SPb HSE, 2 курс ПМИ, осень 2024/25
Конспект лекций по алгоритмам

Собрано 18 ноября 2024 г. в 10:23

Содержание

1. Паросочетания **1**
- 1.1. Определения . . . 1
- 1.2. Сведения . . . 1
- 1.3. Дополняющие чередующиеся пути . . . 1
- 1.4. Поиск паросочетания в двудольном графе . . . 2
- 1.5. Реализация . . . 2
- 1.6. Алгоритм Куна . . . 2
- 1.7. Оптимизации Куна . . . 3
- 1.8. Поиск минимального вершинного покрытия (на практике) . . . 4
- 1.9. Обзор решений . . . 4
- 1.10. Решения для произвольного графа . . . 4
  - 1.10.1. Обзор . . . 4
  - 1.10.2. Читерский подход . . . 5
  - 1.10.3. Связь паросочетаний и определителя . . . 5
  - 1.10.4. Матрица Татта . . . 6

2. Паросочетания-2 **6**
- 2.1. Классификация рёбер двудольного графа . . . 7
- 2.2. Stable matching (marriage problem) . . . 7
- 2.3. Венгерский алгоритм . . . 8
  - 2.3.1. Реализация за $\mathcal{O}(V^3)$ . . . 9
  - 2.3.2. Псевдокод . . . 9
- 2.4. Покраска графов . . . 11
  - 2.4.1. Вершинные раскраски . . . 11
  - 2.4.2. Вершинные раскраски планарных графов . . . 11
  - 2.4.3. Рёберные раскраски . . . 11
  - 2.4.4. Рёберные раскраски двудольных графов . . . 12
  - 2.4.5. Покраска не регулярного графа за $\mathcal{O}(E^2)$ . . . 12

3. Потоки (база) **12**
- 3.1. Основные определения . . . 13
- 3.2. Обратные рёбра . . . 13
- 3.3. Декомпозиция потока . . . 14
- 3.4. Теорема и алгоритм Форда-Фалкерсона . . . 14
- 3.5. Реализация, хранение графа . . . 15
- 3.6. Паросочетание, вершинное покрытие . . . 16
  - 3.6.1. Вершинное покрытие . . . 17
- 3.7. Леммы, позволяющие работать с потоками . . . 17

3.8. Алгоритмы поиска потока . . . 17
3.8.1. Эдмондс-Карп за $\mathcal{O}(VE^2)$ . . . 17
3.8.2. Масштабирование за $\mathcal{O}(E^2 \log U)$ . . . 18

4. Потоки (быстрые) 18
4.1. Алгоритм Диница . . . 19
4.2. Алгоритм Хопкрофта-Карпа . . . 20
4.3. Теоремы Карзанова . . . 21
4.4. Диниц с link-cut tree . . . 21
4.5. Глобальный разрез . . . 22
4.5.1. Алгоритм Штор-Вагнера . . . 22
4.5.2. Алгоритм Каргера-Штейна . . . 22
4.6. (*) Алгоритм Push-Relabel . . . 23

5. Mincost 23
5.1. Mincost k-flow в графе без отрицательных циклов . . . 24
5.2. Потенциалы и Дейкстра . . . 25
5.3. Задачи на mincost поток, паросочетания . . . 25
5.4. Графы с отрицательными циклами . . . 25
5.5. Mincost flow . . . 26
5.6. Полиномиальные решения . . . 27
5.7. (*) Cost Scaling . . . 27

6. Базовые алгоритмы на строках 28
6.1. Обозначения, определения . . . 29
6.2. Поиск подстроки в строке . . . 29
6.2.1. C++ . . . 29
6.2.2. Префикс функция и алгоритм КМП . . . 29
6.2.3. LCP . . . 30
6.2.4. Z-функция . . . 31
6.3. Полиномиальные хеши строк . . . 31
6.3.1. Алгоритм Рабина-Карпа . . . 33
6.3.2. Наибольшая общая подстрока за $\mathcal{O}(n \log n)$ . . . 33
6.3.3. Оценки вероятностей . . . 33
6.3.4. Число различных подстрок (на практике) . . . 34
6.3.5. Строка Туэ-Морса . . . 35
6.4. Алгоритм Бойера-Мура . . . 36

7. Суффиксный массив 38
7.1. Построение за $\mathcal{O}(n \log^2 n)$ хешами . . . 38
7.2. Применение суффиксного массива: поиск строки в тексте . . . 38
7.3. Построение за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$ цифровой сортировкой . . . 38
7.4. LCP за $\mathcal{O}(n)$: алгоритм Касаи . . . 39
7.5. Быстрый поиск строки в тексте . . . 40
7.6. (*) Построение за $\mathcal{O}(n)$: алгоритм Каркайнена-Сандерса . . . 40

7. Бор 41
7.7. Собственно бор . . . 42

7.8. Сортировка строк 42

8. Ахо-Корасик и Укконен **42**
8.1. Алгоритм Ахо-Корасика 43
8.2. Суффиксное дерево, связь с массивом 44
8.3. Суффиксное дерево, решение задач 45
8.4. Алгоритм Укконена 45
8.5. LZSS 46

9. Хеширование **47**
9.1. Универсальное семейство хеш функций 47
9.2. Оценки для хеш-таблицы с закрытой адресацией 47
9.3. Оценки других функций для хеш-таблиц 48
9.4. Фильтр Блюма 48
9.5. Совершенное хеширование 49
9.5.1. Одноуровневая схема 49
9.5.2. Двухуровневая схема 49
9.5.3. (*) Графовый подход 50
9.6. (*) Хеширование кукушки 50

9. Сжатие данных **50**
9.7. Общая информация 51
9.8. Арифметическое кодирование 51
9.8.1. Оценки 52
9.9. Словарное кодирование 52
9.9.1. LZW'84 52
9.10. LZSS 52
9.11. BWT 53
9.12. RLE 53
9.13. MTF (move to front) 53
9.14. Сжатие и предсказания 54
9.15. Итоги и ссылки 54

Лекция #1: Паросочетания

9 сентября 2024

1.1. Определения

Def 1.1.1. *Паросочетание (matching) – множество попарно не смежных рёбер M.*

Def 1.1.2. *Вершинное покрытие (vertex cover) –*
множество вершин C, что у любого ребра хотя бы один конец лежит в C.

Def 1.1.3. *Независимое множество (indpendent set) –*
множество попарно несмежных вершин I.

Def 1.1.4. *Клика (clique) – множество попарно смежных вершин.*

Def 1.1.5. *Совершенное паросочетание – паросочетание, покрывающее все вершины графа.*
В двудольном графе совершенным является паросочетание, покрывающее все вершины меньшей доли.

Def 1.1.6. *Относительно любого паросочетания все вершины можно поделить на*
- *покрытые паросочетанием (принадлежащие паросочетанию),*
- *не покрытые паросочетанием (свободные).*

Обозначения: Matching (M), Vertex Cover (VC или C), Independent Set (IS или I).

1.2. Сведения

Пусть дан граф G, заданный матрицей смежности g_{ij}. Инвертацией G назовём граф G', заданный матрицей смежности $g'_{ij} = 1 - g_{ij}$. тогда независимое множество в G задаёт клику в G', а клика в G задаёт независимое множество в G'.

Следствие 1.2.1. Задачи поиска max клики и max IS сводятся друг к другу.

Lm 1.2.2. Дополнение любого VC – IS. Дополнение любого IS – VC.

Следствие 1.2.3. Все три задачи поиска min VC, max IS, max clique сводятся друг к другу.

Утверждение 1.2.4. Задачи поиска min VC, max IS, max clique NP-трудны.

Утверждение было доказано в прошлом семестре в разделе про сложность.

1.3. Дополняющие чередующиеся пути

Def 1.3.1. *Чередующийся путь – простой путь, в котором рёбра чередуются в смысле принадлежности паросочетанию.*

Def 1.3.2. *Дополняющий чередующийся путь (ДЧП) – чередующийся путь, первая и последняя вершина которого не покрыты паросочетанием.*

Lm 1.3.3. Паросочетание P максимально $\Leftrightarrow \nexists$ ДЧП (лемма о дополняющем пути).

Доказательство. Пусть $\exists$ ДЧП $\Rightarrow$ инвертируем все рёбра на нём, получим паросочетание размера $|P|+1$. Докажем теперь, что если $\exists$ паросочетание M: $|M| > |P|$, то $\exists$ ДЧП. Для этого рассмотрим $S = M \triangledown P$. Степень каждой вершины в S не более двух (одно ребро из M, одно из P) $\Rightarrow$ S является объединением циклов и путей. Каждому пути сопоставим число a_i – разность количеств рёбер из M и P. Тогда $|M| = |P| + \sum_i a_i \Rightarrow \exists a_i > 0 \Rightarrow$ один из путей – ДЧП. ■

Лемма доказана для произвольного графа, но с лёгкостью найти ДЧП мы сможем только для двудольного графа.

1.4. Поиск паросочетания в двудольном графе

Lm 1.4.1. Пусть G – двудольный граф, а P паросочетание в нём. Построим орграф $G'(G,P)$. Вершины такие же, как в G. Рёбра: из первой доли во вторую пустим все рёбра G, а из второй в первую долю только рёбра из P. Тогда есть биекция между путями в G' и чередующимися путями в G.

Lm 1.4.2. Поиск ДЧП в $G \Leftrightarrow$ поиску пути в G' из свободной вершины в свободную.

Следствие 1.4.3. Мы получили алгоритм поиска максимального паросочетания M за $\mathcal{O}(|M|\cdot E)$:

0. $P \leftarrow \varnothing$
1. Попробуем найти путь dfs-ом в $G'(G,P)$
2. `if` не нашли $\Rightarrow$ M максимально
3. `else goto 1`

1.5. Реализация

Важной идеей является применение ДЧП к паросочетания на обратном ходу рекурсии.

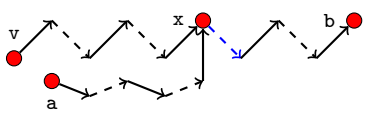
```
1 def dfs(v):
2     used[v] = 1 # массив пометок для вершин первой доли
3     for x in graph[v]: # рёбра из 1-й доли во вторую
4         if (pair[x] == -1) or (used[pair[x]] == 0 and dfs(pair[x])):
5             pair[x] = v # массив пар для вершин второй доли
6             return True
7     return False
```

Граф G' в явном виде мы нигде не строим. Вместо этого, когда идём из 1-й доли во вторую, перебираем все рёбра (`v` $\rightarrow$ `x in graph[v]`), а когда из 2-й в первую, идём только по ребру паросочетания (`x` $\rightarrow$ `pair[x]`).

1.6. Алгоритм Куна

Lm 1.6.1. В процессе поиска максимального паросочетания $\nexists$ ДЧП из $v \Rightarrow$ ДЧП из v уже никогда не появится.

Пусть появился ДЧП p: $v \rightsquigarrow u$ после применения ДЧП q: $a \rightsquigarrow b$. Тогда пойдём по p из v, найдём $x \in p$ – первую вершину в q. Из x пойдём по пути $a \leftrightarrow b$ по ребру паросочетания, дойдём до конца, получим пути $v \rightsquigarrow (a \vee b)$, который существовал до применения q.



■

Получили алгоритм Куна:

```
for v in range(n):
    used = [0] * n
    dfs(v)
```

1.7. Оптимизации Куна

Обозначим за k размер максимального паросочетания. Сейчас время работы алгоритма Куна $\mathcal{O}(VE)$ даже для $k = \mathcal{O}(1)$. Будем обнулять пометки `used[]` только, если мы нашли ДЧП.

```
used = [0] * n
for v in range(n):
    if dfs(v):
        used = [0] * n
```

Алгоритм остался корректным, так как, между успешными запусками `dfs` граф G' не меняется. И теперь работает за $\mathcal{O}(kE)$.

Следующая оптимизация – «жадная инициализация». До запуска Куна переберём все рёбра и те из них, что можем, добавим в паросочетание.

Lm 1.7.1. Жадная инициализация даст паросочетание размера $\geqslant \frac{k}{2}$.

Доказательство. Если мы взяли ребро, которое на самом деле не должно лежать в максимальном паросочетании M, мы заблокировали возможность взять $\leqslant 2$ рёбер из M. ■

При использовании жадной инициализации у нас появляется необходимость поддерживать массив `covered[v]`, хранящий для вершины первой доли, покрыта ли она паросочетанием.

Попробуем теперь сделать следующее:

```
used = [0] * n
for v in range(n):
    if not covered[v]:
        dfs(v)
```

Код работает за $\mathcal{O}(V + E)$. Если паросочетание не максимально, найдёт хотя бы один ДЧП. А может найти больше чем один... в этом и заключается последняя оптимизация «вообще не обнулять пометки»: *пока данный код находит хотя бы один путь, запускаем его.*

Замечание 1.7.2. На практике докажем, что, если мы используем последнюю оптимизацию, «жадная инициализация» является полностью бесполезной.

Напоминание: мы умеем обнулять пометки за $\mathcal{O}(1)$. Для этого помеченной считаем вершины `v`: «`used[v] == cc`», тогда операция «`cc++`» сделает все вершины не помеченными.

Напоминание: если использовать `random_shuffle` рёбер, работает быстрее, в том смысле, что $\max_{test} E[Time(test)]$ уменьшилось (макстест перестаёт быть макстестом).

Замечание 1.7.3. Для последней версии алгоритма авторам конспекта неизвестен тест, на котором достигается время работы $\Omega(VE)$. Если не использовать `random_shuffle` рёбер, есть конструкция для $\Omega(\frac{VE}{\log V})$.

1.8. Поиск минимального вершинного покрытия (на практике)

Lm 1.8.1. $\forall M, VC$ верно, что $|VC| \geqslant |M|$

Доказательство. Для каждого ребра $e \in M$, нужно взять в VC хотя бы один из концов e. ■

Пусть у нас уже построено максимальное паросочетание M. Запустим `dfs` на G' из всех свободных вершин первой доли. Обозначим первую долю A, вторую B. Посещённые `dfs`-ом вершины соответственно A^+ и B^+, а непосещённые A^- и B^-.

Теорема 1.8.2. $X = A^- \cup B^+$ – минимальное вершинное покрытие.

Доказательство. Если бы из $a \in A^+$ было бы ребро в $b \in B^-$, мы бы по нему прошли, и b лежала бы в $B^+ \Rightarrow$ в $A^+ \cup B^-$ нет рёбер $\Rightarrow X$ – вершинное покрытие.
Оценим размер X: все вершины из $A^- \cup B^+$ – концы рёбер паросочетания M, т.к. `dfs` не нашёл дополняющего пути. Более того это концы обязательно разных рёбер паросочетания, т.к. если один конец ребра паросочетания лежит в B^+, то `dfs` пойдёт по нему, и второй конец окажется в A^+. Итого $|X| \leqslant |M|$. Из этого и Lm 1.8.1 следует $|X| = |M|$ и $|X| = \min$. ■

Следствие 1.8.3. $A^+ \cup B^-$ – максимальное независимое множество (Lm 1.2.2).

Замечание 1.8.4. Умеем строить $\min VC$ и $\max IS$ за $\mathcal{O}(V+E)$ при наличии $\max$ паросочетания.

Следствие 1.8.5. $\max M = \min VC$ (теорема Кёнига).

1.9. Обзор решений

Мы изучили алгоритм Куна со всеми оптимизациями.
Асимптотически время его работы $\mathcal{O}(VE)$, на практике же он жутко шустрый.
На графах $V, E \leqslant 10^5$ решение укладывается в секунду.

$\exists$ также алгоритм Хопкрофта-Карпа за $\mathcal{O}(EV^{1/2})$. Его мы изучим в контексте потоков.

В регулярном двудольном графе можно найти совершенное паросочетание за время $\mathcal{O}(V \log V)$.
Статья Михаила Капралова и ко.

1.10. Решения для произвольного графа

1.10.1. Обзор

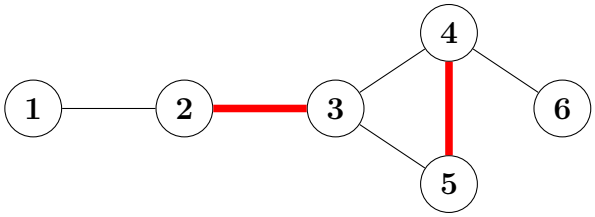
Наиболее стандартным считается решение через сжатие «соцветий» (видим нечётный цикл – сожмём его, найдём паросочетание в новом цикле, разожмём цикл, перестроим паросочетание). Этот подход используют реализации Габова за $\mathcal{O}(V^3)$ и Тарьяна за $\mathcal{O}(VE\alpha)$. Подробно эту тему мы будем изучать на 3-м курсе.

Оптимальный по времени – алгоритм Вазирани, $\mathcal{O}(EV^{1/2})$.

Кроме этого есть два подхода, которые мы обсудим подробнее.

1.10.2. Читерский подход

Давайте на недвудольном графе запустим dfs из Куна для поиска дополняющей цепи...
При этом помечать, как посещённые, будем вершины обеих долей в одном массиве `used`.
С некоторой вероятностью алгоритм успешно найдёт дополняющий путь.



Если мы запускаем dfs из вершины 1, то у неё есть шанс найти дополняющий путь $1 \to 2 \to 3 \to 5 \to 4 \to 6$. Но если из вершины 3 dfs сперва попытается идти в 4, то он пометит 4 и 5, как посещённые, больше в них заходить не будет, путь не найдёт.
Т.е. на данном примере в зависимости от порядка соседей вершины 3 dfs или найдёт, или не найдёт путь. Если выбрать случайный порядок, то найдёт с вероятностью $1/2$.
Это лучше, чем «при некотором порядке рёбер вообще не иметь возможности найти путь», поэтому первой строкой dfs добавляем `random_shuffle(c[v].begin(), c[v].end())`.

На большинстве графов алгоритм сможет найти максимальное паросочетание. $\exists$ графы, на которых вероятность нахождения дополняющей цепи экспоненциально от числа вершин мала.

1.10.3. Связь паросочетаний и определителя

Пусть дан двудольный граф. Если доли имеют размеры n и m, матрицу смежности разумно задавать размера $n \times m$, а не $(n+m) \times (n+m)$, как для обычного.

Пусть $n = m$. Совершенному паросочетанию соответствует перестановка $\pi\colon i \to \pi_i$ и ячейки матрица смежности $A\colon A_{i\,\pi_i} = 1 \Rightarrow$ количество совершенных паросочетаний $N = \sum\limits_{\pi} \prod\limits_{i} a_{i\,\pi_i}$.

Следствие 1.10.1. Чётность числа совершенных паросочетаний $= \det A \bmod 2$

Доказательство. $\det A$ — такая же сумма, только у слагаемых другой знак, но $-1 = 1 \bmod 2$ ■

Замечание 1.10.2. Подсчёт N — трудная задача, её не умеют решать за $2^{o(n)}$.
Поскольку ответ — целое число, формально это не NP-hard, а P#-hard задача.

1.10.4. Матрица Татта

Пусть дан *двудольный* граф $G(L, R, E)$, $|L| = |R| = n$. Определим для него матрицу Эдмондса D размера $n \times n$: $d_{i,j} = x_{i,j}$, если есть ребро из $i \in L$ в $j \in R$. В противном случае $d_{i,j} = 0$.
Здесь $x_{i,j}$ – это n^2 различных переменных.
$\det D$ – это многочлен от n^2 переменных. Легко видеть, что он тождественно равен нулю тогда и только тогда, когда в графе нет совершенного паросочетания (слагаемые в определителе не взаимоуничтожаются, а любое такое слагаемое соответствует совершенному паросочетанию).

Пусть теперь граф произвольный. Определим для него матрицу Татта T.
Для каждого ребра (i, j) элементы $t_{ij} = x_{ij}, t_{ji} = -x_{ij}$.
Остальные элементы равны нулю. Здесь x_{ij} – переменные.
Получается, для каждого ребра неорграфа мы ввели ровно одну переменную.
$\det T$ – многочлен от $n(n-1)/2$ переменных.

Теорема 1.10.3. Татта: $\det T \not\equiv 0 \Leftrightarrow \exists$ совершенное паросочетание.

Доказательство будет в курсе дискретной математики.

Чтобы проверить $\det T \equiv 0$, используем лемму Шварца-Зиппеля: в каждую переменную x_{ij} подставим случайное значение, посчитаем определитель матрицы над полем $\mathbb{F}_p$, где $p = 10^9 + 7$, получим вероятность ошибки не более $\deg(\det T)/p = n/p$.

Время работы $\mathcal{O}(n^3)$ (нахождение определителя матрицы по модулю p), алгоритм умеет лишь проверять наличие совершенного паросочетания.
Алгоритм можно модифицировать сперва для определения размера максимального паросочетания, а затем для его нахождения.
Пример: $n = 3, E = \{(1, 2), (2, 3)\}, T = \begin{bmatrix} 0 & x_{12} & 0 \\ -x_{12} & 0 & x_{23} \\ 0 & -x_{23} & 0 \end{bmatrix}$, подставляем $x_{12} = 9, x_{23} = 7$, считаем $\det \begin{bmatrix} 0 & 9 & 0 \\ -9 & 0 & 7 \\ 0 & -7 & 0 \end{bmatrix} = 0 \Rightarrow$ с большой вероятностью нет совершенного паросочетания.

