Machine Learning

Лекция №5

Снижение размерности

Частота последовательности

Пусть Σ обозначает алфавит, определенный как конечный набор знаков или символов, и пусть $|\Sigma|$ - его мощность. Последовательность или строка определяется как упорядоченный список символов и записывается как $s = s_1 s_2 \ldots s_k$, где $s_i \in \Sigma$- символ с индексом i, также обозначаемый как $s[i]$.

$s[i:j] = s_i s_{i+1} \ldots s_{j-1} s_j$подстрока или последовательность упорядоченных символов в позициях с i по j.

$s[1:i] = s_1 s_2 \ldots s_{i-1} s_i$, где $0 \leqslant i \leqslant n$ – префикс

$s[i:n] = s_i s_{i+1} \ldots s_{n-1} s_n$, где $1 \leqslant i \leqslant n + 1$ – суффикс

Пусть Σ^* будет набором всех возможных последовательностей, которые могут быть построены с использованием символов из Σ, включая пустую последовательность $\emptyset$ (которая имеет нулевую длину).

Пусть $s = s_1 s_2 \ldots s_n$ и $r = r_1 r_2 \ldots r_m$ - две последовательности над Σ. Мы говорим, что r - подпоследовательность s, обозначаемая $r \subseteq s$, если существует взаимно однозначное отображение $\phi: [1, m] \to [1, n]$, такое что $r[i] = s[\phi(i)]$ и для любых двух позиций i, j в r, $i < j \Longrightarrow \phi(i) < \phi(j)$ (s содержит r)

Последовательность r называется последовательной подпоследовательностью или подстрокой s при условии $r_1 r_2 \ldots r_m = s_j s_{j+1} \ldots s_{j+m-1}$, то есть $r[i{:}\, m] = s[j{:}\, j + m - 1]$, где $1 \leqslant j \leqslant n - m + 1$.

Для базы данных $D = \{s_1 s_2 \ldots s_N\}$ из N последовательностей, и для некоторой последовательности r поддержка r в базе данных D определяется как общее количество последовательностей в D, содержащих r

$$\sup(r) = |\{ s_i \in D \mid r \subseteq s_i \}|$$

Относительная поддержка r — это доля последовательностей, содержащих r.

$$r\,\sup(r) = \sup(r)\,/N$$

Учитывая заданный пользователем порог $minsup$, мы говорим, что последовательность r часто встречается в базе данных D, если $sup(r) \geq minsup$.

Частая последовательность ***максимальна***, если она не является подпоследовательностью какой-либо другой частой последовательности.

Частая последовательность ***закрыта***, если она не является подпоследовательностью любой другой частой последовательности с той же поддержкой.

Поиск часто встречающихся последовательностей

Для поиска закономерностей в последовательностях важен порядок символов, и поэтому мы должны рассматривать все возможные ***перестановки* символов** как возможные частые кандидаты.

Пространство поиска последовательности может быть организовано в виде дерева поиска префиксов. Корень дерева на уровне 0 содержит пустую последовательность, причем каждый символ $x \in \Sigma$ является одним из его дочерних элементов. Таким образом, узел, помеченный последовательностью $\mathbf{s} = s_1 s_2 \ldots s_k$ на уровне *k* есть дети вида $\mathbf{s} = s_1 s_2 \ldots s_k s_{k+1}$ на уровне *k+1*.

s – это префикс каждого дочернего элемента $\boldsymbol{s}$`, где $\boldsymbol{s}$` - *расширение* **s**.

Пространство поиска подпоследовательности концептуально бесконечно, поскольку включает в себя все последовательности в Σ^*, то есть все последовательности нулевой или более длины, которые могут быть созданы с помощью символов в Σ.

Пусть *l* обозначает длину самой длинной последовательности в базе данных, тогда в худшем случае необходимо рассмотреть все возможные последовательности длиной до *l*, что дает следующую привязку к размеру пространства поиска:

$$|\Sigma|^1 + |\Sigma|^1 + \cdots + |\Sigma|^l = O(|\Sigma|^l) \quad (10.1)$$

так как на уровне *k* есть $|\Sigma|^k$ возможных подпоследовательностей длины *k*.

