Минимально остовные деревья. CHM.

Сапожников Денис

Contents

1 Определения

Определение (Граф)**.** Граф $G(V, E)$ — это множество вершин V и множество рёбер E — множество пар вершин $(a, b) : a, b \in V$, которые "соединены".

Определение (Подграф)**.** Подграф $G'(V, E') \subset G(V, E)$ — это граф на том же множестве вершин V, и подмножестве рёбер $E' \subseteq E$.

Везде ниже мы будем считать, что $|V| = n, |E| = m$.

Определение (Петля)**.** Ребро $(a, b) \in E$ называется петлёй, если оно соединяет вершину с самой собой, т.е. $a = b$.

Определение (Кратное ребро)**.** Два ребра называются кратными, если они соединяют две одинаковые пары рёбер.

Как правило в задачах указано, что нет петель и кратных рёбер, но если такое не написано, то с ними стоит быть осторожнее!

Определение (Взвешенный граф)**.** Граф называется взвешенным, если есть функция $w : E \to \mathbb{R}$, называемая весом ребра; иначе говоря, каждому ребру сопоставляется вещественное число.

Определение (Дерево)**.** Дерево — это связный граф на n вершинах и $n - 1$ ребре.

2 Задача

Остовное дерево - это подграф, являющийся деревом.
Минимальное остовное дерево (MST) - это остовное дерево минимальное по весу (то есть с минимальной суммой весов рёбер).
Разрезом графа G называется разбиение V на два непересекающихся множества S и T, который дополняют друг друга до V, то есть $S \cup T = V, S \cap T = \varnothing$, и обозначается $\langle S, T \rangle$
Ребро **пересекает разрез** $\langle S, T \rangle$, если один его конец лежит в S, а другой в T.

А, собственно, задача, которую мы будем решать - это нахождение минимально остовного дерева за $O(n^2)$, $O(m \log m)$.

3 Лемма о безопасном ребре

Пусть G' - это подграф некоторого минимального остовного графа $G = (V, E)$

Ребро $(a, b) \notin G'$ называется **безопасным**, если при его добавлении в G', $G' \cup \{(a, b)\}$ тоже является подграфом некоторого минимального остовного дерева исходного графа G.

Лемма (о безопасном ребре)**.** *Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$. Пусть $G' = (V, E')$ — подграф некоторого минимального остовного дерева G, $\langle S, T\rangle$ — разрез G, такой, что ни одно ребро из E' не пересекает разрез, а $e - (u, v)$ — ребро минимального веса среди всех ребер, пересекающих разрез $\langle S, T\rangle$. Тогда ребро $e = (u, v)$ является безопасным для G'.*

Proof. Достроим E' до некоторого минимального остовного дерева, обозначим его T_{min}. Если ребро $e \in T_{min}$, то лемма доказана, поэтому рассмотрим случай, когда ребро $e \notin T_{min}$. Рассмотрим путь в T_{min} от вершины u до вершины v. Так как эти вершины принадлежат разным долям разреза, то хотя бы одно ребро пути пересекает разрез, назовём его e'. По условию леммы $w(e) \leq w(e')$. Заменим ребро e' в T_{min} на ребро e. Полученное дерево также является минимальным остовным деревом графа G, поскольку все вершины G по-прежнему связаны и вес дерева не увеличился. Следовательно $E' \cup \{e\}$ можно дополнить до минимального остовного дерева в графе G, то есть ребро e — безопасное. □

4 Алгоритм Прима

4.1 Идея

Давайте последовательно строить минимально остовное дерево T графа G. Рассмотрим минимальное ребро (a, b) из T в $G \setminus T$. По лемме о безопасном ребре, оно принадлежит MST. Таким образом мы добавили ещё одну вершину b в T.