Лекция #2: Паросочетания-2

16 сентября

2.1. Классификация рёбер двудольного графа

Дан двудольный граф $G = \langle V, E \rangle$. Задача – определить $\forall e \in E$, $\exists$ ли максимальное паросочетание $M_1 \colon e \in M_1$, а также $\exists$ ли максимальное паросочетание $M_2 \colon e \notin M_2$.
Иначе говоря, мы хотим разбить рёбра на три класса:

1. Должно лежать в максимальном паросочетании. `MUST`.
2. Не лежит ни в каком максимальном паросочетании. `NO`.
3. Все остальные. `MAY`.

Решение: для начала найдём любое максимальное паросочетание M.
Если мы найдём класс `MAY`, то $\texttt{MUST} = M \setminus \texttt{MAY}$, $\texttt{NO} = \overline{M} \setminus \texttt{MAY}$.

Lm 2.1.1. $e \in \texttt{MAY} \Leftrightarrow \exists P$ – чередующийся относительно M путь чётной длины или чётный цикл такой, что $e \in P$.

Доказательство. $e \in \texttt{MAY} \Rightarrow \exists$ другое max паросочетание $M' \colon e \in M \triangledown M'$.
Симметрическая разность, как мы уже знаем, состоит из чередующихся путей и циклов.

Посмотрим с другой стороны: если относительно M есть P – чередующийся путь чётной длины или чередующийся цикл, то $P \subseteq \texttt{MAY}$, так как и M, и $M \triangledown P$ являются максимальными, а каждое ребро P лежит ровно в одном из двух. ■

Осталось научиться находить циклы и пути алгоритмически. Для этого рассмотрим тот же граф G', на котором работает dfs из Куна. С чётными путями всё просто: все они начинаются в свободных вершинах, dfs из Куна, запущенный от всех свободных вершин *обеих долей* пройдёт ровно по всем рёбрам, которые можно покрыть чётными путями, и пометит их. А про циклы:

Lm 2.1.2. Ребро e лежит на чётном цикле $\Leftrightarrow$
концы e лежат в одной компоненте сильной связности графа G'.

Следствие 2.1.3. Получили алгоритм построения `MAY` по данному M за $\mathcal{O}(V + E)$.
Следствие 2.1.4. Из `MAY` и M за $\mathcal{O}(E)$ умеем получить `MUST` и `NO`.

2.2. Stable matching (marriage problem)

Сформулируем задачу на языке мальчиков/девочек. Есть n мальчиков, у каждого из них есть список девочек $bs[a]$, которые ему нравятся в порядке от наиболее приоритетных к менее. Есть m девочек, у каждой есть список мальчиков $as[b]$, которые ей нравятся в таком же порядке.
Мальчики и девочки хотят образовать пары.
Никто не готов образовывать пару с тем, кто вообще отсутствует в его списке.
И для мальчиков, и для девочек наименее приоритетный вариант – остаться вообще без пары.
Будем обозначать p_a – пара мальчика a или -1, q_b – пара девочки q или -1.

Def 2.2.1. *Паросочетание называется* **не** *стабильным, если* $\exists$ *мальчик* a *и девочка* b*: мальчику* a *нравится* b *больше чем* p_a *и девочке* b *нравится* a *больше чем* q_b*.*

Def 2.2.2. *Иначе паросочетание называется стабильным*

• **Алгоритм поиска: «мальчики предлагают, девочки отказываются»**

Изначально проинициализируем $p_a = bs[a].\texttt{best}$, далее, пока есть
два мальчика i, j: $b = p_i = p_j \neq -1$, Девочка b откажет тому из них, кто ей меньше нравится.
Пусть она отказала мальчику i, тогда делаем $bs[i].\texttt{remove\_best}(), p_i = bs[i].\texttt{best}$.

Теорема 2.2.3. Алгоритм всегда завершится и найдёт стабильное паросочетание

Доказательство. Длины списков $bs[i]$ убывают $\Rightarrow$ завершится. Пусть в конце мальчик a и девочка b дают не стабильность (не являются парой, но хотят образовать). Это значит, что a перед тем, как образовать пару с p_a, предлагал себя b, а она ему отказала.
Но зачем?! Ведь он ей нравился больше. Противоречие. ■

Аккуратная реализация даёт время $\mathcal{O}(V+E) = \mathcal{O}(\sum_i |bs[i]| + \sum_j |as[j]|)$.

```
def ask(boyIndex): # мальчик предлагает себя
    girlIndex = priorityList[boyIndex].pop_best() # кому?
    if (q[girlIndex] == -1): q[girlIndex] = boyIndex;
    else:
        # нужно ещё сделать, чтобы priority работала за O(1)
        if priority(girlIndex, q[girlIndex]) < prioirity(girlIndex, boyIndex):
            swap(p[girlIndex], boyIndex)
        # в boyIndex сейчас худший из двух вариантов, ему откажем, он предложит следующей
        ask(boyIndex);
for b in boys: ask(b) # собственно алгоритм
```

Утверждение 2.2.4. На практике докажем, что предложенный алгоритм оптимален для мальчиков и предложим версию, оптимальную для девочек.

Теорема 2.2.5. Если граф полный (каждый список содержит все вершины) и размеры долей равны, мы найдём совершенное паросочетание.

Доказательство. Пусть какому-то мальчику отказали все n девочек. После отказа у девочки всегда остаётся лучший кандидат $\Rightarrow$ сейчас у всех девочек одновременно есть кандидаты $= n$ разных мальчиков $\Rightarrow$ всего $\geqslant n+1$ мальчик ?!? ■

Пример использования.
Студенты хотят поступить в ВУЗы. У каждого ВУЗа ограниченное число мест и могут быть разные приоритеты, кого хотят брать. Строим двудольный граф. Каждый студент указывает список ВУЗов в порядке приоритетов, куда хочет поступить. Каждое место-в-вузе получает список студентов в порядке приоритета данного ВУЗа. Ищем стабильное паросочетание, получаем решение, оптимальное для студентов.

2.3. Венгерский алгоритм

Дан взвешенный двудольный граф, заданный матрицей весов a: $n \times n$, где a_{ij} – вес ребра из i-й вершины первой доли в j-ю вершину второй доли. Задача – найти совершенное паросочетание минимального веса.

Формально: найти $\pi \in S_n$: $\sum_i a_{i\pi_i} \to \min$.

Иногда задачу называют *задачей о назначениях*, тогда a_{ij} – стоимость выполнения i-м работником j-й работы, нужно каждому работнику сопоставить одну работу.

Lm 2.3.1. Если $a_{ij} \geqslant 0$ и $\exists$ совершенное паросочетание на нулях, оно оптимально.

Lm 2.3.2. Рассмотрим матрицу $a'_{ij} = a_{ij} + row_i + col_j$, где $row_i, col_j \in \mathbb{R}$.
Оптимальные паросочетания для a' и для a совпадают.

Доказательство. Достаточно заметить, что в $(f = \sum_i a_{i\pi_i})$ войдёт ровно по одному элементу каждой строки, каждого столбца $\Rightarrow$ если все элементы строки (столбца) увеличить на константу C, в не зависимости от π величина f увеличится на $C \Rightarrow$ оптимум перейдёт в оптимум. ■

Венгерский алгоритм, как Кун, перебирает вершины первой доли и от каждой пытается строить ДЧП, но использует при этом только нулевые рёбра. Если нет нулевого ребра, то $x = \min$ на $A^+ \times B^- > 0$. Давайте все столбцы из B^- уменьшим на x, а все строки из A^- увеличим на x.

	B^+	B^-
A^+		$-x$
A^-	$+x$	0

В итоге в подматрице $A^+ \times B^-$ на месте минимального элемента появится 0, в матрице $A^- \times B^+$ элементы увеличатся, остальные не изменятся. При этом все элементы матрицы остались неотрицательными. Рёбра из $A^- \times B^+$ могли перестать быть нулевыми, но они не лежат ни в текущем паросочетании, ни в дереве дополняющих цепей: $M \subseteq (A^- \times B^-) \cup (A^+ \times B^+)$, рёбра дополняющих цепей идут из A^+.

2.3.1. Реализация за $\mathcal{O}(V^3)$

Венгерский алгоритм = V поисков ДЧП.
Поиск ДЧП = инициализировать $A^+ = B^+ = \varnothing$ и не более V раз найти минимум $x = a_{ij}$ в $A^+ \times B^-$. Если $x > 0$, то пересчитать матрицу весов. Посетить столбец j и строку $pair_j$.

Чтобы быстро увеличивать столбец/строку на константу, поддерживаем row_i, col_j.
Реальное значение элемента матрицы: $a'_{ij} = a_{ij} + row_i + col_j$. Увеличение строки на x: row_j `+=` x.
Чтобы найти минимум x, а также строку i, столбец j, на которых минимум достигается, воспользуемся идеей из алгоритма Прима: $w_j = \min\limits_{i \in A^+} \langle a'_{ij}, i \rangle$. Тогда $\langle \langle x, i \rangle, j \rangle = \min\limits_{j \in B^-} \langle w_j, j \rangle$.
Научились находить (x, i, j) за $\mathcal{O}(n)$, осталось при изменении row_i, col_j пересчитать $w_j : j \in B^-$.
col_j `+=` $y \Rightarrow w_k$ `+=` y. А row_i будет меняться только у $i \in A^- \Rightarrow$ на $\min_{i \in A^+}$ не повлияет.

Замечание 2.3.3. Можно выбирать min в множестве $w_j : j \in B^-$ не за линию, а используя кучи.

2.3.2. Псевдокод

Обозначим, как обычно, первую долю A, вторую B, посещённые вершины – A^+, B^+.
Также, как в Куне, если $x \in B$, то `pair[`x`]` $\in A$ – её пара в первой доли.
Строки – вершины первой доли (A). В нашем коде строки – i, v, u.
Столбцы – вершины второй доли (B). В нашем коде столбцы – j.
`pair[`$b \in B$`]` – её пара в A, `pair2[`$a \in A$`]` – её пара в B.

```
1.  row ← 0, col ← 0
2.  for v ∈ A
3.      A⁺ = {v}, B⁺ = ∅ // (остальное в A⁻, B⁻).
4.      for j ∈ B⁻: w[j] = (a[v][j] + row[v] + col[j], v) // (минимум и номер строки)
5.      while (True) // (пока не нашли путь из v в свободную вершину B)
6.          ((x,i),j) = min{(w[j],j): j ∈ B⁻} // (минимум и позиция минимума в A⁺ × B⁻)
7.          // (i - номер строки, j - номер столбца ⇒ a[i,j] + row[i] + col[j] == x)
8.          for i ∈ A⁻: row[i] += x;
```

```
9.          for j ∈ B⁻: col[j] -= x, w[j].value -= x;
10.         // (в итоге мы уменьшили A⁺ × B⁻, увеличили A⁻ × B⁺, пересчитали w[j]).
11.         j перемещаем из B⁻ в B⁺; запоминаем prev[j] = i;
12.         if (u=pair[j]) == -1
13.             break // дополняющий путь: j,prev[j],pair2[prev[j]],prev[pair2[prev[j]]],...
14.         u перемещаем из A⁻ в A⁺;
15.         пересчитываем все w[j] = min(w[j], pair(a[u][j] + row[u] + col[j], u));
16.     применим дополняющий путь v ⇝ j, пересчитаем pair[], pair2[]
```

Текущая реализация даёт время $\mathcal{O}(V^3)$.
Внутри цикла `while` строки 6, 8, 9, 15 работают за $\mathcal{O}(V)$ каждая.
8 и 9 можно улучшить до $\mathcal{O}(1)$, храня специальные величины `addToAMinus`, `addToBMinus`.
6 и 15 можно улучшить до $\langle \mathcal{O}(\log V), deg[x] \cdot \mathcal{O}(1) \rangle$, применив кучу Фибоначчи.
Итого получится $\mathcal{O}(V(E + V \log V))$.

2.4. Покраска графов

2.4.1. Вершинные раскраски

Задача: покрасить вершины графа так, чтобы любые смежные вершины имели разные цвета.
В два цвета (двудольный граф) красит обычный dfs за $\mathcal{O}(V+E)$.
В три цвета красить NP-трудно. В прошлом семестре мы научились это делать за $\mathcal{O}(1.44^n)$.
Во сколько цветов можно покрасить вершины за полиномиальное время?

• **Жадность**

Удалим вершину v из графа → покрасим рекурсивно $G \setminus \{v\}$ → докрасим v.
У вершины v всего deg_v уже покрашенных соседей ⇒
в один из $deg_v + 1$ цветов мы её точно сможем покрасить.

Следствие 2.4.1. Вершины можно покрасить в D+1 цвет за $\mathcal{O}(V+E)$, где $D = \max_v deg_v$.

На дискретной математике будет доказана более сильная теорема:

Теорема 2.4.2. Брукс: все графы кроме нечётных циклов и клик можно покрасить в D цветов.

Кроме теоремы есть алгоритм покраски в D цветов за $\mathcal{O}(V+E)$.

На практике, если удалять вершину v: $deg_v = \min$ и докрашивать её в минимально возможный цвет, жадность будет давать приличные результаты. За счёт потребности выбирать вершину именно минимальной степени нам потребуется куча, время возрастёт до $\mathcal{O}((E+V)\log V)$.

Замечание 2.4.3. Иногда про покраску вершин удобно думать, как про разбиение множества вершин на независимые множества.

2.4.2. Вершинные раскраски планарных графов

Очередная Теорема Эйлера гласит, что для планарного графа $E \leqslant 3 \cdot V - 6 \Rightarrow$ есть вершина степени $\leqslant 5 \Rightarrow$ жадность выше всегда красит в $\leqslant 6$ цветов.

Можно тем же способом покрасить и в 5. Пусть мы докрашиваем вершину v, а у неё 5 разноцветных соседей. Воспользуемся планарностью, упорядочим их по часовой стрелке и заметим, что не могут одновременно существовать два непересекающихся пути $1 \rightsquigarrow 3$, $2 \rightsquigarrow 4$. Поэтому попробуем сперва поменять цвета $c[1] \leftrightarrow c[3]$, для этого dfs-ом из 1 перебирая только вершины цветов $c[1]$, $c[3]$ попробуем найти путь в 3. Если не нашли, перекрасим. Если нашли, то сделаем тоже самое для $\{2,4\}$, там пути точно не будет. После успешного перекрашивания у соседей всего 4 разных цвета, есть 5-й, чтобы докрасить себя.

Мы не хотим укладывать граф, как нам угадать, порядок вершин? Просто переберём все $\frac{5\cdot 4}{2}$ пар соседей $\{u,v\}$. Для какой-то из них не будет пути и получится перекрасить.

2.4.3. Рёберные раскраски

Задача: покрасить *рёбра* графа так, чтобы любые смежные рёбра имели разные цвета.
Попробуем для начала применить ту же жадность: удаляем ребро e из графа, рекурсивно красим рёбра в $G \setminus \{e\}$, докрасим e. У ребра e может быть $2(D-1)$ смежных, где $D = \max_v deg_v$. Значит, чтобы ребро e всегда получалось докрасить, в худшем, нашей жадности нужен $2D-1$ цвет, С другой стороны, поскольку рёбра, инцидентные одной вершине, должны иметь попарно разные цвета, Есть гораздо более сильный результат, который также подробнее будет изучен в курсе дискретной математики.

Теорема 2.4.4. Визинг: рёбра любого графа можно покрасить в $D{+}1$ цвет.

Доказательство теоремы представляет собой алгоритм покраски в $D{+}1$ цвет за $\mathcal{O}(VE)$.

При этом задача «определить, можно ли покрасить в D цветов» NP-трудна.

2.4.4. Рёберные раскраски двудольных графов

С двудольными графами всё проще. Сейчас научимся красить их в D цветов.
Покраска рёбер – разбиения множества рёбер на паросочетания. На последней практике мы доказали, что в двудольном регулярном графе существует совершенное паросочетание.

Следствие 2.4.5. d-регулярный граф можно покрасить в d цветов.

Доказательство. Отщепим совершенное паросочетание, покрасим его в первый цвет. Оставшийся граф является $(d{-}1)$-регулярным, по индукции его можно покрасить в $d{-}1$ цвет. ■

Чтобы покрасить не регулярный граф, дополним его до D-регулярного.

• **Дополнение до регулярного**

1. Если в долях неравное число вершин, добавим новые вершины.
2. Пока граф не является D-регулярным (D – максимальная степень),
   в обеих долях есть вершины степени меньше D, соединим эти вершины ребром.
3. В результате мы получим D-регулярный граф, возможно, с кратными рёбрами.
   Кратные рёбра – это нормально, все изученные нами алгоритмы их не боятся.

Итого рёбра d-регулярного двудольного граф мы умеем красить за $\mathcal{O}(d{\cdot}\texttt{Matching}) = \mathcal{O}(dVE) = \mathcal{O}(E^2)$, а рёбра произвольного двудольного за $\mathcal{O}(D \cdot V \cdot VD) = \mathcal{O}(V^2D^2)$.

Поскольку в полученном регулярном графе есть совершенное паросочетание, мы доказали:

Следствие 2.4.6. Для $\forall$ двудольного G $\exists$ паросочетание, покрывающее все вершины G (в обеих долях) максимальной степени.

Раз такое паросочетание $\exists$, его можно попробовать найти, не дополняя граф до регулярного.

2.4.5. Покраска не регулярного графа за $\mathcal{O}(E^2)$

Обозначим A_D – вершины степени D первой доли, B_D – вершины степени D второй доли.
Уже знаем, что $\exists$ паросочетание M, покрывающее $A_D \cup B_D$.

1. Запустим Куна от вершин A_D, получили паросочетание P.
   Обозначим B_P покрытые паросочетанием P вершины второй доли.
2. Если $B_D \not\subseteq B_P$, чтобы покрыть $X = B_D \setminus B_P$ рассмотрим $M \triangledown P$. Каждой вершине из X в $M \triangledown P$ соответствует или ДЧП, или чётный путь из X в $Y = B_P \setminus B_D$.
3. *Алгоритм:* для всех $v \in X$ ищем путь или в свободную вершину первой доли, или в Y.

Чтобы оценить время работы, обозначим размер найденного паросочетания k_i и заметим, что нашли мы его за $\mathcal{O}(k_iE)$. Все k_i рёбер паросочетания будут покрашены и удалены из графа, то есть, $\sum_i k_i = E$. Получаем время работы алгоритма $\sum_i \mathcal{O}(k_iE) = \mathcal{O}(E^2)$.

Лекция #3: Потоки (база)

23 сентября 2024

3.1. Основные определения

Дан орграф G, у каждого ребра e есть пропускная способность $c_e \in \mathbb{R}$.

Def 3.1.1. *Поток в орграфе из s в t – сопоставленные рёбрам числа $f_e \in \mathbb{R}$:*

$$(\forall \text{ ребра } e \quad 0 \leqslant f_e \leqslant c_e) \ \wedge \ (\forall \text{ вершины } v \neq s,t \quad \sum_{e \in in(v)} f_e = \sum_{e \in out(v)} f_e)$$

Вершина s называется *истоком*, вершина t *стоком*.
Говорят, что по ребру e течёт f_e единиц потока.
Определение говорит «поток течёт из истока в сток и ни в какой вершине не задерживается».

Def 3.1.2. *Величина потока* $|f| = \sum_{e \in out(s)} f_e - \sum_{e \in in(s)} f_e$ *(сколько вытекает из истока).*

Утверждение 3.1.3. В сток втекает ровно столько, сколько вытекает из истока.

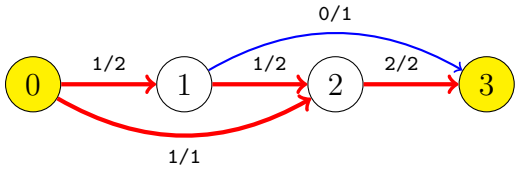
Замечание 3.1.4. $|f|$ может быть отрицательной: пустим по ребру $t \to s$ единицу потока.

Def 3.1.5. *Циркуляцией называется поток величины* 0.

• Примеры потока

Рассмотрим пока граф с единичными пропускными способностями.

1. ∀ цикл – циркуляция.
2. ∀ путь из s в t – поток величины 1.
3. ∀ k не пересекающихся по рёбрам путей из s в t – поток величины k.
4. На картинке поток величины 2, подписи f_e/c_e.



Def 3.1.6. *Остаточная сеть потока f – G_f, граф с пропускными способностями $c_e - f_e$.*

Def 3.1.7. *Дополняющий путь – путь из s в t в остаточной сети G_f.*

Lm 3.1.8. Если по всем рёбрам дополняющего пути p увеличить величину потока на $x = \min_{e \in p}(c_e - f_e)$, получится корректный поток величины $|f| + x$.

3.2. Обратные рёбра

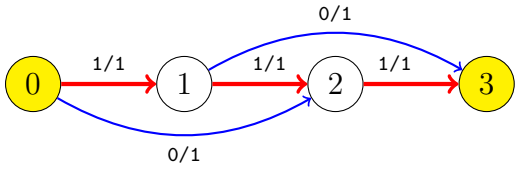
Def 3.2.1. *Для каждого ребра сети G с пропускной способностью c_e создадим обратное ребро e' пропускной способностью* 0. *При этом по определению $f_{e'} = -f_e$.*

Добавим в граф обратные рёбра, упростим определения потока и величины потока:
Теперь ∀ потока f должно выполняться $\forall v \neq s,t \ \sum_{e \in out(v)} f_e = 0$, величина потока $|f| = \sum_{e \in out(s)} f_e$.
Здесь $out(v)$ – множество прямых и обратных рёбер, выходящих из v.

