Реализация схемы «*вверх-прыжок-вниз*»

- Функция *Top-Jump-Bottom ()* реализует схему «*вверх-прыжок-вниз*» для обнаружения на дереве конца заданной подстроки
- Параметры:
  - *substr* – подстрока длины *m*, заведомо совпадающая до предпоследнего символа включительно
  - *pArc* – дуга, на которой был завершен предыдущий поиск
  - *iArcEnd* – индекс последнего символа метки этой дуги на момент завершения поиска (с тех пор дуга могла разделиться)
  - *idxArc* – индекс символа метки этой дуги, на котором был завершен поиск
  - *idxSubstr* – индекс первого несовпавшего символа подстроки
- Возвращает ссылку на дугу завершения поиска новой подстроки &*substr*[1]
- Перевычисляются параметры *idxStr*, *idxArc*, передаваемые по ссылке

Псевдокод стадий «*вверх-прыжок-*»

```
Top-Jump-Bottom (str, substr, m, PArc pArc, iArcEnd, &idxSubstr, &idxArc)
{
  if (!pArc) return NULL; // Переходы невозможны
  PArc pArcNext = NULL; // Результатом будет дуга
  PNode pSrcVert = NULL; // Вершина – цель перемещения “вверх”
  // Мы во внутренней вершине дерева?
  isInnerVert = pArc->pDestVert && idxArc > iArcEnd;
  if (isInnerVert) pSrcVert = (PNode)pArc->pDestVert;
  else pSrcVert = pArc->pSrcVert; // Родительская вершина (стадия “вверх”)
  PNode pRefVert = pSrcVert->pSRef; // “Прыжок” по суффиксной ссылке
  if (!pRefVert) pRefVert = pSrcVert; // У корня суффиксной ссылки нет
```

Псевдокод стадии «*вниз*»

```
//  Поиск от вершины до конца подстроки из m символов (стадия “вниз”)
nCharsUp = isInnerVert ? 1 : idxArc - pArc->iBeg + 1; // Кол-во символов от
                                                      // вершины вниз(+1 - до m)
if  (!pSrcVert->pSRef) --nCharsUp;  // Без ссылки координата сдвинута на 1
nVertChr = m - nCharsUp;  // Индекс первого искомого символа substr
pArcNext = findSuffixTreeArc(str, &substr[nVertChr], nCharsUp, nCharsUp-1,
                                                      pRefVert, idxSubstr, idxArc);
if (!pArcNext)
{  // Ничего не найдено - остаемся на месте
   pArcNext = pRefVert->pArcIn;
   if (pArcNext) idxArc = pArcNext->iEnd + 1;
}
```

Схема «*вверх-прыжок-вниз*» - окончание

```
  idxSubstr += nVertChr;  //  Индекс от начала подстроки
  return  pArcNext;
}
```

- Ниже рассмотрим для наглядности общую схему работы алгоритма – последовательный процесс поиска и продления суффиксов
- Она демонстрирует две смежные фазы и шаблоны обрабатываемых ими подстрок
- Далее приведем основной псевдокод, возвращающий ссылку на корень построенного суффиксного дерева.

Схема обработки подстрок

Шаблон	Фаза, шаг
XXXXXX	(i=5,j=0)
XXXXX	(i=5,j=1)
XXXX	(i=5,j=2)
XXX	(i=5,j=3)
XX	(i=5,j=4)
X	(i=5,j=5)
XXXXXXX	(i=6,j=0)
XXXXXX	(i=6,j=1)
XXXXX	(i=6,j=2)
XXXX	(i=6,j=3)
XXX	(i=6,j=4)
XX	(i=6,j=5)

Комментарии

- На дереве гарантированно вычисляется подстрока до ее предпоследнего символа включительно
- Синим помечен конец подстроки, обнаруживаемый (в начале фазы, j = 0) с помощью функции *Find-SuffixTree-Arc*
- Символ зеленого цвета (j > 0) вычисляется функцией *Top-Jump-Bottom* (по имеющимся суффиксным ссылкам)
- Красным помечен символ, который удлиняет на 1 найденные суффиксы по одному из правил продления 1)-3)
- Красные символы (и соответствующие им внутренние вершины) используются для формирования новых ссылок
- При этом старые суффиксные ссылки сохраняют свою актуальность
- Суффиксная ссылка направлена от более длинной подстроки к более короткой в рамках одной фазы

