Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Д. А. Паращенко

Суффиксные автоматы с сохранением промежуточных версий и их приложения

Магистерская диссертация

Научный руководитель: А. С. Станкевич

Санкт-Петербург
2009

Оглавление

Введение

Структуры данных, которые хранят свои промежуточные версии, имеют намного более широкий круг приложений, чем их варианты без такой возможности [1, 2, 3, 4]. Классическим примером является задача нахождения области, содержащей заданную точку [1]. Она может быть элегантно решена с помощью применения дерева поиска с сохранением промежуточных версий.

Не существует механизма, позволяющего после совершения какие-либо модификаций над обычной структурой данных вернуться к ее предыдущей версии. Структуры данных, хранящие свою хронологию, называются персистентными.

Существует несколько видов персистентных структур данных [1]. Частичная персистентность позволяет модифицировать последнюю версию структуры данных и делать запросы к любой из промежуточных версий. Полная персистентность позволяет совершать запросы и модификации с любой версией структуры данных. При таком типе персистентности версии образуют не линейную, а древовидную структуру. В обоих случаях доступ осуществляется по номеру требуемой версии.

В настоящее время для решения множества строковых задач используются суфиксные деревья, суффиксные массивы и суффиксные автоматы [5, 6, 8, 9, 10]. Также известны методы, позволяющие делать суффиксные деревья персистентными. В продолжении работы [11] возникает вопрос о возможности эффективной реализации персистентных суффиксных автоматов.

Рассмотрены существующие методы модификаций структур данных с целью добавления в них возможности сохранения промежуточных версий приминительно к суффиксному автомату. Предложен метод, позволяющий с небольшими дополнительными затратами сделать суффиксный автомат персистентным.

Свойство персистентности позволяет суффиксному автомату поддерживать помимо операции добавления символа операцию удаления послед-

него символа. При этом последняя операция не вызывает никаких дополнительных временных затрат.

Кроме того, персистентность позволяет использовать суффиксные автоматы в многопоточных приложениях, в которых предъявляются особые требования к быстродействию. Применение персистентной структуры данных позволяет выполнять операции по модификации суффиксного автомата без блокировки операций чтения. В случае суффиксных автоматов это является достоинством, так как модификация суффиксного автомата может занимать до $O(n)$ времени.

Глава 1.

Основные понятия

1.1. Строки

Определение 1. [6, стр. 3] Пусть Σ — конечное множество, называемое *алфавитом*. Под Σ^* будем понимать *множество слов* над алфавитом Σ. *Пустое слово* обозначим за ε. Множество всех непустых слов обозначим за $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Определение 2. [6, стр. 3] Для слова w, под $|w|$ будем понимать *длину* w, то есть количество символов в w.

Определение 3. [6, стр. 4] Строка w называется *подстрокой* строки u, если существуют такие слова x и y, что $u = xwy$.

Определение 4. [6, стр. 4] Строка w называется *префиксом* строки u, если существует такое слово y, что $u = wy$.

Определение 5. [6, стр. 4] Строка w называется *суффиксом* строки u, если существует такое слово x, что $u = xw$.

Определение 6. Множество всех подстрок слова x обозначим за $Fact(x)$.

Определение 7. Множество всех суффиксов слова x обозначим за $Suff(x)$.

Определение 8. [6, стр. 4] Префикс длины k слова w обозначим за $w[1 \dots k]$.

Определение 9. k-ый символ слова w обозначим за $x[k]$.

1.2. Правые контексты

Определение 10. [6, стр. 116] *Правым контекстом* строки x в строке y называется $R_y(x) = x^{-1}Suff(y) = \{s \in \Sigma^* \mid xs \in Suff(y)\}$.

Неформально говоря, правый контекст строки x в строке y состоит из всех суффиксов строки y, идущих непосредственно за строкой x.

Определение 11. [6, стр. 116] Строки u и v называются *конгруэнтными* в строке y ($u \equiv_y v$), если $R_y(u) = R_y(v)$.

Определение 12. [6, стр. 116] Для каждой подстроки u строки y правую позицию первого вхождения u в y обозначим за $end - pos_y(u)$.

Лемма 1. [6, лемма 2.3.1] Пусть $u, v \in Fact(y)$ и $|u| \leq |v|$. Тогда:

- если u является суффиксом v, то $R_y(v) \subseteq R_y(u)$;
- если $R_y(v) = R_y(u)$, то $end - pos_y(u) = end - pos_y(v)$ и $u \in Suff(v)$.

Лемма 2. [6, лемма 2.3.2] Пусть $u, v, w \in Fact(y)$. Если u является суффиксом v, а v — суффиксом w, и $u \equiv_y w$, тогда $u \equiv_y v \equiv_y w$.

Лемма 3. [6, лемма 2.3.3] Пусть $u, v \in \Sigma^*$. Тогда правые контексты u и v либо сравнимы по включению, либо не пересекаются. То есть выполнено хотя бы одно из следующих утверждений:

- $R_y(u) \subseteq R_y(v)$;
- $R_y(v) \subseteq R_y(u)$;
- $R_y(u) \cap R_y(v) = \varnothing$.

Поскольку отношение конгруэнтности является отношением эквивалентности, все строки разбиваются на классы эквивалентности, также называемые классами конгруэнтности.

Определение 13. *Множеством представителей* правого контекста C в строке y называется множество $Repr_y(C) = \{x \in Fact(y) \mid R_y(x) = C\}$.

Теперь дадим определения наибольшего и наименьшего представителя правого контекста.

Определение 14. *Наибольшим представителем* правого контекста C в строке y называется элемент множества $Repr_y(C)$, имеющий наибольшую длину.

Определение 15. *Наименьшим представителем* правого контекста C в строке y называется элемент множества $Repr_y(C)$, имеющий наименьшую длину.

Лемма 4. Множество представителей правого контекста состоит из суффиксов наибольшего представителя, длина которых не меньше длины наименьшего представителя этого контекста. Более формально:

Пусть w_{max} и w_{min} — наибольший и наименьший представители правого контекста C в строке y. В таком случае, s принадлежит $Repr_y(C)$ тогда и только тогда, когда s является суффиксом w_{max} и $|s| \geq |w_{min}|$.

$\triangleright$ Пусть $s \in Repr_y(C)$. Покажем, что $s \in Suff(w_{max})$ и $|s| \geq |w_{min}|$.

По определению 14 $w_{max} \in Repr_y(C)$ и $|w_{max}| \geq |s|$. По лемме 1 из этого следует, что $s \in Suff(w_{max})$. С другой стороны, по определению 15 мы имеем $|s| \geq |w_{min}|$, что и требовалось доказать.

Пусть $s \in Suff(w_{max})$ и $|s| \geq |w_{min}|$. Покажем, что $s \in Repr_y(C)$.

По определению 15 и лемме 1 $w_{min} \in Suff(s)$. Из определений 14 и 15 следует, что $w_{max} \equiv_y w_{min}$. Тогда, по лемме 2 $w_{max} \equiv_y s \equiv_y w_{min}$, что и требовалось доказать. $\triangleleft$

Ниже приведено важное следствие этой леммы.

Следствие 5. Множество представителей некоторого правого контекста однозначно задается его наибольшим представителем и длиной его наименьшего представителя.

$\triangleright$ Рассмотрим правый контекст $R_y(x)$. Пусть w_{max} — его наибольший представитель, и len_{min} — длина его наименьшего представителя. Тогда, по теореме 4 $Repr_y(R_y(x)) = \{z \mid z \in Suff(w_{max}) \wedge |z| \geq len_{min}\}$. $\triangleleft$

Следствие 6. Для любого правого контекста C и числа len существует не более одного слова y длины len, такого, что $R_w(y) = C$.

$\triangleright$ Предположим, что существуют строки y_1 и y_2 длины len такие, что $C = R_w(y_1) = R_w(y_2)$. Докажем, что $y_1 = y_2$. По лемме 1 либо $y_1 \in Suff(y_2)$, либо $y_2 \in Suff(y_1)$. Из того, что y_1 и y_2 являются суффиксами друг друга, и их длины равны, следует, что $y_1 = y_2$, что и требовалось доказать. $\triangleleft$

1.3. Суффикс функция

Приведенные в этом параграфе определения и факты понадобятся в дальнейшем для изучения суффиксного автомата и алгоритма его построения за линейное время.

