Suffix Cactus

王悦同, 徐毅, 徐子涵

江苏省南京外国语学校, 江苏省常州高级中学

February 6, 2014

1. 引入
   - 符号约定
   - 相关概念
   - 中庸之道——后缀仙人掌
2. 原理
   - 定义
   - 性质
3. 使用
   - 构造性质
   - 构造方法
   - 查询
4. 扩展
   - 效率
5. 总结

符号约定

符号约定

- 令 $T = t_1 t_2 \ldots t_n$ 为字符集 Σ 上的字符串。

符号约定

- 令 $T = t_1 t_2 \dots t_n$ 为字符集 Σ 上的字符串。
- 子串 $T_i^j = t_i t_{i+1} \dots t_j (1 \le i \le j \le n)$。

符号约定

- 令 $T = t_1 t_2 \ldots t_n$ 为字符集 Σ 上的字符串。
- 子串 $T_i^j = t_i t_{i+1} \ldots t_j (1 \leq i \leq j \leq n)$。
- 后缀 $T_i = T_i^n$。

符号约定

- 令 $T = t_1 t_2 \dots t_n$ 为字符集 Σ 上的字符串。
- 子串 $T_i^j = t_i t_{i+1} \dots t_j (1 \le i \le j \le n)$。
- 后缀 $T_i = T_i^n$。
- 前缀 $T^j = T_1^j$。

符号约定

- 令 $T = t_1 t_2 \ldots t_n$ 为字符集 Σ 上的字符串。
- 子串 $T_i^j = t_i t_{i+1} \ldots t_j (1 \leq i \leq j \leq n)$。
- 后缀 $T_i = T_i^n$。
- 前缀 $T^j = T_1^j$。
- 如不加说明，举例字符串为 cabacca\$，\$ 标记串的结束。

相关概念

相关概念

后缀 Trie

相关概念

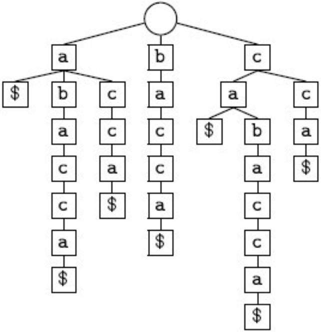
后缀 Trie

$STr(T)$ 为文本串 T 的所有后缀构成的 Trie，其构造时空复杂度均为 $O(n^2)$。

相关概念

后缀 Trie

$STr(T)$ 为文本串 T 的所有后缀构成的 Trie，其构造时空复杂度均为 $O(n^2)$。



相关概念

相关概念

后缀树

相关概念

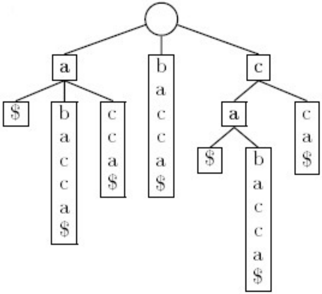
后缀树

将后缀 Trie 中只有一个儿子的结点与儿子合并，其构造时空复杂度均为 $O(n)$。

相关概念

后缀树

将后缀 Trie 中只有一个儿子的结点与儿子合并，其构造时空复杂度均为 $O(n)$。



相关概念

相关概念

后缀数组

相关概念

后缀数组

将文本串的所有后缀按字典序排列，其构造时空复杂度均为 $O(n)$。通常使用倍增算法构造，则时间复杂度为 $O(n\log n)$。

相关概念

后缀数组

将文本串的所有后缀按字典序排列，其构造时空复杂度均为 $O(n)$。通常使用倍增算法构造，则时间复杂度为 $O(n \log n)$。

7	2	4	3	6	1	5
a	a	a	b	c	c	c
$	b	c	a	a	a	c
	a	c	c	$	b	a
	c	a	c		a	$
	c	$	a		c	
	a		$		c	
	$				a	
					$	