Уровневый поиск: GSP

Мы можем разработать эффективный алгоритм поиска последовательностей, который ищет дерево префиксов последовательностей с помощью поиска по уровням или по ширине. Учитывая множество частых последовательностей на уровне *k*, мы генерируем все возможные расширения последовательностей или кандидатов на уровне *k+1*. Затем мы вычисляем поддержку каждого кандидата и отсекаем те, которые не являются частыми. Поиск прекращается, когда более частые расширения невозможны.

Псевдокод для метода уровневого поиска обобщенной последовательности (GSP) показан в алгоритме 10.1. Он использует антимонотонное свойство поддержки для отсечения паттернов кандидатов, то есть: никакая суперсеквенция нечастой последовательности не может быть частой, и все подпоследовательности частой последовательности должны быть частыми.

Префиксное дерево поиска на уровне k обозначается $C^{(k)}$. Первоначально $C^{(1)}$ включает в себя все символы в Σ. Учитывая текущий набор кандидатов *k*-последовательностей $C^{(k)}$, метод сначала вычисляет их поддержку. Для каждой последовательности базы данных $\mathbf{s}_i \in \mathbf{D}$, проверяется, является ли кандидат-последовательность $\mathbf{r} \in C^{(k)}$ подпоследовательностью $\mathbf{s}_i$. Если это так, то поддержку **r** увеличивается. Как только частые последовательности на уровне *k* найдены, генерируются кандидаты на уровень *k+1* (строка 10). Для расширения, каждый лист $\mathbf{r}_a$ расширяется последним символом любого другого листа $\mathbf{r}_b$, который имеет один и тот же префикс (т.е. имеет одного и того же родителя), чтобы получить новый префикс. кандидат (*k + 1*)-последовательность $\mathbf{r}_{ab} = \mathbf{r}_a + \mathbf{r}_b[k]$ (строка 18). Если новый кандидат $\mathbf{r}_{ab}$ содержит любую нечастую *k*-последовательность, мы отсекаем его.

Вычислительная сложность GSP равна $O(|\Sigma|^l)$, согласно ур-ю (10.1), где *l* – длина самой длинной частой последовательности. Сложность ввода-вывода равна $O(l \cdot \mathbf{D})$, потому что вычисляется поддержка целого уровня в одном сканировании базы данных.

Алгоритм 10.1 Алгоритм GSP

GSP(D,Σ, *minsup*)

1. F $\leftarrow \emptyset$
2. C $^{(1)} \leftarrow \{\emptyset\}$ // Начальное префиксное дерево с одиночными символами
3. $k \leftarrow 1$ // k означает уровень
4. Пока C $^{(k)} \neq \emptyset$:
   a) *ComputeSupport*(C $^{(k)}$, **D**)
   b) Для каждого листа $\mathbf{s} \in$ C $^{(k)}$:
      I. Если $sup(\mathbf{r}) \geq minsup$, тогда F $\leftarrow$ F $\cup\{\mathbf{r}, sup(\mathbf{r})\}$
      II. Иначе удалить **s** из C $^{(k)}$
   c) C $^{(k+1)} \leftarrow$ *ExtendPrefixTree(*C $^{(k)}$*)*
   d) $k \leftarrow k + 1$
5. Вернуть F $^{(k)}$

ComputeSupport(C $^{(k)}$, **D**)

1. Для каждого $\mathbf{s}_i \in \mathbf{D}$:
   a) Для каждого $\mathbf{r} \in$ C $^{(k)}$
      i. Если $\mathbf{r} \subseteq \mathbf{s}_i$, то $sup(\mathbf{r}) \leftarrow sup(\mathbf{r}) + 1$

ExtendPrefixTree(C $^{(k)}$)

