

Preface

目前中英文站点上并没有很好的 Suffix Automaton (SAM) 材料, 比较知名的材料有两个, 一是陈立杰的营员交流资料, 二是某俄国人的某 SAM 博文, 但这两个材料都不好懂. 在下反复啃了几天后, 略有所得, 在此记录一下.

Table Of Content

Preface
Table Of Content
Introduction
Automaton: The Definition
State Of SAM
State's Suffix Link
State's Transitions
SAM Online Construction: The Principle
SAM Online Construction: The Pseudocode
SAM: The Time/Space Complexity

Introduction

较为靠谱的 SAM 资料:

- [A short guide to suffix automata - Codeforces](#)
- [MAXimal :: algo :: Суффиксный автомат. Построение и применения](#), 用 [Yandex 提供的机翻服务](#) 勉强可以看懂.
- [后缀自动机：O\(N\)的构建及应用 - wmdcstdio的专栏 - 博客频道 - CSDN.NET](#), 上文的中文翻译, 翻译质量良好.

TL;DR:

- SAM 与 suffix tree 是等价的.
- 建立 SAM 的 time/space complexity: $O(n)$.
- 实现的代码很简单 (小于 40 行), 但理解起来有难度.

SAM 是处理 String 相关问题的有力工具, 包括但不限于以下问题:

- Verify W is substring of T .
- Verify W is suffix of T .
- Count the number of unique substrings in T .
- Count the total length of unique substrings in T .
- Count the number of W in T .
- Find the first position of W in T .
- Find all positions of W in T .
- Find the longest common substring of T_1 and T_2 .
- Find the longest common substring from T_1 to T_k .

- Find the smallest cyclic shift of T .
- Find the k th smallest substring of T .
- Find the shortest string that is not a substring of T .

Automaton: The Definition

Suffix Automaton 里的 Automaton 其实就是 [DFA](#), 以下是 Wiki 中的定义:

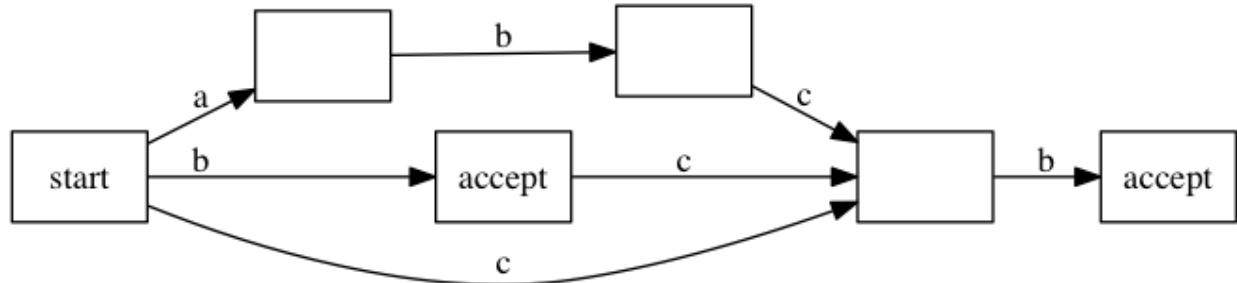
A **deterministic finite automaton** M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of

- a finite set of [states](#) (Q)
- a finite set of input symbols called the [alphabet](#) (Σ)
- a transition [function](#) ($\delta : Q \times \Sigma \rightarrow Q$)
- an initial or [start state](#) ($q_0 \in Q$)
- a set of [accept states](#) ($F \subseteq Q$)

Suffix Automaton, 顾名思义就是可以识别 Suffix 的 Automaton, [A short guide to suffix automata - Codeforces](#) 中有如下的描述:

A **suffix automaton** A for a string s is a minimal finite automaton that recognizes the suffixes of s . This means that A is a directed acyclic graph with an initial node i , a set of *terminal* nodes, and the arrows between nodes are labeled by letters of s . To test whether w is a suffix of s it is enough to start from the initial node of i and follow the arrows corresponding to the letters of w . If we are able to do this for every letter of w and end up in a terminal node, w is a suffix of s .

以下图为例, 下图是 "abcb" 的对应 SAM:



可以看到, 从 start state 出发的 paths, "abcb", "bcb", "cb", "b", 都能到达 accept state, 这些路径的集合即为 "abcb" suffixes 的集合.

State Of SAM

一个关键的问题是, SAM 里的 State 到底有什么含义?

在介绍 State 的含义之前, 需要先了解一个定义. 给定 String T 与 s , 定义

$\text{endpos}(T, s) = \{\text{end} | T[\text{begin} : \text{end}] = s\}$. 也就是说, $\text{endpos}(T, s)$ 表示了所有在 T 中出现的 s 的结束位置(下标). 如果有两个 Strings, s_1 与 s_2 , 如果有 $\text{endpos}(T, s_1) = \text{endpos}(T, s_2)$, 则 s_1 与 s_2 是 **endpos-equivalent** 的.

