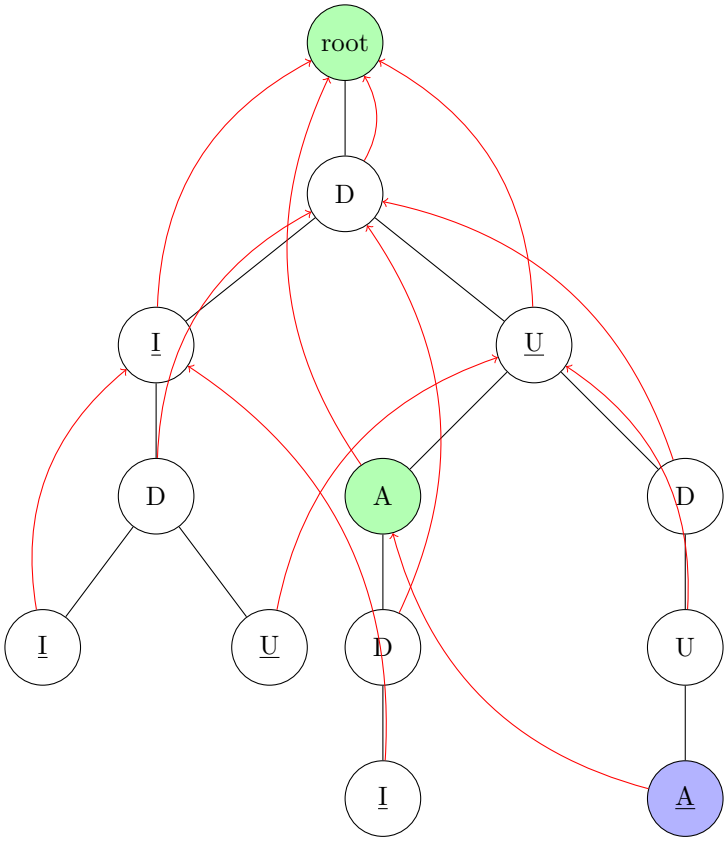


Number of patterns that occur in the text: 3 (DI, DIDU, DU).

Now text[i] = U, we travel to node U from node D. Because this node is not the end of any pattern, we do nothing on this node. We trace back to the node the suffix link of this node points at, which is other node U. This other node U is the end of pattern DU but this pattern has already counted so we do nothing on this node. We countinue to trace back to the node the suffix of this node points at, which is the root. Because this is the root, we stop the procedure.

Step 8:

pos							i		
text	D	I	D	U	D	U	A	D	I

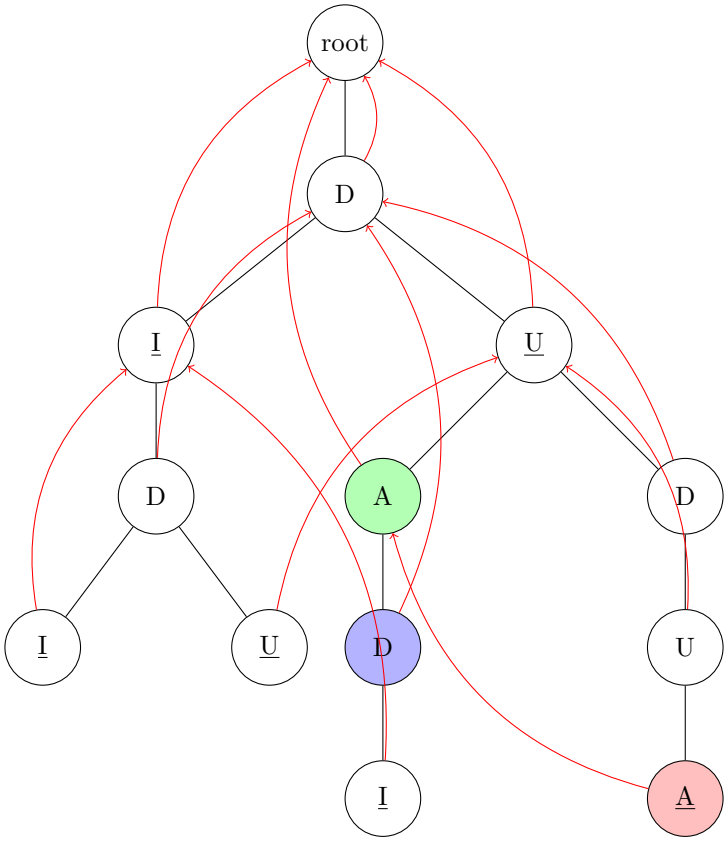


Number of patterns that occur in the text: 4 (DI, DIDU, DU, DUDUA).

Now text[i] = A, we travel to node A from node U. Because this node is the end of pattern DUDUA and this pattern have not been counted, the number of patterns that occur in the text is increased by 1. We trace back to the node the suffix link of this node points at, which is other node A. This other node A is not the end of any pattern so we do nothing on this node. We countinue to trace back to the node the suffix of this node points at, which is the root. Because this is the root, we stop the procedure.

