Лекция 5

Алгоритмы на графах (I).

Граф и его представление в машине. Поиск в глубину. Минимальное остовное дерево.

5.1 Представление графа в машине

Определение 5.1. Ориентированный граф G — это пара конечных множеств (V, E), где V называется множеством вершин, а $E \subseteq V \times V$ — множеством ребер (ребро (v_1, v_2) можно представить себе как стрелку из v_1 в v_2). Будем считать, что в графе нет петель, т.е. $\forall v\ (v, v) \notin E$.

Неориентированный граф отличается от ориентированного тем, что на ребрах не указано направление, т.е. ребра — множества, а не упорядоченные пары; тогда множество ребер $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$.

Часто рассматривают графы с весами — когда каждому ребру приписано некоторое число, называемое весом этого ребра.

Поскольку множество V конечно, можно считать его элементы последовательными натуральными числами.

Вот некоторые способы представить граф в машине (выбор конкретного способа определяется задачей — в каком виде граф нам задан, и алгоритмом — какие операции нам нужны):

- матрица смежности:
  - двумерный массив: $A[i, j] = 1 \Leftrightarrow (i, j) \in E$,
  - матрица с пропусками,
  - неявное задание (при помощи *функции* f): $f(i, j) = 1 \Leftrightarrow (i, j) \in E$;
- массив (из $|V|$ элементов), в котором $A[i]$ — массив/список/дерево ребер, исходящих из вершины i.

Замечание 5.1. Время работы алгоритмов будем выражать не через длину входа, а через $|E|$, $|V|$ и другие параметры графа — это точнее и охватывает большее число случаев — например, когда граф заведомо разреженный (через длину входа всегда сможем выразить).

5.2 Поиск в глубину

Рекурсивно обрабатываем вершины (ориентированного или неориентированного) графа так, чтобы каждую вершину и каждое ребро обработать ровно один раз.

Именно, отмечаем вершину v, которую начали обрабатывать (проставляем момент времени $b[v]$ начала ее обработки). Затем *последовательно* рассматриваем всех ее соседей v_i: если v_i еще не начали обрабатывать, рекурсивно обрабатываем v_i (заметим, что в процессе рекурсивной обработки вершины v_i мы можем обработать v_{i+1}!). Закончив эти рекурсивные вызовы, проставим момент $e[v]$ конца обработки вершины v.

Если после вызова этой рекурсивной процедуры из «главной программы» в графе остались необработанные вершины, берем любую из них и обрабатываем той же процедурой — и так, пока необработанные вершины не кончатся.

Побочным эффектом поиска в глубину является возможность построения *леса поиска в глубину* — совокупности ребер, по которым мы шли, совершая рекурсивные вызовы (*лес* — это граф, являющийся объединением нескольких не связанных между собой деревьев).

Лемма 5.1. *Каждая вершина будет обработана ровно один раз. Каждое ориентированное ребро будет проверено (в цикле поиска соседей) ровно один раз.*

Доказательство. Мы никогда не начинаем обработку вершины, которую уже обрабатывали. □

Лемма 5.2. *Поиск в глубину занимает $O(|E| + |V|)$ операций с вершинами и ребрами, т.е. время $O((|E| + |V|) \log |V|)$.*

Доказательство. Следует из леммы 5.1. Обращений к каждой вершине и данным, с ней связанным, — не более, чем количество ее соседей (значит, всего таких обращений — $O(|E|)$). □

Замечание 5.2. Тут важно представление графа: матрица смежности не подойдет! (Сразу получится время $O(|V|^2)$.)

5.2.1 Топологическая сортировка

Задача о топологической сортировке вершин ориентированного графа: расположить вершины ориентированного графа в таком порядке, чтобы все ребра шли от меньшей (в этом порядке) вершины к большей.

Решение: применить поиск в глубину и отсортировать вершины по убыванию времени окончания обработки.

Лемма 5.3. *В графе без (ориентированных) циклов сортировка по убыванию времени окончания обработки является топологической сортировкой.*

Доказательство. Пусть в графе имеется ребро (v, w), хотя $e[v] < e[w]$. Есть два варианта «скобочной структуры»:

$\{\}()$: $b[v] < e[v] < b[w] < e[w]$, но это противоречит тому, что поиск в глубину делает рекурсивный вызов для всех соседей v, обработка которых еще не началась (в частности, для w);

$\{()\}$: $b[w] < b[v] < e[v] < e[w]$, но это означает, что есть ориентированный путь из w в v, т.е. имеется ориентированный цикл.

$\square$

Замечание 5.3. Если же в графе имеются ориентированные циклы, то топологическую сортировку произвести невозможно.

5.2.2 Компоненты сильной связности

Задача: разбить ориентированный граф на компоненты сильной связности, т.е. на подмножества вершин, в каждом из которых имеется (ориентированный) путь из любой вершины в любую (в обоих направлениях), а для любых двух вершин u и v из разных компонент либо нет пути из u в v, либо нет пути из v в u.

Решение:

- произвести поиск в глубину, найдя время окончания обработки $e[v]$ для каждой вершины;
- произвести поиск в глубину в графе G^t, отличающемся от графа G тем, что ребра в нем идут в противоположном направлении; причем следующую вершину v для вызова процедуры рекурсивного поиска из «главной программы» выбирать в порядке убывания $e[v]$;
- полученные деревья поиска в графе G^t и будут сильно связными компонентами.

Лемма 5.4. *При поиске в глубину, если есть путь из v в w, и вершина v начала обрабатываться до вершины w, то вершина w попадет в то же самое дерево поиска, что и v.*

Доказательство (индукция по длине пути). Пусть y — первая вершина на пути из v в w, не попавшая в то же дерево поиска, что и v, а x — последняя попавшая туда вершина (т.е. x идет непосредственно перед y в этом пути).