中庸之道——后缀仙人掌

中庸之道——后缀仙人掌

- 后缀树进行在线多串匹配的时间复杂度为线性，但其较大的代码量不让人满意。

中庸之道——后缀仙人掌

- 后缀树进行在线多串匹配的时间复杂度为线性，但其较大的代码量不让人满意。
- 后缀数组容易理解且代码量小，但进行在线多串匹配需要二分，同样有所缺憾。

中庸之道——后缀仙人掌

- 后缀树进行在线多串匹配的时间复杂度为线性，但其较大的代码量不让人满意。
- 后缀数组容易理解且代码量小，但进行在线多串匹配需要二分，同样有所缺憾。
- 此时，后缀树与后缀数组的交叉产品后缀仙人掌就有用武之地了。

定义

定义

- 后缀仙人掌与后缀树类似，都基于后缀 Trie。后缀树是将只有一个儿子的结点与儿子合并，而后缀仙人掌是将每个非叶子结点都与一个儿子合并，所形成的连接体称为树枝。

定义

- 后缀仙人掌与后缀树类似，都基于后缀 Trie。后缀树是将只有一个儿子的结点与儿子合并，而后缀仙人掌是将每个非叶子结点都与一个儿子合并，所形成的连接体称为树枝。
- 令 v 为文本串 T 的后缀 Trie $STr(T)$ 的一个结点，满足 v 是根或 v 不是其父亲 w 的第一个儿子（儿子顺序为字典序）。那么，文本串 T 的后缀仙人掌 $SC(T)$ 中有树枝 s 恰好包含从结点 v 到 v 下第一个叶子 u 路径上的所有结点。

定义

定义

- 显然，$STr(T)$ 的每个结点都被 $SC(T)$ 的恰好一根树枝包含。包含 $STr(T)$ 的根的树枝称为根树枝。结点 v 称为树枝 s 的根，结点 u 称为树枝 s 的叶子，结点 w 称为树枝 s 的父亲结点。树枝 s 的深度用 $DEPTH(s)$ 表示，等于结点 w 的深度。根树枝的深度为 0。

定义

- 显然，$STr(T)$ 的每个结点都被 $SC(T)$ 的恰好一根树枝包含。包含 $STr(T)$ 的根的树枝称为根树枝。结点 v 称为树枝 s 的根，结点 u 称为树枝 s 的叶子，结点 w 称为树枝 s 的父亲结点。树枝 s 的深度用 $DEPTH(s)$ 表示，等于结点 w 的深度。根树枝的深度为 0。
- 树枝 s 包含的字符串由 s 包含的结点对应的字符构成，但 s 表示的字符串由根到结点 u 路径上的结点对应的字符构成。因此，$SC(T)$ 的树枝与文本串 T 的后缀一一对应。树枝 s 表示的后缀在文本串 T 中的起始位置用 $SUFFIX(s)$ 表示。那么，树枝 s 包含的字符串为 $T_{SUFFIX(s)+DEPTH(s)}$。

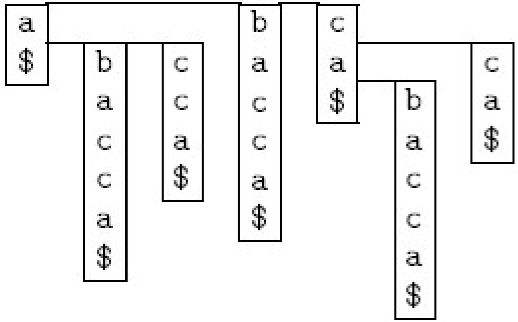
定义

定义

- 与常识不同的是，树枝 s 的父亲树枝为包含结点 v 的左兄弟的树枝。除了根树枝外所有的树枝均有父亲树枝。

定义

- 与常识不同的是，树枝 s 的父亲树枝为包含结点 v 的左兄弟的树枝。除了根树枝外所有的树枝均有父亲树枝。



性质

性质

- 为了方便起见，我们将树枝以其表示的后缀在文本串 T 所有后缀中的字典序排位来编号。因此，$T_{SUFFIX(s)}$ 就是在文本串 T 所有后缀中字典序排第 s 位的树枝。树枝 1 就是根树枝。

