Лекция №12.

Тема: **Организация поиска данных. Часть 2.**

Поиск данных. Цель поиска. Критерии эффективности алгоритмов поиска. Поиск данных в последовательности. Редакционное расстояние.

Ключевые слова: алгоритм Кнута-Морриса-Пратта; префикс-функция; алгоритм Бойера-Мура-Хорспула; расстояние Левенштейна.

Keywords: Knuth–Morris–Pratt algorithm; failure function; Boyer–Moore–Horspool algorithm; Levenshtein distance.

1 Поиск данных в строке

Алгоритм прямого поиска подстроки в строке на больших последовательностях в худшем случае характеризуется большой вычислительной сложностью. Получение и учет дополнительной информации об исходной строке или подстроке повышает эффективность алгоритма поиска данных. Рассмотрим два усложненных алгоритма поиска данных в строке.

1.1 Алгоритм Кнута-Морриса-Пратта

Алгоритм Кнута-Морриса-Пратта (КМП-алгоритм, 1974 г.) – эффективный алгоритм поиска подстроки (pattern) в строке (text), который учитывает структуру pattern путем использования префикс-функции. Префикс-функция позволяет найти подходящий отрезок сдвига, чтобы избежать заведомо бесполезные сравнения элементов pattern с элементами text.

Префикс-функция $\pi(S,i)$ от строки S – длина наибольшего префикса строки $S[0...i-1]$, который одновременно является её суффиксом (исключая вырожденный случай, где префикс совпадает с этой строкой, т.е. $\pi(S,i) = \text{len}(S)$).

Результатом применения префикс-функции к pattern является последовательность целых чисел $d[0...\text{len}(\text{pattern})-1]$, каждый элемент которой определяется следующим образом

$$d[i] = \max_{k=0...i} (k : \text{pattern}[0...k-1] == \text{pattern}[i-k+1...i])$$

Сформируем последовательность d для строки 'колокола'

i	S[i]	Префикс	Суффикс	d[i]
0	'к'	-	-	0
1	'ко'	-	-	0
2	'кол'	-	-	0
3	'коло'	-	-	0
4	'колок'	'**к**олок'	'колок'	1
5	'колоко'	'**ко**локо'	'колок**о**'	2
6	'колокол'	'**кол**окол'	'колок**ол**'	3
7	'колокола'	-	-	0

Свойства последовательности d:

1. Размер d равен размеру pattern (d = [0] * len(pattern)).
2. Каждый элемент d равен целому числу.

3. Возможные числовые значения элементов d можно разделить на 3 группы:

- нулевой элемент d всегда равен нулю;
- элемент d может быть равен нулю, что говорит о несовпадении очередного символа pattern с символом в начале pattern;
- элемент d может быть равен положительному числу, что говорит о совпадении некоторой группы символов в pattern до текущей позиции.

Рассмотрим реализацию префикс-функции на языке Python.

```
# Объявление и инициализация text и pattern.

...

# Объявление и инициализация d.
d = [0] * len(pattern)
j = 0
# Формирование d.
# Нулевой элемент d всегда равен нулю.
for i in range(1, len(pattern)):
    while j > 0 and pattern[i] != pattern[j]:
        j = d[j-1]
    if pattern[i] == pattern[j]:
        j += 1
    # Присвоение значения очередному элементу d.
    d[i] = j
```

В КПМ-алгоритме pattern сопоставляется с text, начиная с нулевых символов каждой из них, то есть «слева-направо». Если нулевые символы совпали, то идет сопоставление последующих символов аналогично прямому поиску подстроки в строке. При первом несовпадении символов КМП-алгоритм начинает использовать информацию из последовательности d для вычисления эффективного смещения pattern относительно text, что обеспечивает меньшее количество сравнений, чем в алгоритме прямого поиска подстроки в строке.

Рассмотрим реализацию КМП-алгоритма на языке Python.

```
j = 0   # Индекс для pattern.
for i in range(0, len(text)):
    while j > 0 and pattern[j] != text[i]:
        # Если встретилось несовпадение, \
        # то индекс pattern устанавливается равным значению \
        # j - 1 элемента d.
        j = d[j - 1]
    if pattern[j] == text[i]:
        # Если встретилось совпадение, \
        # то индекс pattern увеличиваем на 1.
        j += 1
    if j == len(pattern):
        # Если индекс pattern равен длине pattern, \
        # то i - j + 1 - позиция вхождения pattern в text.
        print('Подстрока найдена в позиции', i - j + 1)
        j = d[j - 1]
```

Рассмотрим на примере работу КМП-алгоритма.
Пусть заданы следующие данные

```
text = 'колоколуколокола'
pattern = 'колокола'
```

