Pattern Searching using a Trie of all Suffixes

Problem Statement: Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

As discussed in the previous post, we discussed that there are two ways efficiently solve the above problem.

1) Preprocess Pattern: KMP Algorithm, Rabin Karp Algorithm, Finite Automata, Boyer Moore Algorithm.

2) Preoprocess Text: Suffix Tree

The best possible time complexity achieved by first (preprocssing pattern) is O(n) and by second

(preprocessing text) is O(m) where m and n are lengths of pattern and text respectively.

Note that the second way does the searching only in O(m) time and it is preferred when text doesn't doesn't change very frequently and there are many search queries. We have discussed Suffix Tree (A compressed Trie of all suffixes of Text) .

Implementation of Suffix Tree may be time consuming for problems to be coded in a technical interview or programming contexts. In this post simple implementation of a Standard Trie of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it's a simple Trie instead of compressed Trie.

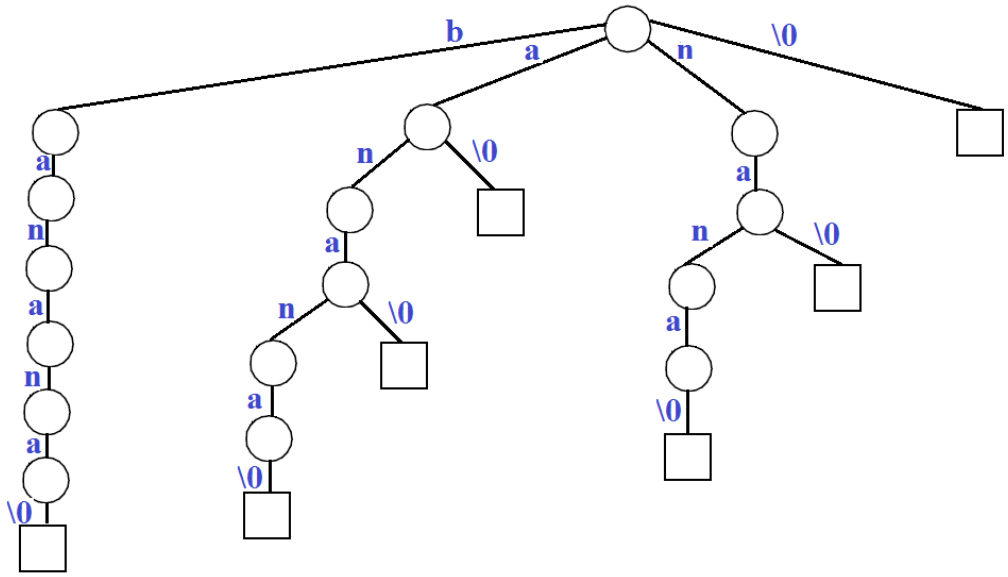
As discussed in Suffix Tree post, the idea is, every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. So if we build a Trie of all suffixes, we can find the pattern in O(m) time where m is pattern length.

Building a Trie of Suffixes
1) Generate all suffixes of given text.
2) Consider all suffixes as individual words and build a trie.

Let us consider an example text "banana\0″ where '\0′ is string termination character. Following are all suffixes of "banana\0″

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a Trie, we get following.



How to search a pattern in the built Trie?
Following are steps to search a pattern in the built Trie.
1) Starting from the first character of the pattern and root of the Trie, do following for every character.
…..**a)** For the current character of pattern, if there is an edge from the current node, follow the edge.
…..**b)** If there is no edge, print "pattern doesn't exist in text" and return.

2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print print all indexes where pattern is present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

