```
    if P(v + 1) ≠ P(i + 1) then
      sp'_i := v
    else
      sp'_i := sp_v;
  end;
```

**Теорема 3.3.2.** *Алгоритм $SP'(P)$ корректно вычисляет все значения $sp'_i$ за время $O(n)$.*

Доказательство. Доказательство ведется индукцией по $i$. Так как $sp_1 = 0$ и $sp'_i \leqslant sp_i$ для всех $i$, то $sp'_1 = 0$, и алгоритм для $i = 1$ корректен. Теперь предположим, что значение $sp'_i$, найденное алгоритмом, корректно для всех $i < k$, и рассмотрим $i = k$. Если $P(sp_k + 1) \neq P(k + 1)$, то очевидно, что $sp'_k = sp_k$, так как префикс $P[1..k]$ длины $sp_k$ удовлетворяет всем требованиям. Следовательно, в этом случае алгоритм находит $sp'_k$ правильно.

Если $P(sp_k + 1) = P(k + 1)$, тогда $sp'_k < sp_k$, и, поскольку $P[1..sp_k]$ — суффикс $P[1..k]$, то $sp'_k$ может быть истолковано как длина наибольшего собственного префикса $P[1..sp_k]$, который совпадает с суффиксом $P[1..sp_k]$, при том, что $P(k + 1) \neq P(sp'_k + 1)$. Но так как $P(k + 1) = P(sp_k + 1)$, условие можно переписать как $P(sp_k + 1) \neq P(sp'_k + 1)$. По индукционному предположению упомянутая длина уже вычислена как $sp_{sp_k}$. Итак, при $P(sp_k + 1) = P(k + 1)$ алгоритм правильно полагает $sp'_k$ равным $sp_{sp_k}$.

Поскольку для каждой позиции трудоемкость алгоритма константная, полное время его работы имеет порядок $O(n)$. □

Интересно сравнить этот классический метод вычисления $sp$ и $sp'$ с методом, использующим основной препроцессинг (т.е. $Z$-значения). В классическом методе сначала вычисляются (слабые) значения $sp$, а затем из них получаются искомые значения $sp'$, тогда как в основном препроцессинге порядок прямо противоположен.

## 3.4. Точный поиск набора образцов

Рассмотрим непосредственное (и важное) обобщение задачи точного поиска на случай множества образцов $\mathscr{P} = \{P_1, P_2, \ldots, P_z\}$, когда требуется обнаружить все вхождения в текст $T$ любого образца из $\mathscr{P}$. Это обобщение называется *множественной задачей точного поиска* (the exact set matching problem). Здесь $n$ будет обозначать суммарную длину всех образцов в $\mathscr{P}$, а $m$, как и ранее, длину $T$. Тогда множественную задачу поиска можно решить за время $O(n + zm)$, применяя любой метод с линейным временем отдельно для каждого из $z$ образцов.

Однако эту задачу можно решить быстрее, чем за $O(n + zm)$, а именно за время $O(n + m + k)$, где $k$ — число вхождений в $T$ образцов из $\mathscr{P}$. Первый метод с такой границей был предложен Ахо и Корасиком [9] *). В этом параграфе мы представим алгоритм Ахо–Корасика; некоторые из доказательств будут оставлены читателю. Столь же эффективный, но более робастный метод для множественной задачи поиска основывается на суффиксных деревьях, он будет рассмотрен в п. 7.2.

*) Имеется более свежее изложение метода Ахо Корасика в [8], где алгоритм используется точно как "приемник", решающий, обнаружилось или нет вхождение в $T$ по меньшей мере одного образца из $\mathscr{P}$. Так как нам захочется явно найти все вхождения, эта версия алгоритма слишком ограничена для ее

**Определение.** *Деревом ключей* (keyword tree) для множества $\mathcal{P}$ называется ориентированное дерево с корнем $\mathcal{K}$, удовлетворяющее трем условиям: 1) каждая дуга помечена ровно одним символом; 2) любые две дуги, выходящие из одной и той же вершины, имеют разные пометки; и 3) каждый образец $P_i$ в $\mathcal{P}$ отображается в некоторую вершину $v$ из $\mathcal{K}$, такую что символы на пути из корня $\mathcal{K}$ в $v$ в точности составляют $P_i$, и каждый лист из $\mathcal{K}$ соответствует какому-либо образцу из $\mathcal{P}$.

Например, на рис. 3.13 показано дерево ключей для множества образцов {*potato, poetry, pottery, science, school*}.