Определение 16. [6, стр. 118] Рассмотрим на множестве $Fact(w)$ функцию s_w, называемую *суффикс функцией* строки w, которая определена для всех $x \in Fact(w) \setminus \{\varepsilon\}$ и равна наидлиннейшему неконгруэнтному x суффиксу x.

Теорема 7. Пусть $x_2 = s_w(x_1)$. Тогда длина наибольшего представителя $R_w(x_2)$ на единицу меньше длины наименьшего представителя $R_w(x_1)$.

$\triangleright$ Из определения 16 следует, что x_2 является суффиксом строки x_1 и $|x_2| < |x_1|$.

Рассмотрим строку y, являющуюся суффиксом строки x_1, длина которого на единицу больше длины строки x_2. То есть $y = x_1[|x_1| - |x_2| - 1 \dots |x_1|]$.

Из определения 16 следует, что $y \equiv_w x_1$ и x_2 является наибольшим представителем $R_w(x_2)$. Из леммы 2 и того факта, что $y \not\equiv_w x_1$, следует, что y является наименьшим представителем $R_w(y) = R_w(x_1)$. По построению $|y| = 1 + |x_2|$, что и требовалось доказать. $\triangleleft$

Лемма 8. Для любой непустой строки x из $Fact(w)$ имеет место соотношение $R_w(x) \subset R_w(s_w(x))$.

$\triangleright$ По определению 16 $s_w(x)$ является суффиксом x, $|s_w(x)| < |x|$ и $R_w(x) \neq R_w(s_w(x))$.

Для завершения доказательства достаточно воспользоваться леммой 1. $\triangleleft$

1.4. Суффиксный автомат

Определение 17. [6, стр. 121] *Суффиксным автоматом* A строки w называется наименьший детерминированный конечный автомат над алфавитом Σ, допускающий все суффиксы строки w и только их.

На рис. 1 приведен суффиксный автомат строки $aabb$.

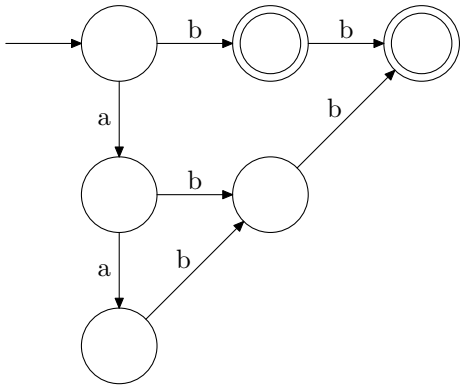

Рис. 1. Суффиксный автомат строки *aabb*

Определение 18. Будем говорить, что состоянию s суффиксного автомата соответствует правый контекст $R_w(x)$, если $R_A(s) = R_w(x)$.

Предложение 9. [6] Существует взаимно однозначное соответствие между множеством состояний суффиксного автомата и множеством непустых правых контекстов в w.

Поэтому, можно говорить о правом контексте состояния, подразумевая под этим соответствующий этому состоянию правый контекст, и о представителях состояния, подразумевая под этим представителя правого контекста этого состояния.

Предложение 10. Множество всех строк, приводящих автомат из начального состояния в некоторое состояние s, совпадает с множеством представителей правого контекста этого состояния.

$\triangleright$ Для доказательства этого факта достаточно заметить, что суффиксный бор является детерминированным конечным автоматом, и воспользоваться определениями 17 и 13. $\triangleleft$

Определение 19. Пусть s — состояние, в котором окажется суффиксный автомат после получения на вход непустого слова x. *Суффиксной ссылкой* из состояния s называется ссылка, ведущая из состояния s в состояние, в котором окажется автомат после получения на вход слова $s(x)$. Обозначим это состояние за $suffix(s)$.

Определение 20. Длину наибольшего представителя состояния s обозначим за $repr_{max}(s)$.

Замечание. В связи с тем, что на сегодняшней день известен алгоритм построения суффиксного автомата за линейное время [6, стр. 126], который в процессе построения суффиксного автомата считает суффикс функцию и длину наибольшего представителя каждого состояния, можно считать, что суффиксный автомат, как структура данных, имеет суффиксные ссылки и для каждого состояния известна длина его наибольшего представителя.

Предложение 11. Пусть состояние s суффиксного автомата не является стартовым. Тогда длина наименьшего представителя состояния s равна $1 + repr_{max}(suffix(s))$.

$\triangleright$ Требуемое равенство следует из определений 19 и 20, а также теоремы 7. $\triangleleft$

Теорема 12. Пусть непустое слово x является некоторым представителем состояния s_1 суффиксного автомата. Пусть s_2 — состояние, в которое придет суффиксный автомат, приняв на вход слово $x[2 \dots |x|]$.

Тогда:

- если $|x| = repr_{max}(suffix(s_1)) + 1$, то s_2 совпадает с $suffix(s_1)$
- если $|x| > repr_{max}(suffix(s_1)) + 1$, то s_2 совпадает с s_1

$\triangleright$ Предположим, что $|x| > repr_{max}(suffix(s_1))+1$. Из определений 19 и 16 следует, что слово $x[2 \dots |x|]$ является представителем $R_w(x)$, что означает, что суффиксный автомат, приняв на вход слово $x[2 \dots |x|]$, окажется в состоянии s_1.

Предположим, что $|x| = repr_{max}(suffix(s_1))+1$. В этом случае, слово $x[2 \dots |x|]$ является представителем $R_w(s_w(x))$, что по определению 16 означает, что суффиксный автомат, приняв на вход слово $x[2 \dots |x|]$, окажется в состоянии $suffix(s_1)$. $\triangleleft$

1.5. Алгоритм построения суффиксного автомата

Суффиксный автомат строки s можно построить, например, следующим образом. Сначала строим бор суффиксов s, а затем минимизируем построенный бор. Однако, у такого алгоритма имеется большой недостаток — он квадратичен. Известен алгоритм, позволяющий строить суффиксный автомат за $O(|s|)$.

В процессе построения суффиксного автомата строки s отрабатываются ее префиксы начиная с ε и заканчивая s. При этом, после обработки каждого префикса w доступна следующая информация:

- суфиикснный автомат строки w со своей функцией переходов;
- суффикс-функция s_w, определенная на состояниях суффиксного автомата;
- функция ln_w, определенная на состояниях суффиксного автомата;
- последнее состояние автомата $last$.

При таком построении автомата, его допускающие состояния заданя неявно. А именно, допускающими являются состояния, лежащие на суффиксном пути из последнего состояния суффиксного автомата.

Итак, алгоритм построения суффиксного автомата основывается на процедуре добавления к базовой строке очередного символа.

Ниже приведена процедура добавления к базовой строке очередного символа.

```
1     protected void addCharacter(char ch) {
2         int p = lastNode;
3         lastNode = createNode(getLn(p) + 1);
4         while (p >= 0 && getEdge(p, ch) == -1) {
5             addEdge(p, ch, lastNode);
6             p = getSuffix(p);
7         }
8         if (p < 0) {
9             setSuffix(lastNode, 0);
10            return;
11        }
12        int q = getTarget(getEdge(p, ch));
13        if (getLn(p) + 1 == getLn(q)) {
14            setSuffix(lastNode, q);
```

```
15          } else {
16              int r = createNode(getLn(p) + 1);
17              for (int edge : getEdges(q)) {
18                  cloneEdge(r, edge);
19              }
20              setSuffix(r, getSuffix(q));
21              setSuffix(q, r);
22              for (int edge = getEdge(p, ch); p
                    >= 0 && edge >= 0 && getTarget(edge)
                    == q; p = getSuffix(p), edge = p
                    < 0 ? -1 : getEdge(p, ch)) {
23                  setTarget(edge, r);
24              }
25              setSuffix(lastNode, r);
26          }
27      }
```

Заметим, что в процессе построения суффиксного автомата совершаются лишь следующие модификации:

- добавление нового состояния (строки 3 и 16);
- добавление нового перехода (строки 5 и 18);
- изменение конечного состояния перехода (строка 23);
- изменение суффикс-ссылки (строки 9, 14, 20, 21, 25).