Квадратичный алгоритм – начало

```
ST-Buid-Online-2 (str)
{
   n = strlen(str);
   // Корень дерева и его начальная дуга
   PNode pTree = ST-Vert-Init-Ex (NULL);
   ST-Arc-Init-Ex (pTree, str[0], 0, 0, NULL, 0);
   //  Последовательное добавление продлеваемых суффиксов
   for  (i = 1; i < n; ++i)
   {  // Фаза i: перебор суффиксов подстроки S[0..i]
      PArc pArcPrev = NULL; iEndPrev = -1;  // Дуга и ее конец на предыдущем шаге
      PNode  pRefFrom = NULL;  // Источник суффиксной ссылки
      idxSubstr = idxArc = 0;    // Координаты остановки поиска
```

Квадратичный алгоритм – поиск суффикса

```
for  (j = 0;  j <= i; ++j)
{ //  Поиск очередного суффикса на дереве
   m = i - j + 1;  // Текущая длина суффикса
   PArc pUVArc = NULL;
   if  (j) pUVArc = Top-Jump-Bottom (str, &str[j], m, pArcPrev, iEndPrev,
                                                          idxSubstr, idxArc);
   else  pUVArc = Find-SuffixTree-Arc (str, &str[j], m, m-1, pTree,
                                                          idxSubstr, idxArc);
   pArcPrev = pUVArc;  // Для «прыжка» на следующей итерации
   // Запомнить конец, т.к. дуга может разделиться на этой же итерации
   iEndPrev = pArcPrev ? pArcPrev->iEnd : -1;
```

Квадратичный алгоритм – продления 1), 3)

```
if (idxSubstr == m)
{   // Суффикс найден -> фиктивное продление
   pRefFrom = NULL; --idxArc; // Для прыжка нужна конечная позиция i-1
   continue;
}
PNode pWNode = NULL;  //  Вершина остановки поиска
if  (!pUVArc)  pWNode = pTree; // Поиск остановился в корне
else  pWNode = pUVArc->pDestVert;  // В середине или конце дуги (U, V)
if (!pWNode && idxArc > pUVArc->iEnd)
{  // Остановка в листе –> продление листа
   ++pUVArc->iEnd; pRefFrom = NULL; continue;
}
```

Квадратичный алгоритм – продление 2)

```
//  Оставшийся вариант - ответвление символа
if (pUVArc && idxArc <= pUVArc->iEnd)
{  // Поиск остановился внутри дуги (U, V), требуется ее разделение
   pWNode = ST-Vert-Init-Ex (pUVArc);  //  Разделяющая вершина
   //  Дуга из W в V
   pWVArc = ST-Arc-Init-Ex (pWNode, str[idxArc], idxArc, pUVArc->iEnd,
                                                pUVArc->pDestVert, pUVArc->iDestVert);
   if  (pUVArc->pDestVert) pUVArc->pDestVert->pArcIn = pWVArc;
   // Дуга из U в W
   pUVArc->pDestVert = pWNode; pUVArc->iDestVert = -1;
   pUVArc->iEnd = idxArc - 1;
```

Квадратичный алгоритм – окончание

```
      if  (pRefFrom)  pRefFrom->pSRef = pWNode; // Суффиксная ссылка в W
      pRefFrom = pWNode; // Ссылка для следующего шага
    }
    else
    {  // W == V, из V ссылка уже есть
      if (pRefFrom)  pRefFrom->pSRef = pWNode;
      pRefFrom = NULL;
    }
  // Добавить новую дугу из вершины W в лист
  ST-Arc-Init-Ex (pWNode, str[i], i, i, NULL, j);
  }
}
return  pTree; }
```

Путь к линейному алгоритму Укконена

- В рассмотренном ранее алгоритме есть «слабые места»; их устранение могло бы повысить эффективность
- *Первое* – вызов функции *Find-SuffixTree-Arc* на 0 шаге каждой фазы (при $j = 0$)
- На 0 шаге ищется наиболее длинная подстрока, стартующая с начала текста
- Поэтому нет суффиксной ссылки для сокращения времени поиска
- *Find-SuffixTree-Arc* реализует перемещение не по символам меток, а по вершинам дерева (подстрока заведомо почти входит)
- Даже логарифмическое время ее работы делает невозможным снижение общей сложности алгоритма до линейной