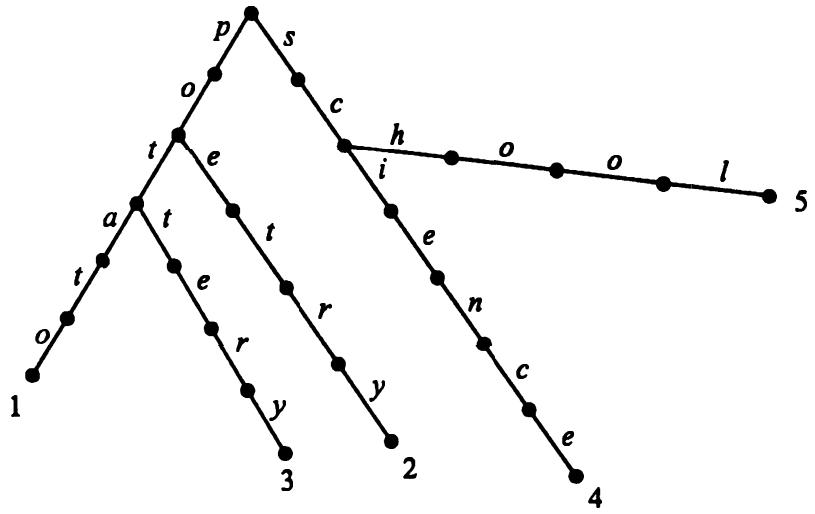


**Рис. 3.13.** Дерево ключей $\mathcal{K}$ с пятью образцами

Ясно, что каждая вершина в дереве ключей соответствует префиксу какого-либо образца из $\mathcal{P}$ и разные префиксы одного образца отображаются в разные вершины дерева.

В предположении, что размер алфавита конечен, легко построить дерево ключей для $\mathcal{P}$ за время $O(n)$. Определим $\mathcal{K}_i$ как (частичное) дерево ключей, кодирующее образцы $P_1, \ldots, P_i$ из набора $\mathcal{P}$. Дерево $\mathcal{K}_1$ состоит из простого пути с $|P_1|$ дугами, начинающегося в корне. Каждая дуга этого пути помечена символом из $P_1$, и, если читать от корня, эти символы составят строку $P_1$. Около конечной вершины этого пути написано число 1. Чтобы сделать $\mathcal{K}_2$ из $\mathcal{K}_1$, найдем самый длинный путь из $r$, в котором символы совпадают по порядку с $P_2$. То есть найдем наибольший префикс $P_2$, который совпадает с символами какого-либо пути, идущего из корня. Этот путь кончается либо исчерпанием $P_2$, либо тем, что в какой-нибудь вершине дерева $v$ совпадение текста пути и образца заканчивается. В первом случае $P_2$ уже есть в дереве, и мы пишем номер 2 в вершине, где путь кончается. Во втором случае мы создаем первый путь, ведущий из $v$, помеченный оставшимися (несовпадающими) символами из $P_2$, и пишем число 2 в конце этого пути. Пример этих двух возможностей показан на рис. 3.14.

В любом из этих двух случаев в $\mathcal{K}_2$ прибавится не больше одной ветвящейся вершины (вершины, у которой детей больше одного), и символы на двух дугах,

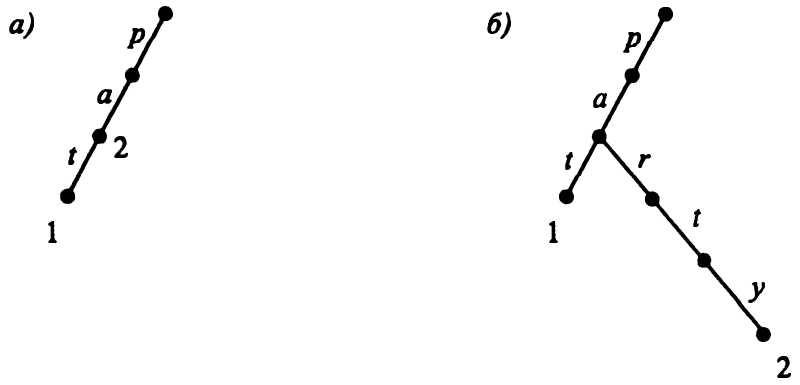


**Рис. 3.14.** Образец $P_1$ — это строка $pat$;
*а* — включение образца $P_2 = pa$; *б* — включение, когда $P_2 = party$

выходящих из этой ветвящейся вершины, будут различны. Мы увидим, что последнее свойство для любого дерева $\mathcal{K}_i$ выполняется индуктивно. В том смысле, что в любой ветвящейся вершине $v$ дерева $\mathcal{K}_i$ все дуги, выходящие из $v$, имеют разные метки.

В общем, для создания $\mathcal{K}_{i+1}$ из $\mathcal{K}_i$ нужно стартовать из корня $\mathcal{K}_i$ и двигаться так далеко, как возможно, по тому (единственному) пути в $\mathcal{K}_i$, на котором метки совпадают по порядку с символами из $P_{i+1}$. Этот путь единственен потому, что в каждой ветвящейся вершине $v$ из $\mathcal{K}_i$ все символы выходящих дуг различны. Если образец $P_{i+1}$ исчерпался (полностью совпал), то вершине, где совпадение закончилось, приписывается номер $i + 1$. Если процесс пришел в вершину $v$, где дальнейшее совпадение невозможно и для очередного символа $P_{i+1}$ подходящей дуги не нашлось, то создается новый путь из $v$, помечаемый оставшейся частью $P_{i+1}$, и конечная вершина этого пути получает номер $i + 1$.