性质

- 为了方便起见，我们将树枝以其表示的后缀在文本串 T 所有后缀中的字典序排位来编号。因此，$T_{SUFFIX(s)}$ 就是在文本串 T 所有后缀中字典序排第 s 位的树枝。树枝 1 就是根树枝。
- 由于性质都比较直观，在这里不再给出证明，如需要可参考相关论文。

性质

- 为了方便起见，我们将树枝以其表示的后缀在文本串 T 所有后缀中的字典序排位来编号。因此，$T_{SUFFIX(s)}$ 就是在文本串 T 所有后缀中字典序排第 s 位的树枝。树枝 1 就是根树枝。
- 由于性质都比较直观，在这里不再给出证明，如需要可参考相关论文。

性质 1

性质

- 为了方便起见，我们将树枝以其表示的后缀在文本串 T 所有后缀中的字典序排位来编号。因此，$T_{SUFFIX(s)}$ 就是在文本串 T 所有后缀中字典序排第 s 位的树枝。树枝 1 就是根树枝。
- 由于性质都比较直观，在这里不再给出证明，如需要可参考相关论文。

性质 1

$$DEPTH(s) = LCP(T_{SUFFIX(s)}, T_{SUFFIX(s-1)})$$

，树枝 s 非根树枝（即 $s > 1$）。

性质

性质

性质 2

性质

性质 2

树枝 $r(r > 1)$ 的父亲树枝是满足 $DEPTH(s) \le DEPTH(r)$ 的比 r 小的编号最大的树枝 s。

性质

性质 2

树枝 $r(r > 1)$ 的父亲树枝是满足 $DEPTH(s) \le DEPTH(r)$ 的比 r 小的编号最大的树枝 s。

性质 3

性质

性质 2

树枝 $r(r>1)$ 的父亲树枝是满足 $DEPTH(s) \leq DEPTH(r)$ 的比 r 小的编号最大的树枝 s。

性质 3

树枝 s 有儿子树枝当且仅当树枝 $s+1$ 是树枝 s 的儿子，此时有

$$s+1 = r_k < \cdots < r_1$$

，$r_1, r_2, \ldots, r_k$ 为 s 从高到低的儿子树枝。

构造

构造

可以看出，后缀仙人掌其实和后缀树还是很相似的——可以视为将后缀树用“左孩子右兄弟”表示法转化后的产物。

构造

可以看出，后缀仙人掌其实和后缀树还是很相似的——可以视为将后缀树用“左孩子右兄弟”表示法转化后的产物。

正是因此，在后缀仙人掌上可以进行很多在后缀树上才能进行（而后缀数组不行）的操作。因此它真正的优势在于：简洁的构建方式。

构造

可以看出，后缀仙人掌其实和后缀树还是很相似的——可以视为将后缀树用“左孩子右兄弟”表示法转化后的产物。

正是因此，在后缀仙人掌上可以进行很多在后缀树上才能进行（而后缀数组不行）的操作。因此它真正的优势在于：简洁的构建方式。

下面给出构造时能用到的几个性质，并介绍一种简单的构建方法。

构造

性质 1

构造

性质 1

后缀仙人掌可以按某种方式构造，使得枝条“从左到右”的顺序和原字符串对应的后缀数组一致。

性质 2

构造

性质 1

后缀仙人掌可以按某种方式构造，使得枝条“从左到右”的顺序和原字符串对应的后缀数组一致。

性质 2

后缀仙人掌如果每个“枝条”画出的长度和对应字符串的长度成正比，那么后缀仙人掌是可平面的。

构造

性质 1

后缀仙人掌可以按某种方式构造，使得枝条“从左到右”的顺序和原字符串对应的后缀数组一致。

