Лекция 6

Алгоритмы на графах (II).

Нахождение кратчайших путей. Изоморфизм деревьев. Максимальный поток.

6.1 Нахождение кратчайших путей

Задача: в ориентированном графе с заданными длинами ребер найти кратчайшие пути из одной вершины s_0 во все остальные. (На самом деле, «найти пути» здесь означает «для каждой вершины v найти длину кратчайшего пути из s_0 в v и предыдущую вершину $p[v]$ на каком-нибудь кратчайшем пути из s_0 в v».)

6.1.1 Алгоритм Дейкстры

В алгоритме Дейкстры предполагается, что длины всех ребер $w(v_1, v_2)$ неотрицательны. Этот алгоритм похож на алгоритм Прима: мы постепенно подсчитываем кратчайшие расстояния $d[s]$ (и находим кратчайшие пути) для все большего числа вершин s — множество S обработанных вершин растет на каждом шаге. Длины путей до вершин, не входящих в S, мы тоже подсчитываем, но на промежуточных этапах это будут еще не кратчайшие пути.

На очередном шаге добавляем к множеству S вершину $u \in V \setminus S$, для которой $d[u]$ минимально (в дальнейшем будет показано, что это автоматически означает, что $d[u]$ уже подсчитано правильно). Далее учитываем этот факт: для каждой соседней с u вершины $v \notin S$ корректируем $d[v]$ следующим образом: $d[v] := \min\{d[v],\ d[u] + w(u, v)\}$. Если $d[v]$ уменьшилось, отмечаем u как предыдущую вершину на кратчайшем пути из s_0 в v.

Для добавления очередной вершины надо ее найти (для этого достаточно $O(|V|)$ операций) и просмотреть всех ее соседей (в сумме для всех вершин — $O(|E|)$ операций). Итого получается $O(|E|) + |V| \cdot O(|V|) = O(|V|^2)$ операций с вершинами, ребрами *и длинами путей.*

Замечание 6.1. Можно найти способ представить данные в алгоритме Дейкстры так, чтобы ограничиться $O((|V| + |E|) \log |V|)$ и даже $O(|V| \log |V| + |E|)$ операциями.

Теорема 6.1. *Алгоритм Дейкстры правильно находит кратчайшие пути.*

Доказательство (индукция по построению множества S). Пусть на очередном шаге мы добавили к множеству S вершину u, но на самом деле имеется путь длины $\delta < d[u]$. Рассмотрим первое ребро (s,t) на этом пути из s_0 в u, выводящее за пределы множества S; пусть $\gamma(t,u)$ — длина остатка этого пути от t до u.

По предположению индукции, $d[s]$ — длина кратчайшего пути до s. Таким образом,

$$\begin{aligned}
\delta &= d[s] + w(s,t) + \gamma(t,u) \geq \\
&\qquad \text{\scriptsize (поскольку алгоритм Дейкстры обязательно учел ребро (s,t) при добавлении вершины s)} \\
&\geq d[t] + \gamma(t,u) \geq \\
&\qquad \text{\scriptsize (по выбору u)} \\
&\geq d[u] + \gamma(t,u) \geq \\
&\geq d[u],
\end{aligned}$$

что противоречит тому, что $\delta < d[u]$. □

6.1.2 Алгоритм Беллмана-Форда

Рассмотрим теперь случай, когда некоторые ребра имеют отрицательную длину. Как видно из доказательства теоремы 6.1, алгоритм Дейкстры может на таком графе работать неправильно.

Упражнение 6.1. Построить конкретный пример (без циклов отрицательной длины), на котором алгоритм Дейкстры работает неправильно. □

Будем предполагать, что в графе нет циклов отрицательной длины, достижимых из s_0 (если они есть, некоторые из кратчайших расстояний будут равны $-\infty$ — в этом случае алгоритм будет выдавать ошибку).