Во время включения в дерево образца $P_{i+1}$ работа для каждой вершины ограничивается константой, так как алфавит конечен и никакие две дуги, выходящие из вершины, не помечены одним и тем же символом. Следовательно, для любого $i$ включение образца $P_{i+1}$ в $\mathcal{K}_i$ занимает время $O(|P_{i+1}|)$, а создание всего дерева ключей — время $O(n)$.

### 3.4.1. Наивное использование дерева ключей для множественного поиска

Поскольку никакие две дуги, выходящие из одной вершины, не помечены одинаковым символом, дерево ключей можно использовать для поиска всех вхождений в $T$ образцов из $\mathcal{P}$. Для начала посмотрим, как искать вхождения образцов из $\mathcal{P}$, которые начинаются с первого символа строки $T$. Нужно следовать единственному пути в $\mathcal{K}$, который совпадает с префиксом $T$, так долго, как возможно. Если на пути найдется вершина, имеющая номер $i$, это значит, что строка $P_i$ входит в $T$, начиная с позиции 1. Больше одной нумерованной вершины можно встретить при условии, что некоторые образцы из $\mathcal{P}$ являются префиксами других образцов из $\mathcal{P}$.

В общем случае, чтобы найти все образцы, которые входят в $T$, мы стартуем с каждой позиции $l$ строки $T$ и идем единственным путем из корня $r$ дерева $\mathcal{K}$, который совпадает с подстрокой $T$, начинающейся с символа позиции $l$. Нумерованные вершины вдоль пути отмечают образцы из $P$, которые входят в $T$, начиная

с позиции $l$. Для фиксированного $l$ прокладка пути в $\mathcal{K}$ требует времени, пропорционального минимуму из $m$ и $n$, так что последовательно наращивая $l$ от 1 до $m$ и проходя через $\mathcal{K}$ для каждого $l$, можно решить точную множественную задачу поиска за время $O(nm)$. Ниже мы уменьшим это время до $O(n + m + k)$, где $k$ — число вхождений.

**Задача о словаре**

Этот простой алгоритм построения дерева ключей без каких-либо дальнейших улучшений эффективно решает специальный случай задачи множественного поиска, называемый *задачей о словаре*. В ней изначально известно и подготовлено множество строк (образующих словарь). Потом появляется последовательность отдельных строк; для каждой из них требуется определить, содержится ли она в словаре. В этом контексте вполне очевидно удобство дерева ключей. Строки словаря закодированы в дереве ключей $\mathcal{K}$, и, когда появляется отдельная строка, проход от корня в дереве $\mathcal{K}$ определяет, есть ли указанная строка в словаре. В этом специальном случае точного множественного поиска необходимо дать ответ, совпадает ли весь текст $T$ (отдельная строка) с какой-нибудь строкой в $\mathcal{P}$.

Вернемся теперь к общей проблеме множественного поиска, где устанавливается, какие строки из $\mathcal{P}$ *содержатся* в тексте $T$.

## 3.4.2. Ускорение: обобщенный метод Кнута–Морриса–Пратта

Изложенный наивный подход к точной задаче множественного поиска аналогичен наивному поиску, который мы обсуждали до того, как ввели метод Кнута–Морриса–Пратта. Последовательное наращивание $l$ на единицу и старт каждый раз от корня $r$ аналогичен наивному точному поиску для отдельного образца, где после каждого несовпадения образец сдвигается только на одну позицию, и сравнения всегда начинаются с левого конца образца. Алгоритм Кнута–Морриса–Пратта улучшает этот наивный алгоритм сдвигом образца больше чем на одну позицию, когда это возможно, и отказом от сравнения символов слева от текущего символа в $T$. Алгоритм Ахо–Корасика использует улучшения того же типа, наращивая $l$ больше чем на единицу и перепрыгивая через начальные части путей в $\mathcal{K}$, когда это возможно. Ключевая идея — такое обобщение функции $sp_i$ (определенной на с. 47 для отдельного образца), которое позволит оперировать с набором образцов. Имеется в виду непосредственное обобщение с одной только тонкостью, которая возникает, если какой-то образец в $\mathcal{P}$ является собственной подстрокой другого образца в $\mathcal{P}$. Поэтому полезно (временно) сделать следующее предположение:

> **Предположение о подстроках.** Никакой образец в $\mathcal{P}$ не является собственной подстрокой любого другого образца в $\mathcal{P}$.

## 3.4.3. Функции неудач для дерева ключей

> **Определение.** Каждая вершина $v$ в $\mathcal{K}$ *помечена* строкой, полученной конкатенацией символов на пути от корня $\mathcal{K}$ до вершины $v$ в порядке их появления. Для этой пометки используется обозначение $\mathcal{L}(v)$. Так что конкатенация символов пути от корня до $v$ произносит строку $\mathcal{L}(v)$.

Например, на рис. 3.15 вершина, на которую указывает стрелка, помечена строкой *pott*.

**Определение.** Для любой вершины $v$ дерева $\mathcal{K}$ определим $lp(v)$ как длину наибольшего собственного суффикса строки $\mathcal{L}(v)$, которая является префиксом некоторого образца из $\mathcal{P}$.