4.2 Алгоритм за $O(n^2)$

Будем для каждой вершины не из MST поддерживать вес минимального ребра, которое ведёт в MST и ∞, если такого нет, а при каждом добавлении новой вершины за $O(n)$ пересчитывать эту информацию. Это будем хранить в массиве min_e, а то, куда ведёт минимальное ребро — в массиве sel_e.

```
const int INF = 1e9; // infinity
int g[n][n]; // if there is no edge, then g[v][u] = INF
vector<int> min_e(n, INF), sel_e(n, -1), used(n);
min_e[0] = 0;
for (int i = 0; i < n; ++i) {
  int v = -1;
  for (int j = 0; j < n; ++j)
    if (!used[j] && (v == -1 || min_e[j] < min_e[v]))
      v = j;
  used[v] = true;
  if (sel_e[v] != -1)
    cout << v << " " << sel_e[v] << '\n';
  for (int u = 0; u < n; u++)
    if (g[v][u] < min_e[u]) {
      min_e[u] = g[v][u];
      sel_e[u] = v;
    }
}
```

4.3 Алгоритм за $O(m \log n)$

Но это на самом деле прошлый алгоритм работает долго на разреженных графах, на них можно находить миностов быстрее.

Будем поддерживать отсортированный набор «торчащих рёбер» ИЗ T. Тогда каждый раз нам нужно находить минимум из этого набора и обновлять его. С этим может справиться стандартные set или $priority\_queue$. При помощи такой реализации алгоритм будет работать за $O(m \log n)$.

```
vector<vector<pair<int, int>>> gr;
const int INF = 1e9;

vector<int> min_e (n, INF), sel_e (n, -1), used(n, false);
min_e[0] = 0;
set<pair<int, int>> q;
q.insert({ 0, 0 });
for (int i = 0; i < n; ++i) {
  if (q.empty()) {
    cout << "No MST!";
```

```
    return 0;
  }
  int v = q.begin()->second;
  q.erase(q.begin());

  used[v] = true;
  if (sel_e[v] != -1)
    cout << v << " " << sel_e[v] << '\n';

  for (auto [u, w] : gr[v]) {
    if (!used[u] && w < min_e[u]) {
      q.erase({ min_e[u], u });
      min_e[u] = w;
      sel_e[u] = v;
      q.insert({ min_e[u], u });
    }
  }
}
```

5 СНМ

Пусть мы хотим (потом поймёте, зачем) придумать структуру, которая умеет в 2 операции:

- Объединить 2 непересекающихся множества
- Узнать для двух элементов, они находятся в разных множествах или в одном

Оказывается, такая структура существует, она называется система непересекающихся множеств (СНМ) и время работы каждой операции равно $O(\alpha(n))$, где $\alpha(n)$ — обратная Аккермана, которая на разумных ограничениях не превосходит 4, обычно такую «почти константу» опускают в оценке времени работы и считают за $O(1)$, что, формально, не верно.

Будем хранить множество в виде корневого дерева, у каждого элемента будет его предок, корень дерева назовём **представителем** множества. Тогда 2 элемента находятся в одном множестве, если их представители равны.

Чтобы объединить 2 множества нужно подвесить корень одного дерева к корню другого. Если делать это в тупую, то поиск представителя может быть очень долгой операцией и работать за $O(n)$.

5.1 Эвристика сжатия путей

5.1.1 Алгоритм

Каждый раз, когда нам нужно найти представителя, найдём его в тупую, идя до корня. После этого пройдемся по всему пути и переподвесим все вершины к корню.

```
int p[N];
void init(int n) {
  for (int i = 0; i < n; i++)
    p[i] = i;
}
int find_p(int v) {
  if (p[v] == v)
    return v;
  else
    return p[v] = find_p(p[v]);
}
void union_set(int a, int b) {
  a = find_p(a);
  b = find_p(b);
  if (a == b)
    return;
  p[a] = b;
}
```

Оказывается, если так делать, то все операции начинают работать за $O(\log n)$ в среднем.

Пусть для удобства вершина - корень, если её предок - это она сама.

5.1.2 Доказательство времени работы

Покажем сначала O-оценку.

Назовём весом вершины w_v размер поддерева v. Размахом ребра (v, u) назовём величину $|w_v - w_u|$. Будем говорить, что ребро имеет класс k, если его размах лежит в полуинтервале $[2^k; 2^{k+1})$.

Сделаем некоторые замечания из работы функции нахождения предка и переподвешивания:

- Размах любого ребра может только увеличиваться, так как либо ребро было удалено при переподвешивании, либо оно добавилось, либо размер предка увеличился, а размер сына уменьшился.
- Класс ребра $\in [0; \log n]$.