Алгоритм Беллмана-Форда пользуется теми же данными $d[v]$ и $p[v]$, что и алгоритм Дейкстры. Он прост: $n-1$ раз повторить следующую операцию: для каждого ребра (u, v), если $d[u] + w(u, v) < d[v]$, то $d[v] := d[u] + w(u, v)$ и $p[v] := u$. Если после всех этих итераций для какого-то ребра по-прежнему $d[u]+w(u, v) < d[v]$, значит, есть цикл отрицательной длины, и можно выдать ошибку.

Очевидно, этот алгоритм затрачивает $O(|V| \cdot |E|)$ операций.

Теорема 6.2. *Алгоритм Беллмана-Форда корректно находит кратчайшие пути либо обнаруживает цикл отрицательной длины.*

Доказательство. Индукция по длине пути (а она $\leq |V| - 1$).

Что же касается цикла отрицательной длины — если он есть, но строгое неравенство $d[u]+w(u, v) < d[v]$ не выполняется ни для одного из его ребер (u, v), сложим все обратные неравенства и получим, что длина этого цикла неотрицательна (противоречие). □

6.1.3 Рекурсивный алгоритм для определения длины кратчайшего пути

Нашей следующей задачей будет определение кратчайших путей для всех пар вершин графа. Перед этим изучим другой алгоритм, основанный на следующем простом соображении.

Очевидно, что кратчайший путь из вершины x в вершину y из $\leq t$ ребер можно разбить на два пути из $\leq \lceil t/2 \rceil$ ребер каждый: кратчайший путь из x в некоторую вершину z и кратчайший путь из z в y.

Эта идея приводит к следующему рекурсивному алгоритму для нахождения *длины* кратчайшего пути из вершины s в вершину t, затрачивающему лишь $O(\log |V|)$ ячеек памяти размера $O(\log |V| + \log \max$ длин ребер) в предположении, что граф задан неявно (т.е. имеется лишь функция, определяющая по u и v длину $w(u, v)$ ребра (u, v)).

```
function shortest(x, y, t) : boolean;
begin
    if x = y then return 0;
    if t = 1 then return w(x, y);                    (*может быть +∞*)

    best := +∞;
    for z := 1 to |V| do
    begin
        new:=shortest(x, z, ⌈t/2⌉)+ shortest(z, y, ⌈t/2⌉);
        if new<best then best:=new;
    end;
    return best;
```

end;

Вызов из главной программы: shortest$(s, t, |V| - 1)$.

Временнáя сложность этого алгоритма велика.

Упражнение 6.2. Определить ее. □

6.1.4 Пути между всеми парами вершин

Подобно тому, как мы уже делали, воспользуемся динамическим программированием, чтобы избавиться от рекурсии. Заодно мы найдем и длины всех кратчайших путей между всеми парами вершин, и сами кратчайшие пути, причем быстрее, чем если бы мы $|V|$ раз повторили алгоритм Беллмана-Форда.

Алгоритм Флойда-Уоршолла. Будем постепенно заполнять трехмерный массив d, где $d_{ij}^{(k)}$ — длина кратчайшего пути из вершины i в вершину j с промежуточными вершинами *только* из множества $\{1, \dots, k\}$.

Значения при $k = 0$ — это длины ребер исходного графа, т.е. $d_{ij}^{(0)} = w(i, j)$. Далее массив заполняется для возрастающих k, а при постоянном k — для всех i и j, по формуле

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}.$$

Корректность алгоритма и время его работы $O(|V|^3)$ очевидны.

Упражнение 6.3. Заодно определить и матрицу p, где p_{ij} — предпоследняя вершина на кратчайшем пути из i в j. □

6.2 Изоморфизм деревьев

Взаимно-однозначное отображение $f : V_1 \to V_2$ между множествами вершин графов (V_1, E_1) и (V_2, E_2) называется *изоморфизмом*, если оно ребро переводит в ребро и наоборот:

$$\forall u, v \in V_1 \quad (u, v) \in E_1 \iff (f(u), f(v)) \in E_2.$$

Графы, между которыми существует изоморфизм, называются *изоморфными*.

Проверка изоморфности графов — очень трудная задача. Мы решим более простую задачу: определим изоморфность деревьев.