1. Для каждого листа $\mathbf{r}_a \in$ C $^{(k)}$
   a) Для каждого листа $\mathbf{r}_b \in$ *ДочерныеУзлы*(*РодительскийУзел*($\mathbf{r}_a$)):
      i. $\mathbf{r}_{ab} \leftarrow \mathbf{r}_a + \mathbf{r}_b[k]$ // расширяем $\mathbf{r}_a$ последним символом $\mathbf{r}_b$ // удалить, если есть нечастые подпоследовательности
      ii. Если $\mathbf{r}_c \in$ C $^{(k)}$, для всех $\mathbf{r}_c \subset \mathbf{r}_{ab}$ таких, что $|\mathbf{r}_c| = |\mathbf{r}_c| - 1$, тогда
         - Добавить $\mathbf{r}_{ab}$ в качестве дочернего элемента $\mathbf{r}_a$ с $sup(\mathbf{r}_{ab}) \leftarrow 0$
   b) Если нет расширений $\mathbf{r}_a$, тогда
      - Удалить $\mathbf{r}_a$ и всех предков $\mathbf{r}_a$ без расширений из C $^{(k)}$
2. Вернуть C $^{(k)}$

Вертикальный поиск последовательности: Spade

Алгоритм Spade (англ. «лопата») использует вертикальное представление базы данных для поиска последовательностей.

Для каждого символа $s \in \Sigma$ сохраняется набор кортежей вида $\langle i, pos(s) \rangle$, где $pos(s)$ – набор позиций в последовательности базы данных $\mathbf{s}_i \in \mathbf{D}$, где появляется символ *s*. Пусть L*(s)* обозначает множество таких кортежей позиции последовательностей для символа *s*, который называется списком позиций. Набор списков позиций для каждого символа $s \in \Sigma$ образует вертикальное представление входной базы данных. Учитывая *k*-последовательность **r**, ее список позиций L(r) поддерживает список позиций для вхождений последнего символа $\mathbf{r}[k]$ в каждой последовательности базы данных $\mathbf{s}_i$, при условии $\mathbf{r} \subseteq \mathbf{s}_i$. Поддержка последовательности **r** – это просто число различных последовательностей, в которых есть **r**, то есть $sup(\mathbf{r}) = |\mathrm{L}(s)|$.

Даны списки позиций для любых двух *k*-последовательностей $\mathbf{r}_a$ и $\mathbf{r}_b$, которые имеют одинаковую длину префикса (*k-1*). Необходимо выполнить последовательные соединения в списках позиций для вычисления поддержки новой (k+1) длины последовательности кандидатов $\mathbf{r}_{ab} = \mathbf{r}_a + \mathbf{r}_b[k]$.

Дан кортеж $\langle i, pos(\mathbf{r}_b[k])\rangle \in \mathrm{L}(\mathbf{r}_b)$, проверяется, существует ли кортеж $\langle i, pos(\mathbf{r}_a[k])\rangle \in \mathrm{L}(\mathbf{r}_a)$, то есть обе последовательности происходят в одной и той же последовательности базы данных $\mathbf{s}_i$. Далее, для каждой позиции $p \in pos(\mathbf{r}_b[k])$ проверяется, существует ли позиция $q \in pos(\mathbf{r}_a[k])$ такая, что $q < p$. Если да, то символ $\mathbf{r}_b[k]$ происходит после последней позиции $\mathbf{r}_a$и, таким образом, p сохраняется как верное вхождение $\mathbf{r}_{ab}$. Список позиций $\mathrm{L}(\mathbf{r}_{ab})$ включает в себя все такие случаи

Последовательное соединение обозначается как $\mathrm{L}(\mathbf{r}_{ab}) = \mathrm{L}(\mathbf{r}_a) \cap \mathrm{L}(\mathbf{r}_b)$.

- **Алгоритм 10.2. Алгоритм Spade**
- // Инициализирующий вызов: F ← ø, $k \leftarrow ø$
- $P \leftarrow \{\langle i, L(s)\rangle \mid s \in \Sigma, sup(s) \geq minsup\}$
- **Spade** (P, *minsup*, F, k):

1. Для каждого $\mathbf{r}_a \in P$
   a) F ← F ∪$\{\mathbf{r}_a, sup(\mathbf{r}_a)\}$
   b) $P_a \leftarrow ø$
   c) Для каждого $\mathbf{r}_b \in P$
      I. $\mathbf{r}_{ab} = \mathbf{r}_a + \mathbf{r}_b[k]$
      II. $L(\mathbf{r}_{ab}) = L(\mathbf{r}_a) \cap L(\mathbf{r}_b)$
      III. Если $sup(\mathbf{r}_{ab}) \geq misup$
         I. $P_a \leftarrow P_a \cup \{\mathbf{r}_{ab}, sup(\mathbf{r}_{ab})\}$
   d) Если $P_a \neq ø$, то *Spade*(P, *minsup*, F, k+1)