Если бы обработка вершины y еще не была начата, когда началась обработка вершины x, то y безусловно попала бы в то же дерево поиска, что и x (а значит, то же, что и v). Если бы обработка y была начата, но не закончена до начала обработки x, ситуация была бы такой же, поскольку одновременно могут обрабатываться только вершины, попадающие в одно дерево поиска. Более того, то же самое было бы, если бы обработка y не была закончена до начала обработки v.

Следовательно, вершина y уже была обработана к тому моменту, когда была начата обработка вершины v. По предположению индукции, y и w попали в одно и то же дерево поиска, но это противоречит тому, что обработка w еще не была начата, когда начала обрабатываться вершина v! □

Теорема 5.1. *Описанная выше процедура корректна.*

Доказательство (в котором мы пользуемся леммой 5.4). 1) Пусть две вершины v и w связаны в обе стороны, но не попали в одно дерево поиска графа G^t. Пусть дерево, содержащее вершину v, возникло раньше. Но тогда w обязана была попасть в это дерево.

2) Пусть пути из v в w нет, но они попали в одно дерево. Достаточно показать, что имеется путь из корня r этого дерева в обе вершины v и w (пока мы только знаем, что есть обратные пути).

Пусть p — последняя вершина на пути из v в r в дереве поиска для графа G^t, которая не лежит в той же компоненте, что и r; а q — следующая за p на этом пути. Нам достаточно показать наличие пути из q в p.

Поскольку $e[r] > e[p],\ e[q]$ в дереве поиска для исходного графа G, имеется два варианта.

(a) Вершина q лежит в поддереве с корнем r. Если p лежит на пути из r в q в этом поддереве, то получаем требуемое утверждение. Если p была обработана раньше q, то также существует путь из q в p в этом дереве. Остается случай, когда p начала обрабатываться позже q (и позже r!). Но, поскольку ее обработка закончилась все же раньше r, то она попала в поддерево с корнем r.

(б) Вершина q была полностью обработана до начала обработки r. Если бы p была обработана раньше q, но по ребру (p, q) мы не пошли, то существовал бы путь из q в p. Значит, p — родитель q. Однако почему вершина r не попала в поддерево поиска с корнем q? Значит, на пути из q в r есть вершина x, которая уже начала обрабатываться до того, как q была обработана, — т.е. есть путь из q в p через x.

□

5.2.3 Упражнения

Упражнение 5.1 (Полусвязность). Ориентированный граф называется полусвязным, если для любых двух его вершин v и w в нем имеется (ориентированный) путь либо из v в w, либо из w в v. Выяснить, является ли данный граф полусвязным. □

Упражнение 5.2 (Мосты и точки раздела). Дан неориентированный связный граф. Точкой раздела называется вершина, при удалении которой граф теряет связность. Мостом называется ребро, при удалении которого граф теряет связность. Найти в данном графе все мосты и точки раздела. □

5.3 Построение минимального остовного дерева

Остовное дерево (неориентированного графа) — это дерево, содержащее все вершины графа и некоторые из его ребер.

Для данного графа с весами построим остовное дерево с минимальным суммарным весом ребер. Будем «растить» дерево постепенно. В процессе построения будет получаться некоторый лес, являющийся подграфом будущего остовного дерева. На каждом шаге будем добавлять к этому лесу ребро минимального возможного веса, соединяющее некоторую компоненту связности C с вершиной *другой* компоненты (минимум берется по всем ребрам, исходящим из компоненты C; выбор компоненты C — произволен).

Лемма 5.5. *Такой алгоритм корректен независимо от способа выбора очередной компоненты (и ребра среди ребер одинакового веса).*

Доказательство (индукция по построению дерева). Пусть, имея лес F, мы добавили к нему ребро (u, v), минимальное по весу из ребер, соединяющих некоторую компоненту U с другими компонентами леса F. Считая (по предположению индукции), что существует минимальное остовное дерево T исходного графа, содержащее лес F, покажем, что существует и минимальное остовное дерево, содержащее $F \cup \{(u, v)\}$.

Итак, пусть T не содержит (u, v). Тем не менее, в T есть путь γ из u в v (не содержащий ребра (u, v)). Рассмотрим первое ребро e на пути γ, выводящее за пределы компоненты U. Наш алгоритм устроен так, что вес ребра e — не меньше веса ребра (u, v). Следовательно, $T \setminus \{e\} \cup \{(u, v)\}$ — искомое минимальное остовное дерево. □

Имеется несколько возможных реализаций этого алгоритма.

Алгоритм Борувки. На очередном шаге выбираем для каждой вершины (одновременно) ребро минимального веса, исходящее из нее. Затем *стягиваем каждую из получающихся из этих ребер компонент связности в одну точку.*

Алгоритм Крускала. Рассматриваем все ребра в порядке, соответствующем весу; очередное ребро добавляем к лесу, если оно соединяет разные компоненты связности.

Алгоритм Прима. Конструируемый нами лес состоит из дерева и независимых вершин. Выбираем ребро минимального веса среди ребер, соединяющих вершины нашего дерева с остальными вершинами.

Легко видеть, что любой из этих методов можно реализовать за $O(|E| \log |V|)$ операций с вершинами *и весами.* Например, в первом из этих методов на каждом шаге количество вершин уменьшается, как минимум, вдвое; каждый шаг можно реализовать за $O(|E|)$ операций с вершинами и ребрами: поиск минимального элемента занимает линейное время, слияние списков, содержащих $|E|$ элементов, — время $O(|E|)$; при стягивании ребра, ведущие в вершины той же компоненты, можно не удалять, а считать бóльшими по весу.