Алгоритм Бойера — Мура

Материал из Википедии — свободной энциклопедии [править | править код]

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 30 июля 2018 года; проверки требуют 28 правок.

Эта статья об алгоритме поиска подстроки; об алгоритме поиска преобладающего элемента последовательности см. Алгоритм большинства голосов Бойера — Мура.

Алгоритм Бойера — Мура	
Назван в честь	Robert S. Boyer[d] и J Strother Moore[d]
Автор	Robert S. Boyer[d] и J Strother Moore[d]
Предназначение	Эффективный поиск подстроки в строке
Структура данных	Строки
Худшее время	$\Theta(n+m)$
Затраты памяти	$\Theta(m+\vert\Sigma\vert)$ или $\Theta(m)$
Медиафайлы на Викискладе	

Алгоритм поиска строки Бойера — Мура — алгоритм общего назначения, предназначенный для поиска подстроки в строке. Разработан Робертом Бойером (рус.) (англ.) и Джеем Муром (рус.) (англ.) в 1977 году[1]. Преимущество этого алгоритма в том, что ценой некоторого количества предварительных вычислений над шаблоном (но не над строкой, в которой ведётся поиск), шаблон сравнивается с исходным текстом не во всех позициях — часть проверок пропускается как заведомо не дающая результата.

Общая оценка вычислительной сложности современного варианта алгоритма Бойера — Мура — $O(n+m)$, если не используется таблица стоп-символов (смотрите ниже), и $O(n+m+|\Sigma|)$, если используется таблица стоп-символов, где n — длина строки, в которой выполняется поиск, m — длина шаблона поиска, Σ — алфавит, на котором проводится сравнение[2].

Содержание [скрыть]

Описание алгоритма [править | править код]

Алгоритм основан на трёх идеях.

1. Сканирование слева направо, сравнение справа налево. Совмещается начало текста (строки) и шаблона, проверка начинается с последнего символа шаблона. Если символы совпадают, производится сравнение предпоследнего символа шаблона и т. д. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и выполняется поиск следующего вхождения подстроки.

Если же какой-то символ шаблона не совпадает с соответствующим символом строки, шаблон сдвигается на *несколько* символов вправо, и проверка снова начинается с последнего символа.

Эти «несколько», упомянутые в предыдущем абзаце, вычисляются по двум эвристикам.

2. Эвристика стоп-символа. (*Замечание:* эвристика стоп-символа присутствует в большинстве описаний алгоритма Бойера — Мура, включая оригинальную статью Бойера и Мура[1], но не является необходимой для достижения оценки $O(n+m)$ времени работы[2]; более того, данная эвристика для своей работы требует $O(|\Sigma|)$ дополнительной памяти и $O(|\Sigma|)$ дополнительного времени на этапе подготовки шаблона.)

Предположим, что мы производим поиск слова «колокол». Первая же буква не совпала — «к» (назовём эту букву *стоп-символом*). Тогда можно сдвинуть шаблон вправо до *последней* его буквы «к».

```
Строка:          * * * * * * к * * * * * *
Шаблон:          к о л о к о л
Следующий шаг:       к о л о к о л
```

Если стоп-символа в шаблоне нет, шаблон смещается за этот стоп-символ.

```
Строка:          * * * * * а л * * * * * * * *
Шаблон:          к о л о к о л
Следующий шаг:               к о л о к о л
```

В данном случае стоп-символ — «а», и шаблон сдвигается так, чтобы он оказался прямо за этой буквой. В алгоритме Бойера — Мура эвристика стоп-символа вообще не смотрит на совпавший суффикс (см. ниже), так что первая буква шаблона («к») окажется под «л», и будет проведена одна заведомо холостая проверка.

Если стоп-символ «к» оказался за другой буквой «к», эвристика стоп-символа не работает.

```
Строка:         * * * * к к о л * * * * *
Шаблон:           к о л о к о л
Следующий шаг:  к о л о к о л
```

В таких ситуациях выручает третья идея алгоритма Бойера — Мура — эвристика совпавшего суффикса.

