Второй курс, осенний 2017/18
Конспект лекций по алгоритмам

Собрано 23 октября 2017 г. в 19:17

Содержание

Лекция #1: Паросочетания

4 сентября

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. Поиск паросочетания в двудольном графе

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 $M\colon |M| > |P|$, 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.

Lm 1.3.4. Пусть G – двудольный граф, а P паросочетание в нём. Построим орграф G', в котором из первой доли во вторую есть 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 1.3.5. Поиск ДЧП в $G \Leftrightarrow$ поиску пути в G' из свободной вершины в свободную.

Следствие 1.3.6. 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`

1.4. 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]`).

1.5. Алгоритм Куна

Lm 1.5.1. Если в процессе поиска максимального паросочетания не существует ДЧП из v, то ДЧП из v уже никогда не появится.

Доказательство. Пусть появился ДЧП $v \rightsquigarrow u$ после применения ДЧП $a \rightsquigarrow b$, тогда в предыдущий момент существовал ДЧП из v в a или b. (**TODO**: картинка). ■

Получили алгоритм Куна:

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

1.6. 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 1.6.1. 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 $\mathcal{O}(V + E)$. И, 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, запускать его.

Замечание 1.6.2. Докажите, 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.

1.7. Поиск минимального вершинного покрытия

Lm 1.7.1. $\forall M, VC$ 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 1.7.2. $X = A^- \cup B^+$ – 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 1.7.1 следует $|X| = |M|$ и $|X| = \max$. ■

Следствие 1.7.3. $A^+ \cup B^-$ – максимальное независимое множество.

Замечание 1.7.4. Мы умеем строить $\min VC$ и $\max IS$ за $\mathcal{O}(V+E)$ при наличии максимального паросочетания.

Следствие 1.7.5. $\max M = \min VC$ (теорема Кёнига).

1.8. 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.

1.9. Решения для произвольного графа

1.9.1. 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.

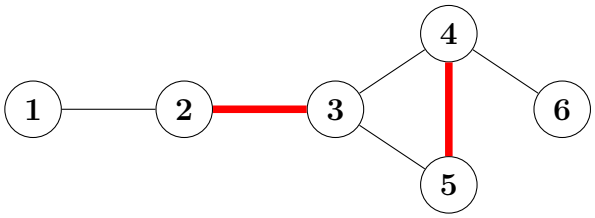
1.9.2. Матрица Татта

Рассмотрим 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 Татта: $\det T \not\equiv 0 \Leftrightarrow \exists$ совершенное паросочетание.
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 n^2/p.
Время работы $\mathcal{O}(n^3)$, 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.

1.9.3. 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. ∃ графы, на которых вероятность нахождения дополняющей цепи экспоненциально от числа вершин мала.

Лекция #2: Паросочетания (продолжение)

11 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. Может лежать в максимальном паросочетании, а может не лежать. `MAY`.
3. Не лежит ни в каком максимальном паросочетании. `NO`.

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, при этом $e \in P$.

Доказательство. Если ребро $e \in \texttt{MAY}$, то $\exists$ другое максимальное 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 $P \subseteq M$, 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.

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

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 $p_a = bs[a].\texttt{first}$, 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 $bs[i].\texttt{remove\_first}(), p_i = bs[i].\texttt{first}$.

Теорема 2.2.3. Алгоритм всегда завершится и найдёт стабильное паросочетание

Доказательство. Длины списков $bs[i]$ убывают $\Rightarrow$ завершится. Пусть мальчик a и девочка b образуют не стабильность. Но 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 $\mathcal{O}(\sum_i |bs[i]| + \sum_j |as[j]|)$: для каждой девочки b поддерживаем $s_b = \{a\colon p_a = b\}$, при присваивании $p_a \leftarrow b$ добавляем a в s_b и, если $|s_b| \geqslant 2$, вызываем рекурсивную процедуру $\texttt{откажиВсемКромеЛучшего}(b)$.

2.3. Венгерский алгоритм

Дан взвешенный двудольный граф, заданный матрицей весов $a\colon n \times n$, 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 $\pi \in S_n\colon \sum_i a_{i\pi_i} \to \min$.

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 $x\colon row_j \texttt{ += } x$.

