rmq 23 октября 2013 в 16:19

Алгоритм Ахо-Корасик

Программирование, C++, Алгоритмы

Из песочницы

Вступление

В посте я постарался избежать сложных дефиниций и строгих мататематических доказательств, а некоторые вещи вообще понятны интуитивно. Алгоритм удобно разбивается на взаимосвязные части, поэтому и уловить принцип его работы не должно составлять труда.

Начальное описание

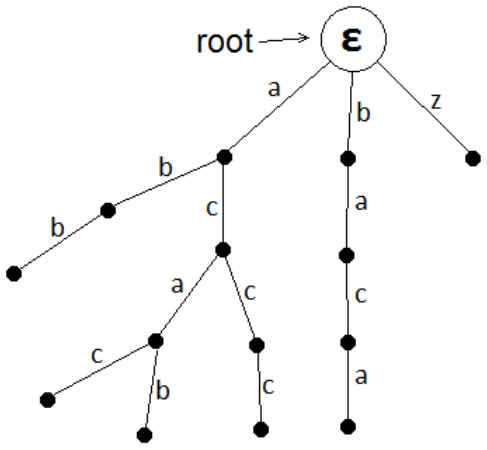
Алгоритм Ахо-Корасик реализует эффективный поиск всех вхождений всех строк-образцов в заданную строку. Был разработан в 1975 году Альфредом Ахо и Маргарет Корасик.
Опишем формально условие задачи. На вход поступают несколько строк pattern[i] и строка s. Наша задача — найти все возможные вхождения строк pattern[i] в s.

Суть алгоритма заключена в использование структуры данных — **бора** и построения по нему **конечного детерминированного автомата**. Важно помнить, что задача поиска подстроки в строки тривиально реализуется за квадратичное время, поэтому для эффективной работы важно, чтоб все части Ахо-Корасика ассимптотически не превосходили линию относительно длинны строк. Мы вернемся к оценке сложности в конце, а пока поближе посмотрим на составляющие алгоритма.

Построение бора по набору строк-образцов

Структура бора

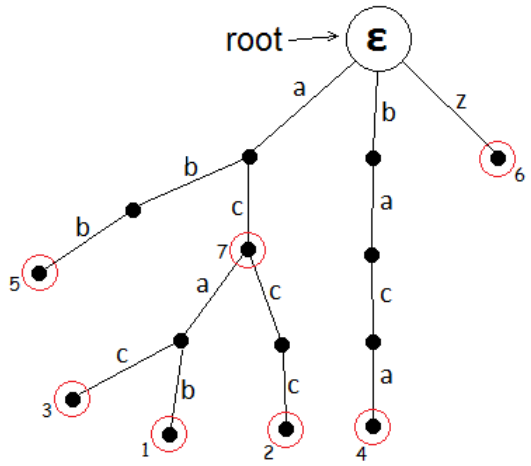
Что же такое бор? Строго говоря, бор — это дерево, в котором каждая вершина обозначает какую-то строку (корень обозначает нулевую строку — ε). На ребрах между вершинами написана 1 буква (в этом его принципиальное различие с суффиксными деревьями и др.), таким образом, добираясь по ребрам из корня в какую-нибудь вершину и контангенируя буквы из ребер в порядке обхода, мы получим строку, соответствующую этой вершине. Из определения бора как дерева вытекает также единственность пути между корнем и любой вершиной, следовательно — каждой вершине соответствует ровно одна строка (в дальнейшем будем отождествлять вершину и строку, которую она обозначает).
Строить бор будем последовательным добавление исходных строк. Изначально у нас есть 1 вершина, корень (root) — пустая строка. Добавление строки происходит так: начиная в корне, двигаемся по нашему дереву, выбирая каждый раз ребро, соответствующее очередной букве строки. Если такого ребра нет, то мы создаем его вместе с вершиной. Вот пример построенного бора для строк: 1)acab, 2)accc, 3)acac, 4)baca, 5)abb, 6)z, 7)ac.



