C-Tree

AmirMohammad Dehghan
Iran Young Scholars Club
amirmd76@gmail.com

A new data structure for working with strings is presented in this paper which can reduce string matching problems to tree problems

Strings are too useful objects in programming, so good data structures can help us work faster or easier with them. In this paper, this paper is going to introduce you a new data structure that's useful in string matching, one of the most important problems with strings. This data structure can reduce many string matching problems to problems about trees, which are well-known and easier to think on.

1 Definition

Trie is one of the most useful data structures for working with strings.

In this paper, we consider that trie is a rooted tree with n vertices and there's a character on each edge and there is at most one edge outgoing from each vertex with a specific character on.

Also, for each vertex v, we define that $s(v)$ is the string we get when we wrote down all characters from the root to v contagiously. Also, we define par_v as the parent of vertex v in this trie. Of course $s(root)$ is empty.

We define f_v for each $v \neq root$ as the highest vertex in the trie like u such that $|s(u)| < |s(v)|$ and $s(u)$ is a substring of $s(v)$. Also, $f(root) = root$.

We can calculate $f_1, f_2, ..., f_n$ in $\mathcal{O}(nc)$ which c is the number of different characters in our Alphabet. First of all, determine the Aho-Corasick of this trie. Consider automaton $aut(v, c)$ is this Aho-Corasick.

Finally we can determine array f by running a dfs on this trie and determine this array recursively. i.e. $f_v = aut(f_{par_v}, ch)$ which ch is the character written on the edge $f_{par_v} - f_v$. $\mathcal{O}(nc)$ is for building Aho-Corasick and $\mathcal{O}(n)$ is for running dfs algorithm on the trie.

C-Tree is a rooted tree with n vertices and its root is the root of trie and for other vertices like v, parent of v in this tree is vertex f_v (it's a rooted tree because for each $v \neq root$, $height(f_v) < height(v)$ in the trie).

Logically, talking about C-Tree is meaningless without trie and Aho-Corasick.

2 Usages

Good thing about C-Tree is, for each (v, u), $s(v)$ is a suffix of string $s(u)$ if and only if v is a parent of u in this C-Tree.

For a string t, define $t(x)$ as its prefix of length x. Also, define $aut(t)$ the vertex we get in Aho-Corasick with giving its characters one by one to automaton aut (Aho-Corasick). i.e. $aut(t) = aut(aut(aut(..., aut(root, t_1), ..., t_{|t|-2}), t_{|t|-1}), t_{|t|})$

Also, consider boolean function $st(v, u)$ returns 1 if and only if v is an ancestor of u in the C-Tree.

This way, the number of occurrences of $S(v)$ in a string t is $\sum_{i=1}^{|t|} st(v, aut(t(i)))$.

So, C-Tree can be used for multiple string matchings in a fast way, and we can reduce string matching problems to tree problems, using it.

Also, among some strings, the largest common suffix of two of them that is a prefix of at least one of this strings, is lca of their end-vertex-in-their-trie in their C-tree.

3 Pseudo Code

Here are pseudo codes of Trie, Aho-Corasick and C-Tree. First of all, you should call function $BuildTrie$, then $BuildAho-Corasick$ and finally $C-Tree$.

Algorithm 1 Trie

```
1:  function BuildTrie(n, string[] s)
2:      root ← 1
3:      next ← 2
4:      new Trie trie[MaxSizeSum][c] ← 0
5:      for i = 1 to n do
6:          cur ← 1
7:          for character ch in s[i] do
8:              if trie[cur][ch] ≠ 0 then
9:                  cur ← trie[cur][ch]
10:             else
11:                 trie[cur][ch] ← next
12:                 next ← next + 1
13:                 cur ← trie[cur][ch]
14:     trie.size ← next − 1
        return trie
```

Algorithm 2 Aho-Corasick

```
1: function BuildAho-Corasick(Trie[][] trie)
2:     n ← trie.size
3:     f[n] ← 1 in all positions
4:     f.size ← n
5:     new Automation aut[n][c] ← 1
6:     queue q ← empty
7:     q.push(1)
8:     while q is not empty do
9:         v ← q.front
10:        q.pop()
11:        for i = 1 to c do
12:            if trie[v][i] ≠ 0 then
13:                q.push(trie[v][i])
14:                aut[v][i] ← trie[v][i]
15:                if v ≠ 1 then
16:                    f[trie[v][i]] ← aut[f[v]][i]
17:            else
18:                if v ≠ 1 then
19:                    aut[v][i] ← aut[f[v]][i]
        return aut and f
```

Algorithm 3 C-Tree

```
    function C-Tree(array[] f)
        n ← f.size
3:      Tree T ← new Tree of size n
        for i = 1 to n do
            T.par[i] ← f[i]
        return T
```

References

[1] de la Briandais, R. (1959). "File Searching Using Variable Length Keys". Proceedings of the Western Joint Computer Conference: 295–298.

[2] Aho, Alfred V.; Corasick, Margaret J. (June 1975). "Efficient string matching: An aid to bibliographic search". Communications of the ACM 18 (6): 333–340.