Например, рассмотрим набор образцов $\mathcal{P} = \{potato, tattoo, theater, other\}$ и его дерево ключей, показанное на рис. 3.16. Пусть $v$ — вершина, помеченная

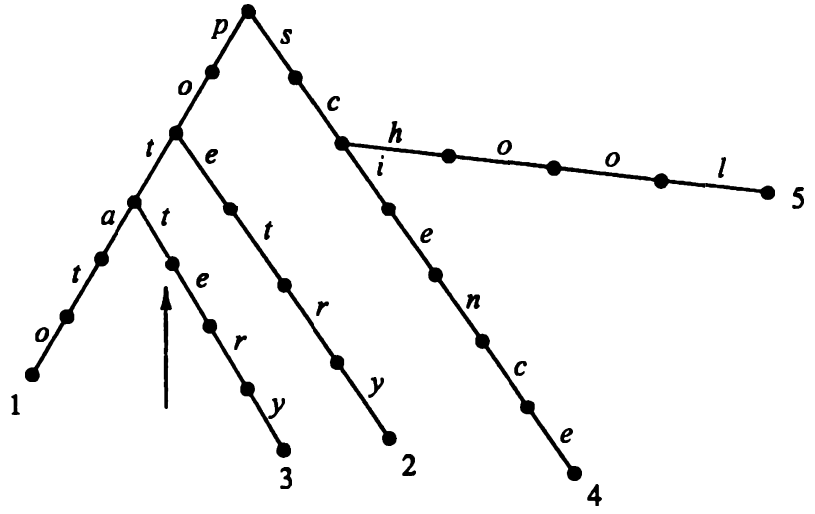


**Рис. 3.15.** Дерево ключей, показывающее пометку узлов

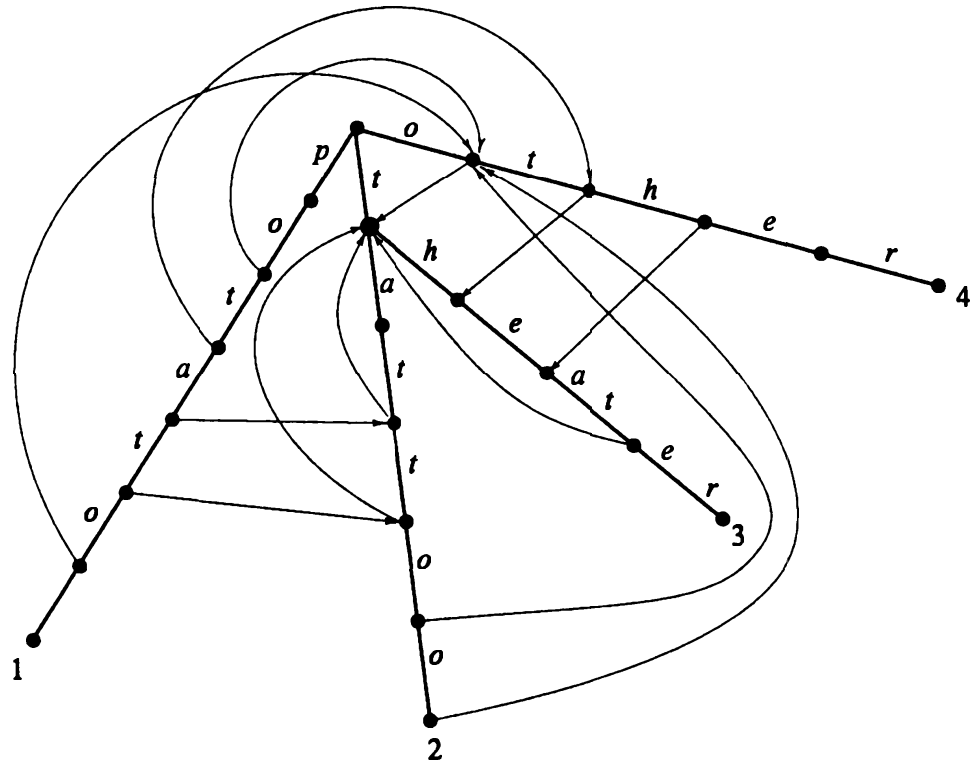


**Рис. 3.16.** Дерево ключей, показывающее связи неудач

строкой *potat*. Так как *tat* — это префикс *tattoo* и наибольший собственный суффикс *potat*, который является префиксом какого-либо образца в $\mathscr{P}$, то $lp(v) = 3$.

**Лемма 3.4.1.** *Пусть $\alpha$ — суффикс строки $\mathscr{L}(v)$ длины $lp(v)$. Тогда существует единственная вершина в дереве ключей, помеченная строкой $\alpha$.*

Доказательство. $\mathscr{K}$ кодирует все образцы из $\mathscr{P}$, и по определению суффикс из $\mathscr{L}(v)$ длины $lp(v)$ есть префикс некоторого образца в $\mathscr{P}$. Так что должен быть путь из корня дерева $\mathscr{K}$, который произносит строку $\alpha$. По построению $\mathscr{K}$ никакие два пути не произносят одну и ту же строку, так что путь единственен, и лемма доказана. □

**Определение.** Для вершины $v$ дерева $\mathscr{K}$ пусть $n_v$ — единственная вершина в $\mathscr{K}$, помеченная суффиксом $\mathscr{L}(v)$ длины $lp(v)$. Если $lp(v) = 0$, то $n_v$ — корень $\mathscr{K}$.

