Содержание

1 Введение

Информационная революция и генная инженерия дали мощный толчок *науке о строках*. Слово *stringology*, вошедшее в обиход специалистов по computer science [3], едва ли является натяжкой:

> за последнее десятилетие накопилось целое море фактов, в том числе замечательных новых постановок задач, теоретических исследований и данных [10].

Все это позволяет говорить о строках как о весьма содержательном объекте.

В своей работе мы хотим показать, что особые структуры данных, порожденные суффиксами данной строки `text`, представляют парадигму для решения *задач о строках*, по охвату и силе сравнимую разве что с поиском в ширину в графовых задачах. А именно, начав с интуитивно очевидной структуры *trie*, мы проследим ход мыслей, приводящих к *суффиксным деревьям*. Далее опишем и поясним алгоритм МакКрейта(McCreight) (Алгоритм M) построения суффиксного дерева. Применим построенный аппарат для решения задач, как то: нахождение наименьшего циклического сдвига строки, наибольшей общей подстроки двух строк и т.д. Наконец, расскажем о дальнейшем направлении нашей работы.

Терминология и обозначения вполне традиционны, вводятся по ходу дела или же ясны из контекста. Значительно перекрывающийся вводный материал желающие найдут в [10],[7],[4] или [5].

1.1 Наш вклад.

Результаты нашей работы:

1. Детальная разработка алгоритма МакКрейта в смысле выделения новых операций, значительно облегчающих понимание

алгоритма в частности и механизма работы суффиксного дерева вообще;

2. Наиболее подробный анализ времени работы Алгоритма M;
3. Реализация таким образом переосмысленного Алгоритма M на языке программирования C с оценкой по памяти $O(n|\Sigma|)$ и по времени $O(n \log |\Sigma|)$;
4. Примеры практического применения нашего программного кода к решению конкретных задач о строках.

1.2 Благодарности

При подготовке работы мы использовали ОС Linux OpenSuSE 11.0. Код программы и исходники LaTeX набирались в редакторе vim 7.1, компилировались на cc v4.3.1, latex, BiBTeX v0.99c. Иллюстрации подготовлены в системе визуализации графов graphViz v2.22.2. Всем разработчикам названных продуктов – огромное спасибо.
Я признателен научному руководителю Ковалеву Александру Демьяновичу – за поощрение в выборе темы; Михаилу Мирзаянову и Сергею Назарову – от них я впервые услышал слово “суффиксные деревья”.

2 Суффиксные деревья: мотивировка

> A complex system that works is
> invariably found to have evolved
> from a simple system that works
>
> Anon.

В рамках нашей работы под $\texttt{trie}(\texttt{text\$})$ мы будем понимать дерево с выделенным корнем, дуги которого помечены символами алфавита Σ, имеющее ровно $n + 1$ листьев ($n = |\texttt{text}|$), и такое, что

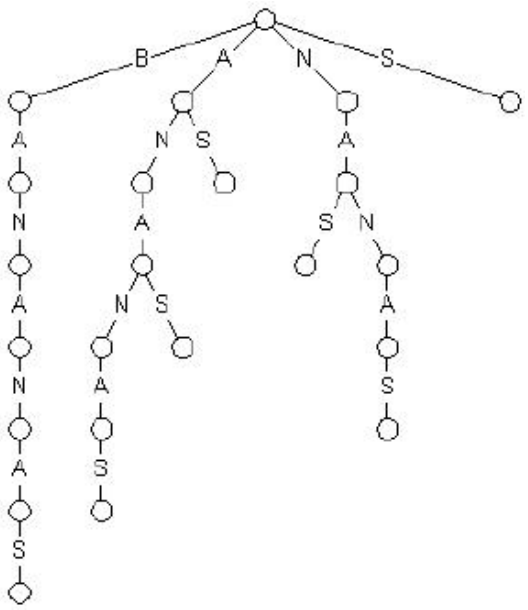

Рис. 1: `trie(BANANA$)` [6]

каждый путь от корня до листа произносит непустой суффикс строки `text$`, и, наоборот, для каждого непустого суффикса строки `text$` существует такой путь. Для каждой вершины $\mathtt{v} \in \mathtt{trie}(\mathtt{text\$})$ и каждого символа $\mathtt{a} \in \Sigma$ существует не более одной `a`-дуги, выходящей из `v`. Этому многословному описанию соотвествует интуитивно очевидная структура (рис.1). Нужно отметить, что роль \$ может и должен выполнять любой нигде в `text` не встречающийся символ.
С $\mathtt{trie}(\mathtt{text\$})$ есть две проблемы:

1. Его наивное построение требует $\mathtt{O}(\mathtt{n}^2)$ действий,
2. Например, `trie` для $\mathtt{text} = \mathtt{a}^\mathtt{n}\mathtt{b}^\mathtt{n}$ требует $\mathtt{O}(\mathtt{n}^2)$ памяти.

Эвристика для сокращения памяти. Пусть мы ищем образец $p = BANK$ в $\mathtt{trie}(\mathtt{BANANA\$})$. Спустившись из корня по `B`-дуге, мы оказываемся в левой верхней вершине (рис. 1), из которой, в свою очередь, исходит единственная дуга. Если мы знаем, что из вершины исходит единственная дуга, то выбор наш “узок, как путь мусульманина в рай”. Поэтому будем метить дуги *словами*(парой индексов

на глобальный массив). Эта эвристика названа в [3] *сжатие цепей* (compacting chains). Справедливо

Наблюдение 1. *Количество вершин в дереве с* `n` *листьями, каждая внутренняя вершина которого имеет не менее двух потомков, не превосходит* `2n` *(*$= \mathtt{O(n)}$*).*

Действительно,

$$n + n/2 + n/2^2 + \ldots \leq \frac{n}{1 - \frac{1}{2}} = 2n.$$

Теперь мы готовы ввести вслед за [G03]

Определение 1. *Пусть* `text$` *– строка над* Σ *,* $|\mathtt{text}| = \mathtt{n}$*. Суффиксное дерево* `T` *для* `text$` *– это ориентированное дерево с корнем* `root`*, имеющее ровно* $\mathtt{n}+1$ *листьев* $\{\mathtt{1}, \mathtt{2}, \ldots, \mathtt{n}+\mathtt{1}\}$*. Каждая внутренняя вершина* $\neq$ `root` *имеет не менее двух детей, а каждая дуга помечена непустой подстрокой строки* `text$`*. Никакие две дуги, выходящие из одной и той же вершины, не могут иметь пометок, начинающихся с одного и того же символа. Для каждого листа* `i` *конкатенация меток дуг на пути от корня к листу* `i` *в точности произносит* $\mathtt{text[i:n]\$}$.