1.6. Персистентные структуры данных

Операции над любой структурой данных можно разделить на два типа:

- модифицирующие структуру данных;

- не модифицирующие структуру данных.

При модификации обычной структуры данных вся информация о ее прежнем состоянии пропадает. В некоторых задачах требуется, чтобы структура данных хранила историю своих изменений.

Определение 21. *Персистентные* структуры данных — это структуры данных, хранящие все свою промежуточные версии.

Существует два вида персистентности.

Определение 22. *Частичная персистентность* означает возможность доступа на чтение к любым версиям структуры данных, и модификации только последней ее версии.

Определение 23. *Полная персистентность* означает возможность совершения любых операций над любой версией структуры данных.

1.7. Персистентные деревья

В этой главе приведен краткий обзор метода, предложенного в работе Слейтера и Тарьяна [12], позволяющего эффективно сделать деревья персистентными.

1.7.1. Структура дерева и поддерживаемые операции

Все выполняемые над деревьями операции можно разделить на две категории:

- модификации — операции, приводящие к изменениям структуры дерева или хранящейся в его вершинах и ребрах дополнительной информации;

- запросы — все остальные операции над деревом.

В то время как с запросами все более-менее ясно, с модификациями дела обстоят не так просто. Необходимо каким-нибудь образом их классифицировать. Модификции бывают следующих видов:

- структурными — добавление или удаление вершины. В связи с тем, что дерево является связной структурой данных, изменение множества вершин не может происходить без модификаций ребер. При создании новой вершины создается ребро, а при удалении — удаляется. Не стоит забывать, что при этом может соответсвующим образом меняться еще одна вершина — второй конец ребра.

- изменение хранящейся в вершине дополнительной информации;

- изменение хранящейся в ребре дополнительной информации.

Модификации первого типа является структурными. Однако, это не означает, что они более или менее приоритетны, чем модификации второго и третьего типов. Все модификации дерева необходимы в равной степени и поддержка лишь некоторых из них почти бессмысленна.

1.7.2. Толстые вершины

Этод метод основывается на следующих соображениях. Во-первых, модификация ребра может быть сведена к модификации его исходной (родительской) вершины. Это сразу же делает ребра дерева immutable объектами. Если выражаться еще точнее, ребра становятся частью вершины.

Что происходит с деревом при модификациях вершины? Для ответа на этот вопрос удобно ввести понятие версии вершины. При создании новой вершины ей присваивается версия дерева, в которой она была добавлена. А при модификции — версия, в которой она была изменена. Таким образом, версия вершины дерева — это версия дерева, в которой произошла последняя модификация вершины.

Заметим, что наличие версий вершин дерева позволяет нам сказать, существовали ли они в некоторой версии дерева. Однако, этой информации не достаточно для получения информации о полном состоянии вершины. Для того, чтобы иметь возможность получать всю информацию о вершинах дерева определенной версии, необходимо при модификации вершины не просто совершать над ней эту модификацию, но и сохранять предыдущую (неизмененную) версию вершины. Другими словами, при изменении вершины необходимо не менять ее, а создавать новую. При таком подходе вершины дерева, как и его ребра, становятся immutable.

Таким образом, в дереве может существовать несколько версий одной и той же вершины. Для ответа на вопрос, какая из этих вершин соответствует дереву определенной версии v, необходимо из всех вершин-кандидатов выбрать вершину с наибольшей версией, не превышающей v.

На рис. 2 продемонстрирован метод толстых вершин при добавлении в дерево новой вершины.

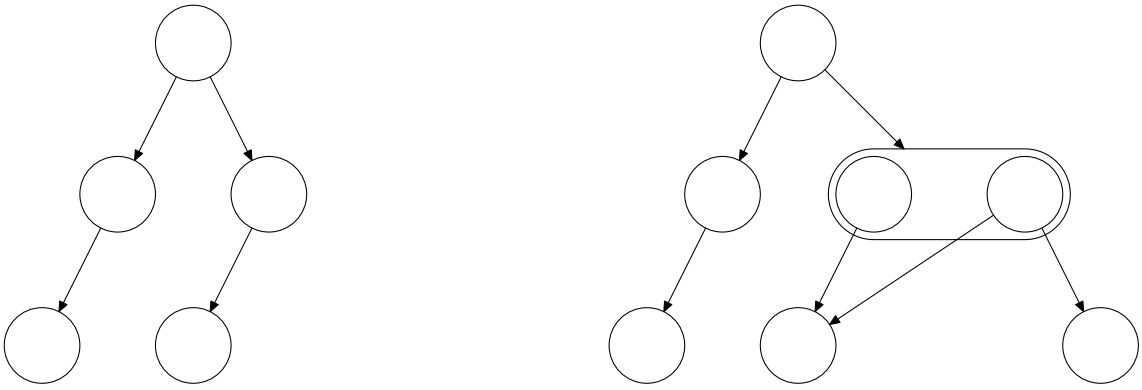

Рис. 2. Метод толстых вершин на дереве

Имеет смысл обхединить в одно множество различные версии одной и той же вершины. Множество таких версифицированных вершин будем называть толстой вершиной. В случае частичной персистентности логично хранить это множество в виде списка, упорядоченного по версии вершины. В случае же полной персистентности можно хранить множество вершин как в виде списка, так и в виде дерева. Это зависит от выбранной модели присвоения версий.

Описанный метод обладает как рядом достоинств, так и рядом недостатков. К его достоинствам можно отнести дополнительные чистые (не амортизированные) $O(1)$ по времени и $O(1)$ по памяти на модификации. К недостаткам — для выполнение запроса над вершиной v тратится дополнительное $O(\log m_v)$ время, где m_v — количество модификаций вершины v.

1.7.3. Копирование путей

В этом методе используются те же принципы версионирования вершин, к как и в методе толстых вешнин. Любые модификации ребер попрежнему сводятся к модификациям вершин. Однако, процесс модифик-

ции вершин отличается. После создания новой (уже измененной) вершины она не добавляется в соответствующую ей толстую вершину. Вместо этого производится модификация входящего в нее ребра, а вместе с ним — родительской вершины. Такой процесс выдывает каскаждую модификацию вершин дерева вплоть до его корня. Поскольку у корня дерева родитель отсутствует, к нему применяется подход толстых вершин.

Таким образом, у такого персистентного дерева число корней (размер толстой вершины-корня) равно числу его версий. При этом, для работы с деревом версии v достаточно найти соответствующий корень. Это можно сделать, например, за $O(\log m)$, где m — количество модификаций дерева. После того, как корень дерева найден, можно с ним работать как с обычным деревом без каких-либо дополнительных временных затрат.

На рис. 3 продемонстрирован метод копирования путей при добавлении в дерево новой вершины.

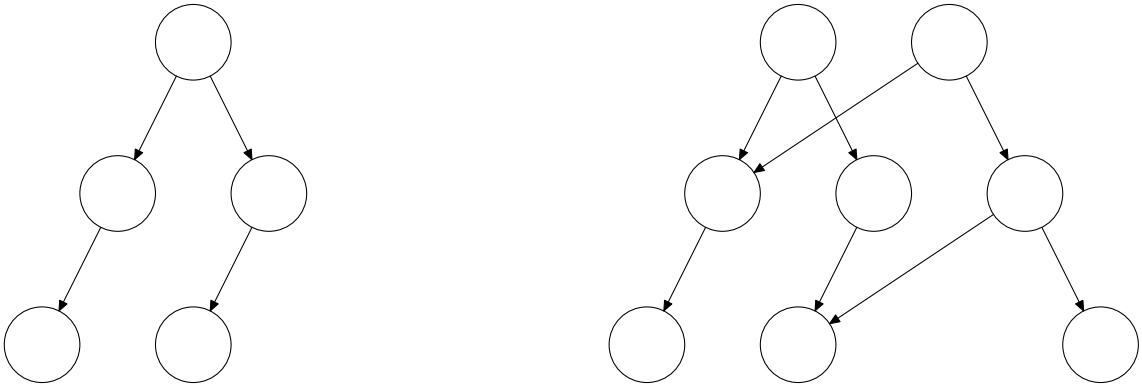

Рис. 3. Метод копирования путей на дереве

Другими словами, идея этого метода состоит в том, чтобы деревья различных версий разделяли между собой как можно больше вершин. В таком случае каждое из деревьев получается в некотором смысле независимым, однако, не происходит дублирования информации, и, как следствие, лишнего расхода памяти.