Последовательность d будет иметь следующие значения

d =	0	0	0	0	1	2	3	0
j =	0	1	2	3	4	5	6	7

Поиск pattern в text:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
text	к	о	л	о	к	о	л	у	к	о	л	о	к	о	л	а
pattern	к	о	л	о	к	о	л	а								
j	0	1	2	3	4	5	6	7								
d	0	0	0	0	1	2	3	0								

↑ ↑

несовпадение элементов
величина сдвига: j = d[j - 1] # j = 3

text	к	о	л	о	к	о	л	у	к	о	л	о	к	о	л	а
pattern					к	о	л	о	к	о	л	а				
j								3	4	5	6	7				
d					0	0	0	0	1	2	3	0				

↑

несовпадение элементов
величина сдвига: j = d[j - 1] # j = 0

text	к	о	л	о	к	о	л	у	к	о	л	о	к	о	л	а
pattern								к	о	л	о	к	о	л	а	
j								0	1	2	3	4	5	6	7	
d								0	0	0	0	1	2	3	0	

↑

несовпадение элементов
j = 0; i +=1

text	к	о	л	о	к	о	л	у	к	о	л	о	к	о	л	а
pattern									к	о	л	о	к	о	л	а
j									0	1	2	3	4	5	6	7
d									0	0	0	0	1	2	3	0

В результате будет достигнут конец pattern и text, а также найдено наличие подстроки в строке, начиная с позиции i – j + 1.

КМП-алгоритм в худшем случае требует len(text) + len(pattern) сравнений и дает хорошие результаты, если несовпадению предшествует некоторое число совпадений.

1.2 Алгоритм Бойера-Мура-Хорспула

Алгоритм Бойера-Мура-Хорспула (БМХ-алгоритм, 1980 г.) является упрощённым алгоритмом поиска подстроки в строке Бойера-Мура (см. https://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm). Рассмотрим особенности БМХ-алгоритма:

1. БМХ-алгоритм выполняет сканирование text слева направо, а сравнение элементов text с элементами pattern справа налево, в отличие от КМП-алгоритма.

Сравнение начинается с последнего символа pattern и символа text, индекс которого (i) равен длине pattern. Если символы совпадают, то выполняется сравнение предпоследнего символа pattern и соответствующего (i - 1) символа text и т.д. Если все символы pattern совпали со сравниваемыми символами text, то pattern содержится в text. При несовпадении символа pattern с соответствующим символом text, pattern смещается вправо на некоторое количество символов.

Величина сдвига определяется с помощью последовательности стоп-символов d. Рассмотрим свойства последовательности d:

1. Размер d равен мощности алфавита text и pattern (необходимо учитывать кодировку символов).
2. Каждый элемент d равен целому числу; максимальное значение, которое принимает элемент d, равно длине pattern. Именно этим значением изначально заполняются все элементы последовательности на первом шаге формирования d.
3. Значения всех элементов d, которые соответствуют символам, фактически принадлежащим pattern, на втором шаге корректируются (до этого все элементы последовательности равны длине pattern). Новое значение каждого такого элемента – это расстояние от самого правого в подстроке вхождения символа до конца подстроки. Для последнего символа в pattern корректировка не выполняется и этот элемент остается равным длине pattern. Если какой-то символ встретился в pattern несколько раз, то его корректировка (уменьшение значения) будет выполнена столько раз, сколько раз этот символ встретился в pattern (кроме, конечно, последнего элемента pattern).

Рассмотрим реализацию алгоритма построения последовательности d на языке Python.

```
# Объявление и инициализация text и pattern.

...

# Длина text.
n = len(text)
# Длина pattern.
m = len(pattern)
# Объявление и инициализация d. \
# Размер d равен мощности алфавита text и pattern.
d = [m] * (ord(max(max(text), max(pattern))) + 1)
# Для последнего символа pattern корректировка не выполняется. /
# Элемент d[m - 1] остается равным m.
for i in range(m - 1):
    d[ord(pattern[i])] = m - i - 1
```

Сформируем последовательность d для pattern = 'CCTTTTGCA'

индекс d	d[ord('C')]	d[ord('T')]	d[ord('G')]	d[ord('A')]	любой другой индекс d
элемент d	1	3	2	9	9