3. Эвристика совпавшего суффикса. Неформально, если при чтении шаблона справа налево совпал суффикс S, а символ b, стоящий перед S в шаблоне (то есть шаблон имеет вид PbS), не совпал, то эвристика совпавшего суффикса сдвигает шаблон на наименьшее число позиций вправо так, чтобы строка S совпала с шаблоном, а символ, предшествующий в шаблоне данному совпадению S, отличался бы от b (если такой символ вообще есть). Формально, для данного шаблона $s[0..m-1]$ считается целочисленный массив suffshift[0..m], в котором suffshift[i] равно минимальному числу $j>0$, такому что $s[i-j]\neq s[i-1]$ (если $i>0$ и $i-j\geq 0$) и $s[i-j+k]=s[i-1+k]$ для любого $k>0$, для которого выполняется $0\leq i-j+k<m$ и $0\leq i-1+k<m$ (для пояснения смотрите примеры ниже). Затем, если при чтении шаблона s справа налево совпало $k-1$ символов $s[m-1],s[m-2],\ldots,s[m-k+1]$, а символ $s[m-k]$ не совпал, то шаблон сдвигается на suffshift[m-k] символов вправо. Например:

```
Строка:          * * * * * * р к а * * * * *
Шаблон:          с к а л к а л к а
Следующий шаг:               с к а л к а л к а
```

В данном случае совпал суффикс «ка», и шаблон сдвигается вправо до ближайшего «ка», перед которым нет буквы «л».

```
Строка:          * * т о к о л * * * * *
Шаблон:          к о л о к о л
Следующий шаг:           к о л о к о л
```

В данном случае совпал суффикс «окол», и шаблон сдвигается вправо до ближайшего «окол», перед которым нет буквы «л». Если подстроки «окол» в шаблоне больше нет, но он начинается на «кол», сдвигается до «кол», и т. д.

Внимание: несовпадение буквы перед ближайшим вхождением совпавшего суффикса является необходимым условием. Если ограничиться только сдвигом до ближайшего вхождения совпавшего суффикса, то алгоритм может работать неприемлемо медленно. Например, при поиске шаблона $cababa\ldots ba$ длины $2n$ в строке $aaa\ldots aababab\ldots ba$, содержащей $2n$ символов «а», за которыми следует строка $baba\ldots ba$ длины $2n$, алгоритм, использующий сдвиги без учёта несовпадения символа, выполняет $O(n^2)$ операций даже при использовании эвристике стоп-символов. С другой стороны доказано[2], что время работы алгоритма БМ, учитывающего несовпадение символов (то есть использующего массив suffshift, определённый выше), равно $O(n+m)$ даже без использования эвристики стоп-символов (данное в книге М. Крошмора и В. Риттера[2] доказательство этого факта является модификацией доказательства Коула 1994 года[3], рассмотревшего только случай непериодических шаблонов).

Обе эвристики требуют предварительных вычислений — в зависимости от шаблона поиска заполняются две таблицы. Таблица стоп-символов по размеру соответствует алфавиту — $O(|\Sigma|)$ (например, если алфавит состоит из 256 символов, то её длина 256); таблица суффиксов — искомому шаблону, то есть $O(m)$.

Опишем подробнее обе таблицы.

Таблица стоп-символов [править | править код]

В таблице стоп-символов указывается последняя позиция в шаблоне s (**исключая последнюю букву**) каждого из символов алфавита. Для всех символов, не вошедших в s, пишем 0, если нумерация символов начинается с 1 (или −1, если нумерация начинается с 0). Например, если $s=abcdadcd$, таблица стоп-символов StopTable будет выглядеть так (таблица приведена для случая строки, нумеруемой с нуля; при нумерации символов с единицы нужно прибавить ко всем числам единицу):

```
Символ             a  b  c  d  [все остальные]
Последняя позиция  4  1  6  5  -1
```

Обратите внимание, для стоп-символа «d» последняя позиция будет 5, а не 7 — последняя буква не учитывается. Это известная ошибка, приводящая к неоптимальности. Для алгоритма БМ она не фатальна (спасает положение эвристика суффикса), но фатальна для упрощённой версии алгоритма БМ — алгоритма Хорспула.

Если при сравнении справа налево шаблона $s[0..m-1]$ со строкой $text[i..i+m-1]$ несовпадение произошло в позиции j, а стоп-символ — $c=text[j]$, то шаблон необходимо сдвинуть на $i'=j-StopTable[c]$ символов.