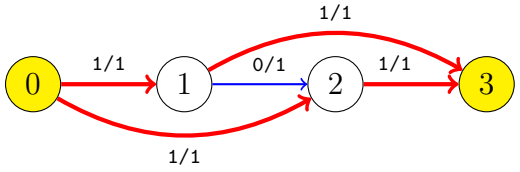
Def 3.2.2. *Ребро называется насыщенным, если $f_e = c_e$, иначе оно ненасыщено.*

Утверждение 3.2.3. Если по прямому ребру течёт поток, обратное ненасыщено.

После добавления обратных рёбер в в G, они появились и в G_f. Поэтому для такого потока из 0 в 3 величины 1 в G_f есть дополняющий путь $0 \to 2 \to 1 \to 3$.



Увеличим поток по пути $0 \to 2 \to 1 \to 3$, получим новый поток. Заметьте, добавляя +1 потока к ребру $2 \to 1$, мы уменьшаем поток по ребру $1 \to 2$.



Замечание 3.2.4. Сейчас мы научимся разбивать поток величины 2 на 2 непересекающихся по рёбрам пути. Если бы мы действовали жадно (*найдём какой-нибудь первый путь, удалим его рёбра из графа, на оставшихся рёбрах найдём второй путь*), нас постигла бы не удача. Потоку же благодаря обратным рёбрам получается даже при неверном первом пути найти дополняющий путь и получить поток размера два.

3.3. Декомпозиция потока

Def 3.3.1. *Элементарный поток – путь из s в t, по которому течёт x единиц потока.*

Def 3.3.2. *Декомпозиция потока f – представление f в виде суммы элементарных потоков (путей) и циркуляции.*

Lm 3.3.3. $|f| > 0 \Rightarrow \exists$ путь из s в t по рёбрам $e\colon f_e > 0$.

• **Алгоритм декомпозиции за $\mathcal{O}(E^2)$**

Пока $|f| > 0$ найдём путь p из s в f по рёбрам $e\colon f_e > 0$,
по всем рёбрам пути p уменьшим поток на $x = \min_{e \in p} f_e$.

Lm 3.3.4. Время работы $\mathcal{O}(E^2)$

Доказательство. По рёбрам $e\colon f_e > 0$ поток только убывает. После отщепления одного пути, как минимум у одного из рёбер f_e обнулится $\Rightarrow$ не более E поисков пути. ■

3.4. Теорема и алгоритм Форда-Фалкерсона

Def 3.4.1. *Для любых множеств $S, T \subseteq V$ определим*

$$F(S,T) = \sum_{a \in S, b \in T} f_{a \to b}, \quad C(S,T) = \sum_{a \in S, b \in T} c_{a \to b}$$

Сумма включает обратные рёбра $\Rightarrow$ на графе из одного ребра $e\colon t \to s, f_e = 1 \quad F(\{s\},\{t\}) = -1$.

Lm 3.4.2. $\forall v \in V \begin{cases} F(\{v\}, V) = 0 & v \neq s, t \\ F(\{v\}, V) = |f| & v = s \end{cases}$

Lm 3.4.3. $\forall S \quad F(S,S) = 0$

Доказательство. В вместе с каждым ребром в сумму войдёт и обратное ему. ■

Lm 3.4.4. $\forall S, T \quad F(S,T) \leqslant C(S,T)$

Доказательство. Сложили неравенства $f_e \leqslant c_e$ по всем рёбрам $e\colon S \to T$. ■

Def 3.4.5. *Разрез – дизъюнктное разбиение вершин $(S,T) : V = S \sqcup T, s \in S, t \in T$.*

Def 3.4.6. *Величина разреза $(S,T) = C(S,T)$.*

Lm 3.4.7. $\forall$ разреза (S,T) $\quad |f| = F(S,T)$

Доказательство. Интуитивно: поток вытекает из s, нигде не задерживается $\Rightarrow$ он весь протечёт через разрез. Строго: $F(S,T) = F(S,T) + F(S,S) = F(S,V) = F(\{s\},V) + 0 + \cdots + 0 = |f|$. ■

Lm 3.4.8. $\forall$ разреза (S,T) и потока f $\quad |f| \leqslant C(S,T)$

Доказательство. $|f| = F(S,T) \leqslant C(S,T)$ (пользуемся леммами Lm 3.4.4 и Lm 3.4.7). ■

Теорема 3.4.9. Форда-Фалекрсона

(1) $|f| = \max \Leftrightarrow \nexists$ дополняющий путь
(2) $\max |f| = \min C(S,T)$ (максимальный поток равен минимальному разрезу)

Доказательство. $\exists$ дополняющий путь $\Rightarrow$ можно увеличить по нему $f \Rightarrow |f| \neq \max$.
Пусть нет дополняющего пути $\Rightarrow$ dfs из s по ненасыщенным рёбрам не посетит t. Множество посещённых вершин обозначим S, обозначим $T = V \setminus S$. Из S в T ведут только e: $f_e = c_e$.
Значит, $|f| = F(S,T) = C(S,T)$. Из леммы Lm 3.4.8 следует, что $|f| = \max, C(S,T) = \min$. ■

• Поиск минимального разреза

Из доказательства теоремы Thm 3.4.9 мы заодно получили алгоритм за $\mathcal{O}(E)$ поиска min разреза по max потоку.

• Алгоритм Форда-Фалкерсона

> Из теоремы следует простейший алгоритм поиска максимального потока: пока есть дополняющий путь p, найдём его, толкнём по нему $x = \min_{e \in p}(c_e - f_e)$ единиц потока.

Утверждение 3.4.10. Если все $c_e \in \mathbb{Z}$, алгоритм конечен.

Время работы алгоритма мы умеем оценивать сверху только как $\mathcal{O}(|f| \cdot E)$.
При $c_e \leqslant polynom(|V|,|E|)$ получаем $|f| \leqslant polynom(|V|,|E|) \Rightarrow$ Ф.Ф. работает за полином.
При экспоненциально больших c_e на практике мы построим тест: время работы $\Omega(2^{V/2})$.

3.5. Реализация, хранение графа

Первый способ хранения графа более естественный:

```
struct Edge {
    int a, b, f, c, rev; // a → b
};
vector<Edge> c[n]; // c[c[v][i].b][c[v][i].rev] - обратное ребро
for (Edge e : c[v]) // перебор рёбер, смежных с v
    ;
```

Второй часто работает быстрее, и позволяет проще обращаться к обратному ребру.
Поэтому про него поговорим подробнее.

```
struct Edge {
    int a, b, f, c; // собственно ребро
    int next; // интрузивный список, список на массиве
};
vector<Edge> edges;
vector<int> head(n, -1); // для каждой вершин храним начало списка
for (int i = head[v]; i != -1; i = edges[i].next)
    Edge e = edges[i]; // перебор рёбер, смежных с v
```

Добавить орребро можно так:

```
void add(a, b, c):
    edges.push_back({a, b, 0, c}); // прямое
    edges.push_back({b, a, 0, 0}); // обратное
```

Заметим, что взаимообратные рёбра добавляются парами ⇒
$\forall i$ к `edges[i]` обратным является `edges[i^1]`.

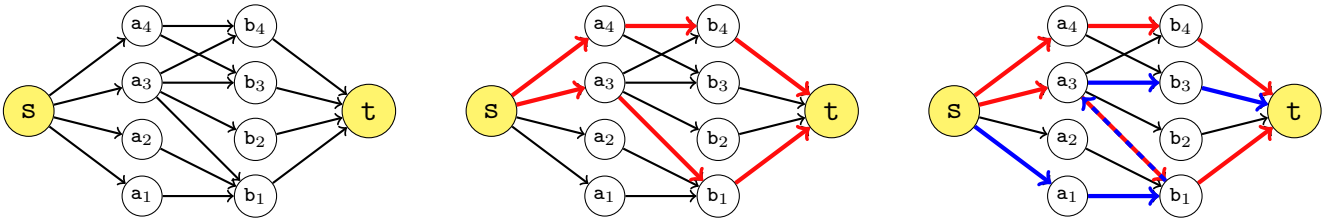
Теперь реализуем алгоритм Ф.Ф. Также, как и Кун, dfs, ищущий путь, сразу на обратном ходу рекурсии будет изменять поток по пути.

```
bool dfs(int v):
    u[v] = 1;
    for (int i = head[v]; i != -1; i = edges[i].next):
        Edge &e = edges[i];
        if (e.f < e.c && !u[e.b] && (e.b == t || dfs(e.b))):
            e.f++, edges[i ^ 1].f--; // не забудьте пересчитать обратное ребро
            return 1;
    return 0;
```

По сути мы лишь нашли путь из s в t в остаточной сети G_f.

Если мы хотим толкать не единицу потока, а $\min_e(c_e - f_e)$, нужно, чтобы dfs на прямом ходу рекурсии насчитывал минимум и возвращал из рекурсии полученное значение.

3.6. Паросочетание, вершинное покрытие



Картинки: собственно сеть ⟶ какой-то поток в сети ⟶ дополняющий путь.

Чтобы с помощью потоков искать паросочетание, добавляем исток и сток, ориентируем рёбра графа из первой доли во вторую. Пропускные способности «из истока» и «в сток» – единицы (ограничение на суммарные поток через вершину). Между долями можно $+\infty$, можно 1.
Алгоритм Форда-Фалекрсона работает за $\mathcal{O}(|f|E) = \mathcal{O}(|M|E) \leqslant \mathcal{O}(VE)$.

Корректность. Следует из двух биекций: (1) между потоками в построенной нами сети и паросочетаниями, (2) между дополняющими путями в нашей цепи и Ч.Д.П. в исходном графе.

• Что нового мы научились делать?

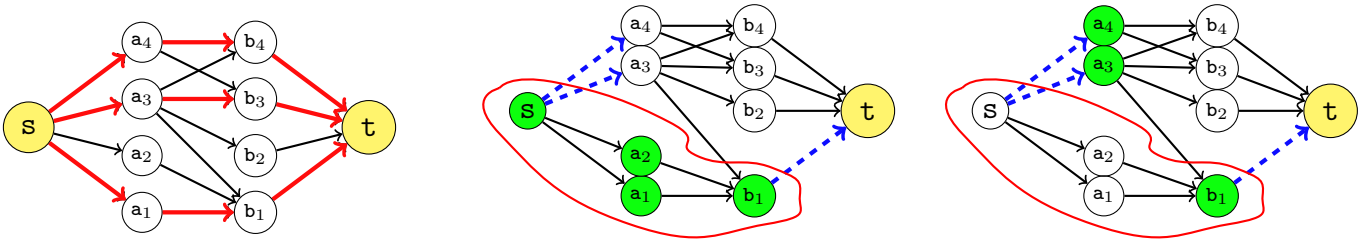
Алгоритм Диница поиска потока (см. дальше) сработает на такой сети быстрее: за $\mathcal{O}(EV^{1/2})$.

Def 3.6.1. *Мультисочетание – обобщение паросочетания, подмножество рёбер графа такое, что для каждой вершины v верно ограничение на максимальную степень w_v.*

Заменим пропускные способности «из истока», «в сток» на w_v. Максимальный поток даст нам максимальное *мультисочетание*. А алгоритм Диница найдёт его за $\mathcal{O}(E^{3/2})$.

3.6.1. Вершинное покрытие

Существует ещё третья биекция: min разрез $\longleftrightarrow$ min вершинное покрытие.



Картинки: max поток $\longrightarrow$ построенный по нему min разрез $\longrightarrow$ min вершинное покрытие.

Рёбрам между долями сделаем $c_e = +\infty \Rightarrow$ min разрезе их, конечно, не будет.
Мы уже умеем строить min cover, используя dfs от Куна: $C = B^+ \cup A^-$.
Осталось заметить, что в построенном разрезе $S \sqcup T$ по построению $S = \{s\} \cup A^+ \cup B^+$.

Новое: научились искать взвешенное минимальное вершинное покрытие в двудольном графе. Для этого поменяем пропускные способности рёбрам «из истока» и «в сток» на веса вершин.

3.7. Леммы, позволяющие работать с потоками

Lm 3.7.1. Условие $0 \leqslant f_e \leqslant c_e$ можно заменить на $(f_e \leqslant c_e) \wedge (-f_e \leqslant 0)$.

То есть, и прямые, и обратные рёбра обладают пропускными способностями, ограничениями сверху на поток, по ним текущий. А про ограничения снизу можно не думать.

Lm 3.7.2. f_1 и f_2 – потоки в $G \Rightarrow f_2 - f_1$ – поток в G_{f_1}.

Доказательство. $\forall$ ребра e имеем $f_{2_e} \leqslant c_e \Rightarrow f_{2_e} - f_{1_e} \leqslant c_e - f_{1_e}$.
Это верно и для прямых, и для обратных. Теперь проверим сумму в вершине:
$\forall v \sum_{e \in out(v)} (f_{2_e} - f_{1_e}) = \forall v \neq s,t \sum_{e \in out(v)} f_{2_e} - \forall v \sum_{e \in out(v)} f_{1_e} = 0 - 0 = 0.$ ■

Следствие 3.7.3. $\forall f_1, f_2 \colon |f_1| = |f_2| \Rightarrow f_2$ можно получить из f_1 добавлением циркуляции из G_{f_1}.

Lm 3.7.4. $|f_2 - f_1| = |f_2| - |f_1|$

Доказательство. Также, как в Lm 3.7.2, распишем сумму для вершины s. ■

Lm 3.7.5. Если f_2 – поток в G_{f_1}, $f_1 + f_2$ – поток в G

Lm 3.7.6. $|f_1 + f_2| = |f_1| + f_2|$

3.8. Алгоритмы поиска потока

3.8.1. Эдмондс-Карп за $\mathcal{O}(VE^2)$

Алгоритм прост: путь ищем bfs-ом, проталкиваем по пути $\min_e(c_e - f_e)$. Конец.

Lm 3.8.1. После увеличения потока по пути, найденному bfs-ом,
для любой вершины v расстояние от истока не уменьшится: $\forall v \; d[s, v] = d[v] \nearrow$.

Доказательство. От противного. Пусть после увеличения потока f по пути p расстояния уменьшились. Расстояния в G_f обозначим d_0, а в G_{f+p} обозначим d_1.

Возьмём вершину v: $d_1[v] < d_0[v]$, а из таких $d_1[v] = \min$. Рассмотрим кратчайший путь q из s в v: $s \rightsquigarrow \cdots \rightsquigarrow x \to v$, $d_1[v] = d_1[x] + 1$. Поскольку v – минимальная по $d_1[v]$ вершина из тех, для кого расстояние уменьшилось, имеем $d_0[x] \leqslant d_1[x]$.

Пусть ребро e: $x \to v$. $e \in q$, $q \in G_{f+p}$. Рассмотрим два случая: $e \in G$, $e \notin G$.

1. $e \in G$, кратчайшие путь $p \leqslant$ какой-то путь `[*]` $\Rightarrow d_0[v] \leqslant d_0[x] + 1 \leqslant d_1[x] + 1 = d_1[v]$. ?!?
2. $e \notin G \Rightarrow p$ прошёл по обратному к $e \Rightarrow d_0[v] = d_0[x] - 1 \leqslant d_1[x] - 1 = d_1[v] - 2$. ?!? ■

`[*]` – здесь и только здесь мы пользовались тем, что p – кратчайший.

• Время работы Эдмондса-Карпа

Толкаем по пути мы всё ещё $\min_e(c_e - f_e) \Rightarrow$ после каждого bfs-а хотя бы одно ребро насытится. Чтобы ещё раз пройти по насыщенному ребру e, нужно сперва уменьшить по нему потока $\Rightarrow$ пройти по обратному к e. Рассмотрим e: $a \to b$, кратчайший путь прошёл через $e \Rightarrow$ $d[b] = d[a] + 1$. Когда кратчайший путь пройдёт через обратное к e имеем

$$d'[a] = d'[b] + 1 \overset{Lm\ 3.8.1}{\geqslant} d[b] + 1 = d[a] + 2$$

Расстояние до a между двумя насыщениями e увеличится хотя бы на 2 $\Rightarrow$ каждое ребро e насытится не более $\frac{V}{2}$ раз $\Rightarrow$ суммарное число насыщений $\leqslant \frac{VE}{2} \Rightarrow$ Э.К. работает за $\mathcal{O}(VE^2)$

Следствие 3.8.2. В графах с $\mathbb{R}$ пропускными способностями $\exists$ max поток.

Доказательство. В оценке времени работы Э.К. мы не пользовались целочисленностью. По завершении Э.К. нет дополняющих путей $\Rightarrow$ по Thm 3.4.9 поток максимален. ■

3.8.2. Масштабирование за $\mathcal{O}(E^2 \log U)$

Будем пытаться сперва найти толстые пути:
перебирать $k \downarrow$, искать пути в остаточной сети, по которым можно толкнуть хотя бы 2^k.
Для этого dfs-у разрешим ходить только по рёбрам: $f_e + 2^k \leqslant c_e$.

```
for k = logU .. 0:
    u <-- 0
    while dfs(s, 2^k):
        u <-- 0
        flow += 2^k
```

Ф.Ф. искал любой путь в G_f, мы ищем в G_f пути толщиной 2^k. Время работы $\mathcal{O}(E^2 \log U)$.

Алгоритм можно соптимизировать, толкая по пути не 2^k, а $\min_e(c_e - f_e) \geqslant 2^k$.
Асимптотика в худшем случае не улучшится. На практике алгоритм ведёт себя как $\approx E^2$.

• Доказательство времени работы

Поток после фазы, на которой мы искали 2^k-пути, обозначим F_k. В остаточной сети G_{F_k} нет пути толщины $2^k \Rightarrow$ есть разрез, для всех рёбер которого верно $c_e - f_e < 2^k$.
Рассмотрим декомпозицию $F_{k-1} - F_k$ на пути. Все пути имеют толщину 2^{k-1}, все проходят через разрез $\Rightarrow$ все по разным рёбрам разреза $\Rightarrow$ их не больше, чем рёбер в разрезе $\leqslant E$.
Доказали, что путей при переходе от F_k к F_{k-1} не более E.

Лекция #4: Потоки (быстрые)

30 сентября

4.1. Алгоритм Диница

У нас уже есть алгоритм Эдмондса-Карпа, ищущий $\mathcal{O}(VE)$ путей за $\mathcal{O}(E)$ каждый.
Построим сеть кратчайших путей, расстояние $s \rightsquigarrow t$ обозначим d.
Слоем A_i будем называть вершины на расстоянии i от s.
Э.К. сперва найдёт сколько-то путей длины d, затем расстояние $s \rightsquigarrow t$ увеличится.

Lm 4.1.1. Пока $\exists$ путь длины d, он имеет вид $s = v_0 \in A_0, v_1 \in A_1, v_2 \in A_2, \ldots, t = v_d \in A_d$

Доказательство. В первый момент это очевидно. Затем в G_f будут появляться новые рёбра – обратные к $v_i \to v_{i+1}$. Все такие рёбра идут назад по слоям $\Rightarrow$ новых рёбер идущих вперёд по слоям не образуется $\Rightarrow$ каждый раз при поиске пути единственный способ за d шагов из s попасть в t – d раз по одному из старых рёбер идти ровно в следующий слой. ■

Научимся искать все пути длины d за $\mathcal{O}(E + dk_d)$, где k_d – количество путей.

Выделим множество E' – рёбра $A_i \to A_{i+1}$ в G_f. Будем запускать dfs по E'.
Модифицируем dfs: если пройдя по рёбру e, dfs не нашёл путь до t, он удалит e из E'.
Каждый из k_d dfs-ов сделал d успешных шагов и x_i неуспешных, но $\sum x_i \leqslant E$, так как после каждого неуспешного шага мы удаляем ребро из E'.

```
1 bool dfs(int v):
2     while (head'[v] != -1):
3         Edge &e = edges[head'[v]];
4         if (e.f < e.c && (e.end == t || dfs(e.b)))
5             // нашли путь
6         head'[v] = e.next;
```

Заметьте, что массив пометок вершин, обычный для любого dfs, тут можно не использовать.

• Алгоритм Диница

Состоит из фаз вида:
(1) запустить bfs, который построил слоистую сеть и нашёл d.
(2) пока в слоистой сети есть путь длины d,
найдём его dfs-ом и толкнём по нему $\min_e(c_e - f_e)$ единиц потока.

Теорема 4.1.2. Время работы алгоритма Диница $\mathcal{O}(V^2E)$.

Доказательство. Фаз всего не более V штук, так как после каждой $d \uparrow$
Фаза с расстоянием d работает за $\mathcal{O}(E + d \cdot k_d)$.

$$\sum(E + d \cdot k_d) \leqslant VE + V\sum k_d \leqslant VE + V(VE)$$

Последнее известно из алгоритма Эдмондса-Карпа. ■

• Алгоритм Диница с масштабированием потока

Масштабирование потока – не только конкретный алгоритм, но и общая идея:

```
1 for (int k = log U; k >= 0; k--)
2     пускаем поток на графе с пропускными способностями (c_e - f_e)/2^k
```

Давайте искать поток именно алгоритмом Диница.

Теорема 4.1.3. Время работы алгоритма Диница с масштабированием $\mathcal{O}(VE\log U)$.