性质 2

后缀仙人掌如果每个“枝条”画出的长度和对应字符串的长度成正比，那么后缀仙人掌是可平面的。

性质 2 的证明十分容易：

构造

性质 1

后缀仙人掌可以按某种方式构造，使得枝条“从左到右”的顺序和原字符串对应的后缀数组一致。

性质 2

后缀仙人掌如果每个“枝条”画出的长度和对应字符串的长度成正比，那么后缀仙人掌是可平面的。

性质 2 的证明十分容易：
从性质 1 看出，如果 T_x 和 T_z 的枝条“穿过”了 T_y 后缀的枝条 $(x < y < z)$

构造

性质 1

后缀仙人掌可以按某种方式构造，使得枝条“从左到右”的顺序和原字符串对应的后缀数组一致。

性质 2

后缀仙人掌如果每个“枝条”画出的长度和对应字符串的长度成正比，那么后缀仙人掌是可平面的。

性质 2 的证明十分容易：

从性质 1 看出，如果 T_x 和 T_z 的枝条“穿过”了 T_y 后缀的枝条 $(x < y < z)$

那么在后缀数组里，T_x 和 T_z 的 LCP 必定不是 height 数组的 min 值了。

构造

构造

下面我们利用后缀数组来构造后缀仙人掌。

构造

下面我们利用后缀数组来构造后缀仙人掌。
在后缀数组中，我们可以以 $O(N \log N)$ 的时间复杂度计算出 $suffix$ 数组（也就是排名对应原数组的位置），然后 $O(N)$ 计算出 $height$ 数组（相邻名次的 LCP）。

构造

下面我们利用后缀数组来构造后缀仙人掌。
在后缀数组中，我们可以以 $O(N \log N)$ 的时间复杂度计算出 $suffix$ 数组（也就是排名对应原数组的位置），然后 $O(N)$ 计算出 $height$ 数组（相邻名次的 LCP）。
我们可以发现，在后缀仙人掌里，每个“枝条”的深度恰好等于该后缀相应的 $height$ 值。

构造

构造

我们定义一个概念“活跃枝条”。一个活跃枝条可能成为以后枝条的父亲。

构造

我们定义一个概念“活跃枝条”。一个活跃枝条可能成为以后枝条的父亲。

一开始只有 1 是活跃枝条。以后，每当插入一个枝条 x 的时候，我们在当前所有 depth 值 $\leq$ 当前后缀的活跃枝条中选择最靠后的那一个。具体如下：

构造

我们定义一个概念“活跃枝条”。一个活跃枝条可能成为以后枝条的父亲。
一开始只有 1 是活跃枝条。以后，每当插入一个枝条 x 的时候，我们在当前所有 depth 值 $\leq$ 当前后缀的活跃枝条中选择最靠后的那一个。具体如下：

- 依次从后往前扫描所有的活跃枝条，如果 $depth_c \geq depth_x$，那么枝条 c 不再是活跃枝条；

构造

我们定义一个概念“活跃枝条”。一个活跃枝条可能成为以后枝条的父亲。
一开始只有 1 是活跃枝条。以后，每当插入一个枝条 x 的时候，我们在当前所有 depth 值 $\leq$ 当前后缀的活跃枝条中选择最靠后的那一个。具体如下：

- 依次从后往前扫描所有的活跃枝条，如果 $depth_c \geq depth_x$，那么枝条 c 不再是活跃枝条；
- 找到需要的枝条 k 后，插入边 $(k, depth_x, x)$，表示枝条 k 在深度 $depth_x$ 处有一条指向枝条 x 的边。