3 Алгоритм МакКрейта: *preliminaries*

Алгоритм МакКрейта является “оффлайновым” – т.е. начинает свою работу лишь после прочтения всей строки `text$`. Алгоритм выполняет $\mathtt{n} = |\mathtt{text}|$ фаз, в ходе каждой из которых текущее дерево расширяется за счет вставки в него очередного (в порядке убывания

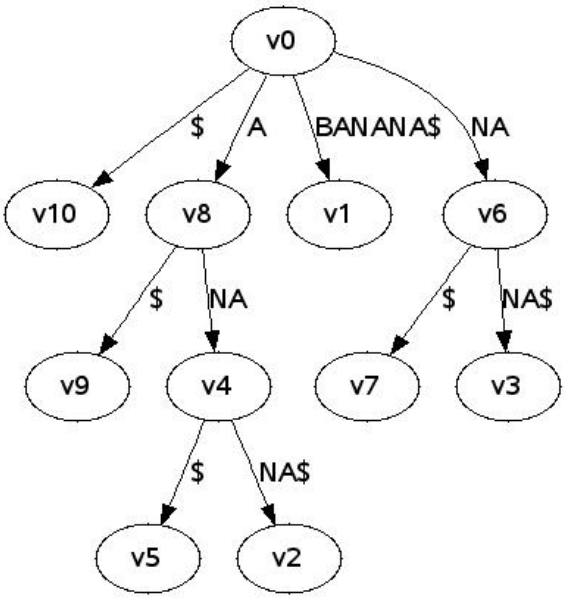

Рис. 2: Суффиксное дерево для BANANA$

длин) суффикса $\texttt{text}[\texttt{i}+1:\texttt{n}]\$$. Наличие \$ гарантирует то, что дерево действительно "расширяется", т.е. в $\texttt{text}\$$ нет *вложенных суффиксов* (nested suffix – [7]).

При вставке суффикса $\texttt{text}[\texttt{i}:\texttt{n}]\$$ фактически происходит следующее: мы спускаемся по соотвествующим дугам до первого расхождения. Это расхождение может произойти как *внутри* дуги, так и в вершине. В первом случае дуга разбивается на месте *неявного узла* (implicit node – [3]). Затем протягиваются соответствующие дуга и родительские связи.

Неявный узел (неявная вершина) – это пара $(\texttt{w}, \alpha)$, где $\texttt{w}$ – вершина текущего дерева $\texttt{T}$, а α – непустой собственный префикс метки дуги, выходящей из $\texttt{w}$.

На этом этапе должно быть ясно, что место, с которого будет расти новый лист $\texttt{i}$ – это $\texttt{lcp}_{1\le \texttt{j}\le \texttt{i}-1}(\texttt{text}[\texttt{i}:\texttt{n}]\$, \texttt{text}[\texttt{j}:\texttt{n}]\$)$, где $\texttt{lcp}$ – наибольший общий префикс. Обозначив этот префикс как $\texttt{head}(\texttt{i})$, будем иметь $\texttt{text}[\texttt{i}:\texttt{n}]\$ = \texttt{head}(\texttt{i})\texttt{tail}(\texttt{i})$. Существенным для алгоритма МакКрейта оказывается

Наблюдение 2 ([S03]). *Наибольший собственный суффикс* $\texttt{head(i)}$ *является префиксом* $\texttt{head(i + 1)}$.

Доказательство. $head(i) = text[i : i + h], i - 1 \leq i + h \leq n, \iff -1 \leq h \leq n - i$. Если $h \leq 0$, то $text[i : i + h] = \epsilon$. А пустая строка является префиксом любой строки, в частности, строки $head(i + 1)$. При $i+h > i$ обозначим $\lambda = text[i]$, $u = text[i + 1 : i + h]$. По определению $\exists 1 \leq j \leq i-1 : text[j : j+h] = text[i : i+h] = \lambda u$. Но тогда u является общим префиксом суффиксов $text[j+1 : n] = text[i+1 : n]$, следовательно, и префиксом $head(i + 1)$. □

Следствие 1. $|\texttt{head(i + 1)}| \geq |\texttt{head(i)}| - \texttt{1}$.

Это следствие будем иметь в виду при временном анализе алгоритма МакКрейта.

На самом деле наибольший собственный суффикс играет куда большую роль и заслуживает отдельного

Определение 2. *Пусть* $\texttt{u}$ *– вершина суффиксного дерева* $\texttt{T(text\$)}$. *Суффикс-функция* $\texttt{s(u)}$ *этой вершины определяется следующим образом:*

- $\texttt{s(u)} = \texttt{root}$, *если* $\texttt{path(u)} = \epsilon$, *i.e.* $\texttt{u} = \texttt{root}$,
- $\texttt{s(u)} = \texttt{v}$, *если* $\texttt{path(u)} = \lambda\texttt{path(v)}$, $\lambda \in \Sigma$.