(Рис. 1)

Обратите внимание на добавление строки 7. Она не создает новых вершин и ребер, а процесс ее добавления останавливается во

внутренней вершине. Отсюда видно, что для каждой строки необходимо дополнительно хранить признак того является она строкой из условия или нет (красные круги).



(Рис. 2)

Отметим также что, две строки в боре имеют общие ребра при условии наличия у них общего префикса. Крайний случай — все строки образцы попарно не имеют одинаковой начальной части. Значит верхняя оценка для числа вершин в боре — сумма длин всех строк + 1(корень).

Реализация

Будем хранить бор как массив вершин, где каждая вершина имеет свой уникальный номер, а корень имеет нулевое значение (root = 0). Возможное описание структуры вершины:

```
//k - размер алфавита
struct bohr_vrtx{
   int next_vrtx[k],pat_num;
   bool flag;
};
```

next_vrtx[i] — номер вершины, в которую мы придем по символу с номером i в алфавите, flag — бит, указывающий на то, является ли наша вершина исходной строкой, pat_num — номер строки-образца, обозначаемого этой вершиной.
Предподсчет длин всех добавляемых строк — лишние затраты по памяти. Будем использовать структуру данных из STL — vector. В нем память выделяется динамически, следовательно дополнительные затраты будут нулевыми. Явным образом вытекает процедура добавление строки (используем 26-буквенный строчный латинский алфавит => k=26).

Функции создания новой вершины и инициализации бора:

Процедура добавление строки-образца в бор:

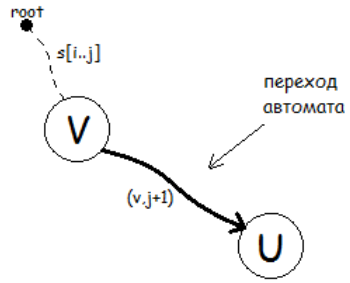
Проверка наличия строки в боре:

Построение автомата по бору

Описание принципа работы

Наша задача — построить конечный детерминированный автомат. Что за штука такая можно посмотреть здесь. Вкратце, состояние автомата — это какая-то вершина бора. Переход из состояний осуществляется по 2 параметрам — текущей вершине v и символу ch. по которорому нам надо сдвинуться из этой вершины. Поконкретнее, необходимо найти вершину u, которая обозначает наидлиннейшую строку, состоящую из суффикса строки v (возможно нулевого) + символа ch. Если такого в боре нет, то идем в корень.

Зачем это нам надо? Предположим, что мы можем вычислить такую вершину быстро, за константное время. Пусть, мы стоим в некой вершине бора, соответствующей подстроке [i..j] строки s, вхождения в которую мы ищем. Теперь найдем все строки бора, суффиксы s[i..j]. Утверждается, что их можно искать быстро (описано далее). После этого, просто перейдем из состояния автомата v в состояние u по символу s[j+1] и продолжим поиск.



(Рис. 3)

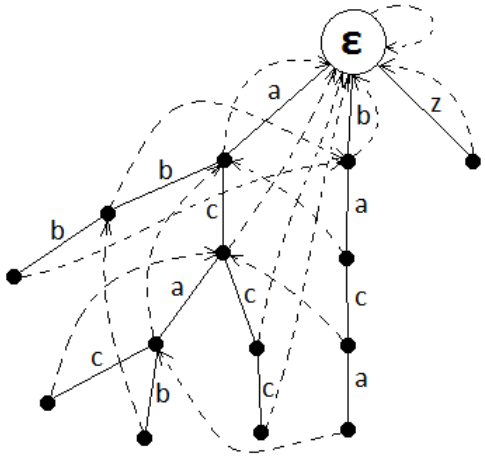
Для реализации автомата нам понадобится понятие суффиксной ссылки из вершины.