Отступление: лексикографическая сортировка. Задача: дано n строк общей длиной L, состоящих из чисел от 1 до k. Требуется отсортировать их в соответствии с лексикографическим упорядочением:

$$s \leq t \iff s \text{ — префикс } t \ \vee \ \exists i \, (s_i < t_i \wedge \forall j < i (s_j = t_j))$$

(как слова в словаре).

Поступим так: отсортируем строки по длине, получим упорядоченный список A. Будем поддерживать массив B, состоящий из k упорядоченных списков строк, сортируя строки по числу s_i, стоящему на одной и той же позиции i (естественно, рассматривая только те строки, где эта позиция есть): просто переместим строку s в список $B[s_i]$. Начинать будем с последней позиции (для строк максимальной длины). На очередном этапе берем строки уже из упорядоченных списков (из оставшейся части списка A, а также из списков $B[1], \ldots, B[k]$) в том порядке, в котором они в этих списках находятся (конечно, сначала нужно из списков $A, B[1], \ldots, B[k]$ сделать списки $B'[1], \ldots, B'[k]$, а уже потом переименовать списки $B'[i]$ в $B[i]$). Ясно, что для строк, попадающих в один список, сохраняется «старое» упорядочение.

Корректность этого алгоритма очевидна. Теперь будем «дорабатывать» его, чтобы ограничиться $O(L + k)$ операциями с символами, указателями на строки и длинами строк (значит, время работы — $O((L+k)\log(L+k))$. Каждую строку мы перемещаем между списками столько раз, сколько в ней символов — всего $O(L)$ перемещений; сортировка строк по длине займет $O(L)$ на вычисление длины и $O(L)$ на сортировку (эта сортировка тоже состоит просто в том, что мы отправляем каждую строку длины l в список $A'[l]$ вспомогательного массива A', а потом сливаем все списки $A'[l]$ в список A); организация списков и их окончательное объединение — $O(k)$ операций. Проблема только в просматривании k списков на каждом шаге: так может получиться $\Omega(kL)$ операций; это много; но ведь списки-то зачастую — пустые (просмотр непустых списков мы, кстати, уже учли в перемещении строк!).

Чтобы не просматривать пустые списки, создадим заранее список пар (l, s_{il}), когда s_{il} действительно существует. Отсортируем этот список сначала по второй компоненте, а потом по первой (также «рассортировывая» соответственно по k или L упорядоченным спискам). После этого легко создать упорядоченные списки символов, встречающихся на позиции l. На все это уйдет $O(L + k)$ операций, зато даст возможность просматривать в исходном алгоритме только непустые списки.

Будем считать, что в двух наших деревьях выделены корни[1]; иначе говоря, зафиксируем значение отображения (будущего изоморфизма) на

[1]Вообще говоря, дерево не обязано иметь корень: *дерево* — это просто неориентированный связный граф, не содержащий циклов.

одной из вершин (для того, чтобы свести общую задачу к этой, достаточно взять *произвольную* вершину, объявить ее корнем, и перебирать вершины, в которые она может отобразиться). Разобьем множество вершин каждого дерева на уровни (по расстоянию вершины от корня). Эти уровни будем обрабатывать от набольшего — к наименьшему (так что в конечном итоге дойдем до корня), приписывая каждой вершине число.

На очередном уровне сначала припишем каждой вершине вектор (быть может, пустой!) чисел, которыми (уже) помечены ее сыновья. Затем отсортируем эти вектора в лексикографическом порядке (отдельно для каждого дерева). Сравним полученные два вектора векторов: они должны совпасть (если нет, деревья неизоморфны).

Перенумеруем *различные* компоненты этих векторов (снова вектора!) последовательными натуральными числами. Это и есть пометки, которые мы припишем соответствующим вершинам.

Если, дойдя до корня, мы не обнаружим разных векторов, деревья изоморфны.

Упражнение 6.4. Докажите, что этот алгоритм корректен (модифицируйте алгоритм так, чтобы он находил и сам изоморфизм) и совершает линейное число операций (придумайте подходящие структуры данных для его эффективной реализации). □

6.3 Задача о максимальном потоке

6.3.1 Лемма о максимальном потоке и минимальном сечении

Дан ориентированный граф $G = (V, E)$ с пропускными способностями ребер, заданными отображением $c : E \to \mathbb{R}$. (Для простоты полагаем $c \equiv 0$ на $V \times V \setminus E$.) Указаны источник $s \in V$ и сток $t \in V$.