在上一小节的例子中, 可以看到每个 state 可以有 ≥ 1 的 indegrees, 这意味着 state u 可以识别 T 的某个 substrings 集合, 设这个集合为 $\text{substrings}(u)$. $\text{substrings}(u)$ 有如下的 property:

对于 SAM 中的任意 state u , $\text{substrings}(u)$ 中的任意两个 substrings 必然是 **endpos-equivalent** 的. 以 $T = "abcb"$ 为例(见上小节的图例), 显然的, “ abc ”, “ bc ”, “ c ” 是 endpos-equivalent 的, 他们的 **endpos** 集合都是 $\{2\}$.

所以, 可以根据 **endpos** 的定义, 将 T 的所有 substrings 分成若干个类, 使每个类中的 substrings 都是 endpos-equivalent 的, 这样的分类结果会与 SAM 中的 states 一一对应.

为了进一步明确 state 的含义, 我们先来思考一个问题: 对于 T 的两个任意的 substrings s_1 与 s_2 , 集合 $\text{endpos}(T, s_1)$ 与 $\text{endpos}(T, s_2)$ 的关系会是怎样的?

For generality, 假设 $\text{length}(s_1) \leq \text{length}(s_2)$. 如果 $\text{endpos}(s_1) \cap \text{endpos}(s_2) \neq \emptyset$, 也就是说, s_1 与 s_2 至少会同时以 T 某个字符作为结尾. 所以, $\text{endpos}(s_1) \cap \text{endpos}(s_2) \neq \emptyset \implies s_1 \sqsupset s_2$, 即 s_1 是 s_2 的 suffix (注意, 包含 $s_1 = s_2$ 的情况).

显然的, 如果 $s_1 \sqsupset s_2$, 则 $\text{endpos}(s_1) \supseteq \text{endpos}(s_2)$, 毕竟 substring 越长, 对 **endpos** 的限制就越多, 集合就越小. 再有, 如果 $\text{endpos}(s_1) \supseteq \text{endpos}(s_2)$, 则必有 $\text{length}(s_1) \leq \text{length}(s_2)$, 由此可推导出 $s_1 \sqsupset s_2$. 综上, $\text{endpos}(s_1) \supseteq \text{endpos}(s_2) \Leftrightarrow s_1 \sqsupset s_2$.

所以, 我们现在可以回答上面提出的那个问题了:

$$\begin{cases} \text{endpos}(s_1) \supseteq \text{endpos}(s_2), & \text{iff } s_1 \sqsupset s_2 \\ \text{endpos}(s_1) \cap \text{endpos}(s_2) = \emptyset, & \text{otherwise} \end{cases}$$

让我们进一步来思考一个问题: 给定一个 state u , $\text{substrings}(u)$ 有什么特性?

给定一个 state u , 定义

- $\text{longest}(u)$ 为 $\text{substrings}(u)$ 中最长的 substring
- $\text{shortest}(u)$ 为 $\text{substrings}(u)$ 中最短 substring
- $\text{maxlen}(u)$ 为 $\text{longest}(u)$ 的长度
- $\text{minlen}(u)$ 为 $\text{shortest}(u)$ 的长度

由于 $\text{endpos}(\text{shortest}(u)) = \text{endpos}(\text{longest}(u))$, 根据上一个问题的结论, 我们可以知道 $\text{shortest}(u) \sqsupset \text{longest}(u)$. 同理, 对于任意的 $s_i \in \text{substrings}(u)$, 必有 $s_i \sqsupset \text{longest}(u)$, 也就是说, $\text{substrings}(u)$ 中所有的 substrings, 必然会是 $\text{longest}(u)$ 的 suffix!

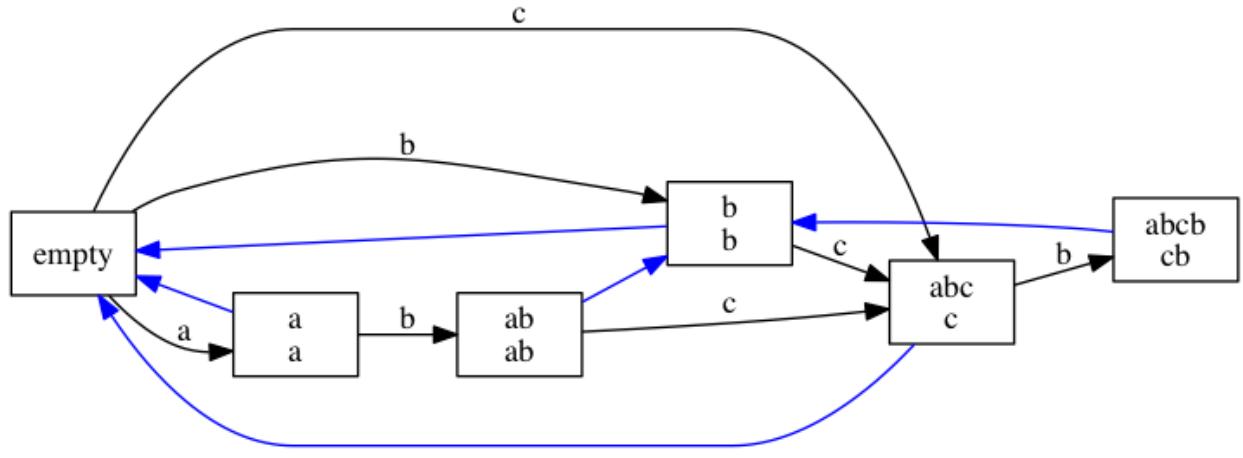
而且, 需要注意的是, 长度在 $[\text{minlen}(u), \text{longest}(u)]$ 的 $\text{longest}(u)$ suffix, 都会被包含在 $\text{substrings}(u)$ 里. 设 s_i 是 $\text{longest}(u)$ 的 suffix 且 $\text{length}(s_i)$ 在 $[\text{minlen}(u), \text{longest}(u)]$ 内, 则必有 $\text{endpos}(\text{shortest}(u)) \supseteq \text{endpos}(s_i) \supseteq \text{endpos}(\text{longest}(u))$, 由于 $\text{endpos}(\text{shortest}(u)) = \text{endpos}(\text{longest}(u))$, 所以 $s_i \in \text{substrings}(u)$ 里的, 得证.