Поиск последовательностей на основе проекций: PrefixSpan

Пусть **D** обозначает базу данных, а $s \in \Sigma$ – любой символ. Проецируемая база данных относительно *s*, обозначаемая $\mathbf{D}_s$, получается путем нахождения первого вхождения *s* в $\mathbf{s}_i$, в позиции *p*. В $\mathbf{D}_s$ сохраняется только суффикс $\mathbf{s}_i$, начинающийся с позиции *p+1*. Любые нечастые символы удаляются из суффикса. Это делается для каждой последовательности $\mathbf{s}_i \in \mathbf{D}$.

Основная идея PrefixSpan - вычисление поддержки только отдельных символов в проецируемой базе данных $\mathbf{D}_s$ и выполнение рекурсивных проекций на частые символы в первую очередь по глубине.

r – это частая подпоследовательность, и $\mathbf{D}_\mathbf{r}$ – ожидаемый набор данных для **r**. Изначально **r** пуст и $\mathbf{D}_\mathbf{r}$ представляет из себя весь входной набор данных **D**.

PrefixSpan сначала находит все частые символы в проецируемом наборе данных. Для каждого такого символа *s* **r** расщиряется, добавляя *s*, чтобы получить новую частую подпоследовательность $\mathbf{r}_s$. Затем создаётся проецируемый набор данных $\mathbf{D}_s$, проецируя $\mathbf{D}_\mathbf{r}$ на символ *s*. Затем рекурсивно вызывается PrefixSpan с $\mathbf{r}_s$ и $\mathbf{D}_\mathbf{s}$

- **Алгоритм 10.3. Алгоритм PrefixSpan**
- // Инициализирующий вызов: $\mathbf{D}_\mathbf{r} \leftarrow \mathbf{D}$, $\mathbf{r} \leftarrow \emptyset$, F$\leftarrow \emptyset$
- **PrefixSpan** (P, *minsup*, F, k):

1. Для каждого $s \in \Sigma$, такого, что $sup(s, \mathbf{D}_\mathbf{r}) \geq misup$
   - *a)* $\mathbf{r}_s = \mathbf{r} + s$ // расширяем **r** символом *s*
   - b) F $\leftarrow$ F $\cup\{(\mathbf{r}_s, sup(\mathbf{D}_\mathbf{r}))\}$
   - *c)* $\mathbf{D}_\mathbf{s} \leftarrow \emptyset$ // создание проекций для символа *s*
   - d) Для каждого $\mathbf{s}_i \in \mathbf{D}_\mathbf{r}$
     - *i.* $\mathbf{s}_i'$ $\leftarrow$ проекция $\mathbf{s}_i$ со ссылкой на символ *s*
     - ii. Удалить все нечастые символы из $\mathbf{s}_i'$
     - iii. Добавить $\mathbf{s}_i'$ в $\mathbf{D}_\mathbf{r}$, если $\mathbf{s}_i' \neq \emptyset$
   - e) Если $\mathbf{D}_s \neq \emptyset$, то *PrefixSpan* ($\mathbf{D}_s$, $\mathbf{r}_s$, *minsup*, F)

Поиск подстрок с помощью суффиксных деревьев

Пусть **s** – последовательность длиной *n*, тогда в **s** содержится не более $O(n^2)$ возможных различных подстрок в **s**. Рассмотрим подстроки длины *w*, из которых в *s* имеется *n-w+1* возможных подстрок в **s**. Сложив все длины подстрок, получим

$$\sum_{w=1}^{n}(n-w+1) = n + (n-1) + \cdots + 2 + 1 = O(n^2)$$

Это гораздо меньшее пространство поиска по сравнению с подпоследовательностями, и поэтому можно разработать более эффективные алгоритмы для решения задачи поиска частых подстрок. Фактически, можно найти все частые подстроки в худшем случае за $O(Nn^2)$ времени для набора данных $\mathbf{D} = \{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_k\}$ с *N* последовательностями.