Ясно, что когда $\texttt{u}$ есть $\texttt{i}$-ый лист, то $\texttt{s(u)}$ есть $\texttt{i + 1}$-ый лист. Когда же $\texttt{u}$ – внутренняя, то подстрока $\texttt{path(u)}$ – *конкатенация всех меток дуг в порядке их прохождения на пути от корня к* $\texttt{u}$ – входит в $\texttt{text}$ с двумя (как минимум) различными продолжениями. Поэтому $\texttt{s(u)}$ существует и тоже является внутренней вершиной. Когда же $\texttt{path(u)} = \lambda \in \Sigma$, $\texttt{s(u)} = \texttt{root}$.
Как раз суффикс-функций и касается

Наблюдение 3. $\mathtt{s}(\mathtt{parent}[\mathtt{u}])$ *является предком* $\mathtt{s}(\mathtt{u})$, $\forall \mathtt{u}$.

Теперь снабдим нашу структуру (которая пока является тем же `trie`, но с сжатыми цепями) указателями, соответствующими значениям суффикс-функции. Назовем этот механизм *суффиксные связи* (или *суффиксные ссылки*).

4 Алгоритм M

4.1 Построение

Так как суффиксное дерево $\mathtt{T}(\mathtt{text\$})$ мыслится нами и как автомат, допускающий суффиксы строки $\mathtt{text\$}$ и только их, то термины "лист" ([10],[3]) и "терминальный узел"([9]) могут использоваться как синонимы.
Алгоритм МакКрейта начинает свою работу с дерева $\mathtt{T_1}$, содержащего два узла: корень (соответствует пустой строке ϵ) и лист 1, соответствующий $\mathtt{text}[\mathtt{1} : \mathtt{n}]\$$. Эти два узла соединены ребром, помеченным $\mathtt{text\$}$. Это дерево также содержит единственную суффиксную связь $\mathtt{s}(\mathtt{root}) = \mathtt{root}$. Затем последовательно для каждого $\mathtt{1} \leq \mathtt{i} \leq \mathtt{n}$ в дерево $\mathtt{T_i}$ добавляются([9]):

1. лист $\mathtt{leaf}(\mathtt{i}+\mathtt{1})$;
2. узел $\mathtt{head}(\mathtt{i}+\mathtt{1})$ (если его еще нет);
3. нисходящее ребро из $\mathtt{head}(\mathtt{i}+\mathtt{1})$ в $\mathtt{leaf}(\mathtt{i}+\mathtt{1})$, метка $\mathtt{tail}(\mathtt{i})$;
4. суффиксная связь $\mathtt{s}(\mathtt{head}(\mathtt{i}))$.

В ходе основного цикла будем поддерживать **инвариант**([9]): в начале $\mathtt{i}$-го шага (это вставка $\mathtt{text}[\mathtt{i}+\mathtt{1} : \mathtt{n}]\$, \mathtt{1} \leq \mathtt{i} \leq \mathtt{n}$) все суффиксные связи нелистовых узлов в дереве $\mathtt{T_i}$ установлены правильно, кроме, быть может, единственного исключения – узла $\mathtt{head}(\mathtt{i})$.
Действия 1 и 3 тривиальны – лишь бы найти $\mathtt{head}(\mathtt{i}+\mathtt{1})$. $\mathtt{s}(\mathtt{head}(\mathtt{i}))$

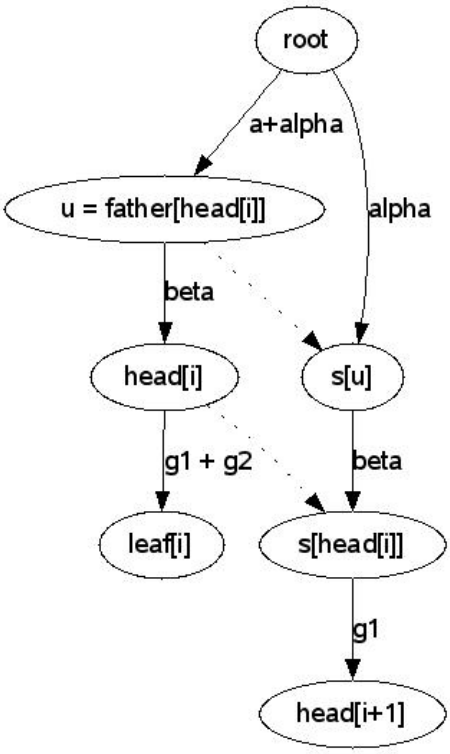

Рис. 3: Примерная конфигурация и главная эвристика алгоритма.

устанавливается в ходе нахождения $\texttt{head}(\texttt{i}+1)$ следующим образом: от суффиксной ссылки узла $\texttt{father}(\texttt{head}(\texttt{i}))$ пройти метку пути от оного до $\texttt{head}(\texttt{i})$(рис.3).

Две вспомогательные функции. Пусть дана вершина $\texttt{u}$ и строка β. Если существование узла $(\texttt{u}, \beta)$(явного или неявного) заведомо известно, то его местонахождение можно вычислить в общем случае быстрее, чем за $O(|\beta|)$. Такая эвристика называется *скачком по счетчику*([10]) и реализуется нами в функции $\texttt{fastscan}(\texttt{u}, \beta)$. Эта функция возвращает явный узел $(\texttt{u}, \beta)$, при необходимости разбив ребро $\texttt{u} \xrightarrow{\omega} \texttt{u}'$, $\omega \in \beta\Sigma^+$ и создав на месте разбиения новую вершину.

Функция $\texttt{slowscan}(\texttt{u}, \gamma)$ укладку γ выполняет посимвольно, до первого расхождения. Если место расхождения – неявный узел, то

ребро разбивается и создается новая вершина.
Пусть расхождение произошло в вершине `v`. Создаётся лист $\mathtt{v}'$ и протягивается дуга $< \mathtt{v}, \mathtt{v}' >$ с меткой "оставшаяся часть строки γ".

Принципиальное отличие двух функций: при вызове `slowscan` заранее известно, что пути с началом в вершине `u` и меткой γ нет. Поэтому эту функцию мы наделили бóльшими полномочиями: она имеет право не только разбивать ребро, но и прививать к вершине новую ветку с оставшейся частью строки γ. Возвращаемое значение – самый "свежий" лист.