直观来讲, 这意味这一个 state 会对应一个"连续的" suffix 集合, 例如 abc, bc, c , 每个 substring 恰好是上一个 substring 的 suffix, 长度与上一个 substring 相差 1. 这真是一个美妙的特性呢!

State's Suffix Link

除了通过 symbol (字符), SAM 的 state 之间还有另外一种状态变化方式, 也就是 suffix link. 上个小节谈到, 给定一个 state u , $\text{substrings}(u)$ 可以通过 $\text{longest}(u)$, $\text{maxlen}(u)$, $\text{shortest}(u)$, $\text{minlen}(u)$ 描述, 而且 $\text{substrings}(u)$ 是"连续"的. 我们将基于上述结论定义 suffix link.

给定 state u, v , 定义 $v = \text{link}(u)$, 意为存在 from u to v 的 suffix link, 其中 $\text{maxlen}(v) + 1 = \text{minlen}(u)$. 例如, 对于 $T = "abcb"$, 有如下的 SAM 结构:

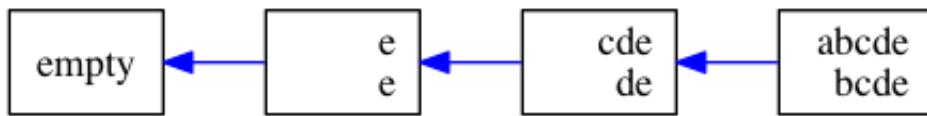


可以看到, 相比于之前的图例, 上图中加入了蓝色的 edges, 这些蓝色的 edges 就是 suffix links. 同时, 在图例中的每个 state u 里标记了 $\text{longest}(u)$ 与 $\text{shortest}(u)$, 以方便理解. 在上图中可以看到, state $["abcb", "cb"]$ (也就是最右边的 state) 通过 suffix link 指向 state $["b", "b"]$, state $["b", "b"]$ 再通过 suffix link 指向 start state. 其中, $\text{ maxlen}(["b", "b"]) + 1 = \text{ minlen}(["abcb", "cb"])$.

为了简化表述, 引入以下与 suffix link 相关的符号与术语:

- T_i : 表示 T 中以 $T[i]$ 结尾的 prefix.
- **Suffix-link path from u to v** : 从 state u 到 state v 的一条使用 suffix link 相连的路径.

直观来讲, SAM 会把 T_i 的 suffix 集合 (注意, 我这里用的不是 substring) $[“, T_i]$ 切成几个区间, 每个区间可以视为一个 state, state 之间使用 suffix link 连接. 所以, 对于某个 state u 满足 $\text{longest}(u) = T_i$, 考虑通过 suffix link 到 start state 的路径上的所有 states, 对这些 states 对应的 substrings 集合做 union 操作, 即可得到 suffix 集合 $[“, T_i]$. 以下是 $\text{longest}(u) = abcde$ 的图例:



可以看到, suffix 集合 $[“, “abcde”]$ 被划分成 $[“, “”]$, $[“e”, “e”]$, $[“de”, “cde”]$, $[“bcde”, “abcde”]$ 4 个区间, 其间通过 suffix link 相连.

State's Transitions

现在我们来深入探讨一下 SAM 里面的 state transition 的含义. 需要注意的是, suffix link 与 state transition 是没有关系的, 原因是 suffix link 并没有被 symbol labeled, 这不符合 DFA 里 *transition function* 的定义. 这也是为什么我在上一小节使用"状态变化"来描述 suffix link, 因为 suffix link 并不是"状态转移".