Как хранить корни? При таком подходе не обязательно хранить корни в виде отсортированного списка версий и производить поиск корня требуемой версии двоичным поиском по этому списку. Можно заметить, что версии корня являются последовательными числами от 1 до m, где m — количество модификаций дерева. Это позволяет хранить корни дерева в структуре данных с прямой адресацией по версии, например, в массиве.

При использовании массива, дополнительное время на доступ к определенной версии дерева равно $O(1)$, дополнительных расходы памяти равны $O(m)$.

Безусловно, отсутвие каких-либо потерь при доступе к определенной версии дерева является сильным достоинством. Однако, описанный подход обладает большим недостатком, а именно — операции по модификации очень дороги. В худшем случае, одна модификация может занимать $O(n)$ времени, где n — размер дерева.

1.7.4. Комбинирование методик

Слейтер и Тарьян [12] разработали метод, обладающий достоинствами обоих подходов. Его идея заключается в том, что толстая вершина может хранить в себе не более двух версий: исходную версию и, возможно, одну модификацию.

При доступе к вершине выполняется проверка того, какую из версий необходимо использовать. Если толстая вершина подвргалась модификации и искомая версия не меньше версии после модификации, следует использовать модифицированную версию вершины. Иначе, следует использовать исходную версию вершины. Эти проверки можно сделать за $O(1)$.

Процесс модификации вершины немного сложнее. В случае, если вершина еще не подвергалась модификациям, к ней добавляется ее новая (модифицированная) версия, и на этом модификация вершины заканчивается.

Если же вершина была ранее изменена, создается ее клон с одновременным приминением к нему выполняемой в данный момент модификации. При этом, вновь созданная толстая вершина является "свежей"в том смысле, что она хранит в себе лишь одну версию и может быть подвержена еще одной модификации до того, как будет произведено очередное ее клонирование. Теперь требуется произвести модификацию родителя измененной вершины для того, чтобы перенаправить соответствующее ребро со "старой"вершины в "новую". Это часть рассуждений вытекает из методики "копирования путей".

Описанный выше метод обладает рядом достоинств. На выполнение операций по доступу и модификациям требуется лишь $O(1)$ дополнительного времени. То есть, этот метод позволяет без дополнительных в ассимптотическом смысле затрат делать любой дерево персистентным.

Глава 2.

Персистентный суффиксный автомат

В этой главе приведены несколько методов, позволяющих сделать суфиксный автомат персистентным. В конце главы проведено сравнение предложенных методов.

В результате анализа алгоритма построения суффиксного автомата, можно обнаружить, что при добавлении к базовой строке очередного символа суффиксный автомат подвергается следующим изменениям:

- добавление нового состояния;
- добавление нового ребра;
- модификация существующего ребра (изменение целевого состояния).

Это, в свою очередь, означает, что после создания состояния единственные его атрибуты, подвергающиеся модификациям — это исходящие переходы. К состоянию может либо добавиться исходящий по новому символу переход, либо измениться целевое состояние уже имеющегося перехода.

2.1. Толстые состояния и толстые переходы

2.1.1. Идея метода

Аналогично ситуации с деревом, можно использовать метод толстых вершин. В таком случае, каждое толстое состояние хранит набор обычных состояний автомата, доступ к которым может осуществляться за время $O(\log m)$, где m — число модификаций состояния.

Поскольку информация, хранящаяся в состоянии, никогда не изменяется, можно применить иной подход и считать, что модификации затрагивают только переходы. Поскольку у перехода может изменяться только

конечное состояние, их можно сгруппировать по начальному состоянию и символу в "толстые переходы".

В таком случае, автомат состоит из состояний и толстых переходов. Для поиска в версии v перехода из состояния s по символу c достаточно найти толстый переход, выходящий из состояния s по символу c, и найти в нем переход, соответствующий версии v. Опять-таки, эту операцию можно проделать за $O(\log m)$, где m, на этот раз, — число модификаций перехода.

2.1.2. Анализ

Для того, чтобы определить эффективность этого метода, необходимо провести оценку числа модификаций перехода, происходящих в процессе построения суффиксного автомата.

Теорема 13. В процессе построения суффиксного автомата каждый его переход в худшем случае подвергается $O(sqrt(n))$ модификациям.

Из доказанной оценки числа модификаций переходов следует, что при использовании предлагаемой реализации персистентного суффиксного автомата, затрачиваемое на операции время может увеличиться не более, чем в $O(\log n)$ раз.

На самом деле, в среднем, затрачиваемое на операции время увеличивается не сильно. На это имеются две причины:

- число толстых в прямом смысле этого слова переходов очень мало;
- строки, при построении суффиксных автоматов которых переходы меняются порядка $O(\log n)$ раз имеют очень специфический вид.

Покажем, почему выполняются эти утверждения.

Предложение 14. Среднее число хранящихся в толстом переходе версий равно $O(1)$.

$\triangleright$ Заметим, что при построении суффиксного автомата переходы модифицируются только после клонирования состояния. Это происходит в последнем цикле алгоритма построения суффиксного автомата. Из доказательства алгоритма построения суффиксного автомата [12] следует, что суммарно модифицируется не более n переходов. Учитывая, что сумма логарифмов меньше логарифма суммы, получаем, что средний размер толстых переходов равен $O(1)$. $\triangleleft$

Лемма 15. Наикратчайшая строка, при построении суффиксного автомата которой переходы изменяются k раз имеет следующий вид: $s_k s_{k-1} s_{k-2} \dots s_2 s_1$, где s_{i-1} является суффиксом s_i.

В рамках работы было проведено исследование числа модификаций переходов при построении суффиксных автоматов строк различных видов. Из табл. 1 видно, что в большинстве случаев число модификаций переходов ограничено константой.

Таблица 1. Максимальное число модификаций переходов при построении суффиксного автомата для строк различного вида над алфавитом {'a', 'b'}

Длина строки	**Число модификаций переходов**	**Теоретически возможное число модификаций переходов**
100000	6	445
200000	6	630
300000	6	773
400000	6	892
500000	6	998
600000	7	1093
700000	7	1181
800000	7	1263
900000	7	1340
1000000	7	1412

На практике строки, при построении суффиксных автоматов которых происходит порядка $O(\log \sqrt{n})$ встречаются настолько редко, что нет смысла уделять им особое внимание.

Из вышесказанного следует, что предложенная реализация персистентных суффиксных автоматов является достаточно эффективной для подавляющегося большинства встречающихся на практике строк.

2.1.3. Вывод

Описанная методика позволяет построить персистентный суффиксный автомат за линейное время с линейным объемом дополнительной памяти. Следовательно, с асимптотически такими же параметрами, что и у обычного суффиксного автомата. При этом, в сравнении с суффиксным автоматом, использование последней версии персистентного суффиксного автомата будет происходить за такое же время, а для промежуточных версий суффиксного автомата переход из некоторого состояния по некоторому символу может потребовать до $O(\log n)$ дополнительного времени.

2.2. Комбинирование толстых вершин и копирования путей

2.2.1. Подход Слейтера и Тарьяна

Предложенный Слейтером и Тарьяном метод [12] неприменим к суффиксным автоматам. Это связано с недревовидной структурой суффиксного автомата. Однако, в своей статье авторы упоминают вариацию их метода, применимого к графам.

Ключевым моментом является ограничение на число входящих в вершину ребер. Предположим, что ни в какой момент времени вершины графа не имеют более m входящих ребер. В таком случае можно позволить толстым вершинам хранить исходную версию вершины и до m ее модификаций.