Схема: две смежные фазы

```
XXXXX     (i=4, j=0)
 XXXX     (i=4, j=1)
  XXX     (i=4, j=2)
   XX     (i=4, j=3)
    X     (i=4, j=4)
XXXXXX    (i=5, j=0)
 XXXXX    (i=5, j=1)
  XXXX    (i=5, j=2)
   XXX    (i=5, j=3)
    XX    (i=5, j=4)
     X    (i=5, j=5)
```

Описание подхода

- При $j > 0$ работают суффиксные ссылки, для шага $j = 0$ такой возможности нет
- Свойства подстрок с индексом $j = 0$: стартуют с начальной позиции текста, а их длины последовательно увеличиваются на 1
- На нулевом шаге рассматривается наиболее длинная подстрока для всех предыдущих фаз ⇒ она всегда заканчивается в листе
- Если на 0 шаге запомнить лист остановки (т.е. входящую дугу *arc0*), то на следующем 0 шаге достаточно лишь продлить его на 1 символ
- Для такой манипуляции достаточно константного времени
- Необходимо еще учесть, что дерево может модифицироваться – дуга *arc0* может разделяться, и тогда ее следует пересохранять

Напоминание: типы продлений

- При поиске может быть не найден лишь последний символ – $S[i]$

  1) Продление листа.

  Неудачный поиск приводит в лист. «Удлиняется» последняя дуга: к ее метке дописывается символ $S[i]$.

  2) Ответвление символа.

  Поиск останавливается во внутренней вершине или на ребре. Из точки остановки выводится дуга в *новый лист*.

  3) Фиктивное продление.

  Искомый суффикс уже существует на дереве (найден). Никаких преобразований дерева не выполняется.

Второе усовершенствование

- Связано со следующим наблюдением: если на некоторой фазе применяется правило 3) – «продленный суффикс найден», то оно будет применяться и на всех оставшихся шагах этой фазы
- Это очевидно: если в $S[0..i]$ содержится некоторая подстрока (в данном случае $S[j..i+1]$), то содержатся и все более короткие подстроки ($S[j+1..i+1]$, $S[j+2..i+1]$, …, $S[i+1..i+1]$)
- Поэтому при нахождении в дереве продленного суффикса можно сразу переходить к следующей фазе алгоритма
- Отсюда – существенная экономия времени работы

Третье усовершенствование

- После применения правила ответвления 2) и создания нового листа к листу всегда будет применяться правило 1)
- В завершении каждой следующей фазы *i* его дуге в качестве индекса символа конца метки должно соответствовать значение *i*
- Метки всех ведущих в листья дуг в качестве *iEnd* будут неявно содержать текущее *i*
- Поле *iEnd* для таких дуг оказывается избыточным, при реализации оно может быть совмещено с полем *iDestVert*
- Реальные продления листьев не требуются вообще

Модификация фазы алгоритма

- Пусть содержательная часть фазы i-1 закончилась на шаге j'
- То есть к суффиксам 0, …, j'-1 применялись лишь правила 1) и 2) ⇒ *каждый из этих суффиксов заканчивается в своем листе*
- На фазе i все эти листья подлежат продлению, теперь осуществляемому неявно
- Поэтому содержательная часть фазы i должна работать лишь с шага $j = j'$ до первого применения правила 3)
- Далее достаточно переопределить значение j' и перейти к следующей фазе алгоритма
- Алгоритм имеет линейную сложность: каждая фаза начинается с того же шага, на котором остановилась предыдущая

Замечания по реализации

- Потребуются модификации функций *Top-Jump-Bottom* и *Find-SuffixTree-Arc*, учитывающие неявный способ хранения концов (*iEnd)* меток дуг, входящих в листья
- Эти функции можно снабдить дополнительным параметром *iLeavesEnd*, получающим индекс *i* текущей фазы в качестве ограничителя указанных дуг
- Использование поля pArc->iEnd можно заменить переменной:

  iEnd = pArc->pDestVert ? pArc->iEnd : iLeavesEnd;

- Ввиду незначительности этих модификаций новые тексты функций не приводятся
- Дополнительный параметр *iLeavesEnd* считается введенным