`fastscan` и `slowscan` приведены по [5], с изменениями. Сам Алгоритм M написан на основе [8]. Наше развитие:

1. Введение двух абстрактных операций:
   1.1 Разбиение ребра – $\mathtt{split}(\mathtt{u}, \mathtt{e}, \mathtt{len})$;
   1.2 Прививка – $\mathtt{planting}(\mathtt{u}, \beta)$;
2. $\mathtt{fastscan}(\mathtt{u}, \beta)$ возвращает узел `v` т.ч. $\mathtt{path}(\mathtt{v}) = \beta$; В результате её работы может появиться не более одной новой вершины (разбиение ребра и преобразование неявного узла в явный);
3. $\mathtt{slowscan}(\mathtt{u}, \gamma)$ полностью укладывает строку γ и возвращает лист `v` т.ч. $\mathtt{path}[\mathtt{v}] = \gamma$; В результате её работы возникает не более двух новых вершин (разбиение ребра и прививка на том месте новой ветки с листом);

Приведем **псевдокод**(рис. 456).

```
construct tree for text[1 : n]$
for i ← 1 to n do :
    if head(i) = root :
        leaf(i + 1) ← slowscan(root, s(tail(i)))
        head(i + 1) ← father[leaf(i + 1)]
        continue
    u ← father(head(i))
    β ← label(father(head(i)), head(i)))
    if u ≠ root :
        w ← fastscan(s(u), β)
    else :
        w ← fastscan(root, s(β))
    if w has only one son :
        head(i + 1) ← w
        leaf(i + 1) ← planting(head(i + 1), tail(i))
    else :
        leaf(i + 1) ← slowscan(w, tail(i))
        head(i + 1) ← father[leaf(i + 1)]
    label(head(i + 1), leaf(i + 1)) ← tail(i + 1)
```

Рис. 4: Подробный Алгоритм M

$\texttt{len} \leftarrow 1$

while $\texttt{len} \leq |\beta|$ **do** :

$\texttt{find edge e} = \texttt{u} \xrightarrow{\omega} \texttt{u}', \ \texttt{w}_1 = \beta_{\texttt{len}}$

if $|\omega| > |\beta| - \texttt{len} + 1$ **then** :

$\texttt{k} \leftarrow (|\beta| - \texttt{len} + 1)$

$//\texttt{split e, with new node v and edges u} \xrightarrow{\omega_1 \ldots \omega_k} \texttt{v and v} \xrightarrow{\omega_{k+1} \ldots \omega_{|\omega|}} \texttt{u}'$

$\texttt{v} \leftarrow \texttt{split}(\texttt{u}, \texttt{e}, \texttt{k})$

return v

if $|\beta| - \texttt{len} + 1 = |\omega|$ **then** : **return** e.v

$\texttt{len} \leftarrow \texttt{len} + |\omega|$

$\texttt{u} \leftarrow \texttt{u}'$

assert 0

Рис. 5: $\texttt{fastscan}(\texttt{u}, \beta)$

$\texttt{len} \leftarrow 1$

while $\exists \texttt{e} = \texttt{u} \xrightarrow{\omega} \texttt{u}' : \ \omega_1 = \gamma_{\texttt{len}}$ **do** :

    $\texttt{k} \leftarrow 1$

    while $\omega_{\texttt{k}} = \gamma_{\texttt{len}}$ and $\texttt{k} \leq |\omega|$ **do** :

        $\texttt{k} \leftarrow \texttt{k} + 1$

        $\texttt{len} \leftarrow \texttt{len} + 1$

    if $\texttt{k} > |\omega|$ **then** :

        $\texttt{u} \leftarrow \texttt{u}'$

    else :

        assert $\omega_{\texttt{k}} \neq \gamma_{\texttt{len}}$

        //split e with new node v and edges $\texttt{u} \xrightarrow{\omega_1 \ldots \omega_{\texttt{k}-1}} \texttt{v}$ and $\texttt{v} \xrightarrow{\omega_{\texttt{k}} \ldots \omega_{|\omega|}} \texttt{u}'$

        $\texttt{v} \leftarrow \texttt{split}(\texttt{u}, \texttt{e}, \texttt{k} - 1)$

        return $\texttt{planting}(\texttt{v}, \ \gamma_{\texttt{len}} \ldots \gamma_{|\gamma|})$

return $\texttt{planting}(\texttt{u}, \ \gamma_{\texttt{len}} \ldots \gamma_{|\gamma|})$

Рис. 6: $\texttt{slowscan}(\texttt{u}, \gamma)$

SPLIT(u, e, len)

```
1  assert len > 0
2  assert len < LENGTH(e)
3  u.list_of_outgoing_edges.erase(e)
4  v ← planting(u, e.i, e.i + len − 1)
5  father[e.v] ← v
6  back_edge[e.v] ← make_arc(e.v, e.i + len, e.j)
7  v.list_of_outgoing_edges.add(back_edge[e.v])
8  return v
```

PLANTING$(u, left, right)$

```
1  if left > right
2      then return u
3  father[v ← new_node] ← u
4  back_edge[v] ← make_arc(v, left, right)
5  u.list_of_outgoing_edges.add(back_edge[v])
6  return v
```

4.2 Анализ

Теорема 1. *Алгоритм M может быть реализован так, чтобы вычислять суффиксное дерево* `T` *для строки* `text` *над упорядоченным алфавитом* Σ *за время* $O(n \log |\Sigma|)$ *и используя* $O(n|\Sigma|)$ *памяти.*

Доказательство. Оценим по отдельности общее время исполнений `fastscan` и `slowscan`. Для каждого шага `i` внешнего цикла время работы `slowscan` в точности равно([9]) $|head(i+1)| - |head(i)| + 1$ (определение $head(i+1)$ исходя из $s(head(i))$). "Сумма-телескоп" и даёт необходимую линейную оценку.

`fastscan` же использует время, пропорциональное количеству посещенных им узлов. Определим $d(v)$ как узловую глубину узла `v`.

Предложение 1 ([8])**.** $d(v) \leq d(s(v)) + 1$.