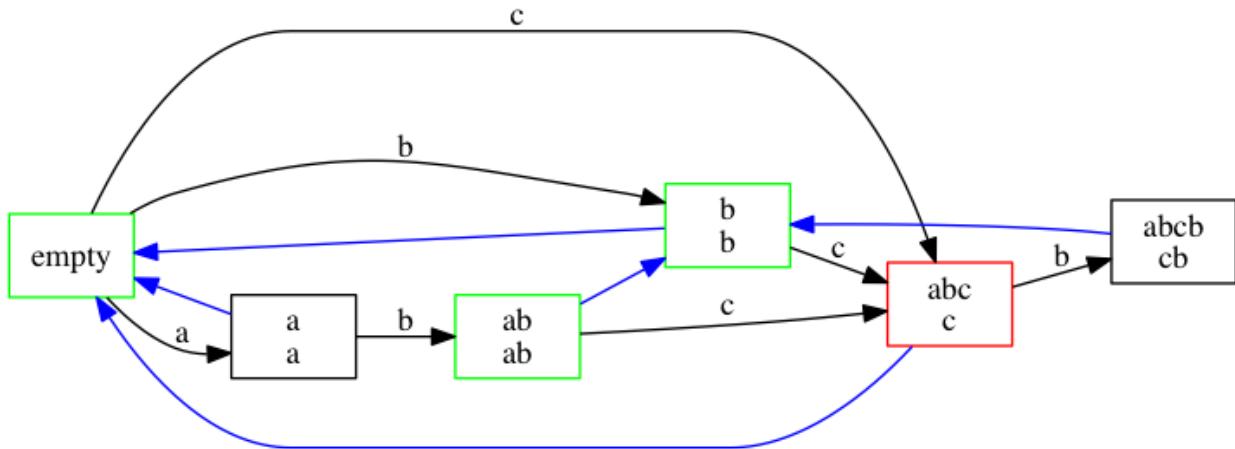
给定一个 state u 与 symbol c , 定义 $\text{trans}(u, c) = v$, 表示从 state u 到 state v 存在 state transition 路径, 该路径通过 c 标识. 在本文的图例中, state transition 一律使用黑色的 edge 表示.

为了深入理解 SAM state transition 的含义, 需要思考一个问题: 如果存在 $\text{trans}(u, c) = v$, state v 的特性与 state u 的特性有什么关系?

由于有 $\text{trans}(u, c) = v$, 那么必有 $\{s_i + c | s_i \in \text{substrings}(u)\} \subseteq \text{substrings}(v)$, 也就是说, 往 state u 的对应的 substrings 后面拼接 symbol c , 得到的新 substrings 集合必然包含于 $\text{substrings}(v)$.

而且, 考虑到 $\text{substrings}(v)$ 里的 substring 都是以某个 symbol c 结尾的, 我们可以知道, 对于 SAM 里所有合法的 state transitions $\text{trans}(u_i, c_i) = v$, 必有 $c_i = c$, 也就是说, 所有以 v 为终点的 transition, 必然带有相同的 symbol!

我们现在来考虑 $\text{indegrees} \geq 1$ 的情况, 还是以 $T = "abcb"$ 为例:



在上图中, 设

- start : start state, 等价于 state $["", ""]$ (最左, green 标记)
- s_1 : state $["b", "b"]$ (green 标记)
- s_2 : state $["ab", "ab"]$ (green 标记)
- v : state $["c", "abc"]$ (red 标记)

所以, 有 $\text{trans}(\text{start}, "c") = \text{trans}(s_1, "c") = \text{trans}(s_2, "c") = v$. 可以看到, $\text{substrings}(v)$ 其实就是往 $\text{substrings}(\text{start}) \cup \text{substrings}(s_1) \cup \text{substrings}(s_2)$ 的所有 substrings 后面拼接上 “ c ” 的新集合! 由此可以引申得到以下性质:

对于 SAM 里的 state v , 设 transition symbol 为 c 则有

$\text{substrings}(v) = \{s_i + c | s_i \in \text{substrings}(u_i), \text{trans}(u_i, c) = v\}$. 通俗来说, 可以通过观察 state transitions 得到 state v 的 $\text{substrings}(u)$.

那么问题来了: 设 $U_v = \{u_i | \text{trans}(u_i, c) = v\}$, U_v 中的 states 有什么关系呢?

我们已经知道, 对于一个 state v , $\text{substrings}(v)$ 中的 substrings 是“连续的”. 由于到 state v 的 transition 都带有相同的 symbol c , 那么由 U_v 做 union 得到的 substrings 集合, 必然也是“连续的”, 即 $\{s_i | s_i \in \text{substrings}(u_i), u_i \in U_v\}$ 是连续的.

回想一下上一小节的 suffix link 定义, $v = \text{link}(u) \implies \text{maxlen}(v) + 1 = \text{minlen}(u)$, 我们可以知道, U_v 中的 states, 必然形成一条 **suffix-link path**! 以上图为例, 有 $U_v = \{\text{start}, s_1, s_2\}$, $\text{start} = \text{link}(s_1)$, $s_1 = \text{link}(s_2)$.

到此, 我们已经基本理解了 SAM 的 state, suffix link, transition 的概念. 下面将会讲解如何构建 SAM.

SAM Online Construction: The Principle

假设我们已经为 T_i 构建了 SAM, 该 SAM 的结构使用两个 variables 追踪:

- start : 指向 start state.
- last : 指向“最后一个” state, $T_i \in \text{substrings}(\text{last})$.