Рассмотрим путь, вдоль которого мы прошли до корня при вызове $find\_parent$. Заметим, что если было 2 ребра класса k, то они превратятся в рёбра класса хотя бы $k+1$ (на самом деле почти всегда большего класса, но этой оценки нам будет достаточно). Таким образом за один вызов этой функции $\log n$ рёбер могут остаться без изменений, а у остальных класс увеличится хотя бы на один, но так как класс может быть числом от 0 до $\log n$, то суммарно класс каждого ребра увеличится не больше чем $n \log n$ раз, то есть амортизированно $\log n$.

Контрпример для Ω-оценки. Построим биномиальное дерево размера $\frac{n}{2}$, у него будет $\frac{n}{4}$ листьев. Будем по очереди вызывать переподвешивание от каждого из листьев к очередной вершине без рёбер. Тогда $find\_parent$ будет работать за хотя бы $\log n - 1$, а значит после $\frac{n}{4}$ запросов мы проделаем хотя бы $\frac{n}{4}(\log n - 1) = O(n \log n)$ действий.

5.2 Ранговая эвристика

Рангом дерева назовём его глубину. Тогда если подвешивать дерево меньшего ранга к дереву большего, то глубина дерева будет не больше $O(\log n)$, а значит и поиск представителя работает за такое же время. Тогда изначальный ранг каждого множества равен 0.

```
int r[N], p[N];
int find_p(int v) {
  return p[v] == v ? v : find_p(p[v]);
}
void union_set(int a, int b) {
  a = find_p(a), b = find_p(b);
  if (a == b) return;
  if (r[a] > r[b]) swap(a, b);
  if (r[a] == r[b]) r[b]++;
  p[a] = b;
}
```

5.3 Эвристика small-to-large

5.3.1 Алгоритм

Об этой эвристике обычно не говорят при разговоре про СНМ, но эта техника часто помогает в других задачах. Давайте переподвешивать меньшее дерево к большему, в таком случае это тоже будет работать за $O(\log n)$ в среднем, а глубина такого дерева будет не больше $\log n$.

```
int sz[N], p[N];
int find_p(int v) {
  return p[v] == v ? v : find_p(p[v]);
}
void union_set(int a, int b) {
  a = find_p(a), b = find_p(b);
  if (a == b) return;
  if (r[a] > r[b]) swap(a, b);
  sz[b] += sz[a];
  p[a] = b;
}
```

5.3.2 Доказательство времени работы

Предположим противное: какую-то вершину мы перелили (то есть переподвесили, когда переподвешивали все поддерево) больше $\log n$ раз. Но такое происходит только в том случае, если в оставшейся части было больше вершин. Тогда после 1-го переподвешивания у нас хотя бы 2 вершины суммарно, после 2-го — 4 и т.д., после $(\log n + 1)$-го хотя бы $2n$, противоречие.

5.4 Объединяем эвристики

Внезапно если объединить ранговую эвристику и эвристику сжатия путей, то алгоритм будет работать за $O(\alpha(n))$. Это мы, конечно же, доказывать не будем.

```
int r[N], p[N];
int find_p(int v) {
  return p[v] == v ? v : p[v] = find_p(p[v]);
}
void union_set(int a, int b) {
  a = find_p(a), b = find_p(b);
  if (a == b)
```

```
8       return;
9     if (r[a] > r[b])
10      swap(a, b);
11    if (r[a] == r[b])
12      r[b]++;
13    p[a] = b;
14  }
```

5.5 Рандомная эвристика

НЕТ, НЕ НАДО, ОНА НЕ РАБОТАЕТ!!!

Иногда у людей, которых пихают рандом везде где можно возникает идея: а давайте случайно подвешивать одно множество к другому, может это тоже работает за $O(\log n)$?

Конечно же нет! Тест очень простой: сделаем простой путь из $\frac{n}{2}$ вершин, и из каждой вершины сделаем добавим одно ребро. Тогда с вероятностью $\frac{1}{2}$ вы подвесите за 1, а с вероятностью $\frac{1}{2}$ за t_{n-1}. То есть если обозначит за t_n глубину дерева, то можно написать её матожидание: $t_n = \frac{1}{2} + \frac{1}{2}t_{n-1}$. По индукции легко показать, что $t_n = \frac{n}{2}$. Проблемы!