Доказательство. Для корня дерева неравенство выполнено. Пусть оно выполнено для всех узлов $\mathtt{u}$ с узловой глубиной $\mathtt{d}(\mathtt{u}) < \mathtt{d}(\mathtt{v})$. Индукционный переход: положим $\mathtt{u} = \mathtt{parent}(\mathtt{v})$. Тогда

$$\mathtt{d}(\mathtt{v}) = \mathtt{d}(\mathtt{u}) + 1 \leq (\mathtt{d}(\mathtt{s}(\mathtt{u})) + 1) + 1 \leq (\mathtt{d}(\mathtt{s}(\mathtt{v})) - 1 + 1) + 1,$$

т.к. $\mathtt{s}(\mathtt{u})$ является предшественником $\mathtt{s}(\mathtt{v})$. □

Так, переход по суффиксной ссылке $\mathtt{s}(\mathtt{father}[\mathtt{head}(\mathtt{i})])$ уменьшает глубину самое большее на 2. Между $\mathtt{s}(\mathtt{father}(\mathtt{i}))$ и $\mathtt{s}(\mathtt{head}(\mathtt{i}))$ могут существовать вершины, которых не было между $\mathtt{father}(\mathtt{i})$ и $\mathtt{head}(\mathtt{i})$, хотя метки обоих путей одинаковы – β. `fastscan` посещает как раз эти вершины и, так как `fastscan` только увеличвает глубину, учет последнего результата даёт желаемую линейную (амортизационно) оценку времени работы.

Более формально, пусть $\mathtt{I_i} = \{\mathtt{l_i}, \mathtt{l_i} + 1, \ldots, \mathtt{r_i} - 1, \mathtt{r_i}\}$ – соответствующие глубины, которые проходит `fastscan` на $\mathtt{i}$-итерации. Заметим, что "отрезки" $\mathtt{I_i}$ упорядочены по неубыванию по левым концам, причем $\mathtt{0} \leq \mathtt{l_1}, \mathtt{r_n} \leq \mathtt{n}$. Согласно **Предложению** $\mathtt{I_i}$ и $\mathtt{I_{i+1}}$ могут пересекаться самое большее по $\{\mathtt{r_i} - 1, \mathtt{r_i}\}$. Тогда мультимножество $\bigcup_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{I_i}$ имеет мощность порядка $\mathtt{O}(\mathtt{n})$.

□

5 Задачи о строках

Линеаризация циклической строки. Рассматриваются эквивалентные в смысле циклических сдвигов строки. Для данной строки требуется указать наименьший в лексикографическом смысле представитель её класса эквивалентности. Другими словами, дано ожерелье, на каждой бусинке которого записана положительная цифра.

Необходимо найти такое место разреза, чтобы получившееся число было наименьшим. Задача допускает много других интерпретаций, например: даны два правых обхода многоугольника, необходимо определить, один ли этот и тот же многоугольник([2]). Поэтому эту задачу также называют задачей канонизации циклической строки [1].
Алфавитно независимый линейный алгоритм решения этой задачи возникает в связи с т.н. *декомпозицией Линдона*. Желающие освоить этот элегантный результат найдут в [4] вместе с [1] исчерпывающий и самодостаточный материал.
Решение. Нетрудно догадаться, что на основе $\mathtt{T}(\mathtt{text} + \mathtt{text\$})$ искомый сдвиг находится за $\mathtt{O}(\mathtt{n} \log |\Sigma|)$. "Все линейные оценки равны, но некоторые равнее других" ([10]).

Статистика для строки. Сколько у строки `text` различных подстрок? (*Указание*: количество нелистовых вершин $\mathtt{trie}(\mathtt{text\$})$).
Решение. Построим суффиксное дерево $\mathtt{T}(\mathtt{text\$})$. $\mathtt{count} \leftarrow 1$. Пробежимся в едином цикле по всем вершинам $\mathtt{u} \neq \mathtt{root}$, увеличивая `count` на длину ребра $< \mathtt{father[u]}, \mathtt{u} >$. Если `u` – лист, то тут же уменьшаем `count` на единицу. Простым тестом может служить строка $\mathtt{text} = \mathtt{a}^{\mathtt{n}/2}\mathtt{b}^{\mathtt{n}/2}$, для которой $\mathtt{count} = (\frac{\mathtt{n}}{2} + 1)^2$.
Замечание. Наш код прошел все тесты на сервере Гданьского Университета (Польша): www.spoj.pl/problems/SUBST1.

Самая длинная повторяющаяся подстрока. В терминах суффиксных деревьев мы ищем самую глубокую внутреннюю вершину. Очевидно, что достаточно перебрать все $\mathtt{head}(\mathtt{i})$.

Наибольшая общая подстрока двух строк. Даны строки α и β. Требуется найти их наибольшую общую подстроку.
Решение. Построим суффиксное дерево $\mathtt{T}(\alpha\$_1\beta\$_2)$, где $\$_1 \neq \$_2$. Листья, соответствующие строке β, пометим числом 2, все остальные вершины и листья – нулем (битовые маски).
Строковая глубина `i`-листа в точности есть $\mathtt{n} + 2 - \mathtt{i}$. Строковые

глубины остальных вершин пометим ∞. Теперь положим все листья в очередь и проведем поиск в ширину: пусть `u` – вершина, только что извлеченная из очереди. Переходим от неё в вершину `v` = `father`[`u`], проведя релаксацию расстояния `dist`[`v`] = `dist`[`u`] − `LENGTH`(`e.i`, `e.j`) и обновление метки вершины `v` по формуле

$$\texttt{mask}[\texttt{v}] = \texttt{mask}[\texttt{v}]\ \texttt{or}\ \texttt{mask}[\texttt{u}].$$