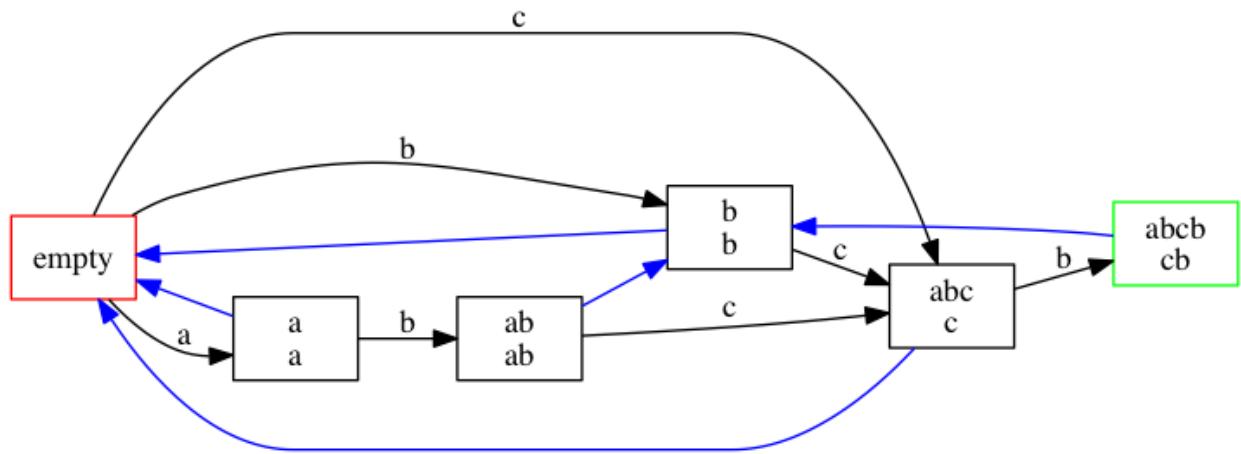
为了简化 construction, 在本小节中, SAM 中的每个 State u 仅维护以下信息:

- $\text{ maxlen}(u)$
- $\text{ link}(u)$
- $\text{ trans}(u, \Sigma)$, 也就是以 u 为起点的 state transitions

start 作为一个特殊的 state, 有:

- $\text{ maxlen}(u) = 0$
- $\text{ link}(u) = \text{null}$

以下是一个简单的图例:



其中, start 是 red state, last 是 green state.

SAM [online](#) construction 的核心问题是: 如何将 $T[i + 1]$ 插入到 T_i 对应的 SAM 里, 插入 c 会导致 SAM 的结构发生什么变化?

如果加入 $T[i + 1]$, 意味着更新后的 SAM 里必定有一个 state u , 满足 $i + 1 \in \text{endpos}(u)$, 而更新前的 SAM 必然不存在这样的 state. 所以, 我们需要创建一个新的 state cur 使

- $i + 1 \in \text{endpos}(\text{cur})$
- $T_{i+1} \in \text{substrings}(\text{cur})$

把 $T[i + 1]$ 加入到 SAM, 是期望新的 SAM 可以识别 T_{i+1} 的 suffix 集合. 这个集合, 恰好是 $\{s_i + T[i + 1] | s_i \sqsupseteq T_i\}$, 也就是说, 这是将 symbol $T[i + 1]$ 拼接到 T_i 的所有 suffix 的末尾之后得到的新集合 (当然, 还需要加入一个 empty string, 不过这个是细节性问题).

显然的, 根据 suffix link 的性质, 要得到 T_i 的 suffix 集合, 只需要访问 suffix-link path from last to start 上的每一个 state u , 将所有的 $\text{substring}(u)$ 合并即可. 由于这些 states 已经代表了 T_i 的 suffix 集合, 只要建立从这些 states 到 state cur 的 transition, 就可以使 SAM 能识别 T_{i+1} 的 suffix 集合.

设 state p 为 suffix-link path from last to start 上当前正在处理的 state, 我们希望可以建立 $\text{trans}(p, T[i + 1]) = \text{cur}$, 使得 $\{s_i + T[i + 1] | s_i \in \text{substrings}(p)\} \subseteq \text{substrings}(\text{cur})$. 那么问题来了, 如果已经存在 $\text{trans}(p, T[i + 1]) = q$ 的 transition, 其中 q 是某个已经存在的 state, 我们该怎么处理?

首先, 如果出现 $\text{trans}(p, T[i + 1]) = q$, 意味着存在某个 suffix $x \sqsupseteq T_i$ (等价的, $x \in \text{substrings}(p)$), 存在 string $(x + T[i + 1])$

- $(x + T[i + 1])$ 是 T_{i+1} 的 suffix (显然).
- $(x + T[i + 1])$ 同时是 T_i 的某个 substring. 等价的, $(x + T[i + 1]) \in substrings(q)$.

在识别到这个 case 之后, 我们可以知道, 原先的 SAM 已经包含了 T_{i+1} 的 suffix 集合的子集, 由此, 可以使用 suffix link 将 state cur 指向 某个 state, 如此更新后的 SAM 即可识别 T_{i+1} 的 suffix 集合.

设某个 state 为 pre , 在识别到 $\text{trans}(p, T[i + 1]) = q$ 的情况下, 需要有 $\text{link}(cur) = pre$, 那么, state pre 需要满足什么样的性质?

显然的, 在处理 state p 的时候, 已经有 $\text{minlen}(cur) = \text{ maxlen}(p) + 2$. 设 state p' 满足 $p = \text{link}(p')$, 根据 suffix link 的定义, 有 $\text{maxlen}(p) + 1 = \text{minlen}(p')$. 由于在处理 state p 的时候, 已经存在 $\text{trans}(p', T[i + 1]) = cur$, 即有 $\text{minlen}(cur) = \text{minlen}(p') + 1$. 所以, $\text{minlen}(cur) = \text{minlen}(p') + 1 = \text{maxlen}(p) + 2$, 得证.