构造

我们定义一个概念“活跃枝条”。一个活跃枝条可能成为以后枝条的父亲。

一开始只有 1 是活跃枝条。以后，每当插入一个枝条 x 的时候，我们在当前所有 depth 值 $\leq$ 当前后缀的活跃枝条中选择最靠后的那一个。具体如下：

- 依次从后往前扫描所有的活跃枝条，如果 $depth_c \geq depth_x$，那么枝条 c 不再是活跃枝条；
- 找到需要的枝条 k 后，插入边 $(k, depth_x, x)$，表示枝条 k 在深度 $depth_x$ 处有一条指向枝条 x 的边。
- 将 x 加入“活跃枝条”。

构造

构造

这段过程的实现如下：

构造

这段过程的实现如下：

```
top = 1;stk[top] = 1;
for (int i = 2; i <= N; i++) {
    while (depth[stk[top]] > depth[i]) --top;
    pushit(stk[top] ,depth[i] ,i);
    ++top;
    stk[top] = i;
}
```

构造

这段过程的实现如下：

```
top = 1;stk[top] = 1;
for (int i = 2; i <= N; i++) {
    while (depth[stk[top]] > depth[i]) --top;
    pushit(stk[top] ,depth[i] ,i);
    ++top;
    stk[top] = i;
}
```

$pushit()$ 是插入一条边的操作，实现时采用将三元组存入 Hash 表的方式，这样以后匹配复杂度就可以保证。

查询

查询

以多串匹配为例，我们来展示一下查询时的操作方式：

查询

以多串匹配为例，我们来展示一下查询时的操作方式：

```
nowx = 1; nowy = 0;
for (int j = 0; j < M; j++) {
    cs = st[j];
    flag = 1;
    while ((suffix[nowx] + nowy > N) || (a[suffix[nowx] + nowy] != cs)) {
        ct = gethash(nowx,nowy);
        if (ct == -1) {
            flag = 0;
            break;
        }
        nowx = ct;
    }
    if (flag == 0) {
        printf("%d\n" ,j);
        break;
    }
    ++nowy;
}
```

查询

查询

在查询开始时，我们位于第一个枝条的首部。

查询

在查询开始时，我们位于第一个枝条的首部。

每次读入一个字符，看当前枝条能否沿着向下走；如果可以，就向下走一步，否则横着沿着边走，直到遇到能向下走的枝条。如果哪一次向下不能走时没有向右的枝条，则匹配结束。

查询

在查询开始时，我们位于第一个枝条的首部。

每次读入一个字符，看当前枝条能否沿着向下走；如果可以，就向下走一步，否则横着沿着边走，直到遇到能向下走的枝条。如果哪一次向下不能走时没有向右的枝条，则匹配结束。

这个过程非常易于理解和实现。对于多串匹配问题，构建后缀仙人掌可以将查询操作的复杂度降到 $\mathrm{O(M)}$（ M 是查询串总长度）。

测试结果

对于 $N = 100000$ 的母串，$Q = 10^7$ 个询问，每个询问长度是 $1 \sim 100$ 的等概率随机数。字符集大小为 2。

测试结果

对于 $N = 100000$ 的母串，$Q = 10^7$ 个询问，每个询问长度是 $1 \sim 100$ 的等概率随机数。字符集大小为 2。

```
The current time is: 14:25:04.54
Enter the new time:
The current time is: 14:25:43.15
Enter the new time:
The current time is: 14:26:08.49
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
FC: no differences encountered

The current time is: 14:26:52.78
Enter the new time:
The current time is: 14:27:37.08
Enter the new time:
The current time is: 14:28:02.60
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
FC: no differences encountered

The current time is: 14:28:48.03
Enter the new time:
The current time is: 14:29:27.06
Enter the new time:
The current time is: 14:29:52.73
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
```