Процесс обновления вершины выглядит следующим образом. Если толстая вершина может хранить еще одну модификацию, делается эта модификация. Если же она полностью заполнена, создается ее свежая копия, и входящие в старую вершину ребра перенаправляются во вновь созданную вершину — выполняется действие, аналогичное копированию путей.

Время работы $O(1)$ достигается именно за счет хранения числа модификаций, не меньшего чем число входящих в вершину ребер. При этом, выполнение запросов занимает $O(\log m) = O(1)$ дополнительного времени.

В связи с тем, что состояния суффиксного автомата могут иметь порядка $O(n)$ входящих ребер, описанный метод становится неприменимым для суффиксных автоматов.

2.2.2. Модификация метода

Для того, чтобы затраты на модификации составляли $O(1)$ необходимо, чтобы на момент копирования путей число накопленных в толстой вершине модификаций было равно числу входящих в нее ребер. При отсутствии ограничения на число входящих в вершину ребер требуется поддерживать счетчик числа входящих ребер и производить копирование путей лишь в описанном выше случае.

Такая адаптация метода позволяет по-прежнему производить модификации без каких-либо дополнительных затрат. Однако, не решается проблема с оценкой времени доступа к определенной версии состояния суффиксного автомата. Действительно, если у некоторого состояния имеется порядка n входящих переходов, то для корректной работы алгоритма в нем требуется хранить порядка n версий этого состояния. Это, в свою очередь, приводит к логарифмическому времени поиска требуемой версии.

2.2.3. Вывод

Описанный метод технически намного более сложен, чем метод толстых переходов. При этом особенности структуры суффиксного автомата не позволяют выполнять модификации и запросы без дополнительных логарифмических временных затрат.

Сложность и относительная неэффективность делают этот метод малопригодным для суффиксных автоматов.

Глава 3.

Множество конечных состояний при построении суффиксного автомата

3.1. Анализ алгоритма построения суффиксного автомата

Классический алгоритм постоения суффиксного автомата [12] состоит из двух этапов:

- построение суффиксного автомата со всеми состояниями, переходами, а также с суффиксными ссылками;
- вычисление множества конечных состояний суффиксного автомата, построенного на предыдущем этапе.

Оба этапа занимают $O(n)$ времени. При этом первый этап итеративен: к базовой строке автомата один за одним добавляются символы строки. Каждая такая операция добавления символа занимает амортизированно $O(1)$ времени, а в худшем случае — $O(n)$. Такая операция является единственной операцией по модификации автомата.

В связи с тем, что хочется научить суффиксный автомат хранить свои промежуточные версии, необходимо добиться, чтобы после каждой операций по модификации автомата он оставался в консистентном состоянии. Это означает, что после каждой модификации требуется пересчитывать множество конечых состояний.

К сожалению, если после каждой модификации автомата проходить по суффиксному пути последнего состояния автомата, каждая операция по его модификации будет выполняться за время $O(n)$. Это превратит исходный алгоритм из линейного в квадратичный. Такие временные затраты неприемлемы.

Для решения описанной проблемы можно пойти двумя путями:

- считать каждое состояние автомата конечным. Такое допущение по-прежнему позволяет решать большое число задач;
- научиться эффективно поддерживать множество конечных состояний автомата.

3.2. Поддерживаемые операции

Теперь стоит изучить, какие операции требуются от множества конечных состояний суффиксного автомата.

Прежде всего, необходимо иметь возможность перечислить все конечные состояния. Это можно сделать, всего-навсего пройдя по суффиксному пути из последнего состояния суффиксного автомата. Никакие дополнительные структуры данных для этого не требуются, так как и последнее состояние автомата, и дерево суффиксных ссылок хранятся со всеми своими промежуточными версиями.

Вторая требуемая операция — для заданной вершины определить, является ли она конечной. Существует несколько приемлемых способов решения этой задачи. Ниже приведен вариант, который кажется автору наиболее простым.

3.3. Дерево суффиксных ссылок

Заметим, что суффиксные ссылки суффиксного автомата образуют дерево с корнем в начальном состоянии автомата.

Определение 24. Это дерево называется *деревом суффиксных ссылок* суффиксного автомата.

Проанализируем, что происходит с множеством конечных состояний суффиксного автомата при добавлении к базовой строке одного символа. Множество конечных состояний состоит из состояний, лежащих на суффиксном пути из последнего состояния автомата. Это означает, что состояние суффиксного автомата является конечным тогда и только тогда, когда в дереве суффиксных ссылок оно является предком последнего состояния суффиксного автомата.

Изучим, что происходит с деревом суффиксных ссылок в процессе построения суффиксного автомата.

При добавлении к базовой строке суффиксного автомата одного символа с деревом суффиксных ссылок происходит одна из следующих операций:

- к одной из вершин дерева подвешивается новая вершина, соответствующая (новому) последнему состоянию суффиксного автомата;
- в результате клонирования состояния суффиксного автомата, одно из ребер дерева суффиксных ссылок делится с добавлением новой промежуточной вершины (соответствующей состоянию-клону). Затем к вновь созданной вершине подвешивается еще одна новая вершина (соответствующая последнему состоянию суффиксного автомата).

Все описанные операции с деревом суффиксных ссылок носят локальный характер. Выражаясь более конкретно, все упомянутые операции можно разложить на комбинацию следующих двух операций:

- деление ребра, сопряженное с добавлением новой вершины;
- подвешивание к некоторой вершине дерева новой вершины посредством нового ребра.

3.4. Сведение к задаче RMQ

В настоящей работе предлагается хранить множество конечных состояний суффиксного автомата в неявном виде. При этом задачу определения, является ли состояние конечным, можно свести к задаче наименьшего общего предка (LCA) [7] на дереве суффиксных ссылок. Последняя, в свою очередь, посредством Эйлерова обхода дерева может быть сведена к задаче RMQ по нахождению максимального значения на определенном отрезке массива.

Заметим, что обе операции — деление ребра и подвешивание новой вершины — вследствие специфики Эйлерова обхода, локально отражаются на исходном массиве задачи RMQ. Точнее, в результате выполнения этих операций над массивом совершаются следующие действия:

- вставка в массив нового элемента;
- увеличение значений элементов, принадлежащих некоторому отрезку. Это требуется для поддержки деления ребра, так как при этом увеличивается глубина вершин, лежащих в его поддереве.

Обычно, задачу RMQ решают с помощью дерева отрезков. Оно может быть построено за $O(n)$ и все требуемые нам операции выполняются за $O(\log n)$. Однако, есть одна небольшая проблема при использовании стандартной реализации дерева отрезков. Остается неясным, как осуществлять вставку в массив нового элемента.

Для решения этой проблемы предлагается использовать несколько модифицированный вариант деревьев отрезков. Таким образом, дерево отрезков можно реализовывать не на подобной куче структуре, а на декартовом дереве с рандомизированным вертикальным ключем.

Случайность вертикального ключа придает декартову дереву свойство сбалансированности. Операция же вставки нового элемента может быть легко реализована при помощи стандартной для декартова дерева операции *split*.

Таким образом, для решения поставленной задачи каждая вершина декартова дерева должна хранить следующую информацию:

- случайный вертикальный ключ;
- неявный горизонтальный ключ — число детей в левом поддереве;
- счетчик инкремента значений элементов, находящихся в поддереве — необходим для поддержки операции увеличения значений элементов некоторого отрезка.

3.5. Вывод

В этой главе предложен метод, позволяющий поддерживать множество допускающих состояний суффиксного автомата в процессе его построения. При этом, для этого расходуется $O(\log n)$ дополнительного времени и $O(1)$ дополнительной памяти на каждой символ базовой строки суффиксного автомата.

В связи с тем, что описанный метод в своей основе имеет древовидную структуру данных, используя статью Слейтера и Тарьяна [12] можно без дополнительных затрат хранить промежуточные версии множества допускающих состояний суффиксного автомата, тем самым, сделав упомянутое множество персистентным.

Глава 4.