因为 $\text{link}(cur) = pre$, 所以必有 $\text{maxlen}(pre) + 1 = \text{minlen}(cur)$, 结合上面的结论, 可以推导出 $\text{maxlen}(pre) = \text{maxlen}(p) + 1$. 这意味着:

- 存在 $\text{trans}(p, T[i + 1]) = pre$
- $\text{longest}(pre) = \text{longest}(p) + T[i + 1]$

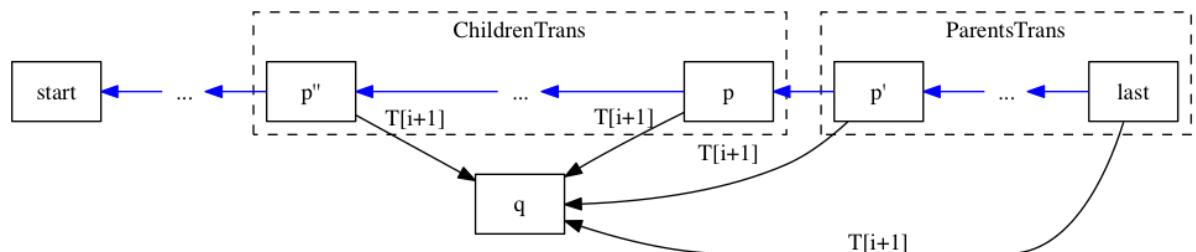
And that's it! 我们需要检测 state q 是否满足作为 pre 的条件:

1. 如果 $\text{maxlen}(p) + 1 = \text{maxlen}(q)$, 那么 state q 就是我们要找的 pre , 直接设 $\text{link}(cur) = q$.
2. 如果 $\text{maxlen}(p) + 1 > \text{maxlen}(q)$, 这意味着 $\text{substring}(pre) \subset \text{substring}(q)$. 这个时候, 需要将满足 pre 条件的部分从 q 里面剥离出去!
3. 显然的, 不可能出现 $\text{maxlen}(p) + 1 < \text{maxlen}(q)$ 的情况.

对于 $\text{maxlen}(p) + 1 > \text{maxlen}(q)$ 的情况, 利用在 state transition 小节得到的结论, 我们可以知道, 对于 state p 所在的 suffix-link path, 必然有 state p' 满足 $p = \text{link}(p')$ 且存在 $\text{trans}(p', T[i + 1]) = q$. 由此, 可以将到 state p 的 transition 分为两个部分:

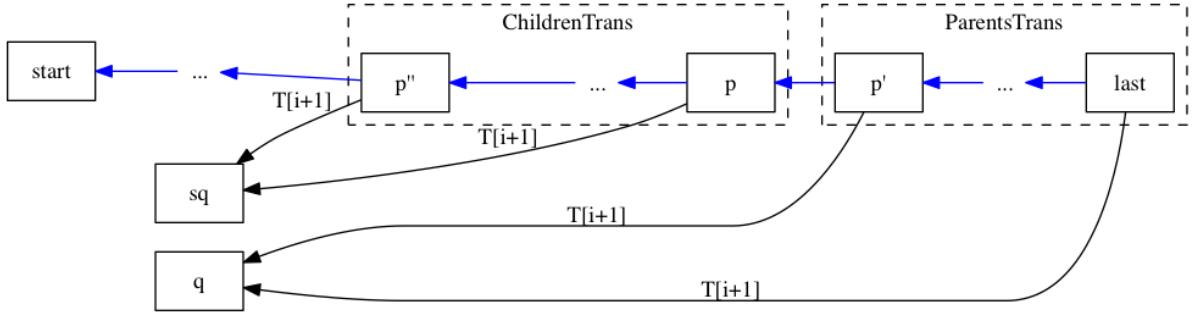
- **ParentsTrans**: 包含满足条件的 state transitions $\text{trans}(u, T[i + 1]) = q$, 其中 state u 满足存在 suffix-link path from u to p' , including p' .
- **ChildrenTrans**: 包含满足条件的 state transitions $\text{trans}(u, T[i + 1]) = q$, 其中 state u 满足存在 suffix-link path from p to u , including p .

以下为对应图例:



其中, state p'' 是 state p 的 child (through suffix link), 是最后一个满足 $\text{trans}(u, T[i + 1]) = q$ 条件的 state, 也就是说, 要么 p'' 是 $start$, 要么不存在 $\text{trans}(\text{link}(p''), T[i + 1])$.

由于 state transition 决定了 $\text{substrings}(q)$, 为了将属于 pre 的部分从 state q 中分离出来, 可以新建一个 state sq , 将 **ChildrenTrans** 集合中的 transitions 全部转向 sq . 如下图所示:



处理完以 state sq 为终点的 transitions 之后, 还需要考虑以 sq 为起点的 transitions. 显然的, 对于所有的 $trans(q, c) = q'$, 也必须存在有 $trans(sq, c) = q'$. 所以, 直接把 state q 维护的 transitions copy 给 state sq 就可以了.