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, x.
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.     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.         ((z,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] == z)
8.         for i ∈ A⁻: row[i] += z;
9.         for j ∈ B⁻: col[j] -= z, w[j].value -= z;
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 (x=pair[j]) == -1
13.            break // дополняющий путь: j,prev[j],pair2[prev[j]],prev[pair2[prev[j]]],...
14.        x перемещаем из A⁻ в A⁺;
15.        пересчитываем все w[j] = min(w[j], pair(a[x][j] + row[x] + col[j], x));
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.
Из них 8 и 9 улучшить до $\mathcal{O}(1)$, храня специальные величины `addToAMinus`, `addToBMinus`.
Строки 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 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 $\to$ покрасим рекурсивно $G \setminus \{v\} \to$ докрасим v.
У вершины v всего deg_v уже покрашенных соседей $\Rightarrow$
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. 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 покраски в D цветов NP-трудна.

2.4.3. Рёберные раскраски двудольных графов

С двудольными графами всё проще. Сейчас научимся красить их в D цветов.
Покраска рёбер – разбиения множества рёбер на паросочетания. В последнем домашнем задании 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 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 $\mathcal{O}(d \cdot \texttt{Matching}) = \mathcal{O}(E^2)$, 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.

2.4.4. 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)

18 сентября

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 $f_e \in \mathbb{R}$:*

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

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.

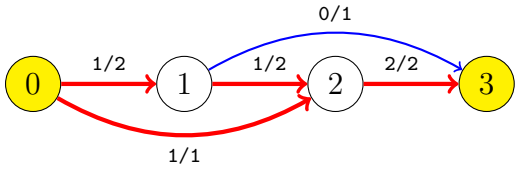
Замечание 3.1.4. $|f|$ может быть отрицательной: пустим по ребру $t \to s$ единицу потока.

Def 3.1.5. *Циркуляцией называется поток величины* 0.

• Примеры потока

Рассмотрим пока граф с единичными пропускными способностями.

1. $\forall$ цикл – циркуляция.
2. $\forall$ путь из s в t – поток величины 1.
3. $\forall$ 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

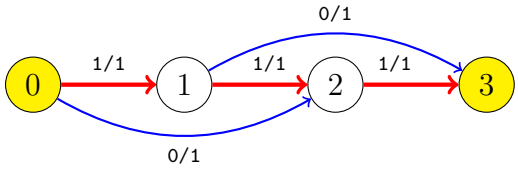
Def 3.2.1. *Для каждого ребра сети G с пропускной способностью c_e создадим обратное ребро e' пропускной способностью* 0. *При этом по определению $f_{e'} = -f_e$.*

Добавим в граф обратные рёбра, упростим определения потока и величины потока:
Теперь $\forall$ 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.

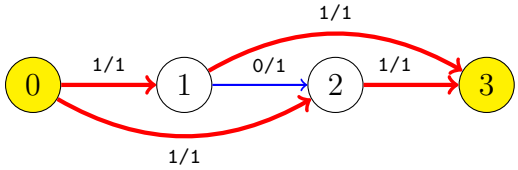
Def 3.2.2. *Ребро называется насыщенным, если $f_e = c_e$, иначе оно ненасыщено.*

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 3.7.2 и 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 3.4.8 следует, что $|f| = \max, C(S,T) = \min$. ■

• Поиск минимального разреза

Из доказательства теоремы 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:

```
1 struct Edge {
2   int a, b, f, c, rev; // a ---> b
3 };
4 vector<Edge> c[n]; // c[c[v][i].b][c[v][i].rev] - обратное ребро
5 for (Edge e : c[v]) // перебор рёбер, смежных с v
6   ;
```

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

```
1 struct Edge {
2   int a, b, f, c; // собственно ребро
3   int next; // интрузивный список, список на массиве
4 };
5 vector<Edge> edges;
6 vector<int> head(n, -1); // для каждой вершин храним начало списка
7 for (int i = head[v]; i != -1; i = edges[i].next)
8   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

TODO

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 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: теперь путь ищем bfs-ом. Конец.

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

Доказательство. **TODO** ■

• 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 $e\colon a \to b$, кратчайший путь прошёл через $e \Rightarrow$
$d[b] = d[a] + 1$. Когда кратчайший путь пройдёт через обратное к e имеем

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

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 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: $f_e + 2^k \leqslant c_e$.