Доказательство. Каждая из фаз масштабирование – алгоритм Диница, который найдёт не более E путей и работает за время (делаем то же, что и в теореме Thm 4.1.2):

$$\sum(E + d\cdot k_d) \leqslant VE + V\sum k_d \leqslant VE + VE = \mathcal{O}(VE)$$

Значит, суммарно все $\log U$ фаз отработают за $\mathcal{O}(VE\log U)$. ■

4.2. Алгоритм Хопкрофта-Карпа

Lm 4.2.1. В единичной сети ($c_e \equiv 1$) фаза Диниц работает за $\mathcal{O}(E)$

Доказательство. Если dfs пройдёт по ребру e, он его в любом случае удалит – и если не найдёт по нему путь, и если найдёт по нему путь: $(c_e = 1) \Rightarrow e$ насытится. ■

Мы уже умеем искать паросочетание за $\mathcal{O}(VE)$ через потоки.
Давайте в том же графе запустим алгоритм Диница.

Теорема 4.2.2. Число фаз Диница на сети для поиска паросочетание не более $2\sqrt{V}$.

Доказательство. Сделаем первые $\sqrt{V}$ фаз, получили поток f, посмотрим на G_f.
В остаточной сети все пути имеют длину хотя бы $\sqrt{V}$. Вспомним биекцию между доппутями в G_f и ДЧП для паросочетания $\Rightarrow$ все ДЧП тоже имеют длину хотя бы $\sqrt{V}$.
Пусть поток f задаёт паросочетание P, рассмотрим максимальное M.
$M \triangledown P$ содержит $k = |M| - |P|$ непересекающихся ДЧП, каждый длины $\geqslant \sqrt{V} \Rightarrow$
$k \leqslant \sqrt{V} \Rightarrow$ Динице осталось найти $\leqslant \sqrt{V}$ путей.
Осталось заметить, что за каждую фазу Диниц находит хотя бы один путь. ■

- **Алгоритм Хопкрофта-Карпа**

```
1  bool dfs(int v):
2      for (x ∈ N(v)): // x - сосед во второй доле
3          if (used2[x]++ == 0) // проверили, что в x попадаем впервые
4              if (pair2[x] == -1 || (dist[pair2[x]] == dist[v]+1 && dfs(pair2[x])):
5                  pair1[v] = x, pair2[x] = v
6                  return 1
7      return 0
8  while (bfs нашёл путь свободной в свободную): // цикл по фазам
9      used2 <-- 0 // пометки для вершин второй доли
10     for (v ∈ A): // вершины первой доли
11         if (pair1[v] == -1): // вершина свободна
12             dfs(v)
```

$\forall$ вершины v второй доли в G_f из v исходит не более одного ребра.
Для свободной вершины это ребро в сток t, для несвободной в её пару в первой доле.
Значит, заходить в v dfs-ам одной фазы Диница имеет смысл только один раз.

Давайте вместо «удаления рёбер» помечать вершины второй доли. Посмотрим на происходящее, как на поиск ДЧП для паросочетания. Поймём, что сток с истоком нам особо не нужны...

4.3. Теоремы Карзанова

Определим пропускную способность вершины:

$$c[v] = \min(c_{in}[v], c_{out}[v]), \text{где } c_{in}[v] = \sum_{e \in in[v]} c_e, c_{out}[v] = \sum_{e \in out[v]} c_e$$

Теорема 4.3.1. Число фаз алгоритма Диницы не больше $2\sqrt{C}$, где $= \sum_v c[v]$.

Доказательство. Заметим, что $\forall$ потока f и вершины $v \neq s, t$ величины $c_{in}[v], c_{out}[v], c[v]$ равны значениям в исходном графе. Запустим первые $\sqrt{C}$ фаз, получим поток f_0, пусть $|f^*| = \max$, рассмотрим декомпозицию $f^* - f_0$. Она состоит из $k = |f^*| - |f_0|$ единичных путей длины хотя бы $\sqrt{C}$ (не считая s и t). Обозначим α_v – сколько путей проходят через v. Тогда:

$$k\sqrt{C} \leqslant \sum_{v \neq s,t} \alpha_v \leqslant \sum_{v \neq s,t} c[v] = C \Rightarrow k \leqslant \sqrt{C}$$

Получили, что число фаз $\leqslant \sqrt{C} + k \leqslant 2\sqrt{C}$. ■

Следствие 4.3.2. Из теоремы следует время работы Хопкрофта-Карпа.

Утверждение 4.3.3. В единичных сетях $C \leqslant E \Rightarrow$ алгоритм Диница работает за $\mathcal{O}(E^{3/2})$.

Теорема 4.3.4. Число фаз алгоритма Диницы не больше $2U^{1/3}V^{2/3}$, где $U = \max_e c_e$.

Доказательство. Запустим первые k фаз (оптимальное k выберем позже), на $(k{+}1)$-й получим слоистую сеть из $\geqslant k{+}1$ слоёв. Обозначим размеры слоёв $a_0, a_1, a_2, \ldots, a_k$.
Тогда величина min разреза не более $\min_{i=1..k}(a_{i-1}a_iU)$.
Максимум такого минимума достигается при $a_1 = a_2 = \ldots a_k = \frac{V}{k}$.
Получили разрез размера $U(\frac{V}{k})^2 \Rightarrow$ осталось не более чем столько фаз $\Rightarrow$

$$\text{всего фаз не более } f(k) = k + U(\tfrac{V}{k})^2. \quad k \uparrow, U(\tfrac{V}{k})^2 \downarrow.$$

Асимптотический минимум f достигается при $k = U(\frac{V}{k})^2 \Rightarrow k^3 = UV^2$, число фаз $\leqslant 2(UV^2)^{1/3}$. ■

4.4. Диниц с link-cut tree

Улучшим время одной фазы алгоритмы Диница с $\mathcal{O}(VE)$ до $\mathcal{O}(E \log V)$.
Построим остовное дерево с корнем в t по входящим не насыщенным рёбрам.

Теперь E раз пускаем поток, по пути дерева $s \rightsquigarrow t$ и перестраиваем дерево.
Для этого находим на пути $s \rightsquigarrow t$ любое одно насытившееся ребро $a \to b$, разрезаем его, и для вершины a добавляем в дерево следующее ребро из $out[a]$. Цикла появиться не может: рёбра идут вперёд по слоям. Зато у a могли просто кончиться рёбра, тогда a объявляем тупиковой веткой развития, и рекурсивно разрезаем ребро, входящее в a.

Заметим, что *link-cut-tree* со *splay-tree* умеет делать все описанные операции за $\mathcal{O}(\log V)$:

- Поиск минимума и позиции минимума величины $c_e - f_e$ на пути.
- Уменьшение величины $c_e - f_e$ на пути.
- Разрезание ребра (cut), проведение нового ребра (link).

Один cut может рекурсивно удалить много рёбер, сильно перестроить дерево. Несмотря на это каждое ребро удалится не более одного раза $\Rightarrow$ суммарное время всех cut – $\mathcal{O}(E \log V)$.

4.5. Глобальный разрез

Задача: найти разбиение $V = A \sqcup B$: $A, B \neq \varnothing, C(A, B) \to \min$.
Простейшее решение: переберём s, t и найдём разрез между ними. Конечно, можно взять $s = 1$.
Время работы $\mathcal{O}(V \cdot Flow)$.
На практике покажем, что на единичных сетях эта идея работает уже за $\mathcal{O}(E^2)$.

4.5.1. Алгоритм Штор-Вагнера

Выберем $a_1 = 1$. Пусть $A_i = \{a_1, a_2, \ldots, a_i\}$. Определим $a_{i+1} = v \in V \setminus A_i \colon C(A_i, \{v\}) = \max$.

Утверждение 4.5.1. Минимальный разрез между a_n и a_{n-1} – $S = \{a_n\}, T = V \setminus S$, где $n = |V|$.

Доказательство. Можно прочесть на e-maxx. ■

Алгоритм: или a_n и a_{n-1} по разные стороны оптимального глобального разреза, и ответ равен $C(\{a_n\}, V \setminus \{a_n\})$, или a_n и a_{n-1} можно стянуть в одну вершину.

Время работы: V фаз, каждая за $\mathcal{O}(Dijkstra) \Rightarrow \mathcal{O}(V(E + V \log V))$.

4.5.2. Алгоритм Каргера-Штейна

Пусть $c_e \equiv 1$. Минимальную степень обозначим k.
$\exists v \colon deg_v = k \Rightarrow C(\{v\}, V \setminus \{v\}) = k \Rightarrow$ в min разрезе не более k рёбер. $E = \frac{1}{2} \sum deg_v \geqslant \frac{1}{2} kV$.
Возьмём случайное ребро e, вероятность того, что оно попало в разрез $Pr[e \in cut] \leqslant \frac{k}{E} \leqslant \frac{2}{V}$.

• **Алгоритм Каргера.**

Пока в графе > 2 вершин, выбираем случайное ребро, не являющееся петлёй, стягиваем его концы в одну вершину. В конце $V' = \{A, B\}$, объявляем минимальным разрезом $V = A \sqcup B$.

Время работы: $T(V) = V + T(V - 1) = \Theta(V^2)$. Здесь V – время стягивания двух вершин.

Вероятность успеха: ни разу не ошиблись с $Pr \geqslant \frac{V-2}{V} \cdot \frac{V-3}{V-1} \cdot \frac{V-4}{V-2} \cdot \ldots \cdot \frac{1}{3} = \frac{2 \cdot 1}{V \cdot (V-1)} \geqslant \frac{2}{V^2}$.
Чтобы алгоритм имел константную вероятность ошибки, достаточно запустить его $V^2 \Rightarrow$ получили RP-алгоритм за $\mathcal{O}(V^4)$.

• **Алгоритм Каргера-Штейна.**

В оценке $\frac{V-3}{V-1} \cdot \frac{V-4}{V-2} \cdot \ldots \cdot \frac{1}{3}$ большинство первых сомножителей близки к 1. Последние сомножители – $\frac{2}{4}, \frac{1}{3}$ напротив весьма малы $\Rightarrow$ остановимся, когда в графе останется $\frac{V}{\sqrt{2}}$ вершин.
Вероятность ни разу не ошибиться при этом будет $\frac{V/\sqrt{2}(V/\sqrt{2}-1)}{V(V-1)} \approx \frac{1}{2}$.
Время, потраченное на $(V - V/\sqrt{2})$ сжатий – $\Theta(V^2)$.

После этого сделаем два рекурсивных вызова от получившегося графа с $V/\sqrt{2}$ вершинами. Алгоритм рандомизированный $\Rightarrow$ ветки рекурсии могут дать разные разрезы $\Rightarrow$ вернём минимальный из полученных.

Время работы: $T(V) = V^2 + 2T(\frac{V}{\sqrt{2}}) = V^2 + 2(\frac{V}{\sqrt{2}})^2 + 4T(\frac{V}{\sqrt{2}^2}) = \cdots = V^2 \log V$.

Вероятность ошибки. Ищем $p(V)$, вероятность успеха на графе из V вершин.

Обозначим $p(V) = q_k, p(V/\sqrt{2}) = q_{k-1}, \ldots, \Rightarrow$
$q_i = \frac{1}{2}(1 - (1-q_{i-1})^2) = q_{i-1} - \frac{1}{2}q_{i-1}^2 \Rightarrow q_i - q_{i-1} = -\frac{1}{2}q_{i-1}^2$. Левая часть похожа на производную $\Rightarrow$ решим диффур $q'(x) = -\frac{1}{2}q(x)^2 \Leftrightarrow \frac{-q'(x)}{q(x)^2} = \frac{1}{2} \Leftrightarrow \frac{1}{q(x)} = \frac{1}{2}x + C \Rightarrow q_k = \Theta(\frac{1}{k}) \Rightarrow p(V) = \Theta(\frac{1}{\log V})$.

Можно пробовать ветвиться не только на две ветки, но именно на две оптимально.

Короткая и быстрая реализация получается через `random_shuffle` исходных рёбер.
Подробнее в разборе практики.

4.6. (*) Алгоритм Push-Relabel

Конспект по этой теме лежит в отдельном файле.

Лекция #5: Mincost

7 октября 2024

5.1. Mincost k-flow в графе без отрицательных циклов

Сопоставим всем прямым рёбрам вес (стоимость) $w_e \in \mathbb{R}$.

Def 5.1.1. *Стоимость потока* $W(f) = \sum_e w_e f_e$. *Сумма по прямым рёбрам.*

Обратному к e рёбру $\overline{e}$ сопоставим $w_{\overline{e}} = -w_e$.

Если толкнуть поток сперва по прямому, затем по обратному к нему ребру, стоимость не изменится. Когда мы толкаем единицу потока по пути `path`, изменение потока и стоимости потока теперь выглядят так:

```
for (int e : path):
    edges[e].f++
    edges[e ^ 1].f--
    W += edges[e].w;
```

Задача mincost k-flow: найти поток $f\colon |f| = k, W(f) \to \min$

При решении задачи мы будем говорить про веса путей, циклов, "отрицательные циклы", кратчайшие пути... Везде вес пути/цикла – сумма весов рёбер (w_e).

Решение #1. Пусть в графе нет отрицательных циклов, а также все $c_e \in \mathbb{Z}$.
Тогда по аналогии с алгоритмом Ф.Ф., который за $\mathcal{O}(k \cdot \texttt{dfs})$ искал поток размера k, мы можем за $\mathcal{O}(k \cdot \texttt{FordBellman})$ найти mincost поток размера k. Обозначим f_k оптимальный поток размера $k \Rightarrow f_0 \equiv 0, f_{k+1} = f_k + path$, где $path$ – кратчайший в G_{f_k}.

Lm 5.1.2. $\forall k, |f| = k \quad (W(f) = \min) \Leftrightarrow (\nexists$ отрицательного цикла в $G_f)$

Доказательство. Если отрицательный цикл есть, увеличим по нему поток, $|f|$ не изменится, $W(f)$ уменьшится. Пусть $\exists f^*\colon |f^*| = |f|, W(f^*) < W(f)$, рассмотрим поток $f^* - f$ в G_f.
Это циркуляция, мы можем декомпозировать её на циклы $c_1, c_2, \ldots, c_k$.
Поскольку $0 > W(f^* - f) = W(c_1) + \cdots + W(c_k)$, среди циклов c_i есть отрицательный. ■

Теорема 5.1.3. Алгоритм поиска mincost потока размера k корректен.

Доказательство. База: по условию нет отрицательных циклов $\Rightarrow f_0$ корректен.
Переход: обозначим f^*_{k+1} mincost поток размера $k{+}1$, смотрим на декомпозицию $\Delta f = f^*_{k+1} - f_k$.
$|\Delta f| = 1 \Rightarrow$ декомпозиция = путь p + набор циклов. Все циклы по Lm 5.1.2 неотрицательны $\Rightarrow$ $W(f_k + p) \leqslant W(f^*_{k+1}) \Rightarrow$, добавив, кратчайший путь мы получим решение не хуже f^*_{k+1}. ■

Lm 5.1.4. Если толкнуть сразу $0 \leqslant x \leqslant \min_{e \in p}(c_e - f_e)$ потока по пути p,
то получим оптимальный поток размера $|f| + x$.

Доказательство. Обозначим f^* оптимальный поток размера $|f| + x$, посмотрим на декомпозицию $f^* - f$, заметим, что все пути в ней имеют вес $\geqslant W(p)$, а циклы вес $\geqslant 0$. ■

Теорема 5.1.5. Пусть $w_e \in \mathbb{Z} \cap [1, W] \Rightarrow$ умеем искать mincost k-flow за $\mathcal{O}(WVE \cdot \texttt{FordBellman})$
Доказательство. Используем алгоритм, толкающий $\min_{e \in p}(c_e - f_e)$.
Также, как в Эдмондсе-Карпе, каждый раз какое-то ребро насытится.
Каждое ребро насытится не более $\frac{maxDist}{2} \leqslant \frac{WV}{2}$ раз $\Rightarrow$ всего $\mathcal{O}(WVE)$ путей ■

5.2. Потенциалы и Дейкстра

Для ускорения хотим Форда-Беллмана заменить на Дейкстру.
Для корректности Дейкстры нужна неотрицательность весов.
В прошлом семестре мы уже сталкивались с такой задачей, когда изучали алгоритм Джонсона.

• **Решение задачи mincost k-flow.**

Запустим один раз Форда-Беллмана из s, получим массив расстояний d_v, применим потенциалы d_v к весам рёбер:

$$e\colon a \to b \Rightarrow w_e \to w_e + d_a - d_b$$

Напомним, что из корректности d имеем $\forall e\ d_a + w_e \geqslant d_b \Rightarrow w_e^{'} \geqslant 0$.
Более того: для всех рёбер e кратчайших путей из s верно $d_a + w_e = d_b \Rightarrow w_e^{'} = 0$.

В G_f найдём Дейкстрой из s кратчайший путь p и расстояния d'_v.
Пустим по пути p поток, получим новый поток $f^{'} = f + p$.
В сети $G_f^{'}$ могли появиться новые рёбра (обратные к p). Они могут быть отрицательными.
Пересчитаем веса:

$$e\colon a \to b \Rightarrow w_e \to w_e + d'_a - d'_b$$

Поскольку d' – расстояния, посчитанные в G_f, все рёбра из G_f останутся неотрицательными.
p – кратчайший путь, все рёбра p станут нулевыми $\Rightarrow$ рёбра обратные p тоже будут нулевыми.

• **Псевдокод**

```
1 def applyPotentials(d):
2     for e in Edges:
3         e.w = e.w + d[e.a] - d[e.b]
4 d <-- FordBellman(s)
5 applyPotentials(d)
6 for i = 1..k:
7     d, path <-- Dijkstra(s)
8     for e in path: e.f += 1, e.rev.f -= 1
9     applyPotentials(d)
```

5.3. Задачи на mincost поток, паросочетания

Чтобы найти паросочетания min веса, достаточно построить поточную сеть для поиска паросочетания и расставить веса: 0 у рёбер, смежных с истоком/стоком w_{ij} у рёбер между долями.
Для поиска паросочетания max веса, ищем mincost поток на весах $-w_{ij}$.

Корректность: биекция с сохранением веса между потоками и паросочетаниями.

Время работы с Дейкстрой: $VE + flow \cdot V^2 = \mathcal{O}(V^3)$.

5.4. Графы с отрицательными циклами

Задача: найти mincost циркуляцию.

Алгоритм Клейна: пока в G_f есть отрицательный цикл, пустим по нему $\min_e(c_e - f_e)$ потока.

Пусть $\forall e\ c_e, w_e \in \mathbb{Z} \Rightarrow W(f)$ каждый раз уменьшается хотя бы на 1 $\Rightarrow$ алгоритм конечен.

Задача: найти mincost k-flow в графе с отрицательными циклами.

Решение #1: добавить ребро $e\colon t \to s, c_e = k, w_e = -\infty$, найти mincost циркуляцию.

Решение #2: найти mincost циркуляцию, перейти от f_0 за k итераций к f_k.
Решение #3: найти любой поток $f\colon |f| = k$, в G_f найти mincost циркуляцию, сложить с f.

5.5. Mincost flow

Задача: найти $f\colon W(f) = \min$, размер f не важен.

Если в графе есть отрицательные циклы $\Rightarrow$ всё равно придётся искать mincost циркуляцию $\Rightarrow$

Решение #1: добавить ребро $e\colon t \to s, c_e = +\infty, w_e = -\infty$, найти mincost циркуляцию.
Иначе можно пытаться сделать лучше (проще).

Решение #2: пытаемся угадать k линейным или бинарным поиском.

Обозначим f_k – оптимальный поток размера k, p_k кратчайший путь в G_{f_k}.

Lm 5.5.1. $W(p_k)\nearrow$, как функция от k.

Доказательство. Аналогично доказательству леммы для Эдмондса-Карпа Lm 3.8.1.
От противного. Был поток f, мы увеличили его по кратчайшему пути p.
Расстояния в G_f обозначим d_0, в G_{f+p} – d_1.
Возьмём $v\colon d_1[v] < d_0[v]$, а из таких ближайшую к s в дереве кратчайших путей.
Рассмотрим кратчайший путь q в G_{f+p} из s в v: $s \rightsquigarrow \cdots \rightsquigarrow x \to v$.
$e = (v \to x), d_1[v] = d_1[x] + w_e, d_1[x] \geqslant d_0[x] \Rightarrow d_1[v] \geqslant d_0[x] + w_e \Rightarrow$ ребра $(x \to v)$ нет в $G_f \Rightarrow$
ребро $(v \to x) \in p \Rightarrow d_0[x] = d_0[v] + w_{\bar{e}} = d_0[v] - w_e \Rightarrow$
$d_1[v] = d_1[x] + w_e \geqslant d_0[x] + w_e = (d_0[v] - w_e) + w_e = d_0[v]$. Противоречие. ■

Следствие 5.5.2. $(W(f_k) = \min) \Leftrightarrow (W(p_{k-1}) \leqslant 0 \wedge W(p_k) \geqslant 0)$.

Осталось найти такое k бинпоиском или линейным поиском. На текущий момент мы умеем искать f_k или за $\mathcal{O}(k \cdot VE)$ с нуля, или за $\mathcal{O}(VE)$ из $f_{k-1} \Rightarrow$ линейный поиск будет быстрее.

5.6. Полиномиальные решения

Mincost flow мы можем бинпоиском свести к mincost k-flow.
Mincost k-flow мы можем поиском любого потока размера k свести к mincost циркуляции.
Осталось научиться за полином искать mincost циркуляцию.

• **Решение #1:** модифицируем алгоритм Клейна, будем толкать $\min_e(c_e - f_e)$ потока по циклу min среднего веса. Заметим, что ($\exists$ отрицательный цикл) $\Leftrightarrow$ (min средний вес < 0).