测试结果

对于 $N = 100000$ 的母串，$Q = 10^7$ 个询问，每个询问长度是 $1 \sim 100$ 的等概率随机数。字符集大小为 2。

```
The current time is: 14:25:04.54
Enter the new time:
The current time is: 14:25:43.15
Enter the new time:
The current time is: 14:26:08.49
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
FC: no differences encountered

The current time is: 14:26:52.78
Enter the new time:
The current time is: 14:27:37.08
Enter the new time:
The current time is: 14:28:02.60
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
FC: no differences encountered

The current time is: 14:28:48.03
Enter the new time:
The current time is: 14:29:27.06
Enter the new time:
The current time is: 14:29:52.73
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
```

```
The current time is: 14:30:37.64
Enter the new time:
The current time is: 14:31:19.21
Enter the new time:
The current time is: 14:31:46.33
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
FC: no differences encountered

The current time is: 14:32:31.63
Enter the new time:
The current time is: 14:33:13.11
Enter the new time:
The current time is: 14:33:38.62
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
FC: no differences encountered

The current time is: 14:34:24.18
Enter the new time:
The current time is: 14:35:05.82
Enter the new time:
The current time is: 14:35:31.34
Enter the new time:
Comparing files symbol.out and SYMBOL.ANS
FC: no differences encountered
```

测试结果

测试结果

一共测试了 10 组数据，后缀数组的平均时间为 $38s$，后缀仙人掌平均时间为 $25s$。除去读入消耗的 $11s$，后缀仙人掌执行时间约是后缀数组的一半。

测试结果

一共测试了 10 组数据，后缀数组的平均时间为 $38s$，后缀仙人掌平均时间为 $25s$。除去读入消耗的 $11s$，后缀仙人掌执行时间约是后缀数组的一半。

对字符集为 4 的时候做了类似的测试，实践表明后缀仙人掌速度有所减慢，约为 $30s$；后缀数组则基本不变，大概为 $40s$ 不到。

可行的优化

可行的优化

后缀数组构造优化

可行的优化

后缀数组构造优化

使用 DC3 等在线性时间内构造后缀数组的方法，可以把构造的时间复杂度降低到 $O(N)$。

可行的优化

后缀数组构造优化

使用 DC3 等在线性时间内构造后缀数组的方法，可以把构造的时间复杂度降低到 $O(N)$。

字符集优化

可行的优化

后缀数组构造优化

使用 DC3 等在线性时间内构造后缀数组的方法，可以把构造的时间复杂度降低到 $O(N)$。

字符集优化

对于字符集较大的情况，实际上在实现的时候可以把连续的若干个兄弟合并起来。这样下来，复杂度与字符集无关。

可行的优化

后缀数组构造优化

使用 DC3 等在线性时间内构造后缀数组的方法，可以把构造的时间复杂度降低到 $O(N)$。

字符集优化

对于字符集较大的情况，实际上在实现的时候可以把连续的若干个兄弟合并起来。这样下来，复杂度与字符集无关。

经过这样的两个优化，后缀仙人掌的时间复杂度可以完全和后缀树等同，而实现比后缀树更加简洁。

总结

总结

后缀仙人掌的优势：

总结

后缀仙人掌的优势：

- 代码实现简单。构造的过程只需要后缀数组即可，查找代码也很短。

总结

后缀仙人掌的优势：

- 代码实现简单。构造的过程只需要后缀数组即可，查找代码也很短。
- 时间复杂度低于后缀数组，常数也比后缀树要小。

总结

后缀仙人掌的优势：

- 代码实现简单。构造的过程只需要后缀数组即可，查找代码也很短。
- 时间复杂度低于后缀数组，常数也比后缀树要小。
- 比后缀自动机更为简洁、明确，方便理解。

总结

后缀仙人掌的优势：

- 代码实现简单。构造的过程只需要后缀数组即可，查找代码也很短。
- 时间复杂度低于后缀数组，常数也比后缀树要小。
- 比后缀自动机更为简洁、明确，方便理解。

总而言之，后缀仙人掌是介于后缀数组和后缀树之间的一种良好的折衷方案，实现简单，便于理解。在一些本来需要使用后缀树的场合，使用后缀仙人掌是一种良好的取代方案。