Определение 6.1. Отображение $f : V \times V \to \mathbb{R}$ называется *потоком* в G, если

$$\forall u, v \in V \quad f(u, v) \le c((u, v)),$$

$$\forall u, v \in V \quad f(u, v) = -f(v, u),$$

$$\forall v \in V \sum_{u:(u,v)\in E} f(u, v) = \sum_{w:(v,w)\in E} f(u, v).$$

Величина $\sum\limits_{w:(s,w)\in E} f(s, w)$ (она же $\sum\limits_{u:(u,t)\in E} f(u, t)$) называется *значением* потока.

Требуется найти поток с наибольшим значением — *максимальный поток*.

Определение 6.2. Сечением в графе называется разбиение V на два непустых непересекающихся множества V_1 и V_2. Также сечением называется множество ребер (v_1, v_2), таких, что $v_1 \in V_1$ и $v_2 \in V_2$ (но не наоборот!). Пропускной способностью сечения называется суммарная пропускная способность этих ребер. Потоком через сечение называется суммарный поток

$$\sum_{v_1 \in V_1,\, v_2 \in V_2} f(v_1, v_2)$$

через эти и обратные (соответственно, с обратным знаком) ребра.

Определение 6.3. Остаточной сетью G_f для данного потока f в графе G называется множество ребер (u, v), для которых $c_f((u,v)) := c((u,v)) - f(u,v) > 0$. Дополняющим путем называется путь γ из s в t в остаточной сети. Очевидно, при наличии дополняющего пути поток можно увеличить на $\min\limits_{e \in \gamma} c_f(e)$.

Лемма 6.1 (о максимальном потоке и минимальном сечении). *Следующие три условия эквивалентны.*

1. *Поток f — максимальный.*
2. *В остаточной сети G_f не имеется дополняющих путей.*
3. *Имеется сечение, пропускная способность которого равна потоку через это сечение.*

Доказательство. «1»⇒«2» и «3»⇒«1» очевидны. Докажем «2»⇒«3». Рассмотрим множество S вершин, достижимых из s в графе G_f. По «2», $V \setminus S \neq \emptyset$. Сечение $S, V \setminus S$ — искомое. □

6.3.2 Алгоритм Форда-Фалкерсона

Алгоритм чрезвычайно прост: будем находить дополняющий путь и увеличивать поток на его значение. Корректность алгоритма очевидна (если он остановится!). Однако, в зависимости от пропускных способностей и конкретной реализации этого алгоритма, время его работы может быть различным (в том числе, он может и зациклиться). Рассмотрим несколько случаев:

Целочисленные пропускные способности. В этом случае алгоритм можно легко реализовать за $O(M|E|)$ операций, где M — значение максимального потока. В самом деле, добавляя каждый дополняющий путь, мы увеличиваем значение потока по крайней мере на единицу. Дополняющий путь же можно найти поиском в глубину (или ширину) и убрать из остаточной сети за линейное количество операций.

Алгоритм Эдмондса-Карпа. Если M велико (или веса — не целочисленные), оценка для предыдущего алгоритма будет не слишком хороша. Покажем, что если искать дополняющие пути поиском в ширину, будет найдено лишь $O(|V| \cdot |E|)$ дополняющих путей, т.е. количество операций будет $O(|V| \cdot |E|^2)$.

> **Отступление: поиск в ширину.** Поиск в ширину преследует ту же цель, что и поиск в глубину: обработать все вершины графа. В отличие от поиска в глубину, основанного на рекурсии, а значит, использующего стек, поиск в ширину можно реализовать при помощи очереди. Очередная вершина вынимается из начала очереди (при инициализации в очередь помещается только одна вершина — корень будущего дерева поиска), обрабатывается, и ее сыновья — те, которых еще нет в очереди, — кладутся в ее конец. Если очередь опустеет, туда кладется следующая необработанная вершина, и т. д. Очевидно, этот алгоритм корректен и использует линейное число операций.