Приложения персистентных суффиксных автоматов

4.1. Суффиксные автоматы в многопоточных приложениях

В последнее время многоядерные процессоры стали очень распространены. Это делает использование многопоточных приложений более привлекательным по сравнению с их однопоточными аналогами.

Однако, использование неперсистентных структур данных может сильно влиять на производительность. При использовании обычных суффиксных автоматов операция по модификации блокирует все операции на суффиксном автомате. Учитывая, что модификация суффиксного автомата может занимать $O(n)$ времени, могут возникать достаточно длительный паузы в обработке запросов.

Использование персистентных суффиксных автоматов позволяет свести число блокировок к минимуму. Следовательно, операции по модификации суффиксного автомата будут блокировать лишь другие операции по модификациям, никак при этом не влияя на выполнение запросов.

4.2. Поддержка суффиксным автоматом операции удаления последнего символа

4.2.1. Постановка задачи

В оригинальном изложении [6] суффиксный автомат как структура данных поддерживает лишь одну операцию модификации — добавление справа к базовой строке одного символа.

Также имеется возможность амортизировано за время $O(1)$ добавлять влево к базовой строке символ. Это сделается с помощью алгоритма Укконена и следующего факта.

Предложение 16. Дерево суффиксных ссылок суффиксного автомата строки s является суффиксным деревом инверсированной строки.

Этот факт позволяет свести процедуру приписывание одного символа слева к базовой строке суффиксного автомата к процедуре приписывания этого символа справа к базовой строке суффиксного дерева. Последнее, в свою очередь, может быть достигнуто применением алгоритма Укконена.

Однако, это отнюдь не является поводом использовать в этих задачах персистентный суффиксный автомат. Свойство персистентности позволяет выполнять операцию удаления символа базовой строки суффиксного автомата амортизированно за время $O(1)$.

4.2.2. Решение 1

Предположим, что имеется персистентный суффиксный автомат для строки s и требуется удалить ее последний символ и перестроить автомат соответствующим образом. Заметим, что можно не выполняя каких-либо дополнительных действий, извлечь суффиксный автомат версии $|s| - 1$ и работать с ним. Это позволит выполнять над ним операции с временными затратами равными не $O(\log n)$, а $O(1)$, так как отсутствует необходимость искать требуемую версию бинарным поиском. Для этого достаточно посмотреть на пару последних версий толстого перехода.

Однако, после удаления большого числа символов временные затраты на доступ к переходам нужной версии могут заметно увеличиться. Для того, чтобы этого не происходило, предлагается хранить в автомате текущую версию и при поиске перехода требуемой версии удалять все более старые переходы.

Такой подход обладает следующими преимуществами:

- время работы операции удаления символа равно $O(1)$;
- время поиска требуемого перехода остается $O(1)$;
- не требуется абсолютно никаких дополнительных действий для подготовки автомата для дальнейших модификаций. При последующих

добавлениях символов в процессе извлечения требуемой версии перехода автоматически удалятся все лишние версии.

У предложенного метода имеются следующие недостатки:

- не освобождается используемая автоматом память до момента обращения к состояниям;
- несмотря на константность времени поиска требуемого перехода, оно лишь асимптотически $O(1)$.

4.2.3. Решение 2

При построении суффиксного автомата можно хранить множества объектов, измененных в каждой версии. Это позволяет при удалении символа игнорировать все изменения в последней версии переходов и состояний.

Описанный подход обладает лишь одним преимуществом по сравнению с предыдущим решением: время поиска требуемого перехода становится полностью константным (без амортизационного анализа). Однако, операция удаления символа занимает $O(1)$ времени лишь амортизировано.

Это может оказаться полезным в задачах, где модификации суффиксного автомата случаются достаточно редко, но при этом предъявляются жесткие требования к запросам над суффиксным автоматом.

4.2.4. Вывод

Использование персистентного суффиксного автомата позволяет добавить в суффиксный автомат поддержку операции удаления последнего символа базовой строки.

Заключение

В работе выполнен анализ существующих методов построения персистентных структур данных, а также разработан эффективный метод построения персистентного суффиксного автомата.

При использовании предлагаемого метода доступ к прежним версиям суффиксного автомата в большинстве случаев не требует значительных дополнительных временных затрат, однако, для некоторых видов строк время выполнения запросов может увеличиться в $O(\log n)$ раз.

Персистентность суффиксных автоматов позволяет более эффективно использовать их в многопоточных приложениях: добавление символов к базовой строке суффиксного автомата не вызывает блокировки запросов. Это является достоинством, так как операции по модификации суффиксного автомата могут занимать до $O(n)$ времени.

Предложен метод, позволяющий поддерживать множество допускающих состояний суффиксного автомата в процессе его построения. При этом требуется $O(\log n)$ дополнительного времени и $O(1)$ дополнительной памяти на каждой символ базовой строки суффиксного автомата.

В связи с тем, что описанный метод основывается на использовании древовидной структуры данных, можно без дополнительных затрат хранить промежуточные версии множества допускающих состояний суффиксного автомата, тем самым, сделав упомянутое множество персистентным.

Список литературы

1. *Karger D.* Persistent Data Structures. Advanced Algorithms: Lecture 2. September 9, 2005.

2. *Bentley J. L., Saxe J. B.* Decomposable searching problems I: Static-to-dynamic transformations // J. Algorithms. 1980. Vol 1, pp. 301-358.

3. *Chazelle B.* Filtering search: A new approach to query-answering // SIAM J. Comput. 1986. Vol. 15, pp. 703-724.

4. *Chazelle B.* How to search in history // Inform. and Control. 1985. Vol 77, pp. 77-99.

5. *Гасфилд Д.* Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. СПб.: Невский диалект; БХВ-Петербург, 2003.

6. *Lothaire M.* Applied Combinatorics on Words // Encyclopedia of Mathematics and its Applications, 2005. Vol. 90. Cambridge University Press, Cambridge.

7. *Bender M., Farach-Colton M.* The LCA Problem Revisited / LATIN 2000, pp. 88-94.

8. *Хопкрофт Дж., Мортвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.

9. *Кормен Т., Лейзерсон Ч., Ривест Л.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.

10. *Sartaj Sahni Dr.* Data Structures, Algorithms, & Applications in Java. Suffix Trees. CISE Department Chair at University of Florida http://www.cise.ufl.edu/s̃ahni/dsaaj/enrich/c16/suffix.htm

11. *Паращенко Д.* Обработка строк на основе суффиксных автоматов. Бакалаврская работа. СПбГУ ИТМО. 2007.

12. *Driscoll J. R., Sarnak N., Sleator D. D., Tarjan R. E.* Making Data Structures Persistent // Journal of Computer and System Sciences. 1989. Vol. 3. No 1, pp. 86-124.

13. *Edelkamp S.* Suffix tree // Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 2007. http://www.nist.gov/dads/HTML/suffixtree.html

14. *Blumer A., Blumer J., Ehrenfeucht A., Hausler D., McConnel R.* Linear size finite automata for the set of all subwords of a word: an outline of results // Bull. Eur. Assoc. Theoret. Commput. Sci. 1983. Vol 21, pp. 12-20.

15. *Blumer A., Blumer J., Ehrenfeucht A., Hausler D., McConnel R.* The smallest automaton recognizing the subwords of a text // Bull. Eur. Assoc. Theoret. Commput. Sci. 1985. Vol 40(1), pp. 31-55.

16. *Eilenberg S.* Automata, Languages, and Machines // Academic Press. 1974. Vol A.

17. *Kuich W., Salomaa A.* Semirings, Automata, Languages. Springer-Verlag. 1986.

18. *Bluma N., Mehlhorn K.* On the average number of rebalancing operations in weight-balanced trees // Theoret. Comput. Sci. 1980. Vol 11, pp. 303-320.

19. *Browna M. R., Tarjan R. E.* Design and analysis of a data structure for representing sorted lists // SIAM J. Comput. 1980. Vol 9, pp. 594-614.

20. *Bayer R., McCreight E.* Organization of large ordered indexes // Acta Inform. 1972. Vol 1, pp. 173-189.

21. *Tarjan R. E.* Updating a balanced search tree in 0(1) rotations // Inform. Process. 1983. Lett. 16, pp. 253-257.