完成 transition 的调整之后, 还需要调整 suffix links:

- $link(sq) = link(q)$, 原因在于, 原先的 $shortest(q)$ 现在已经在 $substrings(sq)$ 集合里, 所以需要调整 suffix link 以保证 suffix-link 的"连续性".
- $link(q) = sq$, 原因同上.

至此, 我们已经从 state q 中分离出满足 pre 特性的 state sq ! 最后, 设 $link(cur) = sq$, Done!

SAM Online Construction: The Pseudocode

利用上一小节的结论, 可以给出将 $T[i]$ 插入到 SAM 的算法步骤 (注意, 上一小节我们讨论的是将 $T[i + 1]$ 插入到 T_i 的 SAM 里, 所以下面的表述会有细微的区别, 但原理是一样的):

1. 新建一个 state cur , 使 $maxlen(cur) = maxlen(last) + 1$.
2. 从 $last$ 开始, 沿着 suffix-link path from $last$ to $start$ 访问每个 state p , 直到 $p = null$ 或者存在 $trans(p, T[i])$ 的情况. 在循环中, 设 $trans(p, T[i]) = cur$.
3. 判断 p 是否为 $null$. 如果 $p = null$, 则设 $link(cur) = start$, 结束后续操作. 反之, p 是某个合法的 state, 且有 $trans(p, T[i]) = q$.
4. 如果 $maxlen(p) + 1 = maxlen(q)$, 设 $link(cur) = q$.
5. 如果 $maxlen(p) + 1 < maxlen(q)$, 从 state q 中拆分出 state sq , 设 $link(sq) = link(q)$, 之后设 $link(q) = sq$, $link(cur) = sq$.
6. 更新 $last$, 使 $last = cur$.

以下是对应的 pseudocode:

```

Procedure Name: AddSymbolToSAM
Input: start, last, T[i]
Output: cur

# (1)
create state cur
maxlen(cur) = maxlen(last) + 1

# (2)
p = last
while p is not null AND trans(p, T[i]) not exists:
    trans(p, T[i]) = cur
    p = link(p)

# (3)
if p is null:
    link(cur) = start
    return cur

q = trans(p, T[i])
if maxlen(p) + 1 = maxlen(q):
    # (4)
    link(cur) = q
else:
    # (5)
    create state sq
    maxlen(sq) = maxlen(p) + 1
    for all trans(q, c) = q':
        trans(sq, c) = q'

while p is valid AND trans(p, T[i]) exists:
    trans(p, T[i]) = sq
    p = link(p)

link(sq) = link(q)
link(q) = sq
link(cur) = sq

reutrn cur

```

所以, 给定一个 string T , 构建对应 SAM 的 pseudocode 如下所示:

```

Procedure Name: CreateSAM
Input: T
Output: start

create state start
last = start

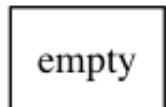
for i in 1 to length(T):
    last = AddSymbolToSAM(start, last, T[i])

return start

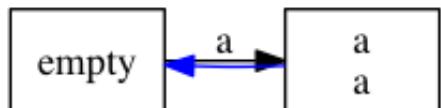
```

Boom! SAM 的算法虽然理解上比较困难, 但是写起来还是很简单的, 基本上用 30-40 行的 C++ 代码就可以写完. 下面给出构建 $T = abcbc$ 的过程, 图片使用 [SAM-PNG](#) + [DOT](#) 生成, [SAM-PNG](#) 是我用 C++ 写的 SAM 实现, 有需要的同学也可以去看一下.

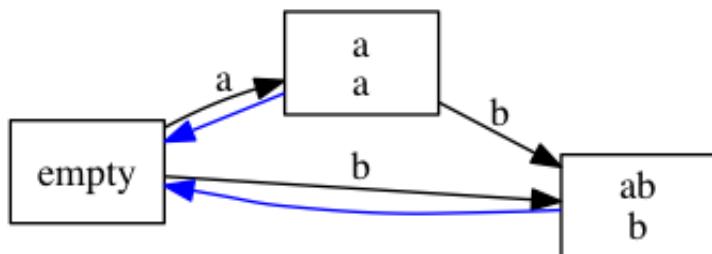
$i = 1$:



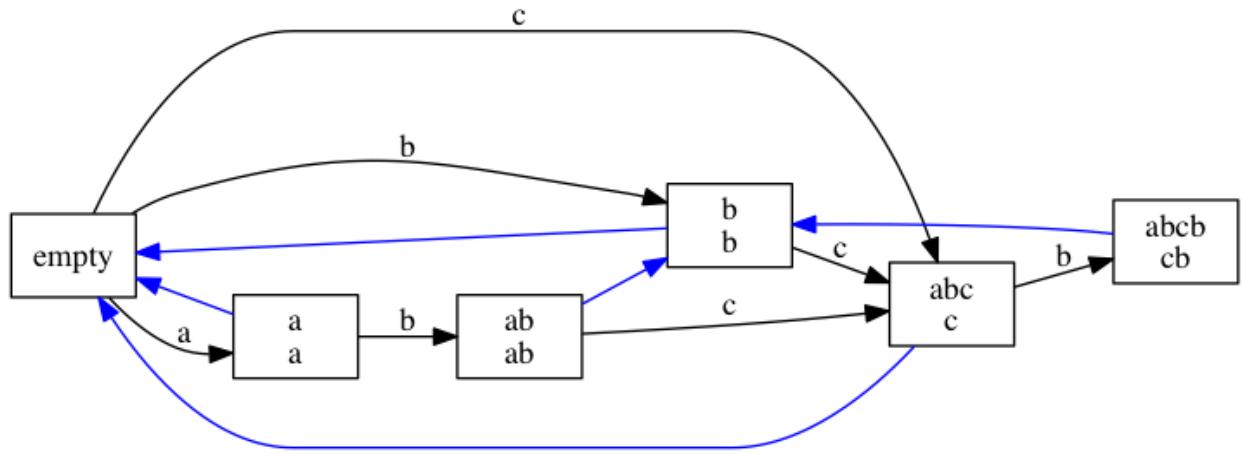
$i = 2$:



