SAM

calabash_boy

2019 年 12 月 8 日

1 问题

现在需要对字符串 S 建立一种数据结构，需要资瓷识别子串操作。即询问字符串 T，判断 T 是否为 S 的子串，以及回答 T 在 S 中的出现次数，要求复杂度 $O(|T|)$。

1.1 Naive 的想法

首先有这样的一个事实：任何 S 的子串 $S[l,r]$，都是 S 的前缀的后缀。具体而言就是 $S[l,r]$ 是 $S[1,r]$ 的后缀。同时每个子串也都是原串的后缀的前缀。

1.2 Naive 的实现

将原串的每一个后缀插入到 $Trie$ 中，这样 $Trie$ 中的每个点，代表的都是原串的后缀的前缀，于是可以实现识别每个子串。预处理复杂度 $O(N^2)$.

1.3 right 集合定义：

对于 S 的每一个子串 T 而言，T 在 S 中的某一些位置出现，那么将 T 在 S 中出现位置的集合定义为 **right 集合**。

1.4 显然的观察 1

在 **Naive 的实现**中的 $Trie$ 中，有很多状态可以合并，即观察到有一些状态代表的子串，其出现次数是相同的，更严格的讲：其每一个出现位置都是相同的。即 **right 集合**相同。

1.5 显然的观察 2

在**显然的观察 1** 中，提出了按照 **right 集合**划分子串的等价类的思想。那么对于任意一个 **right 集合**，我们找到这个等价类中的最长的子串, 记为 $T_{longest}$。那么其他 **right 集合**相同的子串，一定都是 $T_{longest}$ 的后缀。而且是连续的一些后缀，多么显然。。。

1.6 后缀的符号表示

使用 $S_T(length)$ 表示串 T 的长度为 $length$ 的后缀。

1.7 显然的观察 3

令 $T = S$，在**显然的观察 2** 中, 我们知道 T 的一些长度最长的后缀的 **right 集合**和 T 是相同的，于是存在一个分界点，记为 l_p，即 $\forall i \in [l_p + 1, n], S_T(i)$ 和 T 的 **right 集合**相同。而 $\forall i \in [1, l_p], S_T(i)$ 的 **right 集合**都比 T 的要大，翻译一下就是说。。。更短的串，出现位置更多。。。

1.8 后缀链接

至此，对于 T，我们可以选取若干分点，记为 $l_1 = 0, l_2, l_3, ...l_m = n$, 使得 $\forall i \in [l_x + 1, l_{x+1}], S_T(i)$ 都具有相同的 **right 集合**。将他们用一条链链接起来，就形成了**后缀链接**。

1.9 对 Naive 的实现进行优化

由于我们有了很多**显然的观察**，我们现在使用他们对 **Naive 的实现**进行优化，由于我们已经懂了**后缀链接**的意义，因此，我们只需要对 S 的每一个前缀，求解出其完整后缀链接，我们便得到了所有有用的状态。这也就是 SAM 所做的事情。同时也是**后缀树**所做的事情。。。（一下学了两个算法，好开心）

---

2 后缀自动机

2.1 定义

后缀自动机是一个自动机，它能够识别原串的每一个子串。翻译一下：他是一个 DAG，每条边上有一个字符，每一个子串都可以在这个 DAG 上爬出来。其中 DAG 上的每个点与我们上面讲的状态对应。

2.2 符号表示

$nxt[x][c]$ 表示从 x 点，通过字符 c 可以到达的点。

$l[x]$ 表示 x 点所能表示的最长子串长度。

$fa[x]$ 表示 x 在后缀链接上的父亲。

2.3 在线构造 SAM

假设我们已经构造了 $S[1,x]$ 的 SAM，即我们已经可以识别 $S[1,x]$ 的所有子串，我们考虑如何修改得到 $S[1,x+1]$ 的 SAM，即向 SAM 中添加一个字符 $c=S[x+1]$。

显然我们只考虑 $S[1,x+1]$ 的所有后缀。也就是说我们要将 $S[1,x+1]$ 的完整后缀链接放到 SAM 中去。

首先需要新建一个点 np 用来表示 $S[1,x+1]$，他的 **right 集合**用已有 SAM 节点无法表示。记 $last$ 表示 $S[1,x]$ 所在的点。那么显然令 $nxt[last][c]=np$,$l[np]=l[last]+1$.