Таблица суффиксов [править | править код]

Для каждого возможного суффикса t данного шаблона s указываем наименьшую величину, на которую нужно сдвинуть вправо шаблон, чтобы он снова совпал с t и при этом символ, предшествующий этому вхождению t, не совпадал бы с символом, предшествующим суффиксу t. Если такой сдвиг невозможен, ставится $|s|=m$ (в обеих системах нумерации). Например, для $s=aaccbccbcc$ будет:

```
Суффикс        [пустой]         c        cc        bcc             ...   aaccbccbcc
Сдвиг              2            1         6         10             ...           10
Иллюстрация
    было           ?           ?c       ?cc       ?bcc             ...   aaccbccbcc
    стало    aaccbccbcc aaccbccbcc    aaccbccbcc     aaccbccbcc    ...             aaccbccbcc
```

Для $s=$«колокол»:

```
Суффикс       [пустой]        л          ол         кол       ...  олокол        колокол
Сдвиг             1           7           7          4         ...       4              4
Иллюстрация
    было          ?          ?л         ?ол        ?кол       ... ?олокол        колокол
    стало     колокол          колокол    колокол  колокол    ...      колокол        колокол
```

Существует алгоритм вычисления таблицы суффиксов suffshift[0..m] с временем работы $O(m)$.[2] Этот алгоритм основан на тех же идеях, что и алгоритмы вычисления префикс-функции и Z-функции строки[4][5]. Реализация данного алгоритма на C++ выглядит следующим образом:

```
std::vector<int> suffshift(m + 1, m);
std::vector<int> z(m, 0);
for (int j = 1, maxZidx = 0, maxZ = 0; j < m; ++j) {
  if (j <= maxZ) z[j] = std::min(maxZ - j + 1, z[j - maxZidx]);
  while (j + z[j] < m && s[m - 1 - z[j]] == s[m - 1 - (j + z[j])]) z[j]++;
  if (j + z[j] - 1 > maxZ) {
    maxZidx = j;
    maxZ = j + z[j] - 1;
  }
}
for (int j = m - 1; j > 0; j--) suffshift[m - z[j]] = j; //цикл №1
for (int j = 1, r = 0; j <= m - 1; j++) //цикл №2
  if (j + z[j] == m)
    for (; r <= j; r++)
      if (suffshift[r] == m) suffshift[r] = j;
```

В первом цикле в коде воспроизведён код вычисления так называемой Z-функции $z[1..m-1]$, но для перевёрнутой строки $s[0..m-1]$.[5] А именно, для любого j, такого что $0\leq j<m-1$, элемент $z[m-1-j]$ содержит длину длиннейшего суффикса строки $s[0..j]$, который также является суффиксом всей строки s. С помощью массива z далее вычисляется искомый массив suffshift[0..m] (смотрите описание ниже). На каждой итерации последнего цикла j уменьшается на 1, а на каждой итерации вложенного в него цикла k уменьшается на 1. Поэтому, так как $j\geq 0$, $k\geq 0$, и алгоритм вычисления Z-функции работает за $O(m)$ (смотрите, например, соответствующую статью, а также статью[5]), общее время работы данного алгоритма — $O(m)$.

Для доказательства корректности представленного кода удобно представлять себе, что анализируется строка $s'[0..m-1]$, которая равна перевёрнутой строке s. По определению suffshift, имеем suffshift[$m-k$]= j, где j — это наименьшее положительное число, такое что либо 1) строка $s'[j..m-1]$ является префиксом строки $s'[0..k-1]$, либо 2) строка $s'[j..j+k-1]$ равна $s'[0..k-1]$, а символы $s'[j+k]$ и $s'[k]$ отличаются. В случае 2), по определению z, обязательно выполняется $z[j]=k$. Таким образом, пробегая по j от $m-1$ до 1, цикл № 1 находит все значения suffshift, полученные по правилу 2). Для вычисления значений suffshift, полученных по правилу 1), заметим, что, во-первых, если $s'[j..m-1]$ — префикс $s'[0..k-1]$, то обязательно выполняется $j+z[j]=m$, а во-вторых, если suffshift[r] = j для какого-то r, то обязательно $r\leq j$. Опираясь на эти два наблюдения, цикл № 2 вычисляет оставшиеся неинициализированными значения suffshift (то есть такие что suffshift[k] = m).