Суффиксное Дерево

Пусть Σ – алфавит и $\$ \notin \Sigma$ – терминальный символ, используемый для обозначения конца строки. Используя последовательность **s**, терминальный символ добавляется так, что $\mathbf{s} = s_1 s_2 \dots s_n s_{n+1}$, где $s_{n+1} = \$$, а j-й суффикс задается как $\mathbf{s}[j : n+1] = s_j \dots s_n s_{n+1}$.

Суффиксальное дерево последовательностей в базе данных **D**, обозначаемое T, хранит все суффиксы для каждого $\mathbf{s}_i \in \mathbf{D}$ в древовидной структуре, где суффиксы, имеющие общий префикс, лежат на одном и том же пути от корень дерева.

Подстрока, полученная путем объединения всех символов от корневого узла до узла *v*, называется ***меткой узла v*** и обозначается как $\boldsymbol{L(v)}$.

Подстрока, которая появляется на ребре (v_a, v_b), называется ***меткой ребра*** и обозначается как $\boldsymbol{L(v_a, v_b)}$.

Суффиксальное дерево имеет два типа узлов: внутренние и конечные узлы. Внутренний узел в дереве суффиксов (за исключением корня) имеет по крайней мере два дочерних элемента, где каждая метка ребра к дочернему элементу начинается с другого символа. Поскольку терминальный символ уникален, то в дереве суффиксов столько же листьев, сколько уникальных суффиксов во всех последовательностях. Каждый узел-лист соответствует суффиксу одной или нескольких последовательностей в **D**.

Нетрудно получить квадратичное время и объем для алгоритма построения дерева суффиксов. Изначально дерево суффиксов T пусто. Далее, для каждой последовательности $\mathbf{s}_i \in \mathbf{D}$, $|\mathbf{s}_i| = n_i$, генерируются все ее суффиксы $\mathbf{s}_i[j:n+1]$, при $1 \leq j \leq n_i$, каждый суффикс вставляется в дерево, следуя от корня, пока не встретятся либо листья, либо несоответствие в одном из символов вдоль ребра. При достижении листа, вставляется пара *(i, j)* в лист, отмечая, что это *j*-й суффикс последовательности $\mathbf{s}_i$. Если нашлось несоответствие в одном из символов, например, в позиции *p>j*, добавляется внутренняя вершина непосредственно перед несовпадением и создается новый узел-лист, содержащий *(i, j)* с меткой ребра $\mathbf{s}_i[j:n+1]$.

С точки зрения сложности времени и требуемой памяти, алгоритм, описанный выше, требует $O(Nn^2)$ время и памяти, где N – число последовательностей в **D**, а n – самая длинная длина последовательности. Временная сложность вытекает из того, что метод всегда вставляет новый суффикс, начиная с корня дерева суффиксов. Это означает, что в худшем случае он сравнивает $O(n)$ символов на вставку суффикса, давая наихудшую границу $O(n^2)$ для всех n суффиксов. Сложность памяти возникает из-за того, что каждый суффикс явно представлен в дереве, занимая $n + (n-1) + \cdots + 2 + 1 = O(n^2)$ памяти. Для N последовательностей в базе данных, мы получается $O(Nn^2)$ как наихудший случай временных границ и границ памяти.

Частые Подстроки

Метки узлов для узлов с поддержкой хотя бы *minsup* дают набор частых подстрок; все префиксы таких меток узлов также являются частыми. Дерево суффиксов также может поддерживать специальные запросы для поиска всех вхождений в базу данных любой запрашиваемой подстроки **q**. Для каждого символа в **q** мы следуем по пути от корня до тех пор, пока не увидим все символы в **q** или пока не обнаружим несоответствие в любой позиции. Если **q** найден, то набор листьев под этим путем является списком вхождений **q**.

С точки зрения сложности времени запроса, поскольку необходимо сопоставить каждый символ в **q**, $O(|\mathbf{q}|)$ получается в качестве временной границы (предполагая, что $|\Sigma|$ – это константа), которая ***не зависит*** от размера базы данных. Перечисление всех совпадений занимает дополнительное время, для общей временной сложности $O(|\mathbf{q}| + k)$, если есть *k* совпадений.

