Поиск Абелевых подстрок

И. Збань

Version 0.1

Содержание

Введение

Важные определения и факты

$P(s)$ — вектор Парея, вектор частот символов.

$a \equiv b$, если $P(a) = P(b)$.

Абелевы квадраты

1 *Поиск числа абелевых квадратов*

2 *Реализация алгоритма*

Наибольшая общая абелева подстрока

Даны две строки $a, b \in \Sigma^n$. Нужно найти наидлиннейшую строку x такую, что $xs_a \equiv a, xs_b \equiv b$ для некоторых s_a, s_b.

1 *Предыдущий алгоритм*

2 *Общий алгоритм*

2.1 $(n^2 \log \Sigma, n \log \Sigma)$ w.h.p

Будем перебирать все длины l и проверять, есть ли общая абелева подстрока длины l. План: построить $P(t)$ для всех t таких, что $|t| = l$, и $a = xty$ для некоторых $x, y \in \Sigma^*$. После этого проверить, есть ли такая строка r, что $|r| = l, b = xty, P(r) = P(t)$ для некоторого t.

Будем строить векторы $P(t)$ для всех подстрок длины l строк a и b по очереди, переходя от одной подстроки к следующей. Для этого нужно уметь удалять первый символ текущей строки и дописывать в конец новый символ.

Хранить векторы $P(t)$ будем в персистентном массиве, реализованном на персистентном дереве отрезков. Для того, чтобы перейти к следующей строке, нужно уменьшить значение в одной ячейке на 1, и увеличить значение в другой ячейке на 1.

Для того, чтобы научиться сравнивать на равенство две вершины дерева, соответствующие двум векторам $P(t)$, будем при построении считать некоторый хеш от вершины. Будем поддерживать хешмап, в котором для пары хешей $< h_1, h_2 >$ хранится хеш от пары этих чисел, если она уже встречалась. Чтобы посчитать хеш для листа, проверим что-нибудь вроде пары $< -pos, val >$ а для внутренней вершины уже точно посчитаны хеши сыновей $< h(v_l), h(v_r) >$. Когда нам нужно узнать хеш пары $< h_1, h_2 >$, смотрим в мап: если там есть элемент с таким ключом, достаем оттуда соответствующий хеш, иначе кладем туда новый элемент с таким ключом и значением, равным размеру мапа. Значения всех хешей будет принимать значения от 0 до $MapSize - 1$.

Таким образом, после подсчета хеша каждой вершины, для всех подстрок длины l первой и второй строки можно выписать их хеши, и нужно проверить, есть ли в двух массивах одинаковое число. Поскольку все значения не превышают $n \log \Sigma$, это можно сделать используя сортировку подсчетом.

Время работы — n итераций по l, и $n \log \Sigma$ операций для каждой длины: каждая вершина дерева отрезков создается за $O(1)$ w.h.p. используя хешмап. Расходуемая память $n \log \Sigma$ на хранение дерева отрезков и хешмапа.

2.2 $(n^2 \log \Sigma, n^2)$ deterministic

Посмотрим внимательнее на персистентное дерево отрезков из предыдущего решения. Это ациклический орграф, в котором каждая вершина имеет свой уровень (глубину) от 1 до $\log \Sigma$, при чем на каждой глубине по $O(n)$ вершин.

Будем считать хеши всех вершин поднимаясь по уровням от листьев к корням, используя один хешмап размера $O(n^2)$, который умеем очищать за $O(1)$.

База: посчитать хеши от листьев. Хеш листа — $h(< -pos, val >)$, и pos, и val принимают значения порядка $O(n)$. Поэтому можно пройти по всем листам в дереве, и посчитать хеш, обращаясь к хешмапу напрямую и спрашивая, был ли уже такой же лист, и какой у него хеш. Для того, чтобы обойти все листы за их количество, при построении дерева можно в каждый лист складывать ссылку на новый лист, который появляется в следующей версии дерева отрезков.

Переход: посчитан хеш от всех вершин более глубокого уровня. Обратим внимание, что поскольку на каждой глубине $O(n)$ вершин, хеши этих вершин так же будут принимать значения $O(n)$. Поэтому мы можем очистить хешсет и точно так же, как и для листьев, считать новое значение хеша, проверяя, была ли уже такая пара $< h(v_l), h(v_r) >$.