Реализация алгоритма [править | править код]

Пусть массив сдвигов suffshift[0..m] для данного шаблона $s[0..m-1]$ посчитан. Тогда реализация на C++ алгоритма Бойера — Мура для поиска первого вхождения s в тексте $text[0..n-1]$ за $O(n+m)$ времени без применения эвристики стоп-символов выглядит следующим образом[2]:

```
for (int i = 0, j = 0; i <= n - m && j >= 0; i += suffshift[j+1]) {
  for (j = m - 1; j >= 0 && s[j] == text[i + j]; j--);
  if (j < 0) report_occurrence(i);
}
```

Такой алгоритм непригоден для поиска всех вхождений шаблона в текст за $O(m+n)$ времени. Если убрать условие «j >= 0» из внешнего цикла, то алгоритм найдёт все вхождения, но в худшем случае, возможно, выполнит $O(mn)$ операций, в чём легко убедиться, рассмотрев строку, состоящую из одних букв «a». Для поиска всех вхождений используют следующую модификацию, которая работает $O(n+m)$ времени за счёт так называемого правила Галиля[6]:

```
int j, bound = 0; //всегда либо bound = 0, либо bound = m - suffshift[0]
for (int i = 0; i <= n - m; i += suffshift[j+1]) {
  for (j = m - 1; j >= bound && s[j] == text[i + j]; j--);
  if (j < bound) {
    report_occurrence(i);
    bound = m - suffshift[0];
    j = -1; //установить j так, как будто мы прочитали весь шаблон s, а не только до границы bound
  } else {
    bound = 0;
  }
}
```

Правило Галиля основано на следующем несложном наблюдении. Если вхождение s найдено в позиции i, то следующий поиск будет пытаться найти вхождение шаблона в позиции $i'=i+$suffshift[0] и, по определению suffshift, уже известно, что символы $s[i'],s[i'+1],\ldots,s[i+m-1]$ совпадают с символами $s[0],s[1],\ldots,s[m-$suffshift[0]$-1]$. Значит, при выполнении поиска справа налево для определения того, есть ли вхождение шаблона в позиции i', нет смысла проверять символы $s[0],s[1],\ldots,s[m-$suffshift[0]$-1]$. Именно для этого и служит переменная «bound». Доказано, что такая эвристика помогает получить $O(n+m)$ времени для поиска всех вхождений шаблона в строку[6].

Для включения эвристики стоп-символов достаточно строку « `i += suffshift[j+1]` » заменить на следующее выражение в конце основного цикла:

```
if (j < bound) i += suffshift[j+1];
else i += max(suffshift[j+1], j - StopTable[text[i + j]]);
```

Пример работы алгоритма [править | править код]

Искомый шаблон — « `abbad` ». Таблица стоп-символов будет выглядеть как (в данном примере будет удобнее пользоваться нумерацией с единицы)

```
Символ   a  b  d  [остальные]
Позиция  4  3  5  0
```

Таблица суффиксов для всех возможных суффиксов (кроме пустого) даёт максимальный сдвиг — 5.

```
abeccaabadbabbad
abbad
```

Накладываем образец на строку. Совпадения суффикса нет — таблица суффиксов даёт сдвиг на одну позицию. Для несовпавшего символа исходной строки « `c` » (5-я позиция) в таблице стоп-символов записан 0. Сдвигаем образец вправо на `5-0=5` позиций.

```
abeccaabadbabbad
     abbad
```

Символы 3—5 совпали, а второй — нет. Эвристика для « `a` » не работает (`2-4=-2`). Но поскольку часть символов совпала, в дело включается эвристика совпавшего суффикса, сдвигающая шаблон сразу на пять позиций!

```
abeccaabadbabbad
          abbad
```

И снова совпадения суффикса нет. В соответствии с таблицей стоп-символов сдвигаем образец на одну позицию и получаем искомое вхождение образца:

```
abeccaabadbabbad
           abbad
```

Доказательство корректности [править | править код]

Для доказательства корректности алгоритма достаточно показать, что если та или иная эвристика предлагает сдвиг более чем на одну позицию вправо, на пропущенных позициях шаблон не найдётся.

Итак, пусть совпал суффикс *S*, строка-шаблон равна *PbS*, стоп-символ — *a* (в дальнейшем малые буквы — символы, большие — строки).

```
Строка:     * * * * * * * a [-- S --] * * * *
Шаблон:       [--- P ---] b [-- S --]
```

Эвристика стоп-символа. Работает, когда в строке *V* символ *a* отсутствует. Если *P*=*WaV* и в строке *V* нет символа *a*, то эвристика стоп-символа предлагает сдвиг на |*V*|+1 позицию.

```
Строка:     * * * * * * * * * * * * a [-- S --] * * * * * * * *
Шаблон:       [- W -] a [--- V ---] b [-- S --]
Пропустить:             [- W -] a [--- V ---] b [-- S --]
Новый шаг:                  [- W -] a [--- V ---] b [-- S --]
```

Действительно, если в строке *V* нет буквы *a*, нечего пробовать пропущенные |*V*| позиций.