**Определение.** Назовем упорядоченную пару $(v, n_v)$ *связью неудачи*.

На рис. 3.16 представлено дерево ключей для $\mathscr{P} = \{potato, tatoo, theater, other\}$. Связи неудач изображены как указатели от каждой вершины $v$ к вершине $n_v$, где $lp(v) > 0$. Другие связи неудач указывают на корень и в рисунок не включены.

### 3.4.4. Связи неудач ускоряют поиск

Предположим, что мы знаем связь неудачи $v \mapsto n_v$ для каждой вершины $v$ из $\mathscr{K}$. (Позднее мы покажем, как эффективно найти эти связи.) Каким образом связи неудач помогают ускорить поиск? Функция $v \mapsto n_v$ используется в алгоритме Ахо–Корасика (АС) аналогично функции $i \mapsto sp_i$ в алгоритме Кнута–Морриса–Пратта. Здесь $l$, как и раньше, будет отмечать стартовую позицию в $T$ разыскиваемых образцов, а указатель $c$ в $T$ — "текущий символ", который нужно сравнивать с символом из $\mathscr{K}$. Следующий алгоритм использует связи неудач для поиска вхождения в $T$ образца из $\mathscr{P}$:

**Алгоритм поиска АС**

```
l := 1;
  c := 1;
  w := корень 𝒦
  repeat
    while есть дуга (w, w'), помеченная символом T(c) begin
      if w' занумерована образцом i then
        сообщить, что P_i встретилось в t, начиная с позиции l;
      w := w'; c := c + 1;
    end;
    w := n_w; l := c − lp(w);
  until c > m;
```

Чтобы понять назначение функции $v \mapsto n_v$, предположим, что мы прошли по дереву до вершины $v$ и остановились (т. е. символ $T(c)$ не приписан никакой дуге, выходящей из $v$). Мы знаем, что строка $\mathscr{L}(v)$ встречается в $T$, начиная с позиции $l$ и кончая позицией $c - 1$. По определению функции $v \mapsto n_v$ нам гарантировано, что строка $\mathscr{L}(n_v)$ совпадает со строкой $T[c - lp(v)..c - 1]$. Поэтому-то алгоритм и мог

проходить $\mathscr{K}$ от корня до вершины $n_v$, и есть уверенность в совпадении всех символов на этом пути с символами $T$, начиная с позиции $c - lp(v)$. Итак, когда $lp(v) \geqslant 0$, позицию $l$ можно увеличить до $c - lp(v)$, $c$ можно оставить неизмененным и не нужно фактически делать сравнения на пути от корня до вершины $n_v$. На самом деле сравнения следует начать в вершине $n_v$, сопоставляя символ $c$ из $T$ с символами дуг, выходящих из $n_v$.

Например, рассмотрим текст $t = xxpotattooxx$ и дерево ключей, изображенное на рис. 3.16. Когда $l = 3$, текст совпадает со строкой $potat$, а в следующем символе — уже несовпадение. В этой точке $c = 8$, связь неудачи от вершины $v$, помеченной строкой $potat$, указывает на вершину $n_v$, помеченную $tat$, и $lp(v) = 3$. Таким образом, $l$ возрастает до $5 = 8 - 3$, и следующее сравнение будет между символом $T(8)$ и символом $t$ на дуге ниже $tat$.

В этом алгоритме, когда дальнейшие совпадения уже невозможны, $l$ может возрасти больше чем на единицу, избавляя тем самым от повторной проверки символы из $T$ слева от $c$, и при этом мы сохраняем уверенность, что каждое вхождение образца из $\mathscr{P}$, которое *начинается* с символа $c - lp(v)$ строки $T$, будет правильно распознано. Конечно (точно как у Кнута–Морриса–Пратта), мы должны обосновать, что никакое вхождение образца из $\mathscr{P}$ не начинается между старым $l$ и $c - lp(v)$ в $T$, и поэтому $l$ можно увеличить до $c - lp(v)$ без пропуска вхождений. При сделанном предположении, что ни один образец из $\mathscr{P}$ не является собственной подстрокой другого, аргументация почти идентична доказательству теоремы 2.3.2 в анализе метода Кнута–Морриса–Пратта и оставляется как упражнение.

При $lp(v) = 0$ нужно увеличить $l$ до $c$ и начать сравнение с корня $\mathscr{K}$. Остается только случай, когда несовпадение обнаружилось прямо в корне. Тогда $c$ нужно увеличить на 1 и опять начать сравнение с корня.