Решение работает за $\mathcal{O}(VE\log(nC))$ поисков цикла. Цикл ищется алгоритмом Карпа за $\mathcal{O}(VE)$. Доказано будет на практике.

• **Решение #2:** Capacity Scaling.

Начнём с графа $c'_e \equiv 0$, в нём mincost циркуляция тривиальна.
Будем понемногу наращивать c'_e и поддерживать mincost циркуляцию. В итоге хотим $c'_e \equiv c_e$.

```
1  for k = logU..0:
2      for e in Edges:
3          if c_e содержит бит 2^k:
4              c'_e += 2^k // e: ребро из a_e в b_e‘
5              Найдём p - кратчайший путь a_e → b_e
6              if W(p) + w_e ⩾ 0:
7                  нет отрицательных циклов ⇒ циркуляция f оптимальна
8              else:
9                  пустим 2^k потока по циклу p + e (изменим f)
10                 пересчитаем потенциалы, используя расстояния, найденные Дейкстрой
```

Время работы алгоритма $E\log U$ запусков Дейкстры $= E(E + V\log V)\log U$.

Lm 5.6.1. После 9-й строки циркуляция f снова минимальна.

Доказательство. f – минимальная циркуляция до 4-й строки, f' – после.
Как обычно, рассмотрим $f' - f$. Это тоже циркуляция. Декомпозируем её на единичные циклы. Любой цикл проходит через e (иначе f не оптимальна). Через e проходит не более 2^k циклов. Каждый из этих циклов имеет вес не меньше веса $p + e \Rightarrow W(f') \geqslant W(f + 2^k(p + e))$. ■

5.7. (*) Cost Scaling

Задача, которую решаем: mincost circulation.

• **Решение**

База: $f \equiv 0$, если в G_f нет отрицательных рёбер, то f оптимален.

Общая идея: научимся делать шаг $\forall e \in G_f\ w_e \geqslant -2\varepsilon \to \forall e \in G_f\ w_e \geqslant -\varepsilon$.

Когда остановиться? Если исходные $c_e, w_e \in \mathbb{Z}$, $f_e \in \mathbb{Z}$ и $\varepsilon < \frac{1}{n}$, то вес любого цикла $> -1 \Rightarrow$ из целочисленности получаем оптимальность потока.

Как избавиться от отрицательных рёбер? Насытим их. Получим избытки-недостатки, перенаправим избытки в недостатки, разрешая толкать поток только по отрицательным рёбрам. Избыток есть, а исходящие отрицательных рёбер нет? Подгоним потенциалы и получим новое отрицательное.

$\Rightarrow$ схема решения для целых c_e, w_e:

```
// push : поменять потоки рёбра и обратного, пересчитать избытки x[v]
for e do w_e *= n+1
for (int EPS = max|w_e|; EPS >= 1; EPS /= 2)
    for e do if w_e < 0 then push(e, c_e - f_e)
    while ∃ v : x[v] > 0 do
        for e : v → u do push(e, min(x[v], c_e-f_e))
        if x[v] > 0 then что-то сделаем с потенциалами, см. далее
```

$\forall a, b\ \ w_e^p = w_e + p_a - p_b \Rightarrow$ найдём $e\colon v \to u, w_e^p = \min \geqslant 0$ и уменьшим p_v на $w_e^p + \varepsilon$.
Вес входящих в v рёбер увеличился, вес исходящих $\geqslant -\varepsilon$, новый вес ребра e равен $-\varepsilon$.

Если потенциалы здесь воспринимать, как «высоты» с шагом $-\varepsilon$, мы по сути каждую фазу $-2\varepsilon \to -\varepsilon$ запускаем технику push-relabel. Все оценки для push-relabel будут верны. Для того, чтобы показать, что как и «увеличений высот», так и «увеличений потенциалов» мало, заметим, что для всех вершин у которых избыток всё ещё положителен $\exists$ путь по рёбрам $c_e - f_e > 0$ в вершину с отрицательным избытком, для которой на текущей фазе потенциал ещё не менялся.

Время работы на практике. Эксперимент показывает, что на не специальных тестах простейшая реализация (код выше) на графах размера $n = 100$, $m = 1000$ даёт уже $\leqslant 3m$ шагов на фазу. В коде выше мы делим на $\alpha = 2$, можно пробовать и другие значения.
В задачах `assignment` и `mincost` с контеста оптимально $\alpha = 3$.

Поиск mincost потока. Алгоритм выше ищет **циркуляцию**. Добавим ребро $e\colon t \to s$ с нужными w_e, c_e в зависимости от задачи (mincost flow, mincost maxflow, mincost k-flow).

Поиск mincost паросочетания в двудольном графе. Можно добавить ребро $e\colon t \to s, c_e = n$, $w_e = -\max W \cdot (n+1) \cdot n$ и в явном виде использовать код выше. Можно, пользуясь специфичной структурой графа, получить более быстрый алгоритм (см. статью и код ниже).

Краткое описание фазы для паросочетания в двудольном графе. Кун. Для вершин второй доли храним пары в первой. Пытаемся идти в минимальную. Если там уже занято, всё равно идём, вытесняем вершину-старую-пару, для которой предстоит продолжить поиск, и меняем потенциал вершины первой доли на «разницу второго и первого минимума» $+\varepsilon$.

[Lection: Cost scaling (part 1)]
[Lection: Cost scaling (part 2, end of proof)]
[Практически быстрые алгоритмы для задачи о назначениях]
[Код паросочетания через mincost-scaling]
[Код потока через mincost-scaling]

Лекция #6: Базовые алгоритмы на строках

14 октября 2024

6.1. Обозначения, определения

s, t — строки, $|s|$ — длина строки, $\overline{s}$ — перевёрнутая s,
`s[l:r)` и `s[l:r]` — подстроки,
`s[0:i)` — префикс, `s[i:|s|-1]` = `s[i:]` — суффикс.
Σ — алфавит, $|\Sigma|$ — размер алфавита.
Говорят, что s — подстрока t, если $\exists\, l, r\colon s = t[l{:}r)$.

6.2. Поиск подстроки в строке

Даны текст t и строка s. Ещё иногда говорят «*строка (string) t и образец (pattern) s*».
Вхождением s в t назовём позицию i: `s` = `t[i:i+|s|)`.
Возможны различные формулировки задачи поиска подстроки в строке:

(a) Проверить, есть ли хотя бы одно вхождение s в t.
(b) Найти количество вхождений s в t.
(c) Найти позицию любого вхождения s в t, или вернуть -1, если таких нет.
(d) Вернуть множества всех вхождений s в t.

6.2.1. C++

В языке C++ у строк типа `string` есть стандартный метод `find`. Работает за $\mathcal{O}(|s| \cdot |t|)$, возвращает целое число — номер позиции в исходной строке, начиная с которого начинается первое вхождение подстроки или `string::npos`.

Функция из `<cstring>` `strstr(t, s)` ищет s в t. Работает за линию в `Unix`, за квадрат в `Windows`.

В обоих случаях квадрат имеет очень маленькую константу (AVX-регистры).

Все вхождения можно перечислить таким циклом:

```
for (size_t pos = t.find(s); pos != string::npos; pos = st.find(s, pos + 1))
    ; // pos -- позиция вхождения
```

или таким

```
for (char *p = t; (p = strstr(p, s)) != 0; p++)
    ; // p -- указатель на позицию вхождения в t
```

6.2.2. Префикс функция и алгоритм КМП

Def 6.2.1. $\pi_0(s)$ — *длина* max *собственного префикса s, совпадающего с суффиксом s.*

Def 6.2.2. *Префикс-функция строки* `s` — *массив* $\pi(s)\colon \pi(s)[i] = \pi_0\big(s[0{:}i)\big)$.

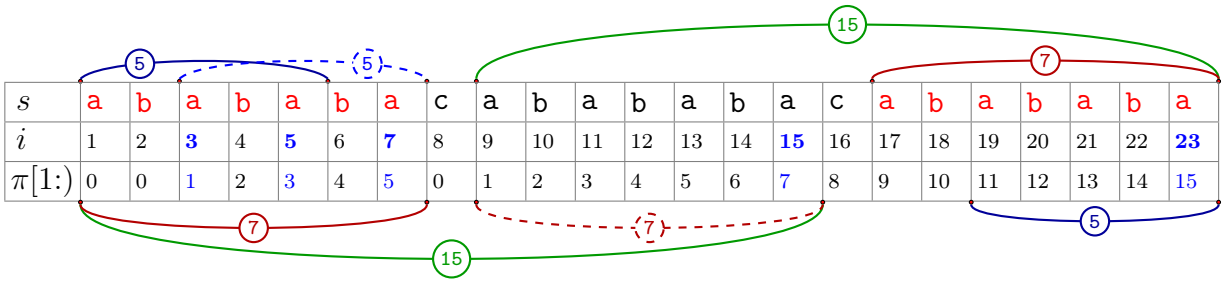
Когда из контекста понятно, о префикс-функции какой строки идёт речь, пишут просто $\pi[i]$.

• Алгоритм Кнута-Мориса-Пратта.

Пусть '#' — любой символ, который не встречается ни в t, ни в s. Создадим новую строку $w = s\#t$ и найдем её префикс-функцию. Благодаря символу '#' $\forall i\ \pi(w)[i] \leqslant |s|$. Такие i, что $\pi(w)[i] = |s|$, задают позиции окончания вхождений s в $w \Rightarrow (j = i - 2|s| - 1)$ — начало вхождения s в t.

• **Вычисление префикс-функции за $\mathcal{O}(n)$.**

Все префиксы равные суффиксам для строки `s[0:i)` это $X = \{i, \pi[i], \pi[\pi[i]], \pi[\pi[\pi[i]]], \dots\}$



s	a	b	a	b	a	b	a	c	a	b	a	b	a	b	a	c	a	b	a	b	a	b	a
i	1	2	**3**	4	**5**	6	**7**	8	9	10	11	12	13	14	**15**	16	17	18	19	20	21	22	**23**
$\pi[1:)$	0	0	1	2	3	4	5	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Пример (см. картинку выше): для всей строки s имеем $X = \{23, 15, 7, 5, 3, 1\}$.

Заметим, что или $\pi[i{+}1] = 0$, или $\pi[i{+}1]$ получается, как $x + 1$ для некоторого $x \in X$.
Будем перебирать $x \in X$ в порядке убывания, получается следующий код:

```
p[1] = 0, n = |s|
for (i = 2; i <= n; i++)
    int k = p[i - 1];
    while (k > 0 && s[k] != s[i - 1])
        k = p[k];
    if (s[k] == s[i - 1])
        k++;
    p[i] = k;
```

Заметим, что с учётом последней строки цикла первая не нужна, получаем:

```
p[1] = k = 0, n = |s|
for (i = 2; i <= n; i++)
    while (k > 0 && s[k] != s[i - 1])
        k = p[k];
    if (s[k] == s[i - 1])
        k++;
    p[i] = k;
```

k увеличится $\leqslant n$ раз $\Rightarrow$ суммарное число шагов цикла `while` не более $n \Rightarrow \mathcal{O}(n)$.

Собственно КМП работает за $\mathcal{O}(|s| + |t|)$ и требует $\mathcal{O}(|s| + |t|)$ дополнительной памяти.
Упражнение: придумайте, как уменьшить количество доппамяти до $\mathcal{O}(|s|)$.

6.2.3. LCP

Def 6.2.3. *`lcp[i,j]` (largest common prefix) для строки s — длина наибольшего общего префикса суффиксов `s[i:]` и `s[j:]`.*

Вычислить массив `lcp` можно за $\mathcal{O}(n^2)$, так как $\texttt{lcp[i,j]} = \begin{cases} 1 + \texttt{lcp[i+1,j+1]} & s[i] = s[j] \\ 0 & \text{иначе} \end{cases}$

Аналогично можно определить и вычислить массив `lcp` для двух разных строк.

6.2.4. Z-функция

Def 6.2.4. *Z-функция — массив z такой, что $z[0] = 0, \forall i{>}0\ z[i] =$ `lcp[0,i]`.*

Для поиска подстроки снова введем $w = s\#t$ и посчитаем $Z(w)$.
Найдем все позиции i: $Z(w)[i] = |s|$. Это позиции всех вхождений строки s в строку t.
Осталось научиться вычислять Z-функцию за линейное время.

```
z[0] = 0;
for (i = 1; i < n; i++)
    int k = 0;
    while (s[i+k] == s[k]) // s[n] == '\x0' ⇒ нет выхода за пределы!
        k++;
    z[i] = k;
```

Приведенный алгоритм работает за $\mathcal{O}(n^2)$. На строке $aaa\ldots a$ оценка n^2 достигается.
Ключом к ускорению является следующая лемма:

Lm 6.2.5. Пусть мы уже знаем $z[l] \Rightarrow$ отрезки `s[0:z[l])` и `s[l:l+z[l])` равны $\Rightarrow$
$\forall i \in [l, l{+}z[l])$ отрезки `s[i-l:z[l])` и `s[i:r)` равны $\Rightarrow z[i] \geqslant \min(r - i, z[i - l])$.

Будем хранить l: $r = l + z[l] = \max$, улучшим код, чтобы получить время $\mathcal{O}(n)$.

```
z[0] = 0, l = r = 0;
for (i = 1; i < n; i++)
    int k = max(0, min(r - i, z[i - l]))
    while (s[i + k] == s[k])
        k++
    z[i] = k
    if (i + z[i] > r) l = i, r = i + z[i]
```

Теорема 6.2.6. Приведенный выше алгоритм работает за $\mathcal{O}(n)$.

Доказательство. $k{+}{+} \Rightarrow r{+}{+}$, а r может увеличиваться $\leqslant n$ раз. ■

6.3. Полиномиальные хеши строк

Основная идея этой секции — научиться с предподсчётом за $\mathcal{O}$(суммарной длины строк) вероятностно сравнивать на равенство любые их подстроки за $\mathcal{O}(1)$.
Например, мы уже умеем считать частичные суммы за $\mathcal{O}(1) \Rightarrow$ можем за $\mathcal{O}(1)$ проверить, равны ли суммы символов в подстроках. Если не равны $\Rightarrow$ строки точно не равны...

Def 6.3.1. *Хеш-функция объектов из мн-ва A в диапазон $[0, m)$ — любая функция $A \to \mathbb{Z}/m\mathbb{Z}$.*

Например, сумма символов строки, посчитанная по модулю 256 — пример хеш-фукнции из множества строк в диапазон $[0, 256)$. Задача — придумать более удачную хеш-функцию.

Зачем нужны хеши (хеш-функции)? Чтобы сравнивать объекты на равенство.

Хеши не совпали $\Rightarrow$ объекты точно не совпали.
Хеши совпали $\Rightarrow$ с некоторой вероятностью объекты всё равно различаются (коллизия хешей) и у нас есть выбор —- или остатоновиться и получить RP алгоритм, или после равенства хешей сравнить сами объекты и получить ZPP алгоритм.

• **Полиномиальная хеш-функция**

Def 6.3.2. *Пусть* $s = s_0 s_1 \dots s_{n-1} \Rightarrow h_{p,m}(s) = (s_0 p^{n-1} + s_1 p^{n-2} + \dots + s_{n-1}) \bmod m$

$h_{p,m}(s)$ — полиномиальный хеш для строки s посчитанный в точке p по модулю m.

По сути мы взяли многочлен (полином) с коэффициентами «символы строки» и посчитали его значение в точке p по модулю m. Можно было бы определить $h_{p,m}(s) = \sum_i s_i p^i$, но при реализации нам будет удобен порядок суммирования из Def 6.3.2.

```
int *h;
void initialize(int n, char* s) {
    h = new int[n + 1]; // h[i] -- хеш префикса s[0:i)
    h[0] = 0; // хеш пустой строки действительно 0...
    for (int i = 0; i < n; i++)
        h[i + 1] = ((int64_t)h[i] * p + s[i]) % m; // 0 < m < 2^31
}
int getHash(int l, int r, char* s) { // [l,r)
    // deg[r - l] = p^(r-l), никогда не пишите здесь лишний if
    T res = (h[r] - (int64_t)h[l] * deg[r - l]) % m;
    return res < 0 ? res + m : res; // остаток мог быть отрицательным
}
```

Чтобы вероятность коллизии была мала нужно:
m — простое большое число.
p — заранее фиксированное случайное число.

Стандартные ошибки:

(a) Символы строки должны быть >0, иначе $\texttt{hash}(00) = \texttt{hash}(0)$.
(b) Переполнения! Например, при подсчёте p^k.
(c) Остаток может быть отрицательными: $(h_r - h_l p^{r-l}) \bmod m$
(d) ± 1 во всех формулах. В коде выше полуинтервал $[l, r)$, индексация с нуля.

Если вы хотите делать вычисления по более большому модулю $\approx 10^{18}$, у вас два пути — или считать два хеша по двум модулям $10^9 + 7$, $10^9 + 9$, или использовать тип `__int128`.

Ещё есть вариант — считать по модулю 2^{32} или 2^{64},
тогда можно везде писать тип `uint_32/uint_64`, и не делать лишних взять по модулю.
Константа станет меньше, а вот работать будет не всегда (ниже подробно разберёмся).

Def 6.3.3. *Коллизия хешей — ситуация вида* $s \neq t,\ h_{p,m}(s) = h_{p,m}(t)$.

Займёмся точными оценками чуть позже, пока предположим, что $\forall$ простого m, если мы выбираем p равновероятно в $[0, m)$, вероятность коллизии при одном сравнении равна $\frac{1}{m}$.
Из умения сравнивать строки на равенство за $\mathcal{O}(1)$ следует алгоритм поиска строки в тексте:

6.3.1. Алгоритм Рабина-Карпа

Можно искать s в t, предподсчитав полиномиальные хеши для t и для каждого потенциального вхождения $[i, i+|s|)$ сравнить за $\mathcal{O}(1)$ хеш подстроки $t[i, i+|s|)$ с хешом s.
Если хеши совпали, то возможно два развития событий: мы можем
или проверить за линию равенство строк, или, не проверяя, выдать вхождение i.

Для задачи поиска одного вхождения в первом случае мы получили `RP`, во втором `ZPP`.
На практике используют оба подхода.
Для задачи поиска всех вхождений проверять каждое вхождение за линию — слишком долго.

• Оптимизируем память.

Преимущество Рабина-Карпа над π-функцией и Z-функцией — реализация с $\mathcal{O}(1)$ доппамяти.
Обозначим $n = |s|$ и $ht_i = h_{p,m}(t[i : i + n))$. Посчитаем $h_{p,m}(s)$, ht_0 и p^n.
Осталось, зная ht_i, научиться считать $ht_{i+1} = ht_i \cdot p - t_i \cdot p^n + t_{i+n}$.

6.3.2. Наибольшая общая подстрока за $\mathcal{O}(n \log n)$

Задача: даны строки s и t, найти w: w — подстрока s, подстрока t, $|w| \to \max$.
Заметим, что если w — общая подстрока s и t, то любой её префикс тоже $\Rightarrow$
функция $f(k) =$ «есть ли у s и t общая подстрока длины k» — монотонный предикат $\Rightarrow$
максимальное k: $f(k) = 1$ можно найти бинпоиском ($\mathcal{O}(\log \min(|s|, |t|)$ итераций).

Осталось для фиксированного k за $\mathcal{O}(|s| + |t|)$ проверить, есть ли у s и t общая подстрока длины k. Сложим хеши всех подстрок s длины k в хеш-таблицу. Для каждой подстроки t длины k поищем её хеш в хеш-таблице. Если нашли совпадение, как и в Рабине-Карпе есть два пути — проверить за линию или сразу поверить в совпадение. Оба метода работают.

6.3.3. Оценки вероятностей

• Многочлены

Пусть m — простое число.

Lm 6.3.4. Число корней многочлена степени n над $\mathbb{Z}/m\mathbb{Z}$ не более n.

Lm 6.3.5. Пусть $t \in \mathbb{Z}/m\mathbb{Z} \Rightarrow Pr[P(t) = 0] = \frac{1}{m}$, где P — случайный многочлен.

Lm 6.3.6. Матожидание числа корней случайного многочлена степени n над $\mathbb{Z}/m\mathbb{Z}$ равно 1.

Первую лемму вы знаете из курса алгебры,
третья сразу следует из второй (m потенциальных корней, каждый с вероятностью $1/m$),
вторая получается из того, что $P(x) = Q(x)(x-t)+r$: у случайного $P(x)$ все r равновероятны.

Нам важна простота модуля m, для непростого оценки неверны.
Например, у многочлена x^{64} при $m = 2^{64}$ любое чётное число является корнем.

• Связь со строками

$h_{p,m}(S) = S(p) \bmod m$, где S — и строка, и многочлен с коэффициентами S_i.
Тогда $h_{p,m}(S) = h_{p,m}(T) \Leftrightarrow (S - T)(p) \equiv 0 \bmod m$.
Запишем несколько следствий из лемм про многочлены.

Следствие 6.3.7. $\forall$ пары $\langle p, m\rangle$ вероятность совпадения хешей случайных строк s и t — $\frac{1}{m}$.

Доказательство. Разность многочленов s и t — случайный многочлен $(s-t)$. Далее Lm 6.3.5. ■

Следствие 6.3.8. $\forall m, \forall s, t\ Pr_p[h_{p,m}(s) = h_{p,m}(t)] \leqslant \frac{\max(|s|,|t|)}{m}$.
По-русски: даны фиксированные строки, выбираем случайное p, оцениваем Pr коллизии.