Использование «начальной дуги» фазы (*arc0*)

```
  XXXXX   (i=5, j=1)
   XXXX   (i=5, j=2)
    XXX   (i=5, j=3)
     XX   (i=5, j=4)
      X   (i=5, j=5)
XXXXXXX   (i=6, j=0)
 XXXXXX   (i=6, j=1)
  XXXXX   (i=6, j=2)
   XXXX   (i=6, j=3)
    XXX   (i=6, j=4)
```

(сюда – переход по *arc0*)

(переход по суффиксной ссылке)

(переход по суффиксной ссылке, конец фазы)

(сюда – переход по *arc0*)

(переход по суффиксной ссылке)

…

Алгоритм Укконена – начало

```
ST-Buid-Online-1 (str)
{
  n = strlen(str);
  PNode pTree = ST-Vert-Init-Ex (NULL); // Корень и начальная дуга
  PArc pArc0 = ST-Arc-Init-Ex (pTree, str[0], 0, 0, NULL, 0);  // Дуга нач. шага
  //  Последовательное добавление продлеваемых суффиксов
  js = 0;  // Начальный шаг фазы
  for  (i = 1; i < n; ++i)
  {  // Фаза i: перебор суффиксов подстроки S[0..i]
    PArc pArcPrev = NULL; iEndPrev = -1;  // Дуга и ее конец на предыдущем шаге
    PNode  pRefFrom = NULL;  // Источник суффиксной ссылки
    idxSubstr = idxArc = 0;    // Координаты остановки поиска
```

Алгоритм Укконена – поиск суффикса

```
for  (j = js;  j <= i; ++j)
{ //  Поиск очередного суффикса на дереве
   m = i - j + 1;  // Текущая длина суффикса
   if  (j == js)
   {  // Начальный шаг
      pArcPrev = pArc0; iEndPrev = i - 1; idxArc = i; idxSubstr = m - 1;
   }
  PArc pUVArc = Top-Jump-Bottom (str, &str[j], m, pArcPrev, iEndPrev, i,
                                                              idxSubstr, idxArc);
   if (idxSubstr == m) { js = j;  break; } // Найден: фиктивное продление
   pArcPrev = pUVArc;  // Для «прыжка» на следующем шаге
```

Алгоритм Укконена – подготовка продления 2)

```
// Запомнить конец поиска, т.к. дуга может разделиться на этом же шаге
if (!pArcPrev) iEndPrev = -1;
else if (!pArcPrev->pDestVert) iEndPrev = i;
else iEndPrev = pArcPrev->iEnd;

PNode pWNode = NULL;  //  Вершина остановки поиска
if  (!pUVArc)  pWNode = pTree; // Поиск остановился в корне
else  pWNode = pUVArc->pDestVert;  // В середине или конце дуги (U, V)
```

Алгоритм Укконена – продление 2)

```
//  Содержательный вариант – ответвление символа
if (pUVArc && idxArc <= iEndPrev)
{  // Поиск остановился внутри дуги (U, V), требуется ее разделение
   pWNode = ST-Vert-Init-Ex (pUVArc);  //  Разделяющая вершина
   //  Дуга из W в V
   pWVArc = ST-Arc-Init-Ex (pWNode, str[idxArc], idxArc, pUVArc->iEnd,
                                                  pUVArc->pDestVert, pUVArc->iDestVert);
   if (pUVArc->pDestVert) pUVArc->pDestVert->pArcIn = pWVArc;
   // Дуга из U в W
   pUVArc->pDestVert = pWNode;  pUVArc->iEnd = idxArc - 1;
   pUVArc->iDestVert = -1;
```

Алгоритм Укконена – окончание

```
        if  (pRefFrom)  pRefFrom->pSRef = pWNode; // Суффиксная ссылка в W
        pRefFrom = pWNode; // Ссылка для следующего шага
      }
      else
      {  // W == V, из V ссылка уже есть
        if (pRefFrom)  pRefFrom->pSRef = pWNode;  pRefFrom = NULL;
      }
    // Добавить новую дугу из вершины W в лист
    pArc0 = ST-Arc-Init-Ex (pWNode, str[i], i, i, NULL, j+1); //  j сдвинуто
    }
  }
  return  pTree;
}
```

Esko Ukkonen