Поэтому использование функции $v \mapsto n_v$ заведомо ускоряет наивный поиск образца из $\mathscr{P}$. Но улучшает ли оно время счета для наихудшего случая? Аргументами того же типа, как при анализе времени поиска (а не препроцессинга) в методе Кнута–Морриса–Пратта (теорема 2.3.3), легко установить, что время поиска для метода Ахо–Корасика имеет порядок $O(m)$. Мы оставляем это как упражнение. Однако мы должны еще показать, как вычислить за линейное время функцию $v \mapsto n_v$.

### 3.4.5. Линейный препроцессинг для функции неудач

Напомним, что для любой вершины $v$ из $\mathscr{K}$ вершина $n_v$ — единственная из $\mathscr{K}$, помеченная суффиксом $\mathscr{L}(v)$ длины $lp(v)$. Следующий алгоритм находит $n_v$ для каждой вершины $v$ из $\mathscr{K}$ с полным временем $O(n)$. Ясно, что если $v$ — корень $r$ или отстоит от корня на один символ, то $n_v = r$. Предположим для некоторого $k$, что $n_v$ вычислено для всех вершин, отстоящих от корня на не более $k$ символов (дуг). Задача в том, чтобы вычислить $n_v$ для вершины $v$, отстоящей от корня на $k + 1$ символов. Пусть $v'$ — отец $v$ в $\mathscr{K}$, а $x$ — символ на дуге от $v'$ к $v$, как показано на рис. 3.17.

Мы ищем вершину $n_v$ и (неизвестную) строку $\mathscr{L}(n_v)$, помечающую путь от корня до этой вершины; мы знаем вершину $n_{v'}$, так как $v'$ отстоит от $r$ на $k$. Точно так же, как в изложении классической препроцессной обработки для метода Кнута–Морриса–Пратта, мы убеждаемся, что строка $\mathscr{L}(n_v)$ должна быть суффиксом $\mathscr{L}(n_{v'})$ (не обязательно собственным), за которым следует символ $x$. Таким образом, первое, что нужно проверить, — существует ли дуга $(n_{v'}, w')$, выходящая из вершины $n_{v'}$

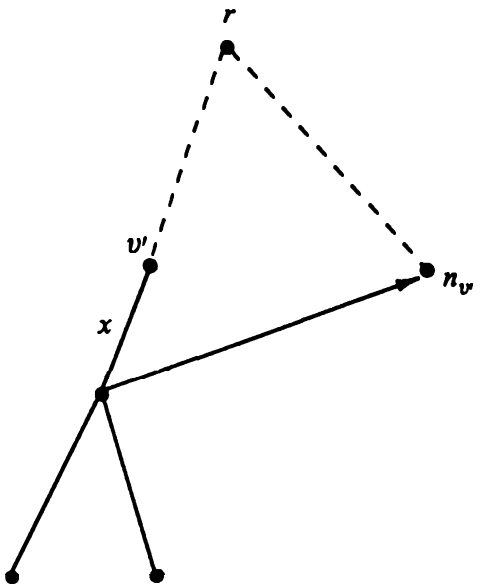


Рис. 3.17. Дерево ключей, используемое для вычисления функции неудач для узла $v$

и помеченная символом $x$. Если такая дуга существует, то $n_v = w'$, и все сделано. В противном случае $\mathscr{L}(n_v)$ есть *собственный* суффикс $\mathscr{L}(n_{v'})$, за которым следует $x$. Тогда проверим следующей вершину $n_{v'}$, чтобы проверить, не найдется ли выходящая из нее дуга, помеченная $x$. (Вершина $n_{v'}$ известна, так как $n_{v'}$ отстоит от корня не больше чем на $k$ дуг.) Продолжая тем же способом с точно тем же обоснованием, как в классическом препроцессинге для метода Кнута–Морриса–Пратта, мы приходим к следующему алгоритму вычисления $n_v$ для вершины $v$:

**Алгоритм** $n_v$
$v'$ — отец $v$ в $\mathscr{K}$;
$x$ — символ на дуге $(v', v)$;
$w := n_{v'}$;
while нет дуги, выходящей из $w$, помеченной $x$, и $w \neq r$
  do $w := n_w$;
if есть дуга $(w, w')$, выходящая из $w$ и помеченная $x$, then
  $n_v := w'$;
else
  $n_v := r$;

Отметим важность предположения, что значение $n_u$ уже известно для каждой вершины $u$, отстоящей не более чем на $k$ символов от $r$.

Чтобы найти $n_v$ для каждой вершины $v$, повторно применим предложенный выше алгоритм ко всем вершинам из $\mathscr{K}$. Будем обходить дерево в ширину, начиная от корня.

**Теорема 3.4.1.** *Пусть $n$ — полная длина всех образцов из $\mathscr{P}$. Полное время, затрачиваемое алгоритмом $n_v$ в применении его ко всем вершинам из $\mathscr{K}$, равно $O(n)$.*

Доказательство. Аргументация этого доказательства прямо обобщает аргументацию из анализа времени при классическом препроцессинге в методе Кнута–

Морриса–Пратта. Рассмотрим единичный образец $P$ из $\mathscr{P}$, имеющий длину $t$, и путь в $\mathscr{K}$ для образца $P$. Будем анализировать время, затраченное алгоритмом для нахождения связей неудач для вершин этого пути так, как будто через проходимые им вершины не пролегают пути для других образцов из $\mathscr{P}$. Такой анализ начислит больше работы, чем ее выполняется на самом деле, но оценка все равно будет линейной.