Таким образом, мы построили дерево и посчитали хеши всех вершин за $(n^2 \log \Sigma, n^2)$ полностью детерменированно.

2.3 $(n^2 \log \Sigma, n)$ deterministic

Начнем с того, что на хранение дерева отрезков у нас сейчас уходит $n \log \Sigma$ памяти, это много. Чтобы уменьшить потребление памяти, можно использовать технику **limited node copying**. Краткое введение, которое будет необходимо для дальнейшего понимания алгоритма:

Вместо того, чтобы после пересоздания очередного листа пересоздать весь путь до корня, будем хранить в каждой вершине дополнительный указатель, изначально нулевой. Когда мы стоим в вершине и знаем, что один из ее сыновей был изменен, в случае, если дополнительный указатель еще не занят, просто установим его на новую версию этого сына и подпишем текущим глобальным временем. После такого изменения все еще несложно обратиться к какой-то версии дерева отрезков: нужно просто при переходе к сыновьям при выборе, куда спускаться, посмотреть, не нужно ли идти по дополнительному указателю.

Можно доказать, что таким образом построенное дерево занимает $O(n)$ памяти, но не будем об этом.

Подсчет хешей для листьев и внутренних вершин в этом решении отличается.

База: подсчет хешей для листьев. Будем считать хеши листьев группами, для каждой позиции все листья, соответствующие одной позиции в массиве вместе. Будем поддерживать счетчик ch — первый еще не использованный хеш. Фиксировав, какую позицию мы сейчас обрабатываем, просто обойдем все листья с этой позицией (для этого можно хранить в каждом листе ссылку на предыдущий лист этой позиции), и листу со значением val присвоим хеш $ch + val$, после чего увеличим ch на $max(val) + 1$.

Переход: посчитали хеши всех (даже больше, об этом далее) вершин на предыдущем уровне. Кроме хешей для вершины мы записываем не только эту вершину, а еще и все ее копии во все времена, которые мы не создавали явно в дереве отрезков (при переходе на послеследующий уровень их можно безопасно удалить, чтобы сохранить линейную память). Чтобы получить для вершины

список всех времен, когда она должна была бы существовать без сжатия, нужно просто взять список всех времен всех трех ее сыновей и смерджить. Пусть у очередной вершины хеши сыновей h_1 и h_2 (можно считать $h_1 < h_2$). Запишем в вектор с номером h_1 напоминание: надо бы посчитать $h(< h_1, h_2 >)$ и записать его в текущую вершину v. После того, как сделали это для всех вершин текущего уровня, можно идти по h_1, очищать хешмап на $O(n)$, и перебирать соответствующее ему h_2. Как обычно, если оно есть в хешмапе, достаем оттуда хеш, иначе сопоставляем ему новый.

После того, как хеши всех вершин посчитаны, можно освобождать память с предыдущего уровня и переходить к следующему. В конце получим посчитанные хеши для всех корней.

Используемое время так и осталось $O(n^2 \log \Sigma)$, а вот требуемая память стала всего $O(n)$.

3 Случай бинарных строк

Отдельный интерес представляет случай $|\Sigma| = 2$. Есть известный алгоритм, описанный, например, в [1], работающий за время $O(n^2/\log n)$.

В этой же статье рассмотрена задача матожидания длины $LCAF$ двух случайных строк длины n и сделано предположение 5.1 о том, что $LCAF_{avg} = n - O(\log n)$. Это преположение выглядит слишком смелым, рассмотрим эту задачу подробнее.

Для начала рассмотрим график $LCAF_{avg}$ от n — матожидания длины НОАП в зависимости от длины строк. График получен генерацией 10 000 случайных строк соответствующей длины и усреднением результатов.

{ Няшный график картиночкой }

N	$LCAF_{avg}(n)$
1000	827.277
2000	1651.76
3000	2487.35
4000	3310.42
5000	4137.15
6000	4960.17
7000	5796.39
8000	6636.79
9000	7452.33
10000	8295.32

Этот график никак не похож на $O(n - \log n)$. Более того, можно доказать другое утверждение:

Заключение

Литература

[1] *Ali Alatabbi, Costas S. Iliopoulos, Alsessio Langiu, M. Sohel Rahman,* Algorithms for Longest Common Abelian Factors // International Journal of Foundations of Computer Science. – 2016. – Т. 27. – №. 05. - С. 529-543