Если ребро `e` =< `father`[`u`], `u` > содержит символ $\$_1$, сделаем также обновление `mask`[`v`] = `mask`[`v`] `or` `1`. Теперь пусть `w` – глубочайшая в смысле `dist` вершина т.ч. `mask`[`w`] = `3`. Тогда `path`[`w`] – искомая наибольшая общая подстрока (`text`[`pos`] = $\$_1$):

```
void bfs( int pos ) {
    int i,j,k,u,v;
    arc *e;
    unsigned int w = 0;

    hd = tl = queue;
    /*memset(mask,0,sizeof(mask));*/
    memset(dist,0xff,sizeof(dist));

    for ( i = 1; i <= n+1; ++i  ) {
        dist[*tl++ = leaf[i]] = LENGTH(i,n+1);
        mask[leaf[i]] = i <= pos ? 0 : 2;
    }

    while ( hd < tl ) {
        if ( mask[u = *hd++] == 3 )
            if ( w < dist[u] )
                w = dist[u];
        if ( u == root || dist[u] <= w )
            continue;
        mask[v = father[u]] |= mask[u];
```

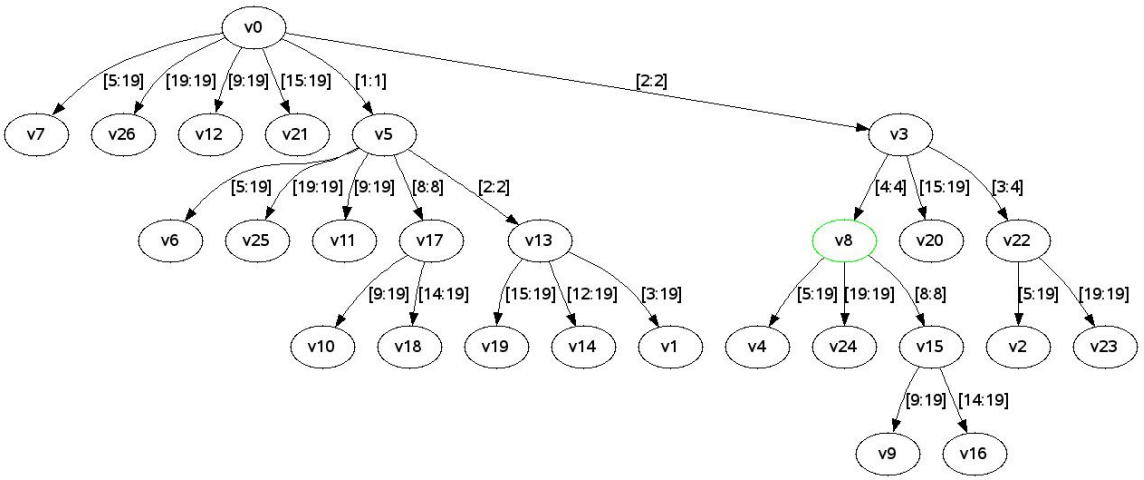

Рис. 7: Вершина `v8` соответствует наибольшей общей подстроке, `path[v8]` = `ba`.

```
        if ((e=back_edge[u]),(e->i<=pos&&pos<e->j))
            mask[v] |= 1;
        if ( dist[v] > dist[u] - LENGTH(e->i,e->j) )
            dist[*tl++ = v]=dist[u]-LENGTH(e->i,e->j);
    }
    printf("%lu\n",w);
}
```

Замечание. Наш код прошел все тесты на сервере Гданьского Университета(Польша) `www.spoj.pl/porblems/LCS`.

Наибольшая общая подстрока k строк. Пусть даны строки `abba`, `baa`, `abaab`, `bba`. Построим суффиксное дерево (*обобщенное суффиксное дерево* – [10]) `T(abba#baa&abaab@bba$)` (рис 7). Глубочайшая вершина с таким свойством, что из неё существует по крайней мере 4 различных пути, при следовании по которым первыми встречаются все четыре различных маркера. Такую вершину можно найти поиском в ширину. Сложность $O(n \log k)$.

6 Заключение

Отметим, что попытка реализовать суффиксные деревья в функциональной парадигме([7]) все же не смогла обойти сложности $\mathtt{O}(|\Sigma|\mathtt{n}^2)$ в худшем случае. Время работы нашей реализации на низкоуровневом языке C (см. Приложение) оценивается $\mathtt{O}(\mathtt{n}\log|\Sigma|)$. Главным преимуществом нашей реализации является её *гибкость*: массив `text` может состоять из элементов произвольной природы, лишь бы на этом множестве был задан полный порядок. С другой стороны, пользуясь тем, что алфавит ASCII индексирован, можно достичь оценки $\mathtt{O}(\mathtt{n})$, напрочь убрав $\log|\Sigma|$. Речь идёт об алгоритме Фараха [3, 9], который основан на несколько иных интуитивных идеях. Дальнейшее направление нашей работы будет связано именно с ним.