Главное — проанализировать, как меняется $lp(v)$ при выполнении алгоритма для каждой очередной вершины $v$ вниз по пути для $P$. Когда $v$ отстоит от корня на одну дугу, $lp(v)$ равно 0. Пусть теперь $v$ — произвольная вершина пути и $v'$ — ее отец. Очевидно, $lp(v) \leqslant lp(v') + 1$, так что во всех исполнениях алгоритма $n_v$ для вершины пути для $P$, $lp()$ увеличится не более чем на $t$. Посмотрим далее, насколько $lp()$ может уменьшиться. При вычислении $n_v$ для любой вершины $v$, $w$ стартует с $n_{v'}$, и таким образом начальная глубина вершины равна $lp(v')$. Однако затем глубина вершины $w$ уменьшается при каждом новом присваивании $w$ (внутри цикла *while*). Когда $n_v$ будет найдено, $lp(v)$ становится равным текущей глубине $w$, так что если $w$ присваивается $k$ раз, то $lp(v) \leqslant lp(v') - k$, и $lp()$ уменьшилось по меньшей мере на $k$. Значит, $lp()$ не может быть отрицательным и вырастает во время вычислений по пути $P$ не больше, чем на $t$. Отсюда следует, что во время всех вычислений для вершин на пути для $P$, число присваиваний внутри цикла *while* будет не больше $t$. Полное затраченное время пропорционально числу присваиваний внутри цикла, и следовательно, все связи неудач по пути для $P$ находятся за время $O(t)$.

Применение этого рассуждения к каждому образцу из $\mathscr{P}$ приводит к выводу, что все связи неудач получаются за время, пропорциональное сумме длин образцов в $\mathscr{P}$ (т.е. за время $O(n)$). □

### 3.4.6. Полный алгоритм Ахо–Корасика: освобождение от предположения о подстроках

До сих пор мы предполагали, что никакой образец из $\mathscr{P}$ не является подстрокой другого. Снимем теперь это условие. Если один образец входит в другой и алгоритм АС (с. 83) использует то же дерево ключей, что и раньше, то $l$ может достигнуть слишком больших значений. Рассмотрим случай, когда $\mathscr{P} = \{acatt, ca\}$ и $T = acatg$. В том виде, как он есть, алгоритм находит совпадение $T$ по пути в $\mathscr{K}$, пока текущим не станет символ $g$. Путь заканчивается в вершине $v$ с $\mathscr{L}(v) = acat$. Ни одна дуга, выходящая из $v$, не помечена $g$, и так как никакой собственный суффикс $acat$ не является префиксом $acatt$ или $ac$, то $n_v$ есть корень $\mathscr{K}$. Таким образом, если вершина $v$ останавливает алгоритм, то он возвращается к корню с $g$ в качестве текущего символа, и $l$ получает значение 5. Затем после одного дополнительного сравнения указатель текущего символа дойдет до $m + 1$, и алгоритм завершает работу, не найдя вхождения $ca$ в $T$. Это случилось потому, что алгоритм сдвигает (увеличивает $l$) так, чтобы найти совпадение самого длинного суффикса $\mathscr{L}(v)$ с префиксом какого-либо образца в $\mathscr{P}$. Вложенные вхождения образцов в $\mathscr{L}(v)$, которые не являются суффиксами $\mathscr{L}(v)$, не влияют на увеличение $l$.

С такой трудностью легко справиться благодаря следующим наблюдениям, доказательства которых мы оставляем читателю.

**Лемма 3.4.2.** *Пусть в дереве ключей $\mathscr{K}$ существует путь из связей неудач (возможно, пустой) от вершины $v$ к вершине, занумерованной образцом $i$. Тогда в $T$*

*должен обнаружиться образец $P_i$, который оканчивается в позиции $c$ (текущий символ), как только во время фазы поиска алгоритма Ахо-Корасика будет достигнута вершина $v$.*

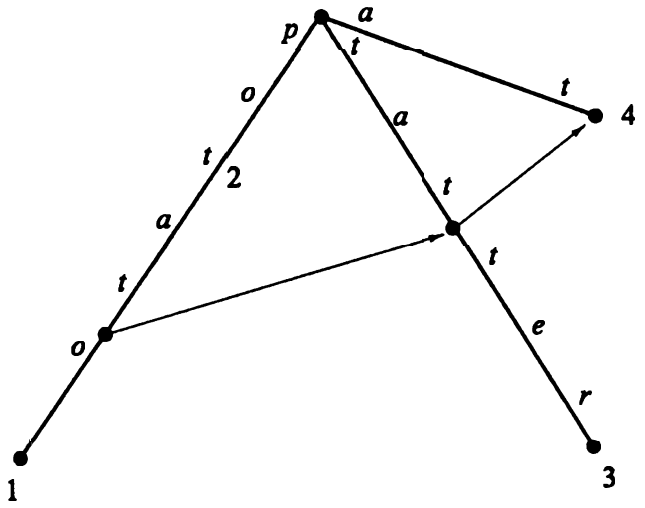


**Рис. 3.18.** Дерево ключей с путем от *potat* до *at* через *tat*

Например, на рис. 3.18 изображено дерево ключей для $\mathscr{P} = \{potato, pot, tatter, at\}$ с некоторыми из связей неудач. Эти связи образуют путь от вершины $v$ с пометкой *potat* к нумерованной вершине с пометкой *at*. Если проход по $\mathscr{K}$ достигает $v$, то $T$ наверняка содержит образцы *tat* и *at*, заканчивающиеся в текущем символе $c$.