Рассмотрим реализацию БМХ-алгоритма на языке Python.

```
pos = -1
i = 0
while n - i >= m and pos == -1:
    # Индекс pattern равен индексу последнего элемента.
    j = m - 1
    # Встретилось совпадение элемента text и pattern.
    while text[i + j] == pattern[j]:
        # Если индекс pattern равен нулю, то поиск закончен.
        # i - позиция первого вхождения pattern в text.
        if j == 0:
            pos = i
            break
        # Уменьшаем индекс pattern на 1, \
        # движемся "справа-налево".
        j -= 1
    # Если встретилось несовпадение, то pattern смещается \
    # вправо на расстояние, рассчитываемое по правилу \
    # стоп-символа (text[i+j]): если text[i+j] в pattern нет, \
    # то pattern смещается за этот символ; если text[i+j] в \
    # pattern присутствует, то pattern смещается на \
    # d[ord(text[i + m - 1])] символов.
    i += d[ord(text[i + m - 1])]
if pos != -1:
    print('Подстрока найдена в позиции', pos)
else:
    print('Элемент не найден.')
```

Рассмотрим на примере работу БМХ-алгоритма.
Пусть заданы следующие данные

```
text = 'колоколуколокола'
pattern = 'колокола'
```

Последовательность d будет иметь следующие значения

индекс d	d[ord('к')]	d[ord('о')]	d[ord('л')]	d[ord('а')]	любой другой индекс d
элемент d	3	2	1	8	8

Поиск pattern в text:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
text	к	о	л	о	к	о	л	у	к	о	л	о	к	о	л	а
pattern	к	о	л	о	к	о	л	а								
j	0	1	2	3	4	5	6	7								
	↑							↑								

несовпадение элементов
величина сдвига: i += d[ord('y')] # i = 8

text	к	о	л	о	к	о	л	у	к	о	л	о	к	о	л	а
pattern									к	о	л	о	к	о	л	а
j									0	1	2	3	4	5	6	7

Эффективность алгоритма БМХ существенно повышается за счет сравнения с конца pattern – в случае несовпадения может происходить смещение на длину всей pattern. Благодаря d, который фиксирует структуру pattern, учитываются повторы символов в pattern и сокращается время на вычисление нового значения индекса text.

В среднем БМХ-алгоритм требует меньше n + m сравнений.

2 Приближенный поиск подстроки в строке

Приближенный поиск pattern в text основан на вычислении расстояния между pattern и text. В общем случае расстояние между pattern и text – метрика, определяющая взвешенное количество стандартных операций редактирования, необходимых для преобразования pattern в text. Выделяют следующие операции редактирования: вставка, удаление, подстановка и транспозиция.

Существует несколько популярных типов расстояний между строками:

1. Расстояние Хемминга (см. https://en.wikipedia.org/wiki/Hamming_distance) – минимальное количество подстановок, необходимых для преобразования pattern в text (pattern и text должны быть одинаковой длины).
2. Расстояние Левенштейна (см. https://en.wikipedia.org/wiki/Levenshtein_distance) – минимальное количество операций вставки, удаления и подстановки, необходимых для преобразования pattern в text.
3. Расстояние Дамерау — Левенштейна (см. https://en.wikipedia.org/wiki/Damerau-Levenshtein_distance) – минимальное количество операций вставки, удаления, подстановки и транспозиции, необходимых для преобразования pattern в text (является модификацией расстояния Левенштейна).

Рассмотрим подробнее расстояние Левенштейна.

Основные области применения расстояния Левенштейна:

- проверка орфографии (в поисковых системах, базах данных, при вводе текста и т.д.);
- сравнение текстовых файлов утилитой diff (https://en.wikipedia.org/wiki/Diff_utility) и ей подобными;
- сравнение генов, хромосом и белков в биоинформатике.

Основные недостатки расстояния Левенштейна:

- при перестановке местами слов или частей слов получаются сравнительно большие расстояния;
- расстояния между совершенно разными короткими словами оказываются небольшими, в то время как расстояния между очень похожими длинными словами оказываются значительными.

Расстояние Левенштейна определяется с помощью матрицы расстояний Левенштейна d. Матрица d может быть построена с помощью алгоритма Вагнера-Фишера (см. https://en.wikipedia.org/wiki/Wagner-Fischer_algorithm).

Рассмотрим реализацию расстояния Левенштейна на языке Python.

```
text = input('Text = ')
pattern = input('Pattern = ')
```

```
    n, m = len(text), len(pattern)
    if n > m:
        # n <= m
        text, pattern = pattern, text
        n, m = m, n

    current = range(n + 1)
    for i in range(1, m + 1):
        previous, current = current, [i] + [0] * n
        for j in range(1, n + 1):
            add, delete, change = previous[j] + 1, current[j-1] + 1,
previous[j-1]
            if text[j-1] != pattern[i-1]:
                change += 1
            current[j] = min(add, delete, change)

    print('Levenshtein distance is', current[n])
```