```
1 for k = logU .. 0:
2   u <-- 0
3   while dfs(s, 2^k):
4     u <-- 0
5     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)

25 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 void dfs(int v) {
2   while (head'[v] != -1) {
3     Edge &e = edges[head'[v]];
4     if (e.f < e.c && (e.b == t || dfs(e.b))) {
5       // нашли путь
6     }
7     head'[v] = e.next;
8   }
9 }
```

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:

```
for (int k = log U; k >= 0; k--)
  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 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

```
void dfs(int v):
  for (x ∈ N(v)): // x - сосед во второй доле
    if (used2[x]++ == 0) // проверили, что в x попадаем впервые
      if (pair2[x] == -1 || (dist[pair2[x]] == dist[v]+1 && dfs(pair2[x])):
        pair1[v] = x, pair2[x] = v
        return 1
  return 0
while (bfs нашёл путь свободной в свободную): // цикл по фазам
  used2 <-- 0 // пометки для вершин второй доли
  for (v ∈ A): // вершины первой доли
    if (pair1[v] == -1): // вершина свободна
      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 $2U^{1/2}V^{2/3}$, 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 $a_{i+1} = v \in V \setminus A_i$: $C(A_i, \{v\}) = \max$.

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.
$\exists v$: $deg_v = k \Rightarrow C(\{v\}, V \setminus \{v\}) = k \Rightarrow$ 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 $Pr[e \in cut] \leqslant \frac{k}{E} = \frac{2}{V}$.

• **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.
Обозначим $p(V) = q_k, p(V/\sqrt{2}) = q_{k-1}, \ldots, \Rightarrow$

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

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

Лекция #5: Mincost

2 октября 2017

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 e' сопоставим $w_{e'} = -w_e$.

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:

```
1 for (int e : path):
2   edges[e].f++
3   edges[e ^ 1].f--
4   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 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, 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 $\geqslant 0$. ■

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. 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 циркуляцию в графе с отрицательными циклами.

Решение #1: найти за $|W(f)|$ итераций mincost циркуляцию, перейти от f_0 за k итераций к f_k.
Решение #2: найти любой поток $f\colon |f| = k$, в G_f найти mincost циркуляцию, сложить с f.

5.4. Mincost flow

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

Lm 5.4.1. Обозначим f_k – оптимальный поток размера k, p_k кратчайший путь в G_{f_k}.
Тогда $W(p_k) \nearrow$, как функция от k.

Доказательство. **TODO** (аналогично доказательству леммы для Эдмондса-Карпа). ■

Следствие 5.4.2. $(W(f_k) = \min) \Leftrightarrow (W(p_{k-1} \leqslant 0 \wedge W(p_k) \geqslant 0)$.

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

5.5. 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 (∃ отрицательный цикл) ⇔ (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
10         пересчитаем потенциалы, используя расстояния, найденные Дейкстрой
```

Время работы алгоритма $E \log U$ запусков Дейкстры – $E(E + V \log V) \log U$.
Осталось доказать, что после 9-й строки у наша циркуляция минимальна:

TODO

• **Решение #3:** Cost Scaling.

TODO
Cost scaling (часть 1)
Cost scaling (часть 2)

Лекция #6: Базовые алгоритмы на строках

9 октября 2017

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 задач 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:

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

или таким

```
1 for (char *p = t; (p = strstr(p, s)) != 0; p++)
2   ; // 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* $\pi(s)$: $\pi(s)[i] = \pi_0(s[0{:}i))$.

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 $w \Rightarrow (j = i - 2|s|)$ – начало вхождения s в t.

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

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

TODO : картинка

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;
```

Вычисление префикс функции работает за $\mathcal{O}(n)$, так как k увеличится $\leqslant n$ раз $\Rightarrow$ суммарное число шагов цикла `while` не более 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 `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 = 0; i < n; i++)
  int k = 0;
  while (s[i + k] == s[k])
    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. $\forall l < i < l + z[l] = r$ имеем `s[0:z[l])` = `s[l:r)` и `s[i-l:z[l])` = `s[i:r)`.

Следствие леммы: $z[i] \geqslant \min(r - i, z[i] - l)$. Логично взять l: $r = l + z[l] = \max$.
Немного модифицируем код, чтобы получить асимптотику $\mathcal{O}(n)$.

```
z[0] = 0, l = r = 0;
for (i = 0; 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. ■

6.2.5. 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 $\Theta(|m|/|n|)$.

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.

• Полиномиальная хеш-функция

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

$h_{p,m}(s)$ – 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 6.3.2.

```
typedef unsigned long long T;
T *h; // важна беззнаковость типа, чтобы не было undefined behavior
void initialize( int n, char* s ) {
  h = new T[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] = ((__int128)h[i] * p + s[i]) % m; // 0 ⩽ m < 2^63
}
T getHash( int l, int r, char* s ) { // [l,r)
  return h[r] - h[l] * deg[r - l]; // deg[r - l] = p^{r-l}, никогда не пишите здесь лишний if
}
```

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 $ht_{i+1} = ht_i \cdot p - t_i \cdot p^n$.