И напротив:

**Лемма 3.4.3.** *Пусть в ходе работы алгоритма достигнута вершина $v$. Тогда образец $P_i$ появляется в $T$, заканчиваясь в позиции $c$, только если $v$ имеет номер $i$ или существует путь из связей неудач из $v$ в вершину с номером $i$.*

Итак, полный алгоритм поиска таков.

**Алгоритм полного АС-поиска**

```
l := 1;
c := 1;
w := корень;
repeat
  while существует дуга (w, w') с пометкой T(c) begin
    if w' занумерована образцом i или существует путь из связей неудач
        из w' в вершину с номером i
      then сообщить о вхождении P_i в T, начиная с позиции c;
    w := w'; c := c + 1;
  end;
  w := n_w; l := c - lp(w);
until c > n;
```

**Реализация**

Леммы 3.4.2 и 3.4.3 указывают на высоком уровне, как найти все вхождения образцов в текст, однако нужно уточнить детали реализации. Наша цель — научиться строить

дерево ключей, определять функцию $v \mapsto n_v$ и иметь возможность выполнять алгоритм полного АС поиска за время $O(m+k)$. Нам понадобится дополнительный указатель в каждой вершине $\mathcal{K}$, который мы назовем *связью выхода* (output link).

Связь выхода (если она существует) в вершине $v$ указывает на нумерованную вершину (вершину, соответствующую концу образца из $\mathcal{P}$), отличную от $v$ и достижимую из $v$ за наименьшее число связей неудачи. Связи выхода можно получить за время $O(n)$ при выполнении препроцессинга для $n_v$. Когда определено значение $n_v$, возможная связь выхода из вершины $v$ определяется следующим образом. Если $n_v$ — нумерованная вершина, то связь выхода из $v$ указывает на $n_v$; если $n_v$ не занумерована, но имеет связь выхода на вершину $w$, то связь выхода из $v$ указывает на $w$; в противном случае $v$ не имеет связи выхода. Здесь видно, что связь выхода может указывать только на нумерованную вершину и путь из связей выхода от любой вершины $v$ проходит через все нумерованные вершины, достижимые из $v$ путем, составленным из связей неудач. Например, на рис. 3.18 вершины для *tat* и *potat* получат свои связи выхода на вершину для *at*. Работа по добавлению связей выхода требует лишь константного времени на вершину, так что полное время для алгоритма $n_v$ остается $O(n)$.

Располагая связями выхода, мы можем определить все вхождения в $T$ образцов из $\mathcal{P}$ за время $O(m+k)$. Как раньше, поскольку нумерованная вершина встречается во время полного АС-поиска, вхождение распознается и регистрируется. Вдобавок, если встречается вершина $v$, у которой есть связь выхода, алгоритм должен пройти путь связей выхода из $v$, регистрируя вхождение, кончающееся в позиции $c$ из $T$ для каждой связи этого пути. Когда прохождение по этому пути достигнет вершины без связи выхода, мы возвращаемся вдоль пути к вершине $v$ и продолжаем выполнять алгоритм полного АС-поиска. Так как во время этого прохода не происходит сравнений символов, в обеих фазах, создания и поиска, число сравнений по-прежнему имеет границу $O(n+m)$. Далее, хотя число проходов по связям выхода может превысить линейную границу, каждый проход связи выхода обнаруживает вхождение образца, так что полное время алгоритма имеет оценку $O(n+m+k)$, где $k$ — полное число вхождений. В результате мы имеем:

**Теорема 3.4.2.** *Если $\mathcal{P}$ — набор образцов с полной длиной $n$ и полная длина текста $T$ равна $m$, то все вхождения в $T$ образцов из $\mathcal{P}$ можно найти за время $O(n)$ на препроцессинг плюс $O(m+k)$ на поиск $k$ вхождений. Это верно даже без предположения о подстроках.*

В дальнейшем (п. 6.5) мы обсудим другие аспекты реализации, которые влияют на практическое осуществление как метода Ахо–Корасика, так и метода суффиксного дерева.

## 3.5. Три приложения точного множественного поиска

### 3.5.1. Сравнение с ДНК или библиотекой идентифицированных белков

В молекулярной биологии есть ряд прикладных направлений, при развитии которых созданы относительно устойчивые библиотеки интересных или распознанных подстрок ДНК или белков. Наша первая важная иллюстрация будет связана