Суффиксные ссылки

Назовем суффиксной ссылкой вершины v указатель на вершину u, такую что строка u — наибольший собственный суффикс строки v, или, если такой вершины нет в боре, то указатель на корень. В частности, ссылка из корня ведет в него же. Нам понадобятся суффиксные ссылки для каждой вершины в боре, поэтому немного изменим структуру вершины и процедуру создание вершины, введя дополнительную переменную suff_link.

Изменения в коде:

Вот пример расстановки суф. ссылок для бора на рис. 1:

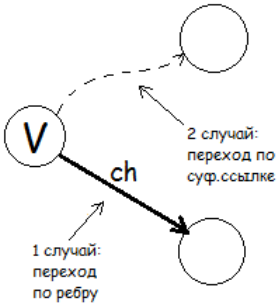


(Рис. 4)

Реализация автомата

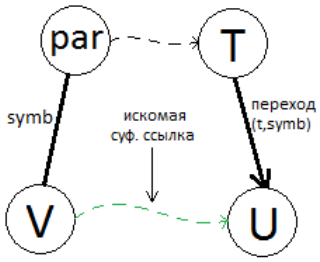
Вернемся к задача быстрого перехода между состояниями автомата. Очевидно, что всего возможных переходов существует bohr.size()*k, так как для для каждой возможной вершиной и каждым возможным символом в алфавите нужно знать переход. Предподсчет существенно снизит среднее время работы алгоритма, поэтому воспользуемся идеями ленивой динамики — будем

пройдем по нему, в обратном случаем пройдем по суффиксной ссылке и запустимся рекурсивно от новой вершины. Почему это работает, догадаться не трудно.



(Рис. 5)

Вопрос лишь в корректном получение суф. ссылки от вершины. В этой задаче тоже можно использовать ленивую динамику. Эвристика заключена в следующем: для получения суф. ссылки вершины v (строки s[i..j]) спустимся до ее предка par, пройдем по суф. ссылке par и запустим переход от текущей вершины t по символу symb, который написан на ребре от par до v. Очевидно, что сначала мы попадем в наибольший суфикс s[i..j-1] такой что, он имеет ребро с символом symb, потом пройдем по этому ребру. По определению, получившаяся вершина и есть суффикная ссылка из вершин v.



(Рис. 6)

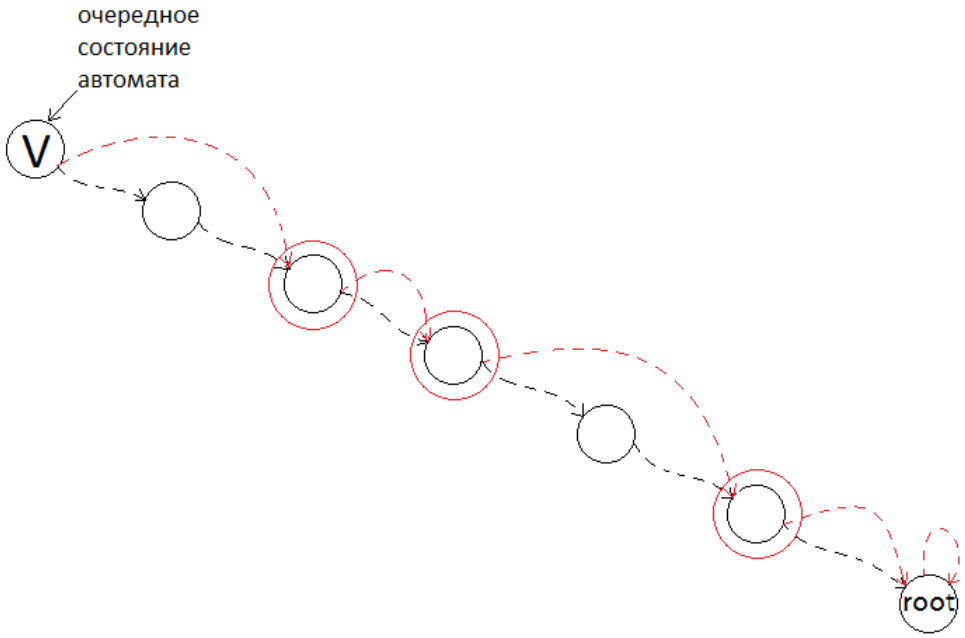
Итак, видно, что функции получения суффиксной ссылки и перехода из состояния автомата взаимосвязаны. Их удобная реализация представляет 2 функции, каждая из которых рекурсивно вызывает другую. База обоих рекурсий — суф. ссылка из корня или из сына корня ведет в корень.