Если же в шаблоне нет символа *a*, то эвристика стоп-символа предлагает сдвиг на |*P*|+1 позицию — и также нет смысла пробовать пропущенные |*P*|.

```
Строка:     * * * * * * * * a [-- S --] * * * * * * * *
Шаблон:         [--- P ---] b [-- S --]
Пропустить:         [--- P ---] b [-- S --]
Новый шаг:                    [--- P ---] b [-- S --]
```

Эвристика совпавшего суффикса. Сама неформальная фраза — «наименьшая величина, на которую нужно сдвинуть вправо шаблон, чтобы он снова совпал с S, но символ перед данным совпадением с S (если такой символ существует) отличался бы от b» — говорит, что меньшие сдвиги бесполезны.

Варианты [править | править код]

Алгоритм Бойера — Мура — Хорспула [править | править код]

Алгоритм Бойера — Мура — Хорспула (АБМХ) работает лучше алгоритма Бойера — Мура (АБМ) на *случайных* текстах — для него оценка в среднем лучше.

АБМХ использует только эвристику стоп-символа; при этом за стоп-символ берётся символ входной строки, который соответствует последнему символу шаблона, **независимо от того, где случилось несовпадение**.

Поскольку реальные поисковые образцы редко имеют равномерное распределение, АБМХ может дать как выигрыш, так и проигрыш по сравнению с АБМ.

Алгоритм Чжу — Такаоки [править | править код]

На коротких алфавитах (например, при сравнении участков ДНК алфавит состоит всего из четырёх символов: А, Т, Г, Ц) эвристика стоп-символа отказывает уже на коротких суффиксах. Простейший способ улучшить работу АБМ в таких условиях — вместо одного стоп-символа строить таблицу для пары символов: несовпавшего и идущего перед ним[7]. Такой алгоритм был назван алгоритмом Чжу — Такаоки.

На предварительную обработку расходуется $O(m+|\Sigma|^2)$ времени.

Алгоритм «турбо-Бойера — Мура» [править | править код]

Алгоритм «турбо-Бойера — Мура» разработан группой учёных во главе с М. Крошмором, предлагает другой подход к коротким алфавитам и заодно решает вторую проблему — квадратичную сложность в худшем случае.

Помимо эвристики стоп-символа и эвристики совпавшего суффикса, применяется третья эвристика — **эвристика турбосдвига**[8].

Пусть первый раз совпал суффикс *UV* (и сработала эвристика суффиксов, обеспечив полное перекрытие этого суффикса), второй раз — более короткий *V* (возможно, *V*=∅).

```
                                              !                      !
входная строка:      * * * * * * * * * * # [-U-] [V] * * * * * * * * # [V] * * * * * *
1. Совпало UV:         * [-U-] [V] * * * * [-U-] [V]
2. Затем совпало V:                      * [-U-] [V] * * * * * * [-U-] [V]
Сдвиг по эвристике суффиксов:                * [-U-] [V] * * * * * * [-U-] [V]
Турбосдвиг:                                    * [-U-] [V] * * * * * * [-U-] [V]
```

По рисунку видно, что минимально возможный сдвиг — |*U*|. В противном случае два символа, обозначенные восклицательными знаками, во входной строке разные, а в шаблоне одинаковые. В этом и заключается эвристика турбосдвига.

Алгоритм выполняет свою работу за $2n$ сравнений до первого совпадения в худшем случае.

Сравнение с другими алгоритмами [править | править код]

Достоинства [править | править код]

Алгоритм Бойера — Мура на «хороших» данных очень быстр[уточнить], а вероятность появления «плохих» данных крайне мала. Поэтому он оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск[9]. Разве что на коротких текстах выигрыш не оправдает предварительных вычислений.

Недостатки [править | править код]

Алгоритмы семейства Бойера — Мура не расширяются до приблизительного поиска, поиска любой строки из нескольких.

Сравнение не является «чёрным ящиком» (только если применяется эвристика стоп-символа), поэтому при реализации наиболее быстрого поиска приходится либо рассчитывать на удачную работу оптимизатора, либо вручную оптимизировать поиск на языке ассемблера.