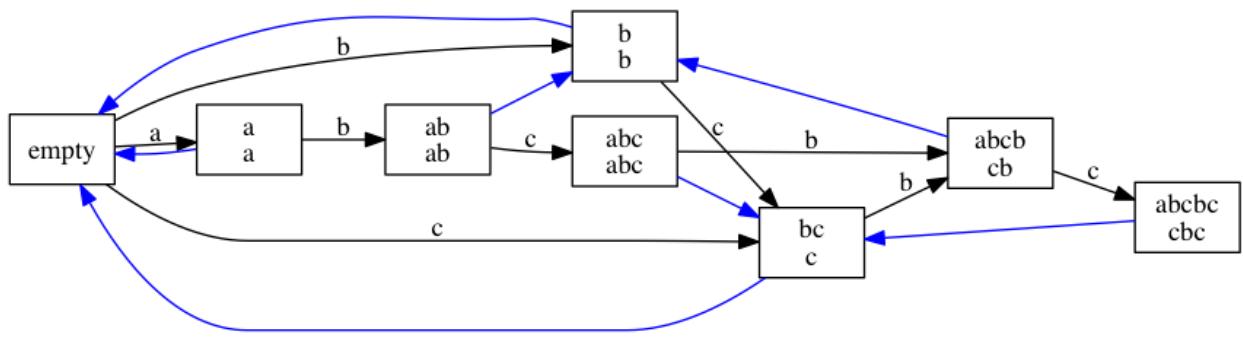
$i = 3$:



$i = 4$:



$i = 5$:



SAM: The Time/Space Complexity

问题: 对于 $n = \text{length}(T)$, 求 SAM 的 time/space complexity.

先来考虑 space complexity. 由于 SAM 本质上是 graph, 所以在考虑 space complexity 的时候, 需要考虑 state (node) 与 transition (edge) 的数量.

根据 SAM 的构建过程, 往 SAM 里加入 $T[1], T[2]$ 时, 仅会增加 1 state, 对于剩下的 $T[3]$ 至 $T[n]$, 每次处理最多增加 2 states, 所以, state 的上限是 $1 + 2 + 2(n - 2) = 2n - 1$, 即 state 的数量是 $O(n)$.

为了得出 transition 的数量, 先定义出两类 transition. 给定一个 transition $\text{trans}(u, c) = v$:

- 如果 $\text{maxlen}(u) + 1 = \text{maxlen}(v)$, 则称之为 continuous transition.
- 反之, 称之为 discontinuous transition.

由于 state 的上限是 $2n - 1$, 所以 continues transition 数量的上限是 $2n - 2$, 即 $O(n)$.

对于每个 discontinuous transition $\text{trans}(u, c) = v$, 设 s_u 为从 start 到 u 的最长 path, 这个 path 中必然只包含 continuous transition. 设 s_v 为从 v 到 accept state 的, 最长的只包含 continuous transition 的路径, 由于每个 state 要么是 accept state, 要么存在以这个 state 为起点的一个或多个 continuous transition, 所以 s_v 也必然存在. 所以, 对于所有的 discontinuous transition $\text{trans}(u, c) = v$, 必然有 $(s_u + c + s_v) \sqsubset T$, 而且不同的 discontinuous transition 对应的 $(s_u + c + s_v)$ 各不相同. 因为 T 的 unique suffixes 数量的上限是 n , 所以 discontinuous transition 的数量也是 $O(n)$.

所以, transition 的数量是 $O(n)$.

由此, SAM 的 space complexity 是 $O(n)$.

接着我们来考虑 time complexity. 根据 SAM 的构建算法, 影响 time complexity 的操作有三种:

1. 第 (2) 步的创建 transitions.
2. 第 (5) 步的 copy transitions from q to sq .
3. 第 (5) 步的重定向 transitions *ChildrenTrans*.

显然, (1) 与 (2) 只涉及创建新的 transition, 由于 transition 的上限是 $O(n)$, 所以均摊后 (1) 与 (2) 类操作的复杂度是 $O(n)$. 至于 (3) 我就不知道怎么证了, 我找到了一些 paper 说这个是可以证的, 但是具体怎么搞我没想清楚. 如果你有比较好的证明方法, 请务必告诉我, 我请你喝咖啡!