Step 9:

pos						i			
text	D	I	D	U	D	U	A	D	I

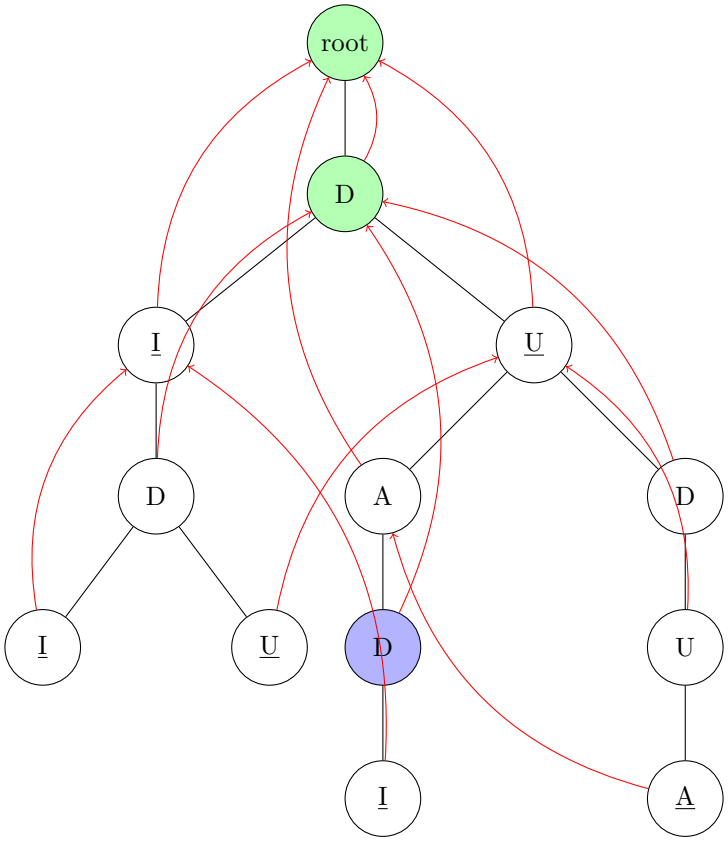


Number of patterns that occur in the text: 4 (DI, DIDU, DU, DUDUA).

Now text[i] = D, but node U doesn't have child node D, we trace back to the node the suffix of node U points at, which is the other node A. The other node A has child node D so we travel to that node D.

Step 10:

pos						i			
text	D	I	D	U	D	U	A	D	I

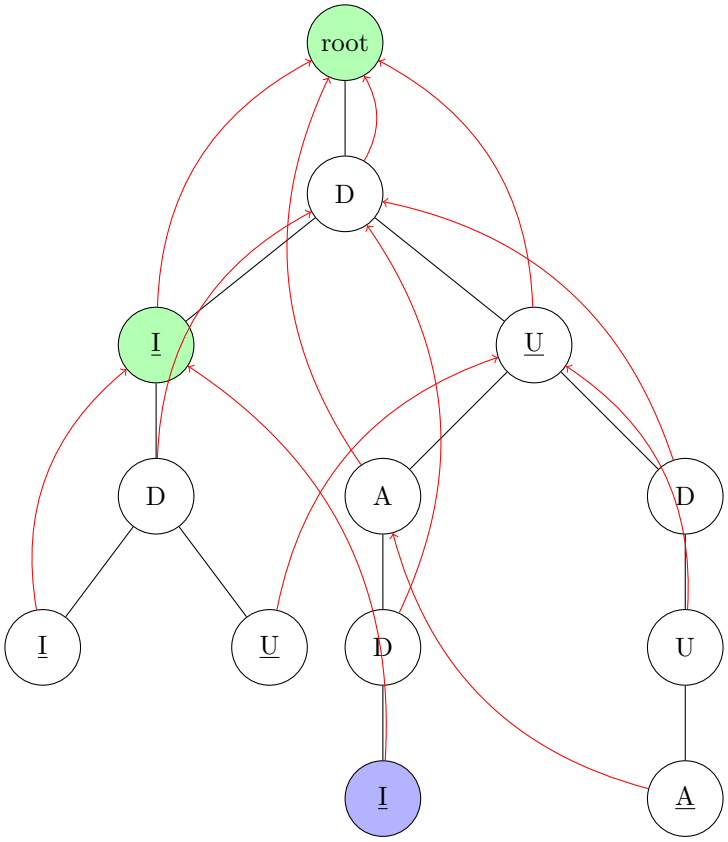


Number of patterns that occur in the text: 4 (DI, DIDU, DU, DUDUA).

This node D is not the end of ant pattern so we do thing at this node. We trace back to the node the suffix of this node points at, which is the other node D. This node D is also not the end of any pattern so we do nothing at this node. We continue to trace back to the node the suffix of this node points at, which is the root. Because this is the root, we stop the procedure.

Step 11:

pos						i			
text	D	I	D	U	D	U	A	D	I



Number of patterns that occur in the text: 5 (DI, DIDU, DU, DUDUA, DUADI).