A Реализация на языке C

```
1  /*
2   */
3  #include <assert.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  enum { L, R };
8  enum { Red, Black };
9  #define which(x) (x->p->son[L] == x ? L : R)
10 #define color(x) (x->c)
11 #define flip(x)  (x->c ^= 1)
12 #define LEN ((1UL<<17)+1UL)
13 #define N LEN
14 enum { DELIM = '$', root = 0 };
15 #define M (2*LEN+1)
16 #define LENGTH(i,j) ((i)<=(j)?((j)-(i)+1):((i)-(j)+1))
```

```
17
18 char text[LEN];
19 int n;
20
21 typedef struct {
22     int v;    // vertex id
23     int i,j; // text[i:j]
24 } arc;
25
26 arc *make_arc( int v, int i, int j ) {
27     arc *e = (arc *)malloc(sizeof *e);
28     e->v = v, e->i = i, e->j = j;
29     return e; // e = <father[e.v], e.v>
30               // label<father[e.v],e.v> = text[i:j]
31 }
32
33 int cmp_arcs( arc *a, arc *b ) {
34     if ( text[a->i] != text[b->i] )
35         return text[a->i] < text[b->i] ? -1 : 1;
36     return 0;
37 }
38
39 /*
40  * Red-Black Tree stores
41  * all the edges outgoing from
42  * suffix-tree node.
43  */
44 typedef struct cell {
45     struct cell *son[2], *p; // left/right son, parent
46     char c;                  // color
47     arc *e;
48 } cell;
```

```
49
50 /*
51  * NIHIL -- sentinel;
52  */
53 cell *NIHIL;
54
55 typedef struct {
56       cell *root;
57       size_t count;
58 } tree;
59
60 tree *init_tree() {
61       tree *t = (tree *)malloc(sizeof *t);
62       t->root = NIHIL, t->count = 0;
63       return t;
64 }
65
66 void rotate( tree *t, cell *x, int i ) {
67       cell *y;
68       assert( (y = x->son[i^1]) != NIHIL );
69       assert( (x->son[i^1] = y->son[i]) == NIHIL ||\
70             (y->son[i]->p = x) != NIHIL );
71       (y->p = x->p) != NIHIL ? (x->p->son[which(x)] = y):\
72                   (t->root = y);
73       x->p = y, y->son[i] = x;
74 }
75
76 void tree_push( tree *t, arc *e ) {
77       cell *x,*y,*g,**hold;
78       int i;
79
80       for ( y = NIHIL, x = t->root, hold = &t->root;; ) {
```

```
81            if ( x == NIHIL ) {
82                x = (*hold) = (cell *)malloc(sizeof *x);
83                x->son[L] = x->son[R] = NIHIL, x->p = y;
84                x->c = Red, x->e = e, ++t->count;
85                break;
86            }
87            if ( !( i = cmp_arcs(x->e,e)) )
88                assert (0);
89            i = i < 0 ? R : L;
90            y = *hold, hold = &x->son[i], x = x->son[i];
91        }
92        while ( x != t->root && color(x->p) == Red ) {
93            i = which(x->p), g = x->p->p;
94            y = g->son[i^1];
95            if ( color(y) == Red ) {
96                flip (x->p), flip (y), flip (g), x = g;
97                continue;
98            }
99            if ( which(x) == (i^1) )
100               x = x->p, rotate(t,x,i );
101           flip (x->p), flip (g), rotate(t,g,i^1);
102       }
103       t->root->c = Black;
104   }
105
106   tree *q[M];
107   int head[N],leaf[N],m;
108   int father[M],suf[M];
109   arc *back_edge[M];
110
111   arc *get_arc( tree *t, int k ) {
112       cell *x;
```

```
113     if ( !t ) return NULL;
114     for ( x=t->root; x!=NIHIL && text[x->e->i]!=text[k];)
115         x = x->son[text[x->e->i] < text[k] ? R : L];
116     return x == NIHIL ? NULL : x->e;
117 }
118
119 void fixup( tree *t, cell *x ) {
120     cell *w;
121     int i;
122
123     while ( x != t->root && color(x) == Black ) {
124         i = which(x);
125         assert( (w = x->p->son[i^1]) != NIHIL );
126         if ( color(w) == Red )
127             flip(x->p), flip(w), rotate(t,x->p,i);
128         else if ( color(w->son[i^1]) == Black &&\
129                   color(w->son[i])   == Black )
130             flip(w), x = x->p;
131         else if ( color(w->son[i^1]) == Black &&\
132                   color(w->son[i])   == Red )
133             flip(w), flip(w->son[i]), rotate(t,w,i^1);
134         else {
135             w->c = x->p->c, flip(w->son[i^1]);
136             x->p->c = Black;
137             rotate(t,x->p,i);
138             x = t->root;
139         }
140     }
141     x->c = Black;
142 }
143
144 cell *erase( tree *t, cell *z ) {
```

```
145         cell  *x,*y;
146
147         if  ( !z  ||  z == NIHIL )
148             return NULL;
149
150         if  ( (y = z->son[L]==NIHIL||z->son[R]==NIHIL?z:\
151                 z->son[R]) != NIHIL ) {
152             for  ( ;y->son[L] != NIHIL; y = y->son[L] );
153             z->e = y->e;
154         }
155         x = y->son[L] != NIHIL ? y->son[L] : y->son[R];
156         (x->p=y->p)!=NIHIL ? (y->p->son[which(y)]=x):\
157                 (t->root = x);
158         if  ( color(y) == Black )
159             fixup(t,x);
160         --t->count;
161
162         return y;
163
164 }
165
166 cell  *find( tree  *t,  arc  *e ) {
167         cell  *x = t->root;
168         int  i;
169         for  (;x && x!=NIHIL && (i=cmp_arcs(x->e,e));)
170             x = x->son[i < 0 ? R : L];
171         return x == NIHIL ? NULL : x;
172 }
173 /*
174  * We're done with Red-Black Tree implementation
175  */
176
```

```
177 /*
178  * Thing relevant to Suffix Tree begin here
179  */
180 void initialize_suff_tree () {
181       assert( root == 0 );
182       m = root+1;
183       memset( suf, -1, sizeof(suf) );
184       memset( head, root, sizeof(head) );
185       q[root] = init_tree(), suf[root] = root;
186       q[leaf[1] = m++] = init_tree();
187       father[leaf[1]] = root;
188       back_edge[leaf[1]] = make_arc(leaf[1],1,n+1);
189       tree_push( q[root], back_edge[leaf[1]] );
190 }
191
192 /*
193  * introduce a new leaf "v" as
194  * the son of "u" and set label<u,v> = text[left:right]
195  */
196 int planting( int u, int left, int right ) {
197       int v;
198
199       if ( left > right )
200             return u;
201       q[v = m++] = init_tree();
202       tree_push(q[father[v] = u],\
203       back_edge[v] = make_arc(v,left,right));
204       return v;
205 }
206
207 /*
208  * split the edge "e" outgoing from vertex "u";
```

```
209  * return the newly created vertex ``v''
210  */
211 int split( int u, arc *e, int len ) {
212     int v, w = e->v;
213     cell *tmp;
214
215     assert( len > 0 );
216     assert( len < LENGTH(e->i,e->j) );
217     assert( e->i + len <= e->j );
218     assert( tmp = erase(q[u],find(q[u],e)) );
219     v = planting( u, e->i, e->i+len-1 );
220     father[w] = v;
221     back_edge[w] = make_arc( w, e->i+len, e->j );
222     tree_push( q[v], back_edge[w] );
223     return v;
224 }
225
226 /*
227  * Precondition: text[left:right] is fully contained
228  * in a path starting from ``root''
229  */
230 int fastscan( int u, int left, int right ) {
231     arc *e;
232     int k,cur,v;
233
234     if ( left > right )
235         return root;
236
237     for ( cur = left; cur <= right; ) {
238         assert( e = get_arc(q[u],cur) );
239         assert( text[e->i] == text[cur] );
240         if ( LENGTH(cur,right)<LENGTH(e->i,e->j) ) {
```

```
241                 k = LENGTH(cur,right);
242                 v = split(u,e,k);
243                 return v;
244             }
245             if ( LENGTH(cur,right)==LENGTH(e->i,e->j) )
246                 return e->v;
247             cur += LENGTH(e->i,e->j);
248             u = e->v;
249         }
250
251         assert( 0 ); // execution should never reach here
252     }
253
254     /*
255      * "cur" never exceeds "right" because of the DELIM,
256      * so we needn't add (cur <= right) condition
257      */
258     int slowscan( int u, int left, int right ) {
259         int k,cur,v;
260         arc *e;
261         for ( cur = left; e = get_arc(q[u],cur); ) {
262             for (k=e->i;text[k]==text[cur]&&k<=e->j; ++cur,++k);
263             if ( k > e->j ) { u = e->v; continue; }
264             assert( k <= e->j && text[k] != text[cur] );
265             v = split(u,e,LENGTH(e->i,k)-1);
266             return planting(v,cur,right);
267         }
268         return planting(u,cur,right);
269     }
270
271     void McCreight() {
272         int i,j,k,
```

```
273             left , right ,
274             u,v,w;
275
276         initialize_suff_tree ();
277
278         for ( i = 1; i <= n; ++i ) {
279
280             if ( head[i] == root ) {
281                 leaf [i+1] = slowscan(root,i+1,n+1);
282                 head[i+1] = father[leaf [i+1]];
283                 continue;
284             }
285
286             u       = father[head[i ]];
287
288             left    = back_edge[head[i]]->i;
289             right = back_edge[head[i]]->j;
290
291             if ( u == root )
292                 w = fastscan(root, left +1, right );
293             else
294                 w = fastscan(suf[u], left , right );
295
296             assert( q[w] && q[w]->count >= 1 );
297
298             suf[head[i ]] = w;
299
300             if ( q[w]->count == 1 ) {
301                 head[i+1] = w;
302                 left = back_edge[leaf[i]]->i ;
303                 right = back_edge[leaf[i]]->j ;
304                 leaf [i+1] = planting(head[i+1],left , right );
```

```
305             }
306             else {
307                 left  = back_edge[leaf[i]]->i;
308                 right = back_edge[leaf[i]]->j;
309                 leaf[i+1] = slowscan( w, left, right );
310                 head[i+1] = father[leaf[i+1]];
311             }
312         }
313 }
314
315 int main() {
316     int i;
317
318     NIHIL = (cell *)malloc(sizeof *NIHIL);
319     NIHIL->p = NIHIL, NIHIL->c = Black;
320
321     fgets( text+1, sizeof(text), stdin );
322     for ( n = 0; text[n+1] && text[n+1] != '\n'; ++n );
323     assert( !text[n+1] || text[n+1] == '\n' );
324     text[n+1] = DELIM;
325     McCreight();
326     return 0;
327 }
```