Доказательство. Подставим многочлен $(s-t)$ в лемму Lm 6.3.4. ■

Теперь пусть сравнений строк было много.

Теорема 6.3.9. Пусть дано множество **случайных** различных строк, и сделано k сравнений $\langle p, m\rangle$ хешей каких-то из этих строк $\Rightarrow$ вероятность существования коллизии не более $\frac{k}{m}$.

Доказательство. $Pr[\text{коллизии}] \leqslant E[\text{коллизий}] = k \cdot Pr[\text{коллизии при 1 сравнении}] = k/m$. ■

Теорема 6.3.10. Пусть дано множество **произвольных** различных строк длины $\leqslant n$, выбрано случайное p и сделано k сравнений $\langle p, m\rangle$ хешей каких-то из этих строк $\Rightarrow$ вероятность существования коллизии $\leqslant \frac{nk}{m}$.

Доказательство. Суммарное число корней у k многочленов степени $\leqslant n$ не более nk. ■

Замечание 6.3.11. На самом деле оценка $\frac{nk}{m}$ из Thm 6.3.10 не достигается. В практических рассчётах можно смело пользоваться оценкой $\frac{k}{m}$ из Thm 6.3.9.

6.3.4. Число различных подстрок (на практике)

Рассмотрим два решения, оценим их вероятности ошибок.
Решение #1: сложить хеши всех $\frac{1}{2}n(n-1)$ подстрок в хеш-таблицу.
Решение #2: отдельно решаем для каждой длины, внутри сложим хеши всех $\leqslant n$ подстрок в хеш-таблицу, просуммируем размеры хеш-таблиц.

В первом случае, у нас неявно происходит $\approx n^4/8$ сравнений подстрок $\Rightarrow$ вероятность наличия коллизии $\approx \frac{n^4/8}{m} \Rightarrow$ при $n = 1000$ нам точно не хватит 32-битного модуля,
при $n = 10\,000$ для $m \approx 10^{18}$ вероятность коллизии $\approx \frac{1}{800}$.

Во втором случае $\approx \sum_{i=1..n} i^2/2 \approx n^3/6$ сравнений подстрок $\Rightarrow$ вероятность наличия коллизии $\approx \frac{n^3/6}{m} \Rightarrow$ при $n = 10\,000$ для $m \approx 10^{18}$ вероятность коллизии $\approx \frac{1}{6\cdot 10^6}$.

6.3.5. Строка Туэ-Морса

[Пост на codeforces в тему]

Сейчас мы построим коллизию (пару строк) для полиномиального хеша $\langle p, m = 2^k \rangle$. $\forall p, k$.

Def 6.3.12. *Определим строку Туэ — Морса. Сначала введём строки S_i:*

- $S_0 = 0$
- $S_1 = 01$
- $S_2 = 0110$
- $S_n = S_{n-1}\overline{S_{n-1}}$ *($\overline{x}$ — отрицание x)*

Каждая строка является префиксом всех последующих.
Обозначим $t = S_\infty = \lim_{i\to\infty} S_i$. $S_\infty \in \{0,1\}^{\mathbb{N}}$. S_∞ и есть строка Туэ — Морса.

• Построение (два способа)

1. Строим рекурсивно по определению.
2. `s[i] = std::popcount<uint64_t>(i) mod 2` (кол-во единичных битов в двоичной записи i)

p.s. Раньше вместо `popcount` была `__builtin_popcount`, но это было давно, уже у всех `C++20`.

Второй способ доказывается по индукции $S_k \to S_{k+1}$.

• Коллизия

Обозначим $H(s) = \texttt{hash}(s)$. Для достаточно большого i: $H(S_i) = H(\overline{S_i}) \Leftrightarrow H(S_i) - H(\overline{S_i}) = 0$.
Докажем по индукции, что $H(S_i) - H(\overline{S_i}) = (p-1)(p^2-1)\dots(p^{2^{i-1}}-1)$:

$S_n = S_{n-1}\overline{S_{n-1}}$, $\overline{S_n} = \overline{S_{n-1}}S_{n-1}$ $|S_n| = 2^n$. Пусть $f(n) = H(S_n) - H(\overline{S_n}) \Rightarrow f(n) = (H(S_{n-1}) - H(\overline{S_{n-1}})) \cdot p^{2^{n-1}} + (H(\overline{S_{n-1}}) - H(S_{n-1})) = f(n-1)(p^{2^{n-1}}-1) = (p-1)(p^2-1)\dots(p^{2^{n-1}}-1)$.

Вспомним, что $m = 2^k$. Когда $f(n)$ занулится?
Если p чётно, то при $i \geqslant k$ имеем $p^i = 0 \Rightarrow$ для всех строк длины $>k$ тривиально строится.
Если p нечётно, то заметим, что $p^{2^i}-1 = (p+1)(p-1)(p^2-1)(p^4-1)\dots(p^{2^{i-1}}-1) \Rightarrow$ содержит хотя бы $i+1$ чётных сомножителей $\Rightarrow f(n) \vdots 2^{1+2+3+\dots+n}$. Достаточно, чтобы $f(n) \vdots 2^k$.
$1+2+3+\dots+n \geqslant k \Leftrightarrow \frac{1}{2}n(n+1) \geqslant k \Leftarrow n \geqslant \sqrt{2k}$.

Итого получили, что при $n \geqslant \sqrt{2k}$ всегда $H(S_n) = H(\overline{S_n}) \Rightarrow$ есть коллизия для строк длины $2^{\sqrt{2k}}$.
Пример $m = 2^{64}, k = 64$ длина $2^{\sqrt{128}} \leqslant 2^8 = 256$.

6.4. Алгоритм Бойера-Мура

Даны текст t и шаблон s. Требуется найти хотя бы одно вхождение s в t или сказать, что их нет. БМ — алгоритм, решающий эту задачу за время $\mathcal{O}(\frac{|t|}{|s|})$ в среднем и $\mathcal{O}(|t| \cdot |s|)$ в худшем.

• Наивная версия алгоритма

```
for (p = 0; p <= |t| - |s|; p++)
    for (k = |s| - 1; k >= 0; k--)
        if (t[p + k] != s[k])
            break;
    if (k < 0)
        return 1;
```

То есть, мы прикладываем шаблон s ко всем позициям t, сравниваем символы с конца.

• Оптимизации

Каждый раз мы сдвигаем шаблон на 1 вправо.
Сдвинем лучше сразу так, чтобы несовпавший символ текста `t[p+k]` совпал с каким-либо символом шаблона. Эта оптимизация называется «правилом плохого символа».

```
for (i = 0; i < |s|; i++)
    pos[s[i]].push_back(i); // для каждого символа список позиций
for (p = 0; p <= |t| - |s|; p += dp)
    for (k = |s| - 1; k >= 0; k--)
        if (t[p + k] != s[k])
            break;
    if (k < 0)
        return 1
    auto &v = pos[t[p + k]]; // нужно в v найти последний элемент меньший k
    for (i = v.size() - 1; i >= 0 && v[i] >= k; i--)
        ;
    dp = (k - (i < 0 ? -1 : v[i])); // сдвигаем так, чтобы вместо s[k] оказался s[v[i]]
```

Вторая оптимизация «правило хорошего суффикса» — использовать информацию, что суффикс u = `s[k+1:]` уже совпал с текстом $\Rightarrow$ нужно сдвинуть s до следующего вхождения u. Тут нам поможет Z-функция от $\overline{s}$: $|u| = |s|-k-1$, мы ищем `shift[|u|]` $= \min j \colon z(\overline{s})[j] \geqslant |u|$, и сдвигаем на j. На самом деле мы даже знаем, что следующий символ не совпадает, поэтому ищем $\min j \colon z(\overline{s})[j] = |u|$.

```
z <-- z_function(reverse(s))
for (j = |s| - 1; j >= 0; j--)
    shift[z[j]] = r;
```

В итоге алгоритм Бойера-Мура сдвигает шаблон на $\max(x, y)$, где $x =$ `dp`, $y =$ `shift[|s|-k-1]`).

Время и память, требуемые на предподсчёт — $\Theta(|s|)$. Предподсчёт зависит только от s.

Пример выполнения Бойера-Мура:

t = «abcabcabbababa», s = «baba»

a	b	c	a	b	c	a	b	b	a	a	a	b	a	b	a	
b	a	b	a													x = 3, y = 2
			b	a	b	a										x = 3, y = 2
						b	a	b	a							x = 1, y = 2
								b	a	b	a					x = 1, y = 2
										b	a	b	a			x = 1, y = 2
												b	a	b	a	ok

Алгоритм можно продолжить модифицировать, например искать $\min j$:
после сдвига на j, совпададут с шаблоном все уже открытые символы в t.
Тогда в первом же шаге примера сдвиг будет на 4 вместо трёх.

Другой пример: $s = \underbrace{aa\ldots a}_{n}$ в строке $t = \underbrace{bb\ldots b}_{m}$. Тогда каждый раз мы сравниваем лишь один символ, а сдвигаемся на n позиций $\Rightarrow$ время $\Theta(|m|/|n|)$.

Лекция #7: Суффиксный массив

21 октября 2024

Def 7.0.1. *Суффиксный массив s – отсортированный массив суффиксов s.*

Суффиксы сортируем в лексикографическом порядке. Каждый суффикс однозначно задается позицией начала в $s \Rightarrow$ на выходе мы хотим получить перестановку чисел от 0 до $n-1$.

• **Тривиальное решение:** `std::sort` отработает за $\mathcal{O}(n \log n)$ операций '<' $\Rightarrow$ за $\mathcal{O}(n^2 \log n)$.

7.1. Построение за $\mathcal{O}(n \log^2 n)$ хешами

Мы уже умеем сравнивать хешами строки на равенство, научимся сравнивать их на "`>/<`".
Бинпоиском за $\mathcal{O}(\log(min(|s|, |t|)))$ проверок на равенство найдём $x = lcp(s, t)$.
Теперь $less(s, t) = (s[x] < t[x])$. Кстати, в `C/C++` после строки всегда идёт символ с кодом 0.

Получили оператор меньше, работающий за $\mathcal{O}(\log n)$ и требующий $\mathcal{O}(n)$ предподсчёта.
Итого: суффмассив за $\mathcal{O}(n + (n \log n) \cdot \log n) = \mathcal{O}(n \log^2 n)$.

При написании сортировки нам нужно теперь минимизировать в первую очередь именно число сравнений $\Rightarrow$ с точки зрения `C++::STL` быстрее будет работать `stable_sort` (MergeSort внутри).

Замечание 7.1.1. Заодно научились за $\mathcal{O}(\log n)$ сравнивать на больше/меньше любые подстроки.

7.2. Применение суффиксного массива: поиск строки в тексте

Задача: дана строка t, приходят строки-запросы s_i: "является ли s_i подстрокой t".

Предподсчёт: построим суффиксный массив p строки t.
В суффиксом массиве сначала лежат все суффиксы $< s_i$, затем $\geqslant s_i \Rightarrow$ бинпоиском можно найти $\min k\colon\ t[p_k{:}] \geqslant s_i$. Осталось заметить, что (s_i – префикс $t[p_k{:}]$) $\Leftrightarrow$ (s_i – подстрока t).

Внутри бинпоиска можно сравнивать строки за линию, получим время $\mathcal{O}(|s_i| \log |t|)$ на запрос.
Можно за $\mathcal{O}(\log |t|)$ с помощью хешей, для этого нужно один раз предподсчитать хеши для t, а при ответе на запрос насчитать хеши s_i. Получили время $\mathcal{O}(|s_i| + \log |t| \cdot \log |s_i|)$ на запрос.

В Section 7.5 мы улучшим время обработки запроса до $\mathcal{O}(|s_i| + \log |t|)$.

7.3. Построение за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$ цифровой сортировкой

Заменим строку `s` на строку `s#`, где `#` – символ, лексикографически меньший всех в `s`.
Будем сортировать циклические сдвиги `s#`, порядок совпадёт с порядком суффиксом.
Длину `s#` обозначим n.

Решение за $\mathcal{O}(n^2)$: цифровая сортировка.
Сперва подсчётом по последнему символу, затем по предпоследнему и т.д.
Всего n фаз сортировок подсчётом. В предположении $|\Sigma| \leqslant n$ получаем время $\mathcal{O}(n^2)$.
Суффмассив, как и раньше задаётся перестановкой начал... теперь циклических сдвигов.

Решение за $\mathcal{O}(n \log n)$: цифровая сортировка с удвоением длины.
Пусть у нас уже отсортированы все подстроки длины k циклической строки `s#`.
Научимся за $\mathcal{O}(n)$ переходить к подстрокам длины $2k$.

Давайте требовать не только отсортированности но и знания "равны ли соседние в отсортированном порядке". Тогда линейным проходом можно для каждого i насчитать тип (цвет) циклического сдвига $c[i]$: $(0 \leqslant c[i] < n) \wedge (\texttt{s[i:i+k)} < \texttt{s[j:j+k)} \Leftrightarrow c[i] \leqslant c[j])$.

Любая подстрока длины $2k$ состоит из двух половин длины k $\Rightarrow$
переход $k \to 2k$ – цифровая сортировка пар $\langle c[i], c[i+k]\rangle$.

Прекратим удвоение k, когда $k \geqslant n$. Порядки подстрок длины k и n совпадут.

Замечание 7.3.1. В обоих решениях в случае $|\Sigma| > n$ нужно первым шагом отсортировать и перенумеровать символы строки. Это можно сделать за $\mathcal{O}(n \log n)$ или за $\mathcal{O}(n + |\Sigma|)$ подсчётом.

Реализация решения за $\mathcal{O}(n \log n)$.
`p[i]` – перестановка, задающая порядок подстрок длины `s[i:i+k)` циклической строки `s#`.
`c[i]` – тип подстроки `s[i:i+k)`.
За базу возьмём $k = 1$

```
bool sless( int i, int j ) { return s[i] < s[j]; }
sort(p, p + n, sless);
cc = 0; // текущий тип подстроки
for (i = 0; i < n; i++) // тот самый линейный проход, насчитываем типы строк длины 1
    cc += (i && s[p[i]] != s[p[i-1]]), c[p[i]] = cc;
```

Переход: (у нас уже отсортированы строки длины k) $\Rightarrow$ (уже отсортированы строки длины $2k$ по второй половине) $\Rightarrow$ (осталось сделать сортировку подсчётом по первой половине).

```
// pos - массив из n нулей
for (i = 0; i < n; i++)
    pos[c[i] + 1]++; // обойдёмся без лишнего массива cnt
for (i = 1; i < n; i++)
    pos[i] += pos[i - 1];
for (i = 0; i < n; i++): // p[i] - позиция начала второй половины
    int j = (p[i] - k) mod n; // j - позиция начала первой половины
    p2[pos[c[j]]++] = j; // поставили подстроку s[j,j+2k) на правильное место в p2
cc = 0; // текущий тип подстроки
for (i = 0; i < n; i++) // линейным проходом насчитываем типы строк длины 2k
    cc += (i && pair_of_c(p2[i]) != pair_of_c(p2[i-1]))), c2[p2[i]] = cc;
c2.swap(c), p2.swap(p); // не забудем перейти к новой паре (p,c)
```

Здесь `pair_of_c(i)` – пара $\langle$`c[i], c[(i + k) mod n]`$\rangle$ (мы сортировали как раз эти пары!).

Замечание 7.3.2. При написании суффмассива в контесте рекомендуется, прочтя конспект, написать код самостоятельно, без подглядывания в конспект.

7.4. LCP за $\mathcal{O}(n)$: алгоритм Касаи

Алгоритм Касаи считает LCP соседних суффиксов в суффиксном массиве. Обозначения:

- $p[i]$ – элемент суффмассива,
- $p^{-1}[i]$ – позиция суффикса `s[i:]` в суффмассиве,
- $next_i = p[p^{-1}[i] + 1]$, $lcp_i = \mathrm{LCP}(i, next_i)$. Наша задача – насчитать массив lcp_i.

Утверждение 7.4.1. Если у i-го и j-го по порядку суффикса в суффмассиве совпадают первые k символов, то на всём отрезке $[i, j]$ суффмассива совпадают первые k символов.

Lm 7.4.2. Основная идея алгоритма Касаи: $lcp_i > 0 \Rightarrow lcp_{i+1} \geqslant lcp_i - 1$.

Доказательство. Отрежем у `s[i:]` и `s[next`$_i$`:]` по первому символу.
Получили суффиксы `s[i+1:]` и какой-нибудь `r`.
(`s[i:]` $\neq$ `s[next`$_i$`:]`) $\wedge$ (первый символ у них совпадал) $\Rightarrow$
(`r` в суффмассиве идёт после `s[i+1:]`) $\wedge$ (у них совпадает первых $lcp_i{-}1$ символов) $\overset{Prop\ 7.4.1}{\Rightarrow}$
у `s[i+1:]` и `s[next`$_{i+1}$`]` совпадает хотя бы $lcp_i{-}1$ символ $\Rightarrow lcp_{i+1} \geqslant lcp_i - 1$. ■

Собственно алгоритм заключается в переборе $i \nearrow$ и подсчёте lcp_i начиная с $\max(0, lcp_{i+1}{-}1)$.

Задача: уметь выдавать за $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ LCP любых двух суффиксов строки s.

Решение: используем Касаи для соседних, а для подсчёта LCP любых других считаем RMQ. RMQ мы решили в прошлом семестре. Например, Фарах-Колтоном-Бендером за $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.

7.5. Быстрый поиск строки в тексте

Представим себе простой бинпоиск за $\mathcal{O}(|s| \log(|text|))$. Будем стараться максимально переиспользовать информацию, полученную из уже сделанных сравнений.

Для краткости $\forall k$ обозначим k-й суффикс (`text[p`$_k$`:]`) как просто k.
Инвариант: бинпоиск в состоянии $[l, r]$ уже знает $lcp(s, l)$ и $lcp(s, r)$.
Сейчас мы хотим найти $lcp(s, m)$ и перейти к $[l, m]$ или $[m, r]$.
Заметим, $lcp(s, m) \geq max\{min\{lcp(s, l), lcp(l, m)\}, min\{lcp(s, r), lcp(r, m)\}\} = x$.
Мы умеем искать $lcp(l, m)$ и $lcp(r, m)$ за $\mathcal{O}(1) \Rightarrow$ `for (`$lcp(s, m) = x$`; можем; `$lcp(s, m)$`++)`.

Кстати, $lcp(l, m)$ и $lcp(r, m)$ не обязательно считать Фарах-Колтоном-Бендером, так как, аргументы lcp – не произвольный отрезок, а вершина дерева отрезков (состояние бинпоиска). Предподсчитаем lcp для всех $\leqslant 2|text|$ вершин и по ходу бинпоиска будем спускаться по Д.О.

Теорема 7.5.1. Суммарное число увеличений на один $lcp(s, ?)$ не более $|x|$

Доказательство. Сейчас бинпоиск в состоянии l_i, m_i, r_i. Следующее состояние: l_{i+1}, r_{i+1}.
Предположим, $lcp(s, l_i) \geqslant lcp(s, r_i)$. Будем следить за величиной $z_i = \max\{lcp(s, l_i), lcp(s, r_i)\}$.
Пусть $lcp(s, m_i) < z_i \Rightarrow lcp(s, m) = x \wedge l_{i+1} = l_i \Rightarrow z_{i+1} = z_i$. Иначе $x = z_i \wedge z_{i+1} = lcp(s, m_i)$. ■

7.6. (*) Построение за $\mathcal{O}(n)$: алгоритм Каркайнена-Сандерса

На вход получаем строку s длины n, при этом $0 \leqslant s_i \leqslant \frac{3}{2}n$.
Выход – суффиксный массив. Сортируем именно суффиксы, а не циклические сдвиги.

Допишем к строке 3 нулевых символа. Теперь сделаем новый алфавит: $w_i = (s_i, s_{i+1}, s_{i+2})$.
Отсортируем w_i цифровой сортировкой за $\mathcal{O}(n)$, перенумеруем их от 0 до $n{-}1$.
Запишем все суффиксы строки s над новым алфавитом:

$t_0 = w_0 w_3 w_6 \dots$
$t_1 = w_1 w_4 w_7 \dots$
$t_2 = w_2 w_5 w_8 \dots$
$\dots$
$t_{n-1} = w_{n-1}$

Про суффиксы t_{3k+i}, где $i \in \{0, 1, 2\}$, будем говорить “суффикс i-типа”.

Запустимся рекурсивно от строки $t_0 t_1$. Длина $t_0 t_1$ не более $2\lceil \frac{n}{3} \rceil$.

Теперь мы умеем сравнивать между собой все суффиксы 0-типа и 1-типа.

Суффикс 2-типа = один символ + суффикс 0-типа $\Rightarrow$
их можно рассматривать как пары и отсортировать за $\mathcal{O}(n)$ цифровой сортировкой.

Осталось сделать merge двух суффиксных массивов.
Операция merge работает за линию, если есть "`operator <`", работающий за $\mathcal{O}(1)$.
Нужно научиться сравнивать суффиксы 2-типа с остальными за $\mathcal{O}(1)$.

$\forall i, j\colon t_{3i+2} = s_{3i+2}t_{3i+3}$, $t_{3j} = s_{3j}t_{3j+1} \Rightarrow$ чтобы сравнить суффиксы 2-типа и 0-типа, достаточно уметь сравнивать суффиксы 0-типа и 1-типа. Умеем.