Приложение. Исходный код персистентного суффиксного автомата и программа по оценке его производительности

```
package ru.ifmo.paraschenko.masters;

import java.util.List;

/**
 * @author Dmitry Paraschenko
 */
public interface PersistentGraph {
    public int createNode();
    public int getEdge(int source, char ch);
    public List<Integer> getEdges(int source);
    public int addEdge(int souce, char ch, int target
      );
    public int getTarget(int edge);
    public void setTarget(int edge, int target);
    public char getChar(int edge);
    public int getMods(int edge);
    public int getMaxMods();
    public int getNodeCount();
    public int getEdgeCount();
    public int cloneEdge(int state, int edge);
}
```

```
package ru.ifmo.paraschenko.masters;

import ru.ifmo.paraschenko.LargeInputSizeException;

import java.util.ArrayList;

/**
 * @author Dmitry Paraschenko
 */
public class PersistentSuffixAutomaton extends
  PersistentGraphAdapter {
    public static final char SUFFIX = 0;
    private ArrayList<Integer> ln;
    private int lastNode;
    private int length;

    public PersistentSuffixAutomaton(PersistentGraph
      graph) throws LargeInputSizeException {
        super(graph);
        ln = new ArrayList<Integer>();
        lastNode = createNode(0);
        length = 0;
    }

    public PersistentSuffixAutomaton(PersistentGraph
      graph, String string) throws
      LargeInputSizeException {
        this(graph);
```

```
        if (string.length() > 100000) {
            throw new LargeInputSizeException();
        }
        for (char ch: string.toCharArray()) {
            addCharacter(ch);
        }
    }

    private int createNode(int log) {
        int node = createNode();
        addEdge(node, SUFFIX, -1);
        ln.add(log);
        return node;
    }

    private int getLn(int p) {
        return ln.get(p);
    }

    protected void addCharacter(char ch) {
        length++;
        int p = lastNode;
        lastNode = createNode(getLn(p) + 1);
        while (p >= 0 && getEdge(p, ch) == -1) {
            addEdge(p, ch, lastNode);
            p = getSuffix(p);
        }
        if (p < 0) {
```

```
            setSuffix(lastNode, 0);
            return;
        }
        int q = getTarget(getEdge(p, ch));
        if (getLn(p) + 1 == getLn(q)) {
            setSuffix(lastNode, q);
        } else {
            int r = createNode(getLn(p) + 1);
            for (int edge : getEdges(q)) {
                cloneEdge(r, edge);
            }
            setSuffix(r, getSuffix(q));
            setSuffix(q, r);
            for (int edge = getEdge(p, ch); p
                >= 0 && edge >= 0 && getTarget(edge)
                == q; p = getSuffix(p), edge = p
                < 0 ? -1 : getEdge(p, ch)) {
                setTarget(edge, r);
            }
            setSuffix(lastNode, r);
        }
    }

    public int getSuffix(int source) {
        return getTarget(getEdge(source, SUFFIX));
    }

    public void setSuffix(int source, int target) {
```

```
            setTarget(getEdge(source, SUFFIX), target);
        }

        public int length() {
            return length;
        }
    }

    package ru.ifmo.paraschenko.masters;

    import java.util.HashMap;
    import java.util.List;
    import java.util.ArrayList;

    /**
     * @author Dmitry Paraschenko
     */
    public class SuffixAutomatonHelper {
        public static boolean containsSubstring(
            PersistentGraph auto, String str) {
            int node = 0;
            for (char ch : str.toCharArray()) {
                int edge = auto.getEdge(node, ch);
                if (edge < 0) {
                    return false;
                }
                node = auto.getTarget(edge);
            }
            return true;
```

```
}

public static Result getMaxLnSum(
  PersistentSuffixAutomaton auto) {
    HashMap<Integer, Result> map = new HashMap<
      Integer, Result>();
    return getMaxLnSum(auto, map, 0);
}

private static Result getMaxLnSum(
  PersistentSuffixAutomaton auto, HashMap<Integer
  , Result> map, int v) {
    if (map.get(v) != null) {
        return map.get(v);
    }
    Result max = null;
    for (int edge : auto.getEdges(v)) {
        if (auto.getChar(edge) ==
          PersistentSuffixAutomaton.SUFFIX) {
            continue;
        }
        Result cost = new Result(auto.getMods(
          edge), getMaxLnSum(auto, map, auto.
          getTarget(edge)));
        if (max == null || max.compareTo(cost)
           < 0) {
            max = cost;
        }
```

```
        }
        map.put(v, max);
        return max;
    }

    public static class Result implements Comparable<
      Result>{
        public double cost;
        public ArrayList<Integer> list;
        public String string;

        public Result(int mods, Result res) {
            list = new ArrayList<Integer>();
            list.add(mods);
            cost = mods > 3 ? Math.log(mods) / Math.
              log(2) : 0;
            if (res != null) {
                list.addAll(res.list);
                cost += res.cost;
            }
        }

        public int compareTo(Result that) {
            return Double.compare(this.cost, that.
              cost);
        }

        public String toString() {
```

```
            return cost + " " + list + " " + string;
        }
    }
}

package ru.ifmo.paraschenko.masters;

import java.util.List;

/**
 * @author Dmitry Paraschenko
 */
public class PersistentGraphAdapter implements
  PersistentGraph {
    private PersistentGraph graph;

    public PersistentGraphAdapter(PersistentGraph
      graph) {
        this.graph = graph;
    }

    public int getEdge(int source, char ch) {
        return graph.getEdge(source, ch);
    }

    public int addEdge(int source, char ch, int
      target) {
        return graph.addEdge(source, ch, target);
    }
```

```
public int createNode() {
    return graph.createNode();
}

public int getTarget(int edge) {
    return graph.getTarget(edge);
}

public void setTarget(int edge, int target) {
    graph.setTarget(edge, target);
}

public List<Integer> getEdges(int source) {
    return graph.getEdges(source);
}

public char getChar(int edge) {
    return graph.getChar(edge);
}

public int getMods(int edge) {
    return graph.getMods(edge);
}

public int getMaxMods() {
    return graph.getMaxMods();
}
```

```
    public int getNodeCount() {
        return graph.getNodeCount();
    }

    public int getEdgeCount() {
        return graph.getEdgeCount();
    }

    public int cloneEdge(int state, int edge) {
        return graph.cloneEdge(state, edge);
    }
}
```

```
package ru.ifmo.paraschenko.masters;

import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

/**
 * @author Dmitry Paraschenko
 */
public class FatNodesPersistentGraph {
    private int nodeCount;
    private int edgeCount;
    private List<List<Integer>> edges;
    private List<Character> chars;
    private List<Integer> targets;
```

```
public FatNodesPersistentGraph() {
    edges = new ArrayList<List<Integer>>();
    chars = new ArrayList<Character>();
    targets = new ArrayList<Integer>();
}

public int createNode() {
    edges.add(new ArrayList<Integer>());
    return nodeCount++;
}

public int getEdge(int source, char ch) {
    for (int edge : getEdges(source)) {
        if (getChar(edge) == ch) {
            return edge;
        }
    }
    return -1;
}

public List<Integer> getEdges(int source) {
    return Collections.unmodifiableList(edges.get
      (source));
}

public void addEdge(int souce, char ch, int
  target) {
```

```
            int edge = edgeCount++;
            chars.add(ch);
            targets.add(target);
            edges.get(souce).add(edge);
        }

        public int getTarget(int edge) {
            return targets.get(edge);
        }

        public void setTarget(int edge, int target) {
            targets.set(edge, target);
        }

        public char getChar(int edge) {
            return chars.get(edge);
        }
    }
```

```
package ru.ifmo.paraschenko.masters;

import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

/**
 * @author Dmitry Paraschenko
 */
```

```
public class DefaultGraph implements PersistentGraph
    {
     private int nodeCount;
     private int edgeCount;
     private List<List<Integer>> edges;
     private List<Character> chars;
     private List<Integer> targets;
     private List<Integer> mods;
     private int max_mods;

     public DefaultGraph() {
         edges = new ArrayList<List<Integer>>();
         chars = new ArrayList<Character>();
         targets = new ArrayList<Integer>();
         mods = new ArrayList<Integer>();
     }

     public int createNode() {
         edges.add(new ArrayList<Integer>());
         return nodeCount++;
     }

     public int getEdge(int source, char ch) {
         for (int edge : getEdges(source)) {
             if (getChar(edge) == ch) {