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 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 стороками**

$h_{p,m}(S) = S(p) \bmod m$, 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 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 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 6.3.10 не достигается. В практических рассчётах можно смело пользоваться оценкой $\frac{k}{m}$ из 6.3.9.

6.3.4. Число различных подстрок

Рассмотрим два решения, оценим их вероятности ошибок.
Решение #1: сложить хеши всех $n(n-1)/2$ 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 $\approx \frac{1}{6 \cdot 10^6}$.

Лекция #7: Суффиксный массив

16 октября 2017

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 $x = lcp(s, t)$.
Результатом сравнения s и t на больше/меньше будет "`s[x] < t[x]`".

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`.

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, и приходит куча строк s_i, запрос "является ли s_i подстрокой t".

Предподсчёт: построим суффиксный массив строки t. Отвечать на запрос будем бинпоиском:
(s_i – подстрока t) $\Leftrightarrow$ (s_i является началом какого-то суффикса t).
В суффиксом массиве p 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 разд. 7.6 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 $\langle c[i], c[i+k]\rangle$.

Осталось заметить, что $\forall k \geqslant n$ порядок строк длины k совпадёт с порядком строк длины n.

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 k.
`c[i]` – тип подстроки `s[i:i+k)` циклической строки `s#`
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, можно это воспринимать, как "уже отсортированы строки длины $2k$ по второй половине" $\Rightarrow$ осталось сделать сортировку подсчётом по первой половине.

```
pos <-- 0
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++) { // позиция начала второй половины - p[i]
  int j = (p[i] - k) mod n; // j - позиция начала первой половины
  p2[pos[c[j]]++] = j;
}
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

Собственно 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.

Полезное утверждение: 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 7.4.1. 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 первом символу, получили суффиксы `s[i+1:]` и какой-нибудь r. Поскольку `s[i:]` $\neq$ `s[next`$_i$`:]`, а первый символ у них совпадал, `s[i+1:]` и r идут в суффмассиве в таком же порядке и совпадает у них $lcp_i - 1$ символ $\Rightarrow$ 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 $i \searrow$ 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. Построение за $\mathcal{O}(n)$: алгоритм Каркайнена-Сандерса

На вход получаем строку s длины n, при этом $0 \leqslant s_i \leqslant \frac{3}{2}n$.
Хотим построить 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 $t_0 t_1$. Длины $t_0 t_1$ 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.

TODO : псевдокод

7.6. Быстрый поиск строки в тексте

TODO : переписать

$\mathcal{O}(|s| + \log(|text|))$.

Сделаем бинпоиск по суффиксному массиву, на котором уже насчитали lcp. Будем поддерживать lcp искомой строки и границ бинпоиска. Посмотрим на строку посередине отрезка. Ясно, что $lcp(s, m) \geq max\{min\{lcp(s, l), lcp(l, m)\}, min\{lcp(s, r), lcp(r, m)\}\}$. Так что lcp с m мы начинаем находить с этого максимума, а не с начала строки. $lcp(l, m) = min$ на отрезке, $lcp(r, m)$ – аналогично, два других lcp мы уже знаем. Пусть мы построили дерево отрезков на значениях lcp. Заметим, что каждый раз интересующий нас диапазон уменьшается ровно вдвое. Т.е, мы как бы всё это время спускаемся вниз по дереву отрезков. Т.е., если хранить, в какой вершине мы сейчас стоим, можем получать минимум на отрезке за $\mathcal{O}(1)$.

Далее, почему все поиски lcp суммарно работают за длину строки. $lcp(s, r) + lcp(s, l)$ не уменьшаются, т.к. не уменьшается ни минимум, ни максимум из этих двух величин ($lcp(s, m)$ больше либо равен минимума, т.е. если он станет новым минимумом, то минимум не уменьшится, если новым максимумом, то аналогично). А как только мы увеличиваем $lcp(s, m)$ руками, т.е., продолжаем втупую идти с некоторого места строки, то сейчас $lcp(s, m) = max\{lcp(s, l), lcp(s, r)\}$.

Лекция #8: Ахо-Корасик и Укконен

23 октября 2017

8.1. Бор

TODO

8.2. Алгоритм Ахо-Корасика

TODO

8.3. Суффиксное дерево, связь с массивом

TODO

8.4. Алгоритм Укконена

TODO