«Искать путь из s в t поиском в ширину» означает «запустить поиск в ширину, начав его с вершины s, и прервать, как только в очередь попадет вершина t». Заметим, что поиск в ширину находит путь, состоящий из наименьшего количества ребер (кратчайший): в нем сначала обрабатываются вершины, находящиеся на расстоянии 1 от s, затем — на расстоянии 2, и т. д.

Назовем ребро e в дополняющем пути γ *критическим*, если на нем достигается $\min\limits_{e \in \gamma} c_f(e)$. При удалении очередного дополняющего пути хотя бы одно из его ребер оказывается критическим. Значит, для доказательства искомой верхней оценки достаточно доказать следующую лемму.

Лемма 6.2. *Ребро может стать критическим лишь $O(|V|)$ раз.*

> **Замечание 6.2.** Заметим, что ребра пропадают, оказываясь критическими на выбранном дополняющем пути, а появляются, когда обратные ребра оказываются (необязательно критическими) на выбранном дополняющем пути.

Доказательство. Будем обозначать через $d(x, y)$ расстояние между вершинами в очередной остаточной сети.

Лемма 6.3 (к лемме 6.2). *Расстояние $d(s, y)$ не убывает.*

Доказательство. Пусть оно уменьшилось, а поток при этом увеличился с f до f'. НУО можем считать, что y — ближайшая к s в $G_{f'}$ вершина с таким свойством. Рассмотрим вершину x, предшествующую y на кратчайшем пути из s в y в $G_{f'}$. До этой вершины, по предположению, расстояние не уменьшилось.

Следовательно, ребро (x, y) возникло при переходе от f к f', т.е. соответствующий дополняющий путь содержал (y, x). Он был кратчайшим; значит, было $d(s, x) - 1 = d(s, y)$. Но $d(s, x)$ не уменьшилось, и $d(s, y)$ стало больше, чем $d(s, x)$; следовательно, $d(s, y)$ увеличилось! □

Ребро (u, v) пропадает тогда и только тогда, когда оно становится критическим в каком-то дополняющем пути. Поскольку это ребро лежит на кратчайшем пути, до этого было $d(s, v) = d(s, u)+1$. Ребро (u, v) появляется вновь, если ребро (v, u) входит в какой-то дополняющий путь. Поскольку теперь уже (v, u) лежит на кратчайшем пути, перед этим моментом было $d(s, u) = d(s, v) + 1$. Но эти расстояния не убывают (по лемме 6.3)! Следовательно, в промежутке между двумя моментами, когда ребро (u, v) побывает критическим, расстояние $d(s, v)$ увеличится на 2. Но оно не может стать больше $|V| - 1$! □

6.3.3 Применение алгоритма Форда-Фалкерсона для нахождения максимального паросочетания в двудольном графе

Определение 6.4. Неориентированный граф (V, E) называется *двудольным*, если множество его вершин разбито на два непересекающихся множества V_1 и V_2, таких, что $\forall i \, \forall u, v \in V_i \;\; \{u, v\} \notin E$.

Определение 6.5. Паросочетанием в неориентированном графе (V, E) называется множество $E' \subseteq E$, такое, что $\forall e_1, e_2 \in E' \;\; e_1 \cap e_2 = \emptyset$.

Найдем в двудольном графе паросочетание, состоящее из наибольшего количества ребер. Для этого модифицируем граф: сделаем его ориентированным (направление всех ребер — из V_1 в V_2), добавим источник s и ребра (s, v_1) для всех $v_1 \in V_1$, а также сток t и ребра (v_2, t) для всех $v_1 \in V_2$. Пропускные способности всех ребер положим равными единице. Применим алгоритм Форда-Фалкерсона. Как нетрудно заметить, ребра, по которым будет течь найденный им максимальный поток, и дадут максимальное паросочетание. Поскольку в нем не может быть более $|V|/2$ ребер, алгоритм совершит лишь $O(|V| \cdot |E|)$ операций.