接下来我们要确定 $fa[np]$。由于 $S[1,x+1]$ 的每一个后缀，都是 $s[1,x]$ 的后缀增加一个字符 $c=S[x+1]$ 得到，因此我们其实只需要考察 $S[1,x]$ 的 **right 集合**就可以得知 $S[1,x+1]$ 的 **right 集合**如何变化。

沿着 $last$ 的后缀链接走，如果一个点 $p=fa[fa..[last]]$ 没有字符 c 的转移，那么增加 $nxt[x][c]=np$.

直到我们找到了后缀链接上某点 $p=fa[fa...[last]]$，他原本已经有字符 c 的转移，令 $q=nxt[p][c]$, 显然我们不需要再添加字符 c 的转移了，那么剩下的事情只剩下 $fa[np]$ 等于啥。

由**后缀链接**的定义，如果 $l[q]=l[p]+1$，则 $fa[np]=q$。

如果 $l[q]>l[p]+1$，you **后缀链接**定义，我们指导 $fa[np]$ 节点的 l 一定等于 $l[p]+1$，因此我们将原节点 q，分割为两部分：一部分代表长度

$[l[p]+2, l[q]]$，另一部分代表长度 $[l[fa[q]]+1, l[p]+1]$.

分割出的两个点，显然应该拥有相同的转移，即 nxt 相同。所以新建节点 nq，令 $nxt[nq]=nxt[q]$，$l[nq]=l[p]+1$，这一步实现了分割长度。然后令 $fa[nq]=fa[q]$，$fa[q]=nq$，这一步保证了后缀连接的正确性，这一步可以简单的类比于在链表中插入一个节点。之后我们就可以令 $fa[np]=nq$ 了。之后修改 p 的剩下的后缀链接，使得原来通过字符 c 指向 q 的，都重新指向 nq.

2.4 复杂度证明

简单的由 1.8，复杂度并不能很清楚的证明。

但是由 2.3，我们很清楚的看到：复杂度为 $O(|\Sigma|n)$

好像跟说了句：复杂度显然正确，没啥区别。。。

一切确实就是这么显然。。。

---

3 应用

回到我们的问题，我们要识别每个子串，很容易，在 SAM 上沿着字符边爬，能爬到就是子串，爬不到就不是子串。

emmmm。我们还需要统计子串出现次数。

3.1 统计子串出现次数

SPOJ Substrings

https://www.spoj.com/problems/NSUBSTR

由于后缀链接的定义，我们知道在 SAM 的一个节点中，它代表了一些连续长度的串，且其 right 集合相同。那么必然的，出现次数也相同，等于 right 集合大小。

如何求 right 集合大小呢，我们考虑每一个前缀 $\forall x \in [1, n], S[1, x]$，我们找到所有包含 x 这个位置的节点，给他们的 right 集合大小 +1 即可。显然这些节点是前缀 $S[1, x]$ 代表的整条后缀链接。

而由构造过程观察到所有点的后缀链接，其实形成的是一棵树。

那么问题变成了，树上 n 次修改，每次把根到一个点的路径都 $+1$，仿佛闻到了傻逼题的味道？

做法为：开一个 cnt 数组，将 S 在 SAM 上运行一遍，将每个前缀所在点的 $cnt = 1$. 然后进行 树上差分dp，或者**拓扑更新**，whatever。

3.2 统计本质不同子串个数

BZOJ 4516

又闻到了傻逼题的味道？

$Ans = \sum l[x] - l[fa[x]]$

3.3 后缀树的拓扑序

我们发现在使用 SAM 的时候，经常需要在后缀树上进行**拓扑更新**，而递归函数常数巨大，常规的拓扑排序……常数也不小。对于这样一个特殊的后缀树，我们是否可以快速处理出拓扑序，然后避免递归呢？

使用基数排序，以每个节点 x 的 $l[x]$ 为关键字，排序之后即为拓扑序。因为沿着后缀树往根的方向走，$l[x]$ 单调变小。

---

4 一些不厉害的题目

HDU 4641

POJ 1509

5 一些厉害的题目

codeforces gym 100962 D Deep Purple

SPOJ 7258

6 模板

https://github.com/4thcalabash/ACM-Code-Library /blob/master/String/AutoMachine/Suffix_Automaton.cpp