Для начала изменим структуру вершины и процедуру создания новой вершины:

Полная реализация автомата требует предобъявления одной из функций:

Выявление «хороших» суффиксных ссылок

С автоматом несложно определить сам алгоритм: считываем строку, двигаемся из состояния в состояние по символам строки, в каждом из состояний двигаемся по суф. ссылками, то есть по суффиксам строки в позиции автомата, проверяя при этом наличие их в боре.
Все бы ничего, но оказывается, что этот вариант Ахо-Корасика имеет квадратичную ассимптотику относительно N — длинны считываемой строки s. Действительно, для строки из состояния v можно найти v.length() суффиксов, а переход из состояний может просто увеличивать на 1 длину этой строки. Цимес в том, чтобы двигаясь по суф. ссылкам попадать только в заведомо имеющиеся среди строк-образцов. Введем понятие «хороших» суф. ссылок suff_flink. Так, bohr[v].suff_flink — это ближайший суффикс, имеющийся в боре, для которого flag=true. Число «скачков» при использование таких ссылок уменьшится и станет пропорционально количеству искомых вхождений, оканчивающихся в этой позиции.



(Рис. 7)

Снова мальца изменим структуру вершины и процедуру добавления:

Вычислять их довольно просто, все той же ленивой динамикой. Введем функцию подсчета «хорошой» суф. ссылки. Если для вершине по суф. ссылке flag=true, то это и есть искомая вершина, в ином случае рекурсивно запускаемся от этой же вершины.

Вычисление хорошей суф. ссылки:

Реализация поиска по автомату

Поиск реализуется тривиально. Нам потребуется процедура хождения по «хорошим» суф. ссылкам check(v,i) из текущей позиции автомата v учитывая, что эта позиция оканчивается на i-ую букву в слове s.

check(v,i)

Вот и сам поиск:

А вот пример работы:

Оценка сложности и способы хранения

Существующий вариант алгоритм проходит циклом по длине s (N=s.length()), откуда его уже можно оценить как O(N*O(check)), но так как check прыгает только по заведомо помеченным вершинам, для которых flag=true, то общую ассимптотику можно оценить как O(N+t), где t — количество всех возможных вхождений всех строк-образцов в s. Если быть точным и учитывать вычисления автомата и суф. ссылок, то алгоритм работает O(M*k+N+t), где M=bohr.size(). Память — константные массивы размера k для каждой вершины бора, откуда и выливается оценка O(M*k).

Оказывается, другой способ хранение, а конкретно, обращения к алфавиту, способен изменить эту оценку. Будем использовать отображение map<char,int> вместо массива. Читаем здесь и здесь, видим, что структура данных map из STL реализована красно-черным деревом, а время обращения к его элементам пропорционально логарифму числа элементов. В нашем случае — двоичному логарифму размера алфавита k (что практически константа). Общее время — O((M+N)*log k+t). На практике, это значительнее быстрее массива. Map вовсе не хранит лишних ячеек памяти для элементов, поэтому память пропорциональна количеству ребер в боре (а следовательно и количеству вершин в боре, т.к. в дереве с M вершин — M-1 ребер). Количество вычислений переходов автомата, очевидно, пропорционально длине строки. Получившаяся оценка — O(M+N).