Now text[i] = I, we travel to node I from node D. Because this node is the end of pattern DUADI and this pattern have not been counted, the number of patterns that occur in the text is increased by 1. We trace back to the node the suffix link of this node points at, which is other node I. This other node A is the end of pattern DI but this pattern has already been counted so we do nothing on this node. We countinue to trace back to the node the suffix of this node points at, which is the root. Because this is the root, we stop the procedure.

Result

There are 5 patterns occur in the text DIDUDUADI.

4.5 Complexity Analysis

Trie Construction

Each pattern is inserted into the trie, where the total length of all patterns is denoted as m. Since inserting each character takes $O(1)$, the total time complexity of constructing the trie is:

$$O(m)$$

Suffix Link Construction

The suffix links are built using a depth-first search (DFS) traversal of the trie. Each node and edge is processed once, leading to a complexity of:

$$O(m)$$

Text Searching

Let n be the length of the text. During the search:

- If a transition exists in the trie, we move forward in $O(1)$.
- If there is a mismatch, suffix links guide the search backward.

In the worst case, suffix links may be followed multiple times. However, since each character in the text is processed at most once along with backtracking via suffix links, the total complexity remains:

$$O(n + m)$$

Best Case

If the text mostly follows the trie transitions without needing fail links, then each character is processed in $O(1)$, leading to:

$$O(n)$$

Worst Case

If suffix links are frequently used, a character may trigger multiple backtracking steps. However, across the entire search, the number of such backtracking steps is bounded by $O(n + m)$. Thus, the worst-case complexity is:

$$O(n + m)$$

Conclusion

The Aho-Corasick algorithm provides an efficient way to search multiple patterns in a text with the following time complexities:

- **Best case:** $O(n)$, when suffix links are rarely used.
- **Worst case:** $O(n+m)$, when suffix links cause multiple jumps.

5 Similarities and Dissimilarities Between Aho-Corasick and KMP

Similarities

Both Aho-Corasick and Knuth-Morris-Pratt (KMP) are efficient pattern matching algorithms that share the following characteristics:

1. **Preprocessing for Efficient Searching**: Both algorithms preprocess the pattern(s) before searching.
2. **Failure Links for Efficient Backtracking**: Both use suffix links to avoid unnecessary comparisons (KMP uses the Longest Prefix Suffix (LPS) array, while AC uses suffix links in a trie).
3. **Linear Time Complexity in Searching**: Both achieve an efficient search time of $O(n)$, where n is the length of the text.
4. **Pattern Matching Without Backtracking**: Neither algorithm requires explicit backtracking in the text; instead, they use precomputed failure information to move efficiently.

Dissimilarities

Feature	**Aho-Corasick (AC)**	**Knuth-Morris-Pratt (KMP)**
Purpose	Searches for multiple patterns simultaneously	Searches for a single pattern
Preprocessing Complexity	$O(M)$, where M is the total length of all patterns	$O(m)$, where m is the length of the pattern
Search Complexity	$O(n+z)$, where n is the text length and z is the number of matches	$O(n)$, where n is the text length
Data Structure Used	Trie with failure (or suffix) links	LPS (Longest Prefix Suffix) array
Space Complexity	$O(M \cdot \sigma)$ in worst-case, where σ is the alphabet size	$O(m)$
Pattern Matching Approach	Simultaneously matches multiple patterns	Sequentially matches a single pattern
Common Use Cases	Spam filtering, intrusion detection, dictionary matching	Substring search in text editors, string matching problems

Table 1: Comparison of Aho-Corasick and KMP Algorithms [H]

Conclusion

- **Aho-Corasick** is preferable when searching for multiple patterns at the same time.

- **KMP** is more suitable for efficiently searching a single pattern in a text.

6 Team Progress

No.	Member Name	Assigned Tasks	Completion	Timeline
1	Trịnh Duy Nhân	- Analyze and identify the Aho-Corasick algorithm - Preprocess multiple patterns using Trie data structure - Perform string matching - Write report sections 4 and 5 - Create PowerPoint slides	100%	12/03 – 03/04
		- Present the preprocessing and matching process of the Aho-Corasick algorithm, followed by a comparison of similarities and differences between the two algorithms	100%	05/04
2	Võ Đình Cao Minh Hào	- Analyze and identify the KMP algorithm - Preprocess strings using Trie - Perform string matching - Write report sections 2 and 6 - Create PowerPoint slides	100%	12/03 – 03/04
		- Present the preprocessing and string matching process of the KMP algorithm	100%	05/04
3	Phan Thanh Trúc Quân	- Write report sections 1, 3, and 7	100%	12/03 – 03/04
		- Present the definition, problem, and main idea of both KMP and Aho-Corasick algorithms	100%	05/04

7 Presentation Video

Click here to watch the video

References

[1] Trịnh Quang Anh. Thuật toán KMP. In *VNOI Wiki*, 2024.

[2] jakobkogler and tcNickolas. Prefix function. In *Algorithms for Competitive Programming*.