B Визуализация суффиксного дерева

Приведенный ниже кусок кода можно состыковать с системой визуализации `graphViz v2.22.2 dot` для раскладки суффиксного дерева в графический файл.

```
1 void printout( int u ) {
2     cell *x,*y;
3     int k,i,j;
```

```
4       char tmp;
5       arc *e;
6
7       if ( u >= m || q[u]->count <= 1 )
8           return ;
9
10      for (x=q[u]->root; x->son[L]!=NIHIL; x=x->son[L]);
11      for ( k = q[u]->count; k; x = y ) {
12          e = back_edge[x->e->v];
13          tmp = text[e->j+1], text[e->j+1] = '\0';
14          printf("v%d -> v%d [label = \"%s\"];\n",\
15                  u,x->e->v,text + e->i);
16          text[e->j+1] = tmp;
17          printout(x->e->v);
18          if ( !-- k ) break;
19          if ( x->son[R] != NIHIL )
20              for ( y = x->son[R]; y->son[L] != NIHIL;\
21                      y = y->son[L] );
22          else
23              for ( y = x->p; y != NIHIL && which(x) == R;\
24                      x = y, y = y->p );
25      }
26  }
27  <...some code here...>
28      printf("digraph G {\n"), printout( root ), puts("\n}");
29  <...some code here...>
```

Для подробностей рекомендуем обратиться к dotguide.pdf из документации к `graphViz`.

Список литературы

[1] A.Apostolico and M.Crochemore, Optimal Canonization of All Substrings of a String, 1989.

[2] C.Iliopoulus and S.Rahman, Indexing Circular Patterns.

[3] M.Crochemore and W.Rytter, Jewels of Stringology, World Scientific Publishing Co., 2002.

[4] M.Lothaire, Combinatorics on Words, Cambridge University Press, 1997.

[5] M.Maab, Suffix Trees and Their Applications, Tech. report, Munchen Technical University, 1999.

[6] M.Neslon, Fast String Searching With Suffix Trees, 1996.

[7] R.Giegerich and S.Kurtz, Suffix Trees in the Functional Programming Paradigm.

[8] T.Mailund, McCreight's suffix tree construction algorithm, 2007.

[9] W.Smyth, Computing Patterns in Strings, Pearson Education Limited, 2003.

[10] Д.Гасфилд, Строки, деревья и последовательности в алгоритмах, Невский Диалект, БХВ-Петербург, Санкт-Петербург, 2003.