$\forall i, j\colon t_{3i+2} = s_{3i+2}t_{3i+3}$, $t_{3j+1} = s_{3j+1}t_{3j+2} \Rightarrow$ чтобы сравнить суффиксы 2-типа и 1-типа, достаточно уметь сравнивать суффиксы 0-типа и 2-типа. Только что научились.

• **Псевдокод.**

Пусть у нас уже есть `radixSort(a)`, возвращающий перестановку.

```
1  def getIndex(a): # новая нумерация, O(|a| + max_i a[i])
2      p = radixSort(a)
3      cc = 0
4      ind = [0] * n
5      for i in range(n):
6          cc += (i > 0 and a[p[i]] != a[p[i-1]])
7          ind[p[i]] = cc
8      return ind
9
10 def sufArray(s): # 0 ⩽ s_i ⩽ 3/2 n
11     n = len(s)
12     if n < 3: return slowSlowSort(s)
13     s += [0, 0, 0]
14     w = getIndex( [(s[i], s[i+1], s[i+2]) for i in range(n)] )
15     index01 = range(0, n, 3) + range(1, n, 3) # с шагом 3
16     p01 = sufArray( [w[i] for i in index01] )
17     pos = [0] * n
18     for i in range(len(p01)): pos[index01[p01[i]]] = i # позиция 01-суффикса в p01
19     index2 = range(2, n, 3)
20     p2 = getIndex( [(w[i], pos[i+1]) for i in index2] )
21     def less(i, j): # i mod 3 = 0/1, j mod 3 = 2
22         if i mod 3 == 1: return (s[i],pos[i+1]) < (s[j],pos[j+1])
23         else: return (s[i],s[i+1],pos[i+2]) < (s[j],s[j+1],pos[j+2])
24     return merge(p01 ∘ index01, p2 ∘ index2, less)
25     # ∘ - композиция: index01[p01[i]], ...
```

Для $n \geqslant 3$ рекурсивный вызов делается от строго меньшей строки:
$3 \to 1{+}1,\ 4 \to 2{+}1,\ 5 \to 2{+}2,\ \ldots$.

Неравенством $s_i \leqslant \frac{3}{2}n$ мы в явном виде в коде нигде не пользуемся.
Оно нужно, чтобы гарантировать, что `radixSort` работает за $\mathcal{O}(n)$.

Есть и другие идеи построения суффиксного массива за линию.
Из более быстрых и современных стоит отметить [Nong, Zhang & Chan (2009)].
Реализации более быстрого SA-IS: [google-implementation], [SK-implementation].

Лекция #7: Бор

21 октября 2024

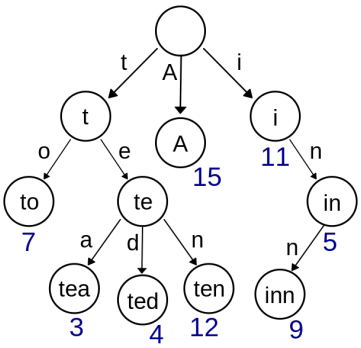
7.7. Собственно бор

Бор – корневое дерево. Рёбра направлены от корня и подписаны буквами. Некоторые вершины бора подписаны, как конечные.

Базовое применение бора – хранение словаря `map<string, T>`.

Пример из wiki бора, содержащего словарь
$\{\text{A}:15,\ \text{to}:7,\ \text{tea}:3,\ \text{ted}:4,\ \text{ten}:12,\ \text{i}:11,\ \text{in}:5,\ \text{inn}:9\}$.

Для строки `s` операции `add(s)`, `delete(s)`, `getValue(s)` работают, как спуск вниз от корня.



Самый простой способ хранить бор: `vector<Vertex> t;`, где `struct Vertex { int id[|Σ|]; }`;
Сейчас рёбра из вершины `t` хранятся в массиве `t.id[]`. Есть другие структуры данных:

Способ хранения	**Время спуска по строке**	**Память на ребро**
`array`	$\mathcal{O}(\|s\|)$	$\mathcal{O}(\|\Sigma\|)$
`list`	$\mathcal{O}(\|s\| \cdot \|\Sigma\|)$	$\mathcal{O}(1)$
`map (TreeMap)`	$\mathcal{O}(\|s\| \cdot \log \|\Sigma\|)$	$\mathcal{O}(1)$
`HashMap`	$\mathcal{O}(\|s\|)$ с большой const	$\mathcal{O}(1)$
`SplayMap`	$\mathcal{O}(\|s\| + \log S$	$\mathcal{O}(1)$

Иногда для краткости мы будем хранить бор массивом `int next[N][|Σ|];`. `next[v][c] == 0` $\Leftrightarrow$ ребра нет.

7.8. Сортировка строк

Если мы храним рёбра в структуре, способной перебирать рёбра в лексикографическом порядке (не хеш-таблица, не список), можно легко отсортировать массив строк:
(1) добавить их все в бор, (2) обойти бор слева направо.
Для `SplayMap` и n и строк суммарной длины S, получаем время $\mathcal{O}(S + n \log S)$.
Для `TreeMap` получаем $\mathcal{O}(S \log |\Sigma|)$.

Замечание 7.8.1. Если бы мы научились сортировать строки над произвольным алфавитом за $\mathcal{O}(|S|)$, то для $\Sigma = \mathbb{Z}$, получилась бы сортировка целых чисел за $\mathcal{O}(|S|)$.

Часто размер алфавита считают $\mathcal{O}(1)$.
Например строчные латинские буквы – 26, или любимый для биологов $|\{A, C, G, T\}| = 4$.

Лекция #8: Ахо-Корасик и Укконен

11 ноября 2024

8.1. Алгоритм Ахо-Корасика

Даны текст t и словарь $s_1, s_2, \ldots, s_m$, нужно научиться искать словарные слова в тексте. Простейший алгоритм, отлично работающий для коротких слов, – сложить словарные слова в бор и от каждой позиции текста i попытаться пройти вперёд, откладывая суффикс t_i вниз по бору, и отмечая все концы слов, которые мы проходим. Время работы – $\mathcal{O}(|t| \cdot \max|s_i|)$.

Ту же асимптотику можно получить, сложив все хеши всех словарных слов в хеш-таблицу, и проверив, есть ли в хеш-таблице какие-нибудь подстроки t длины не более $\max|s_i|$.

Давайте теперь соптимизируем первое решение также, как префикс-функция, позволяет простейший алгоритм поиска подстроки в строке улучшить до линейного времени. Обобщение префикс-функции на бор – суффиксные ссылки:

Def 8.1.1. $\forall$ *вершины бора* v:
$str[v]$ – *строка, написанная на пути от корня бора до* v.
$suf[v]$ – *вершина бора, соответствующая самому длинному суффиксу* $str[v]$ *в боре.*

$\forall$ позиции текста i насчитаем вершину бора v_i: $str[v_i]$ – суффикс $t[0{:}i)$, $|str[v_i]| \to \max$.
Пересчёт v_i**:**

```
v[0] = root, p = root
for (i = 0; i < |t|; i++)
    while (next[p][t[i]] == 0) // нет ребра
        p = suf[p]
    v[i+1] = p = next[p][t[i]]
```

Чтобы цикл `while` всегда останавливался введём фиктивную вершину `f` и сделаем `suf[root] = f`, $\forall c$ `next[f][c] = root`.

Поиск словарных слов. Пометим все вершины бора, посещённые в процессе: `used[`v_i`] = 1`. В конце алгоритма поднимем пометки вверх по суффиксным ссылкам: `used[v]` $\Rightarrow$ `used[suf[v]]`. Для i-го словарного слова при добавлении мы запомнили вершину `end[i]`, тогда наличии этого слова в тексте лежит в `used[end[i]]`. Также можно насчитывать число вхождений.

Суффссылки. Чтобы всё это счастье работало осталось насчитать суффссылки.

Способ #1: полный автомат.

```
suf[v] = go[suf[parent[v]]][parent_char[v]];
go[v][c] = (next[v][c] ? next[v][c] : go[suf[v]][c]);
```

Мы хотим, чтобы от `parent[v]` и `suf[v]` всё было уже посчитано $\Rightarrow$ нужно или перебирать вершины в порядке bfs от корня **(1a)**, или считать эту динамику рекурсивно-лениво **(1b)**.

Способ #2: пишем bfs от корня и пытаемся продолжить какой-нибудь суффикс отца.

```
q <-- root
while q --> v:
    z = suf[parent[v]]
    while next[z][parent_char[v]] == 0:
```

```
5           z = suf[z]
6       suf[v] = next[z][parent_char[v]]
7       for (auto[c,vertex] : next[v]) q <-- c
```

Этот способ экономнее по памяти, если `next` – не массив, а, например, `map<int,int>` **(2)**.

Теорема 8.1.2. Время построения линейно от длины суммарной строк, но не от размера бора.

Доказательство. Линейность от размера бора ломается на примере «бамбук длины n из букв a, из листа которого торчат рёбра по n разным символам». Линейность от суммарной длины строк следует из того, что если рассмотреть путь, соответствующий $\forall$ словарному слову s_i, то при вычислении суффссылок от вершин именно этого пути, указатель z в `while` всё время поднимался, а затем опускался не более чем на 1 $\Rightarrow$ сделал не более $2|s_i|$ шагов. ■

Сравнение способов.

Пусть размер алфавита равен k, число вершин бора V, сумма длин строк в словаре S.
Заметим $V \leqslant S$, но может быть сильно меньше, если у строк длинный общий префикс.

(1a) Ровно $\Theta(k \cdot V + S)$ времени, $\Theta(k \cdot V)$ памяти.
(1b) В худшем случае $k \cdot V$, но на практике за счёт ленивости быстрее.
(2) $\Theta(S)$ времени, $\Theta(V)$ памяти (линия и там, и там). Времени именно S, не V.

8.2. Суффиксное дерево, связь с массивом

Def 8.2.1. *Сжатый бор: разрешим на ребре писать не только букву, но и строку. При этом из каждой вершины по каждой букве выходит всё ещё не более одного ребра.*

Def 8.2.2. *Суффиксное дерево – сжатый бор построенный из суффиксов строки.*

Lm 8.2.3. Сжатое суффиксное дерево содержит не более $2n$ вершин.

Доказательство. Индукция: база один суффикс, 2 вершины, добавляем суффиксы по одному, каждый порождает максимум +1 развилку и +1 лист. ■

• Построение суффдерева из суффмассива+LCP

Пусть мы уже построили дерево из первых i суффиксов в порядке суффмассива. Храним путь от корня до конца i-го. Чтобы добавить $(i+1)$-й, поднимаемся до высоты $LCP(i, i{+}1)$ и делаем новую развилку, новый лист. Это несколько `pop`-ов и не более одного `push`-а. Итого $\mathcal{O}(n)$.

• Построение суффмассива+LCP из суффдерева

Считаем, что дерево построено от строки $s\$ \Rightarrow$ (листья = суффиксы).
Обходим дерево слева направо. Если в вершине используется неупорядоченный `map` для хранения рёбер, сперва отсортируем их. При обходе выписываем номера листьев-суффиксов.
$LCP(i, i+1)$ – максимально высокая вершина, из пройденных по пути из i в $i+1$.
Время работы $\mathcal{O}(n)$ или $\mathcal{O}(n \log |\Sigma|)$.

8.3. Суффиксное дерево, решение задач

• Число различных подстрок.

Это ровно суммарная длина всех рёбер. Так как любая подстрока есть префикс суффикса ⇒ откладывается от корня дерева вниз до «середины» ребра.

• Поиск подстрок в тексте.

Строим суффдерево от текста. ∀ строку s можно за $\mathcal{O}(|s|)$ искать в тексте спуском по дереву.

• Общая подстрока k строк.

Построим дерево от $s_1\#_1 s_2\#_2 \dots s_k\#_k$, найдём самую глубокую вершину, в поддереве которой содержатся суффиксы k различных типов. Время работы $\mathcal{O}(\sum |s_i|)$, оптимально по асимптотике. Константу времени работы можно улучшать за счёт уменьшения памяти – строить суффдерево не от конкатенации, а лишь от одной из строк.

8.4. Алгоритм Укконена

Обозначение: $ST(s)$ – суффиксное дерево строки s.
Алгоритм Укконена – онлайн алгоритм построения суффиксного дерево. Нам поступают по одной буквы c_i, мы хотим за амортизированное $\mathcal{O}(1)$ из $ST(s)$ получать $ST(sc_i)$.

За квадрат это делать просто: храним позиции концов всех суффиксов, каждый из них продлеваем вниз на c_i, если нужно, создаём при этом новые рёбра/вершины.

Ускорение #1: суффиксы, ставшие листьями, растут весьма однообразно – рассмотрим ребро $[l, r)$, за которое подвешен лист, тогда всегда происходит `r++`. Давайте сразу присвоим $[l, \infty)$.

Теперь опишем жизненный цикл любого суффикса:
рождается в корне, ползёт вниз по дереву, разветвляется, становится саморастущим листом.
Нам интересно обработать только момент разветвления.

Lm 8.4.1. ⌝ Суффикс длины k не разветвился ⇒ все более короткие тоже не разветвились.

Доказательство. Суффикс длины k не разветвился ⇒ он встречался в s как подстрока.
Все более короткие являются его суффиксами ⇒ тоже встречаются в s ⇒ не разветвятся. ■

Ускорение #2: давайте хранить только позицию самого длинного неразветвившегося суффикса. Пока он спускается по дереву, ничего не нужно делать. Как только он разветвится, нужно научиться быстро переходить к следующему по длине (отрезать первую букву).

Ускорение #3: отрезать первую букву = перейти по суффссылке, давайте от всех вершин поддерживать суффссылки. Если мы были в вершине, когда не смогли пойти вниз, теперь всё просто, перейдём по её суффссылке. Если же мы стояли посередине ребра и создали новую вершину v, от неё следует посчитать суффссылку. Для этого возьмём суффссылку её отца $p[v]$ и из $suf[p[v]]$ спустимся вниз на строку, соединяющую $p[v]$ и v.

```
1 void build(char *s):
2     int N = strlen(s), VN = 2 * Ns;
3     int vn = 2, v = 1, pos; // идём по ребру из p[v] в v, сейчас стоим в pos
4     int suf[VN], l[VN], r[VN], p[VN]; // «ребро p[v] → v» = s[l[v]:r[v])
5     map<char,int> t[VN]; // собственно рёбра нашего бора
6     for (int i = 0; i < |Σ|; i++) t[0][i] = 1; // 0 = фиктивная, 1 = корень
7     l[1] = -1, r[1] = 0, suf[1] = 0;
```

```
    for (int n = 0; n < N; n++):
        char c = s[n];
        auto new_leaf = [&]( int v ) {
            p[vn] = v, l[vn] = n, r[vn] = ∞, t[v][c] = vn++;
        };
        go:;
        if (r[v] <= pos) { // дошли до вершины, конца ребра
            if (!t[v].count(c)) { // по символу c нет ребра вперёд, создаём
                new_leaf(v), v = suf[v], pos = r[v];
                goto go;
            }
            v = t[v][c], pos = l[v] + 1; // начинаем идти по новому ребру
        } else if (c == s[pos]) {
            pos++; // спускаемся по ребру
        } else {
            int x = vn++; // создаём развилку
            l[x] = l[v], r[x] = pos, l[v] = pos;
            p[x] = p[v], p[v] = x;
            t[p[x]][s[l[x]]] = x, t[x][s[pos]] = v;
            new_leaf(x);
            v = suf[p[x]], pos = l[x]; // вычисляем позицию следующего суффикса
            while (pos < r[x])
                v = t[v][s[pos]], pos += r[v] - l[v];
            suf[x] = (pos == r[x] ? v : vn);
            pos = r[v] - (pos - r[x]);
            goto go;
        }
```

Теорема 8.4.2. Суммарное время работы n первых шагов равно $\mathcal{O}(n)$.

Доказательство. Понаблюдаем за величиной z «число вершин на пути от корня до нас». Пока мы идём вниз, z растёт, когда переходим по суффссылке, z уменьшается максимум на 1 $\Rightarrow$ возьмём потенциал $\varphi = -z$, суммарное число шагов вниз не больше n. ■

8.5. LZSS

Решим ещё одну задачу – сжатие текста алгоритмом LZSS.
В отличии от использования массива, дерево даёт чисто линейную асимптотику и простейшую реализацию – насчитаем для каждой вершины $l[v]$ = самый левый суффикс в поддереве и при попытки найти $j < i$: $LCP(j, i) = \max$ будем спускаться из корня, пока $l[v] < i$.

Лекция #9: Хеширование

18 ноября 2024

9.1. Универсальное семейство хеш функций

[wiki] [итмо-конспект] [Carter,Wegman'1977]

Def 9.1.1. *Хеш-функция. Сжимающее отображение* $h : U \to M$, $|U| > |M|$, $|M| = m$.

Def 9.1.2. *Универсальная система хеш-функций (1-я версия определения).*
Множество хеш-функций $\mathcal{H}$ *– универсальная система, если*

$$\forall x, y \; x \neq y \colon \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \tfrac{1}{m}$$

Def 9.1.3. *Универсальная система хеш-функций (2-я версия определения ≡ 1-й).*

$$\forall x, y \; x \neq y \colon \sum_{h \in \mathcal{H}} [h(x) = h(y)] \leq \tfrac{|\mathcal{H}|}{m}$$

Теорема 9.1.4. $\mathcal{H}_{p,m} = \{(a,b) : x \to ((ax+b) \mod p) \mod m\}$, $a \in [1,p), b \in [0,p)$.

Утверждение: $\mathcal{H}_{p,m}$ универсально для $|U| = [0,p)$ и $M = [0,m)$ $(m \leqslant p)$.

Доказательство. Зафиксируем пару x, y. Проверим для неё определение универсальности.

$f(x) = ax + b \bmod p$, $f(y) = ay + b \bmod p$

$\big[f(x) = f(y) \Leftrightarrow a(x-y) \equiv 0 \bmod p\big] \Rightarrow (a \neq 0, x \neq y)$ по модулю p коллизий нет и
у нас есть биекция $\langle a, b \colon a \neq 0\rangle \leftrightarrow \langle f(x), f(y) \colon f(x) \neq f(y)\rangle$.

Осталось понять, для скольки хеш-функций $f(x) \equiv f(y) \bmod m$?

$$\sum_{(a,b),a\neq 0} [f(x) \equiv f(y) \bmod m] \overset{\text{биекция}}{=} \sum_{f(x)\neq f(y)} [f(x) \equiv f(y) \bmod m] \overset{(*)}{\leqslant} \tfrac{p(p-1)}{m} = \tfrac{|\mathcal{H}|}{m}$$

(*) Зафиксируем $f(x)$, будем перебирать $f(y) = f(x)+1, f(x)+2, \ldots$ В каждом блоке из m вариантов, только последний даёт $f(x) \equiv f(y) \bmod m \Rightarrow$ таких $f(y)$ ровно $\lfloor \frac{p-1}{m} \rfloor \leqslant \frac{p-1}{m}$. ■

9.2. Оценки для хеш-таблицы с закрытой адресацией

[wiki] [wiki-probability] [2-choice-hashing]

Пусть у нас есть универсальное семейство хеш-функций $\mathcal{H}$.
Хеш-таблица с закрытой адресацией (на списках) – вероятностный алгоритм.
Вся вероятностная часть заключается в том, что мы выбираем случайную $h \in \mathcal{H}$.

Операции `Find(x)`, `Del(x)` работают за длину списка.
Оценим для них матожидание времени работы.

$E[time(find(z))] = E[\text{длины списка}] = \sum_{x \in table} Pr[h(x) = h(z)] = 1 + (n-1)\cdot\frac{1}{m} = \mathcal{O}(1 + \frac{n}{m})$, где
n – количество элементов в таблице, а m – размер таблицы (и диапазон хеш-функции).

Замечание 9.2.1. Мы оценили именно матожидание времени работы. Матожидание средней длины списка всегда $\frac{m}{n}$, даже если все элементы всегда класть в один список.

Теперь несколько утверждений без доказательства.

Утверждение 9.2.2. E[максимальной длины списка] $= \mathcal{O}(\log n)$

Утверждение 9.2.3. 2-choice hashing. Модифицируем хеш-таблицу: будем использовать две хеш-функции h_1, h_2 и при добавлении элемента будем выбирать из списков $a[h_1(x)], a[h_2(x)]$ список меньшей длины. Тогда E[максимальной длины списка] $= \Theta(\log\log n)$.

9.3. Оценки других функций для хеш-таблиц

Популярна функция $x \to x \bmod n$, где n размер таблицы (число списков, длина массива открытой адресации). Поскольку все x-ы могут иметь одинаковый остаток по модулю n, $\forall n \exists$ контртест. Новая идея, давайте брать случайное $n \in ¶ \cap [N, 2N)$, и хеш функцию из семейства $\mathcal{H} = size \in ¶ \cap [n, 2n), x \to x \bmod size$.

Lm 9.3.1. $\forall x \neq y\ Pr_{h\in\mathcal{H}}[h(x) = h(y)] \leqslant \frac{\log_n m}{n/\ln n}$.

Доказательство. Коллизия $(x - y) \equiv 0 \Rightarrow$ оцениваем вероятность попасть в простой делитель $(x - y)$. У числа до m не более $\log_n m$ простых делителей $\geqslant n$, простых на $[n, 2n) \approx \frac{n}{\ln n}$. ■

Для хеш-таблицы на списках нам этого хватает. У хеш-таблицы с открытой адресацией есть ещё проблема с тестом: $[1, 2, 3, \ldots, \frac{n}{4}] + \frac{n}{4}$ случайных ключей, на нём работает за квадрат.