Если текст изменяется редко, а поиск проводится часто (например, поисковой машиной), можно провести индексацию текста. Алгоритм поиска по индексу быстрее[уточнить] алгоритма Бойера — Мура.

На больших алфавитах (например, Юникод) таблица стоп-символов может занимать много памяти. В таких случаях либо обходятся хеш-таблицами, либо дробят алфавит, рассматривая, например, 4-байтовый символ как пару двухбайтовых, либо (что проще всего) пользуются вариантом алгоритма Бойера — Мура без эвристики стоп-символов.

Существует ряд модификаций алгоритма Бойера — Мура, нацеленных на ещё большее ускорение — например, турбо-алгоритм, обратный алгоритм Колусси[10] и другие.

См. также [править | править код]

- Алгоритм Кнута — Морриса — Пратта
- Алгоритм Рабина — Карпа
- Алгоритм Ахо — Корасик

Примечания [править | править код]

1. ↑ 1 2 Boyer, Moore, 1977.
2. ↑ 1 2 3 4 5 6 Crochemore, Rytter, 2002.
3. ↑ Cole, 1994.
4. ↑ Гасфилд, 2003.
5. ↑ 1 2 3 MAXimal :: algo :: Z-функция строки и её вычисление. Дата обращения: 14 марта 2017. Архивировано 26 апреля 2017 года.
6. ↑ 1 2 Galil, 1979.
7. ↑ Zhu-Takaoka algorithm Архивная копия от 16 декабря 2008 на Wayback Machine (англ.)
8. ↑ Turbo-BM algorithm Архивная копия от 16 декабря 2008 на Wayback Machine (англ.)
9. ↑ Exact string matching algorithms — Introduction Архивная копия от 16 декабря 2008 на Wayback Machine (англ.)
10. ↑ Reverse Colussi algorithm Архивная копия от 9 марта 2016 на Wayback Machine (англ.)

Литература [править | править код]

Имеется викиучебник по теме «Алгоритм Бойера — Мура»

- *Кормен Т. Х., Лейзерсон Ч. Е., Ривест Р. Л., Штайн К.* Алгоритмы: построение и анализ = Introduction to Algorithms / под ред. С. Н. Тригуба; пер. с англ. И. В. Красиков, Н. А. Орехов, В. Н. Романов. — 2-е изд. — М.: Вильямс, 2005. — 801 с. — ISBN 5-8459-0857-4.
- *Crochemore M., Rytter W.* Jewels of Stringology. — Singapore: World Publishing Scientific Co. Pte. Ltd., 2002. — 310 с. — ISBN 981-02-4782-6.
- *Boyer R. S., Moore J. S.* A fast string searching algorithm // Communications of the ACM. — 1977. — Т. 20, № 10. — С. 762—772. — doi:10.1145/359842.359859.
- *Cole R.* Tight bounds on the complexity of the Boyer-Moore string matching algorithm // SIAM Journal on Computing. — 1994. — Т. 23, № 5. — С. 1075—1091. — doi:10.1137/S0097539791195543.
- *Galil Z.* On improving the worst case running time of the Boyer-Moore string matching algorithm // Communications of the ACM. — 1979. — Т. 22, № 9. — С. 505—508. — doi:10.1145/359146.359148.
- *Гасфилд Д.* Строки, деревья и последовательности в алгоритмах: информатика и вычислительная биология = Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology / пер. с англ. И. В. Романовский. — СПб.: Невский Диалект, 2003. — 654 с. — ISBN 5-7940-0103-8.

Строки [скрыть]	
Меры схожести строк	Расстояние Дамерау — Левенштейна • Расстояние Левенштейна • Расстояние Хэмминга • Сходство Джаро — Винклера
Поиск подстроки	**Алгоритм Бойера — Мура** • Алгоритм Бойера — Мура — Хорспула • Алгоритм Кнута — Морриса — Пратта • Алгоритм Рабина — Карпа • Префикс-функция • Z-функция • Алгоритм Ахо — Корасик
Палиндромы	Дерево палиндромов • Алгоритм Манакера
Выравнивание последовательностей	Алгоритм Нидлмана — Вунша • Алгоритм Смита — Ватермана
Суффиксные структуры	Суффиксный массив • Суффиксный автомат • Суффиксное дерево • Префиксное дерево
Другое	Синтаксический анализ • Сопоставление с образцом • Наибольшая общая подпоследовательность • Наибольшая общая подстрока

Категории: Поиск подстроки | Эвристические алгоритмы