                 return edge;
             }
         }
```

```
        return -1;
    }

    public List<Integer> getEdges(int source) {
        return Collections.unmodifiableList(edges.get
          (source));
    }

    public int addEdge(int source, char ch, int
      target) {
        int edge = edgeCount++;
        chars.add(ch);
        targets.add(target);
        edges.get(source).add(edge);
        mods.add(1);
        max_mods = Math.max(max_mods, 1);
        return edge;
    }

    public int getTarget(int edge) {
        return targets.get(edge);
    }

    public void setTarget(int edge, int target) {
        targets.set(edge, target);
        int new_mods = mods.get(edge) + 1;
        mods.set(edge, new_mods);
        max_mods = Math.max(max_mods, new_mods);
```

```
    }

    public char getChar(int edge) {
        return chars.get(edge);
    }

    public int getMods(int edge) {
        return mods.get(edge);
    }

    public int getMaxMods() {
        return max_mods;
    }

    public int getNodeCount() {
        return nodeCount;
    }

    public int getEdgeCount() {
        return edgeCount;
    }

    public int cloneEdge(int state, int edge) {
        int new_edge = addEdge(state, getChar(edge),
          getTarget(edge));
        return new_edge;
    }
}
```

```
package ru.ifmo.paraschenko.masters;

import junit.framework.TestCase;
import ru.ifmo.paraschenko.LargeInputSizeException;

import java.util.List;
import java.util.Arrays;
import java.util.Random;
import java.util.ArrayList;

/**
 * @author Dmitry Paraschenko
 */
public class GraphAnalysis extends TestCase {
    private static final String[] STRINGS = {
        "aaa",
        "aab",
        "bab",
        "bbb",
        "aaaa",
        "aabb",
        "abba",
        "aaaaa",
        "aabaa",
        "aabab",
        "aabba",
        "abbaa",
        "abbab",
```

```
    "abbba",
    "abbbb",
};

private Random random = new Random(20090523);
private List<String> strings;


public void testLinearSuffixAutomaton() throws
  LargeInputSizeException {
    for (int len = 1; len <= 20; len++) {
        System.out.println(len + ": " +
          testString("", len));
    }
}

private SuffixAutomatonHelper.Result testString(
  String string, int len) throws
  LargeInputSizeException {
    if (len > 0) {
        SuffixAutomatonHelper.Result res1 =
          testString(string + "a", len - 1);
        SuffixAutomatonHelper.Result res2 =
          testString(string + "b", len - 1);
        if (res1.compareTo(res2) >= 0) {
            return res1;
        } else {
            return res2;
```

```
            }
        } else {
            PersistentSuffixAutomaton auto = new
              PersistentSuffixAutomaton(new
              DefaultGraph(), string);
            SuffixAutomatonHelper.Result res =
              SuffixAutomatonHelper.getMaxLnSum(auto)
              ;
            res.string = string;
            return res;
        }
    }

    public void singleFactoryTest() throws
      LargeInputSizeException {
        for (String str : getStrings()) {
            testString(str);
        }
    }

    private void testString(String string) throws
      LargeInputSizeException {
        PersistentSuffixAutomaton auto = new
          PersistentSuffixAutomaton(new DefaultGraph
          (), string);
        System.out.println(SuffixAutomatonHelper.
          getMaxLnSum(auto));
    }
```

```
    private List<String> getStrings() {
        if (strings != null) {
            return strings;
        }
        strings = new ArrayList<String>(Arrays.asList
          (STRINGS));
        for (int len = 1; len < 100; len++) {
            for (int alpha = 2; alpha <= 4; alpha++)
               {
                strings.add(randomString(len, alpha))
                   ;
            }
        }
        return strings;
    }

    private String randomString(int len, int alpha) {
        StringBuilder sb = new StringBuilder();
        for (; len-- > 0;) {
            sb.append((char) (random.nextInt(alpha)
               + 'a'));
        }
        return sb.toString();
    }
}
```

```
package ru.ifmo.paraschenko;
```

```
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;

/**
 * @author Dmitry Paraschenko
 */
public class Test {
    public static void main(String[] args) throws
      LargeInputSizeException {
        test1(15);
    }

    private static void test3(int maxLen) throws
      LargeInputSizeException {
        int base = 2;
        for (int len = 1; len <= maxLen; len++) {
            ArrayList<String> best = new ArrayList<
              String>();
            double max = 0;
            for (int mask = 0; mask < Math.pow(base,
              len); mask++) {
                String str = generateString(len, mask
                  , base);
                double cnt = testDouble(str);
                if (cnt == max) {
                    best.add(str);
                } else if (cnt > max) {
```

```
                    max = cnt;
                    best.clear();
                    best.add(str);
                }
            }
            System.out.println("results for length="
                + len + " (count=" + max + "): " +
                best);
        }
    }

    private static double testDouble(String str)
        throws LargeInputSizeException {
        PersistentLinearSuffixAutomaton automaton =
            new PersistentLinearSuffixAutomaton(str);
        Set<String> set = new HashSet<String>();
        double max = 0;
        String maxs = "";
        for (int left = 0; left < str.length(); left
            ++) {
            for (int right = left + 1; right <= str.
                length(); right++) {
                String substr = str.substring(left,
                    right);
                if (set.contains(substr)) {
                    continue;
                }
                set.add(substr);
```

```
                double tmp = automaton.
                  getDoubleStatistics(substr);
                if (max < tmp) {
                    max = tmp;
                    maxs = substr;
                }
            }
        }
        return max;
    }

    private static String buildWorstString(int mod) {
        String str = "";
        for (int cnt = mod; cnt >= 0; cnt--) {
            for (int i = 0; i < cnt; i++) {
                str += "a";
            }
            str += "b";
        }
        return str;
    }

    private static void test2(int maxLen) throws
      LargeInputSizeException {
        for (int len = 1; len <= maxLen; len++) {
            ArrayList<String> best = new ArrayList<
              String>();
            int max = 0;
```

```
            for (int mask = 0; mask < Math.pow(3, len
              ); mask++) {
                String str = generateString(len, mask
                  , 3);
                int cnt = testInt(str);
                if (cnt == max) {
                    best.add(str);
                } else if (cnt > max) {
                    max = cnt;
                    best.clear();
                    best.add(str);
                }
            }
            System.out.println("results for length="
               + len + " (count=" + max + "):");
            for (String str : best) {
                System.out.println(str + " : " +
                  testString(str));
            }
        }
    }

    private static void test1(int maxLen) throws
      LargeInputSizeException {
        for (int len = 1; len <= maxLen; len++) {
            String maxs = "0";
            int max = 0;
            String str = null;
```

```
            for (int mask = 0; mask < 1 << len; mask
              ++) {
                String ts = generateString(len, mask
                  , 2);
                String tmp = testString(ts);
                if (maxs.compareTo(tmp) < 0) {
                    maxs = tmp;
                }
                int tmpi = testInt(ts);
                if (max < tmpi) {
                    max = tmpi;
                    str = ts;
                }
            }
            System.out.println(len + ": " + max + ":
              " + maxs + " " + str);
        }
    }

    private static String testString(String str)
      throws LargeInputSizeException {
        PersistentLinearSuffixAutomaton automaton =
          new PersistentLinearSuffixAutomaton(str);
        return automaton.getCharacteristics();
    }

    private static int testInt(String str) throws
      LargeInputSizeException {
```

```
        PersistentLinearSuffixAutomaton automaton =
          new PersistentLinearSuffixAutomaton(str);
        return automaton.getMaxModified();
    }

    private static String generateString(int len, int
        mask, int base) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < len; i++) {
            sb.append((char) ('a' + (mask % base)));
            mask /= base;
        }
        return sb.reverse().toString();
    }
}
```