Алгоритм линейного времени Укконена
Достижение линейной памяти

Если алгоритм хранит все символы на каждой метке ребра, то сложность памяти равна $O(n^2)$, и также невозможно достичь линейного времени построения. Хитрость заключается в том, чтобы явно не хранить все метки ребер, а использовать метод сжатия ребер, когда хранятся только начальная и конечная позиции метки ребер во входной строке **s**. То есть, если метка ребра задана как $\mathbf{s}[i:j]$, то она представляется как интервал $[i, j]$.

С точки зрения сложности памяти, когда добавляется новый суффикс к дереву T, оно может создать не более одного нового внутреннего узла. Поскольку существует *n* суффиксов, в T есть *n* листьев и не более *n* внутренних узлов. Имея не более *2n* узлов, дерево имеет не более *2n–1* ребер, и поэтому общее пространство, необходимое для хранения интервала для каждого ребра, равно $2(2n - 1) = 4n - 2 = O(n)$.

Достижение Линейного Времени

Метод Укконена – это *онлайн* алгоритм, то есть при заданной строке $\mathbf{s} = s_1 s_2 \ldots s_n\$$ он строит полное суффиксальное дерево поэтапно. Фаза *i* строит дерево до *i*-го символа в **s**, то есть обновляет дерево суффиксов из предыдущей фазы, добавляя следующий символ s_i. Пусть T_i , обозначим суффиксальное дерево вплоть до *i*-го префикса $\mathbf{s}[i:j]$ с $1 \leq j \leq n$. Алгоритм Укконена строит T_i из T_{i-1} , убедившись, что все суффиксы, включая текущий символ s_i, находятся в новом промежуточном дереве T. Другими словами, на *i*-й фазе он вставляет в дерево T_i все суффиксы $\mathbf{s}[i:j]$ от *j* = 1 до *j* = *i*. Каждая такая вставка называется *j*-м расширением *i*-й фазы. Как только обработается терминальный символ в позиции *n*+1, получится конечное суффиксальное дерево T для **s**.

Алгоритм 10. 4. Наивный алгоритм Укконена Неявные суффиксы

Эта оптимизация утверждает, что в фазе *i*, если *j*-е расширение $\mathbf{s}[j:i]$ найдено в дереве, то любые последующие расширения также будут найдены, и, следовательно, нет необходимости обрабатывать дальнейшие расширения в фазе *i*. Таким образом, суффиксальное дерево T в конце фазы *i* имеет неявные суффиксы, соответствующие расширениям *j*+1 через *i*.

- **NaiveUkkonen**(**s**):

1. $n \leftarrow |\mathbf{s}|$
2. $\mathbf{s}[n+1] \leftarrow \$$ // добавляем терминальный символ
3. T $\leftarrow \emptyset$ // добавляем пустую строку в качестве корень
4. Для каждого $i = 1, \ldots, n+1$ // *i*-ая фаза – построение T_i
   - Для каждого $j = 1, \ldots, i$ // расширение *j* на фазе *i*
   - // Вставка $\mathbf{s}[j:i]$ в суффиксное дерево
     a) Найти конец пути с меткой $\mathbf{s}[j:i-1]$ в T
     b) Вставить s_i в конец пути
5. Вернуть T

Неявные расширения

Пусть текущая фаза равна *i*, а $l \leq i-1$ – последняя явная фаза. Суффикс в предыдущем дереве T_{i-1}. Все явные суффиксы в T_{i-1} имеют реберные метки вида $[x: i+1]$, ведущие к соответствующим листовым узлам, где начальная позиция *x* является узлом, но конечная позиция должна быть *i*-1, потому что s_{i-1} был добавлен в конец этих путей в фазе *i*-1. В текущей фазе *i* мы должны были бы расширить эти пути, добавив s_i в конце.

Если заменить $[x: i-1]$ на $[x: e]$, то на фазе *i*, если установить *e=i*, то сразу же все *l* существующих суффиксов будут ***неявно*** расширены до $[x: i]$. Таким образом, в одной операции приращения *e* произошло расширение от 1 до *l* для фазы *i*.