Following is C++ implementation of the above idea.

```
// A simple C++ implementation of substring search using trie of suffixes
#include<iostream>
#include<list>
#define MAX_CHAR 256
using namespace std;

// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTrieNode
{
private:
    SuffixTrieNode *children[MAX_CHAR];
    list<int> *indexes;
public:
    SuffixTrieNode() // Constructor
    {
        // Create an empty linked list for indexes of
        // suffixes starting from this node
        indexes = new list<int>;

        // Initialize all child pointers as NULL
        for (int i = 0; i < MAX_CHAR; i++)
          children[i] = NULL;
    }

    // A recursive function to insert a suffix of the txt
    // in subtree rooted with this node
    void insertSuffix(string suffix, int index);

    // A function to search a pattern in subtree rooted
    // with this node.The function returns pointer to a linked
    // list containing all indexes where pattern is present.
    // The returned indexes are indexes of last characters
    // of matched text.
    list<int>* search(string pat);
};

// A Trie of all suffixes
class SuffixTrie
{
private:
    SuffixTrieNode root;
public:
    // Constructor (Builds a trie of suffies of the given text)
    SuffixTrie(string txt)
    {
        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
```

```
        // insertSuffix() in SuffixTrieNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substr(i), i);
    }

    // Function to searches a pattern in this suffix trie.
    void search(string pat);
};

// A recursive function to insert a suffix of the txt in
// subtree rooted with this node
void SuffixTrieNode::insertSuffix(string s, int index)
{
    // Store index in linked list
    indexes->push_front(index);

    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character
        char cIndex = s.at(0);

        // If there is no edge for this character, add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTrieNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1), index+1);
    }
}

// A recursive function to search a pattern in subtree rooted with
// this node
list<int>* SuffixTrieNode::search(string s)
{
    // If all characters of pattern have been processed,
    if (s.length() == 0)
        return indexes;

    // if there is an edge from the current node of suffix trie,
    // follow the edge.
    if (children[s.at(0)] != NULL)
        return (children[s.at(0)])->search(s.substr(1));

    // If there is no edge, pattern doesn't exist in text
    else return NULL;
}

/* Prints all occurrences of pat in the Suffix Trie S (built for text)*/
void SuffixTrie::search(string pat)
{
    // Let us call recursive search function for root of Trie.
    // We get a list of all indexes (where pat is present in text) in
```

```
    // variable 'result'
    list<int> *result = root.search(pat);

    // Check if the list of indexes is empty or not
    if (result == NULL)
        cout << "Pattern not found" << endl;
    else
    {
       list<int>::iterator i;
       int patLen = pat.length();
       for (i = result->begin(); i != result->end(); ++i)
         cout << "Pattern found at position " << *i - patLen<< endl;
    }
}

// driver program to test above functions
int main()
{
    // Let us build a suffix trie for text "geeksforgeeks.org"
    string txt = "geeksforgeeks.org";
    SuffixTrie S(txt);

    cout << "Search for 'ee'" << endl;
    S.search("ee");

    cout << "\nSearch for 'geek'" << endl;
    S.search("geek");

    cout << "\nSearch for 'quiz'" << endl;
    S.search("quiz");

    cout << "\nSearch for 'forgeeks'" << endl;
    S.search("forgeeks");

    return 0;
}
```

Output:

```
Search for 'ee'
Pattern found at position 9
Pattern found at position 1

Search for 'geek'
Pattern found at position 8
Pattern found at position 0

Search for 'quiz'
Pattern not found

Search for 'forgeeks'
Pattern found at position 5
```

Time Complexity of the above search function is O(m+k) where m is length of the pattern and k is the number of occurrences of pattern in text.

This article is contributed by Ashish Anand. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Related Topics:

- Recursively print all sentences that can be formed from list of word lists
- Check if a given sequence of moves for a robot is circular or not
- Find the longest substring with k unique characters in a given string
- Function to find Number of customers who could not get a computer
- Find maximum depth of nested parenthesis in a string
- Find all distinct palindromic sub-strings of a given string
- Find if a given string can be represented from a substring by iterating the substring “n” times
- Suffix Tree Application 6 – Longest Palindromic Substring

Tags: Pattern Searching

Writing code in comment? Please use ideone.com and share the link here.

tiny.So how can be used with efficient space utilization.

• Reply • Share ›

Anurag Singh ➚ Chandra Pratap Prajapati • 5 months ago

This is where suffix tree comes in. Searching All Patterns article talks about this.

• Reply • Share ›

Игорь Писарев • 8 months ago

using Ukkonen's algorithm the trie can be built in O(n).

• Reply • Share ›

Anurag Singh ➚ Игорь Писарев • 5 months ago

Searching All Patterns article talks about this.

• Reply • Share ›

Guest • 8 months ago

using Ukkonen's algorithm the trie can be built in O(n)

• Reply • Share ›

sk • 8 months ago

It's very expensive in case of large (length of txt) string .To build a Trie we need lot of computation as
1. build trie for N char
2. add N-1 char
.
.
N. add last char
now search so building trie is O(n^2) and search O(m)

• Reply • Share ›

Anurag Singh ➚ sk • 5 months ago

This is where suffix tree comes in. Searching All Patterns article talks about this.

• Reply • Share ›

Siva Krishna • 8 months ago

Here you are not considering the time for building suffix trie

2 • Reply • Share ›

Altair ➚ Siva Krishna • 8 months ago

That is what they meant by 'Preprocessing text' .

1 • Reply • Share ›

Detect Cycle in a Directed Graph · 2 hours ago

- Andy Toh

  My compile-time solution, which agrees with the...

  Dynamic Programming | Set 16 (Floyd Warshall Algorithm) · 2 hours ago

- lucy

  because we first fill zero in first col and...

  Dynamic Programming | Set 29 (Longest Common Substring) · 2 hours ago

- lucy

  @GeeksforGeeks i don't n know what is this long...

  Dynamic Programming | Set 28 (Minimum insertions to form a palindrome) · 3 hours ago

- manish

  Because TAN is not a subsequence of RANT. ANT...

  Given two strings, find if first string is a subsequence of second · 3 hours ago