9.4. Фильтр Блюма

[wiki] [итмо-конспект]

Прелюдия.
Мы хотим вероятностную структуру данных, которая умеет делать всего две операции – `add`, `find`. Преимущество нашей структуры над хеш-таблицей — очень мало памяти, $\mathcal{O}(n)$ бит.

Собственно структура.
Хотим хранить не более n x-ов. Для этого у нас есть m бит и k хеш-функций $h_1, \ldots, h_k$.

```
1 bitset<m> a; // m нулей
2 Add(x): a[h_1(x)] = a[h_2(x)] = ... = a[h_k(x)] = 1;
3 Find(x): return a[h_1(x)] & a[h_2(x)] & ... & a[h_k(x)];
```

Собственно алгоритм окончен, осталось разобраться, с какой вероятностью это работает.
Во-первых, если `find` вернул 0, элемент точно ещё не был добавлен.
Оценим ошибку `find`-а, который вернул 1. При оценке для простоты предположим, что все nk значений, которые вернули k хеш-функций данных n элементов, равномерно распределены.

Pr[в i-й ячейке 0] $= (1-1/m)^{kn} \approx exp(\frac{-kn}{m}) \Rightarrow Pr$[ложного срабатывания `Find`] $= \left(1 - exp\left(\frac{-kn}{m}\right)\right)^k$

При фиксированных $\langle n, m\rangle$ какое оптимально выбрать k? Будем искать $k = \alpha\frac{m}{n}$: $(1 - e^{-\alpha})^{\alpha\frac{m}{n}} \to \min \Leftrightarrow (1 - e^{-\alpha})^\alpha \to \min$, обозначим $e^{-\alpha} = t \Rightarrow (1 - t)^{-\ln t} \to \min \Leftrightarrow e^{-\ln(1-t)\ln(t)} \to \min \Leftrightarrow \ln(1 - t)\ln(t) \to \max \Rightarrow t = \frac{1}{2} \Rightarrow -\alpha = \ln\frac{1}{2} \Leftrightarrow \alpha = \ln 2 \Rightarrow k = (\ln 2) \cdot \frac{m}{n}$ и $Pr[error] = 2^{-k} = 0.6185^{m/n}$.

Например, если нам дали лишь $5n$ бит на всё про всё, мы достигли вероятности ошибки ≈9%.

• Применение и улучшения

[bio'2013]. В биоинформатике, сборка генома в небольшой памяти.
[xor-filter]. Если мы знаем ключи заранее, можно сделать оптимальнее по памяти.
[stanform-slides]. Фильтры: bloom, cuckoo, xor, cupled-xor.

Если у нас есть в оперативной (быстрой) памяти только фильтр Блюма от большого множества. Фильтр Блюма влезает, большая структура не влезает: всегда сперва спросили у фильтра Блюма «а есть ли вообще во мне такое» и, только если есть, спросили сложную структуру.

Любое место, где поможет кеширование информации «есть ли во мне такое» в быстрой памяти.

Говорят, можно применять в block-list (по ip, по id канала, по нику) и т.д.

9.5. Совершенное хеширование

[wiki] [итмо-конспект] [cs.cmu.edu] [practice:bbhash] [Fredman'1984]

Вспомним, как мы ищем младший бит 64-битного целого x.

```
uint64_t i2(uint64_t x) { return x & (~x + 1); }
```

Таким образом мы получили младший бит i в форме 2^i.
Осталось сделать последнее действие, $2^i \to i$.

```
int table[67];
void init() { forn(i, 64) table[2^i % 67] = i; }
int get(uint64_t i2) { return table[i2 % 67]; }
```

Здесь все остатки по модулю 67 различны $\Rightarrow$
мы получили детерминированный алгоритм поиска номера младшего бита за $\mathcal{O}(1)$.

$x \to x \bmod 67$ – пример совершенной хеш-функции для множества
фиксированных ключей $\{x_0 = 2^0, x_1 = 2^1, \dots, x_{63} = 2^{63}\}$.

Def 9.5.1. *Совершенная хеш-функция – любая инъективная функция.*
То есть, хеш-функция, у которой по определению не возникает коллизий.
Чтобы построить такую хеш-функцию, нам, конечно, нужно заранее знать набор ключей $x_1, x_2 \dots, x_k$ (см. пример с младшим битом).

Задача построения совершенной хеш-функции.
Дан набор ключей $x_1, x_2, \dots, x_n$, найти такую функцию
$h\colon X = \{x_1, x_2, \dots, x_n\} \to [0, m) \cap \mathbb{Z}$, что все $h(x_i)$ различны и функция вычислима за $\mathcal{O}(1)$.
m будем называть размером хеш-функции (m в итоге станет размером массива хеш-таблицы).

9.5.1. Одноуровневая схема

Возьмём $m = n^2$ и любое универсальное семейство хеш-функций $\mathcal{H}\colon X \to [0, m)$.
Например, $\mathcal{H} = \{h_{a,b}(x) = ((ax + b) \bmod p) \bmod m\}$, $m \leqslant p$, p простое.

E(количества коллизий) $= \frac{1}{2}n(n-1) \cdot \frac{1}{m} < \frac{1}{2} \Rightarrow$ с вероятностью хотя бы $\frac{1}{2}$ коллизий нет $\Rightarrow$

Алгоритм: берём случайную $h \in \mathcal{H}$, пока не повезёт. E(времени работы) $= \mathcal{O}(n)$, размер n^2.

9.5.2. Двухуровневая схема

Возьмём $m = n$ и любое универсальное семейство хеш-функций $\mathcal{H}\colon X \to [0, m)$.
Выберем случайную $h \in \mathcal{H}$ и посмотрим на списки $A_y = \{x \mid h(x) = y\}$.
Для каждого A_y построим одноуровневую совершенную хеш-функцию f_y.
Оценим суммарный диапазон всех внутренних уровней:

$$E(\sum_y |A_y|^2) = \sum_{ij} Pr[h(x_i) = h(x_j)] = n \cdot 1 + n(n-1) \cdot \tfrac{1}{n} = 2n-1$$

Осталось внутренние уровни выписать в один массив.
Для этого берём префиксные суммы: `pref_sum[y+1] = pref_sum[y] + `$|A_y|^2$.

Итого, наша хеш-функция: `int hash(x) { y = h(x); return pref_sum[y] + f[y](x); }`

E(времени работы) $= \mathcal{O}(n)$, E(размера) $= 2n-1$.

9.5.3. (*) Графовый подход

[[RandomGraphs|book]]
Начнём с забавного факта

Утверждение 9.5.2. Рассмотрим случайны неорграф из $3n$ вершин и n рёбер.
Pr[отсутствия циклов] $\geqslant \frac{1}{2}$.

Строим совершенную хеш-функцию от $x_1, \ldots, x_n$.
Возьмём $m = 3n$ и любое универсальное семейство хеш-функций $\mathcal{H}\colon X \to [0, m)$.
Выберем $h_1, h_2 \in H$. Каждому x_i сопоставим ребро $\langle h_1(x_i), h_2(x_i)\rangle$.
Если граф не ацикличен, выберем другие хеш-функции, повторим. Пусть ацикличен.

Тогда запишем значения f в вершинах и определим $hash(x_i) = (f[h_1(x_i)] + f[h_2(x_i)]) \bmod n$.

Как записать значения f, чтобы было верно $hash(x_i) = i$? Граф – лес. В каждом дереве в корне пишем ноль (любое число), остальные числа расставит `dfs` по дереву.

9.6. (*) Хеширование кукушки

[wiki] [Pagh,Rodler'2001]

За сколько работает хеш-таблица на списках?
`Add` за $\mathcal{O}(1)$ в худшем, `Find` и `Del` за $\mathcal{O}(1)$ в среднем.

Сейчас мы придумаем, как сделать наоборот:
`Add` за $\mathcal{O}(1)$ в среднем, `Find` и `Del` за $\mathcal{O}(1)$ в худшем.

Заведём массив a размера $m = 3n$ и две хеш-функции: $h_1, h_2 \in \mathcal{H}_m$.
$\forall$ элемент x всегда живёт в одной из двух ячеек — $h_1(x)$ или $h_2(x)$.
`Find` и `Add`, очевидно, обращаются не более чем к двум ячейкам — $h_1(x)$ и $h_2(x)$.
`Add` сперва ищет свободную среди $h_1(x), h_2(x)$. Если заняты обе, начнём выталкивать элементы:

```
1 AddIfBothAreOccupied(x):
2     i = h1(x)
3     while (a[i] != -1):
4         y = a[h1(x)], a[h1(x)] = x, x = y
5         i = h1[y] + h2[y] - i // вторая из двух ячеек, где может жить y
```

Почему и за сколько работает `Add`?
Представим себе граф с рёбрами между $h_1(x)$ и $h_2(x)$. Как мы уже знаем, он с большой вероятностью ацикличен (значит `Add` хотя бы не циклится). Теперь осталось дождаться, пока голос Белы Боллобаша (автор книги Random Graphs) не нашепчет нам недостающую мудрость — глубина случайного дерева $\mathcal{O}(\log n)$.

Лекция #9: Сжатие данных

18 ноября 2024

9.7. Общая информация

Def 9.7.1. *Архиватор: обратимая* $f\colon \Sigma^* \to \Sigma^*$.

Часто для простоты $\Sigma = \{0,1\}$, так как $\forall$ информацию можно закодировать битами.
Кодирование: преобразование $s \to f(s)$. *Декодирование:* преобразование $f(s) \to s$.

Теорема 9.7.2. Нет идеального архиватора: $\nexists f\colon \forall s\ |f(s)| < |s|$

Доказательство. Кол-во строк дины $n = 2^n$, а меньшей длины $= 1 + 2 + \cdots + 2^{n-1} = 2^n - 1$. ■

Теорема 9.7.3. Есть непортящие архиваторы $\forall f \exists f'\colon |f'(s)| \leqslant \min(|f(s)|, |s|) + 1$.

Доказательство. $f'(s) = |s| < |f(s)|\ ?\ 0s : 1f(s)$ ■

• Как мы уже умеем кодировать?

7-битное сжатие: тексты используют < 128 символов (a-z, A-Z, 0-9, знаки препинания).
Всем n символов положили в массив из $7n$ бит, разбили его на $\lceil \frac{7n}{8} \rceil$ байт.

Хаффман: $c \to code_c$, коды должны быть префиксными для однозначного декодирования.
Повторим, как можно хранить коды для Хаффмана: [конспект 2-го семестра].
Лучшее, что можем: $z \cdot (2 + \log|\Sigma|)$, где z – количество символов с ненулевой частотой.

9.8. Арифметическое кодирование

[wiki], [github:implementation], [MattMahoney|book], [MattMahoney|code].

Пример #1, на котором Хаффман сработает плохо: $k = 3$, $cnt_0 = cnt_1 = cnt_2 = 10^6$.
Коды Хаффмана: $code_0 = 0$, $code_1 = 01$, $code_2 = 10$. А хотелось бы равные длины.

Пример #2, на котором Хаффман сработает плохо: $k = 2$, $cnt_0 = 10^6$, $cnt_1 = 1$.
Коды Хаффмана: $code_0 = 0$, $code_1 = 1$. А хотелось бы сильно неравные длины.

Как можно было бы сделать лучше: закодируем не один, а сразу несколько символов.
Первый случай: есть $3^5 = 243 < 256$ строк длины 5 $\Rightarrow$ можно за 8 бит кодировать $\forall$ 5 символов.
Второй случай: чтобы закодировать k нулей используем один символ 0. А оставшиеся $2^k - 1$ комбинаций кодируем, как символ $1 + k$ бит.

• Идея для общего случая

Вероятность появления i-го символа $p_i = cnt_i / \sum cnt_i$, $\sum p_i = 1$.

Отсортируем $p_i \downarrow$, поделим отрезок $[0,1]$ на $|\Sigma|$ отрезков длины p_i.
Для каждого полученного отрезка рекурсивно поделим в такой же пропорции и так далее.
Любой строке s соответствует вершина дерева, подотрезок $[0,1]$.

Кодирование строки s. Начнём с отрезка $[0,1]$. Будем генерировать биты: 0 – пойти в левую половину отрезка, 1 – пойти в правую половину. Целимся в середину отрезка, соответствующего s, если наш отрезок целиком внутри отрезка-строки, кодирование закончено.

Декодирование строки s. Знаем p_i. Строим то же самое дерево, также по нему спускаемся.

9.8.1. Оценки

[Gibbs inequality], [Kraft-McMillan inequality].

Теорема 9.8.1. Kraft-McMillan: по кодам можно декодировать текст iff $\sum 2^{-|code_i|} \leqslant 1$.

Теорема 9.8.2. Gibbs' inequality: если p_i – вероятность появления символа i, $\sum p_i = 1$, то оптимальная длина кода равна $-\log_2 p_i$, а средний битовый вес одного символа $\sum p_i \log_2 p_i$.

В случае арифметического кодирования длины кодов вещественные. Пусть $s = c_1 c_2 \ldots c_n$. Длина отрезка, соответствующего s: $l = \prod p_{c_i}$. Достаточно выписать k бит: $2^{-k} < \frac{1}{2} l$.

$$2^{-k} < \tfrac{1}{2} l \Leftrightarrow k > 1 + (-\log \prod p_{c_i}) = 1 + \sum (-\log_2 p_{c_i})$$

$\Rightarrow$ общая длина кода – сумма вещественных длин кодов $+\mathcal{O}(1)$. Арифметическое кодирование можно воспринимать как обобщение Хаффмана: $\mathbb{Z}$ длины кодов заменили на $\mathbb{R}$ длины кодов.

9.9. Словарное кодирование

Давайте сопоставлять короткий код длинным словам.

Пример, как это можно использовать для сжатия текста: пробегаемся по исходному текста, все английские слова (максимальные по включению отрезки букв) кидаем в словарь.
Новый алфавит = исходный + слова словаря.

Кодирование. Записываем все слова из словаря, как `char*` (конец строки – символ с кодом 0), символами старого алфавита. Записываем текст символами нового алфавита. Символ нового алфавита весит больше бит. Оптимизация: новый алфавит скормить Хаффману.

9.9.1. LZW'84

[wiki], [итмо-конспект].

Как эффективно строить словари? `LZW` – пример алгоритма, как можно без оверхеда на хранение словаря получить кодирование со словарём.

Кодирование: есть словарь-бор, в который изначально добавлены все строки длины 1. Идём по тексту и параллельно спускаемся по бору. Как только выходим за пределы бора, добавляем новое слово в словарь-бор, и записываем код того слова, на котором вышли из бора.

Пример: $k = 3$, `ababc`. Пишем 0 (`a`) и добавляем в словарь `ab` (код 3), пишем 1 (`b`), добавляем в словарь `ba` (код 4), пишем 3 (`ab`), добавляем в словарь `abc` (код 5), пишем 2 (c).

Кодирование быстрое: один проход по данным. Чтобы словарь d не разрастался (нам нужно $\lceil \log |d| \rceil$ бит на один код), можно его приводить к исходному состоянию при превышении 4096. LZW получил известность, как формат хранения картинок `.gif` в 1987-м.

9.10. LZSS

Главная идея: можно повторять уже написанную часть. Для этого запомним пару $\langle p, n \rangle$ – откуда и сколько символов повторить. Пример: `hello`$\langle 3, 10 \rangle \to$ `hellolololololo`. В данном примере «откуда» = абсолютная позиция. Альтернатива: «на сколько символов назад откатиться».

Мы умеем быстро выбирать пару $\langle p, n \rangle$: $n = \max$ с помощью суффиксного массива, что даёт

асимптотику $\mathcal{O}(n)$ для сжатия текста длины n. На практике разберёмся, как оптимально кодировать и скрестим с Хаффманом, получим алгоритм, который жмёт тексты лучше `zip/rar`.

9.11. BWT

Пример: `harryharry` → `hhyyaarrrr`
Пример: `harry,oh,harry,harry` → `hyyhhho,y,,aaarrrrrr`

Преобразование: отсортировать все циклические сдвиги строки и вернуть последний столбец. Суффмассив мы умеем строить за $\mathcal{O}(n \log n)$ и даже $\mathcal{O}(n)$ (Каркайнен-Сандерс). По последнему столбцу мы умеем однозначно восстанавливать исходную строку (см.практику).

Замечание 9.11.1. Всегда нужно пытаться в конце применить Хаффмана. Хуже по Thm 9.7.3 не будет. В данном случае лучше тоже не будет, т.к. мы лишь переставили буквы, частоты не поменялись.

9.12. RLE

Пример: `aaaabbbcc` → `4a3b2c`.

Преобразование: группы одинаковых символов записывать, как $\langle n, c \rangle$.

Если нам нужно сжать bmp-картинку «красный квадрат», это прекрасная идея. Как это может пригодиться при сжатии текстовых данных? После `BWT` у нас появятся группы одинаковых символов. Пусть слово `gladiolus` встречается k раз в тексте $s \Rightarrow$ будет k суффиксов, начинающихся с `diolus`, все они заканчиваются на «`a`», и в суффиксном массиве скорее всего идут подряд (разве что есть другие слова с суффиксом `diolus`) $\Rightarrow$ получили «`aaa..a`» в `BWT`(s).

9.13. MTF (move to front)

Пример: `999000559` → `900100302`.
Преобразование: встретили символ $c \Rightarrow$ сдвигаем его в алфавите на первую позицию.

```
1 void MTF(vector<int> &text):
2     int k = *max_element(all(text)) + 1; // размер алфавита
3     int ord[k]; // порядок символов в алфавите
4     iota(ord, ord+k, 0); // изначально тождественная перестановка
5     for (int &c : text): // c - символ, который сейчас выписываем
6         int p = ord^{-1}[c]; // позиция символа c в алфавите
7         c = p; // собственно MTF (если ord тождественная, ничего не поменяли)
8         rotate(ord, ord + p, ord + k); // сдвинули символ c (позиция p) в начало
```

При данной реализации время работы $\mathcal{O}(nk)$ с малой константой. Можно хранить `ord` в декартовом дереве по неявному ключу, тогда получим $\mathcal{O}(n \log k)$. `MTF`$^{-1}$ такое же по коду и времени, изучим его на практике.

Посмотрим, как это работает: `MTF`$(222222) =$ `200000`, `MTF`$(333333) =$ `300000` $\Rightarrow$ в тех же ситуациях, когда `RLE` работает, `MTF` даст результат ещё лучше с точки зрения «скормим результат Хаффману». `MTF`$(27272727) =$ `27111111` $\Rightarrow$ `MTF` справляется с более широким классом закономерностей, чем `RLE`. Ещё плюс: в отличии от `RLE` мы не меняем исходный алфавит.

Мы получили возможные архиваторы:
$t \to$ `BWT` $\to$ `RLE` $\to$ `ARI`
$t \to$ `BWT` $\to$ `MTF` $\to$ `ARI`

$t \to$ `BWT` $\to$ `MTF` $\to$ `RLE` $\to$ `ARI`
Здесь `ARI` – арифметическое кодирование. Архиваторы упорядочены от плохого к хорошему. Аккуратная реализация «$t \to$ `BWT` $\to$ `MTF` $\to$ `RLE` $\to$ Хаффман»
будет работать на текстовых данных уже лучше чем обычные `*.zip`, `*.rar`.

9.14. Сжатие и предсказания

В процессии декодирования наше состояние: мы уже верно выписали i символов, какой символ (или даже слово) выписать следующим? Чтобы верно ответить на этот вопрос, нам дают подсказку «сколько-то следующих бит из архива» $\Rightarrow$ если мы хорошо умеем предсказывать текстовые данные (например, выдавать 3 длинных варианта, один из которых почти наверняка подойдёт), мы умеем хорошо кодировать (2 бита = или один из этих вариантов или «не один не подошёл, вот вам символ, что выписать дальше»).

Конкретно, как это можно использовать на практике:

1. Хаффман, который строит массив частот для каждого c: что с какой вероятность встречается непосредственно после c?

2. Тоже самое на расширенном словами алфавите.

3. Вариант с предварительным обучением. Пусть у нас есть библиотека английской классики, z. Построим суффиксный автомат от z (минимальный автомат, допускающий все суффиксы строки и только их). Сделаем его частью программы-архиватор (не частью архива). При сжатии текста t будем параллельно идти по автомату z и смотреть «какие символы с какой вероятностью могут быть следующими?». В зависимости от этих вероятностей получаем разные длины кодов для текущего символа (в каждой вершине автомата построили свои коды Хаффмана, но частоты хранить в архиве не нужно, т.к. они однозначно выводятся из автомата).

4. К идеи (3) расширить алфавит английскими словами.

9.15. Итоги и ссылки

Мы знаем Хаффмана. Умеем расширять алфавит словарными словами.
Мы знаем `LZSS`, на практике разберёмся, как оптимально написать и скрестить с Хаффманом.
Мы знаем цепочку `BWT` $\to$ `MTF` $\to$ `RLE` $\to$ Хаффман.

Реализация наших знаний для текстовых данных даст следующие результаты:

TODO

`[Some datasets, competition]`
`[Some benchmarks]`
`[src/release: paq8px]`. Крутой медленный архиватор: 445k $\to$ 95k.
`[src/release: lib with GPT-2 model]`. Крутой медленный архиватор: 445k $\to$ 59k.
`[wiki]`
`[Matt Mahoney | ShortBook]`
`[Better than RLE,MTF]`