Хитрость пропуска/подсчета

Для *j*-го расширения фазы *i* необходимо искать подстроку $\mathbf{s}[j: i-1]$, так что можно добавить s_i в конец. Эта строка должна существовать в T_{i-1}, потому что символ s_{i-1} обработан на предыдущем этапе. Таким образом, вместо поиска каждого символа в $\mathbf{s}[j: i-1]$, начиная с корня, сначала идет подсчёт количества символов на ребре, начинающемся с символов s_i; пусть эта длина равна m. Если m больше длины подстроки (т.е. если $m > i-j$), то подстрока должна заканчиваться на этом ребре, поэтому совершается переход в позицию $i-j$ и вставляется s_i. Если $m < i-j$, то можно перейти непосредственно к дочернему узлу, например v_c, и искать оставшуюся строку $\mathbf{s}[j+m: i-1]$ из v_c, используя ту же технику пропуска/подсчета. При такой оптимизации стоимость расширения становится пропорциональной количеству узлов на пути, а не количеству символов в $\mathbf{s}[j: i-1]$.

Суффиксальные ссылки

Для каждого внутреннего узла v_a поддерживается связь с внутренним узлом v_b, где $L(v_b)$ – непосредственный суффикс $L(v_a)$. В расширении j-1 пусть v_p обозначает внутренний узел, под которым находится $\mathbf{s}[j: i-1]$, и пусть m – длина метки узла v_p. Чтобы вставить j-е расширение $\mathbf{s}[j: i]$, осуществляется переход по суффиксной ссылке из v_p, в другой узел, например v_s, и ищется оставшееся подстроку $\mathbf{s}[j+m-1: i-1]$ из v_s. Использование суффиксных ссылок позволяет прыгать внутри дерева для различных расширений, в отличие от поиска из корня каждый раз.

Алгоритм 10. 5. Алгоритм Укконена

- **Ukkonen** (**s**):

1. $n \leftarrow |\mathbf{s}|$
2. $\mathbf{s}[n+1] \leftarrow \$$ // добавляем терминальный символ
3. T $\leftarrow \emptyset$ // добавляем пустую строку как корень
4. $l \leftarrow 0$
5. Для каждого $i = 1, \ldots, n+1$ // *i*-ая фаза – построение T_i
   - Для каждого $j = 1, \ldots, i$ // расширение *j* на фазе *i*

- // Вставка $\mathbf{s}[j:i]$ в суффиксное дерево
  a) Найти конец пути с меткой $\mathbf{s}[j:i-1]$ в T с использованием пропуска/подсчета и суффиксальных ссылок
  b) Если $\mathbf{s}_i \in$T // неявные суффиксы, то
     - break
  c) Иначе
     i. Вставить s_i в конец пути
     ii. Установить последний явный суффикс *l*, если необходимо

6. Вернуть T

Алгоритм Укконена имеет временную сложность $O(n)$ для последовательности длины *n*, поскольку он выполняет только постоянный объем работы (амортизируемый), чтобы сделать каждый суффикс явным. Для каждой фазы определенное число расширений выполняется неявно, просто увеличивая *e*. После *i* расширений от $j = 1$ до $j = i$, скажем, что *l* выполняются неявно. Для остальных расширений остановка совершается в первый раз, когда некоторый суффикс неявно находится в дереве; пусть это расширение будет *k*. Таким образом, фаза *i* должна добавлять явные суффиксы только для суффиксов *l*+1 через *k*-1. Для создания каждого явного суффикса выполняется постоянное число операций, которое включает в себя следование за ближайшей суффиксной связью, пропуск/подсчет для поиска первого несоответствия и вставку, если это необходимо, нового конечного узла суффикса. Поскольку каждый лист становится явным только один раз, а количество шагов пропуска/подсчета ограничено $O(n)$ по всему дереву, получается наихудший случай времени алгоритма $O(n)$. Таким образом, общее время по всей базе данных из *N* последовательностей равно $O(Nn)$, если *n* – самая длинная длина последовательности.