6 Алгоритм Крускала

Зачем вообще нужна была структура CHM? С помощью неё очень удобно поддерживать множество вершин, которые находятся в одной компоненте связности. Если мы хотим добавить ребро, которое соединяет 2 вершины из разных компонент, то их можно легко объединить.

Отсортируем все рёбра по возрастанию веса. Пусть есть несколько компонент связности — частей нашего минимального остова. Тогда рассмотрим минимальное по весу ребро. Если оно соединяет 2 вершины из разных компонент, то по лемме о безопасном ребре оно входит в MST, добавим его и соединим компоненты. Если ребро соединяет 2 вершины из одной компоненты, то перейдём к следующему по весу ребру и т.д.

Таким образом мы построим MST за время $O(m \log m)$ на сортировку и $O(m\alpha(m))$ на запросы, итого $O(m \log m)$.

```
vector<pair<int, pair<int, int>>> edges(m); // { weight, { u, v } }
//*read and initialization*
sort(edges.begin(), edges.end());
for (auto [w, e] : edges)
  if (find_p(e.first) != find_p(e.second)) {
    cout << e.first << ' ' << e.second << '\n';
    union_set(e.first, e.second);
  }
```

А теперь попробуйте сами придумать, как, используя лемму о безопасном ребре, доказать корректность алгоритма.

7 Алгоритм Борувки

Об этом алгоритме редко говорят, но тем не менее, он есть и идейно он очень простой и красивый. А ещё именно он используется на практике, потому что его можно распаралеллить.

Алгоритм состоит из 3 шагов:

1. Изначально каждая вершина графа G — тривиальное дерево (то есть состоит из одной вершины), а ребра не принадлежат никакому тривиальному дереву. То есть у нас есть лес (набор деревьев).

2. Для каждого дерева T из леса найдем минимальное инцидентное ему ребро. Добавим все такие ребра.

3. Повторяем шаг 2 пока в графе не останется только одно дерево T.

В алгоритме так же стоит быть аккуратными с рёбрами равного веса (например, в треугольнике с рёбрами по 1).

Теорема. *Алгоритм находит MST.*

Proof. Очевидно, что в результате работы алгоритма получается дерево. Пусть T — минимальное остовное дерево графа G, а T' — дерево полученное после работы алгоритма.

Покажем, что $w(T) = w(T')$.

Предположим обратное $w(T) > w(T')$. Пусть ребро e' — первое добавленное ребро дерева T', не принадлежащее дереву T. Пусть P — путь, соединяющий в дереве T вершины ребра e'.

Понятно, что в момент, когда ребро e' добавляли, какое-то ребро P (назовем его e) не было добавлено. По алгоритму $w(e) \geq w(e')$. Однако тогда $T - e + e'$ — остовное дерево веса не превышающего вес дерева T. Следовательно $w(T) = w(T')$. □

Этот алгоритм делает $O(\log n)$ итераций, так как на каждой итерации в каждое дерево добавляется по новому ребру и, следовательно, деревьев становится хотя бы в 2 раза меньше. Каждая итерация работает за $O(m)$, и итоговая асимптотика $O(m \log n)$.

8 Свойства MST

И последнее, о чём стоит упомянуть – это свойства минимально остовных деревьев:

1. Максимальный остов также можно искать (например, заменив все веса рёбер на противоположные алгоритмы не требуют неотрицательности весов рёбер).
2. Минимальный остов единственен, если веса всех рёбер различны. В противном случае, может существовать несколько минимальных остовов
3. Минимальный остов также является остовом, минимальным по произведению всех рёбер (предполагается, что все веса положительны). В самом деле, если мы заменим веса всех рёбер на их логарифмы, то легко заметить, что в работе алгоритма ничего не изменится, и будут найдены те же самые рёбра.
4. Минимальный остов является остовом с минимальным весом самого тяжёлого ребра. Яснее всего это утверждение понятно, если рассмотреть работу алгоритма Крускала.
5. Критерий минимальности остова: остов является минимальным тогда и только тогда, когда для любого ребра, не принадлежащего остову, цикл, образуемый этим ребром при добавлении к остову, не содержит рёбер тяжелее этого ребра. В самом деле, если для какого-то ребра оказалось, что оно легче некоторых рёбер образуемого цикла, то можно получить остов с меньшим весом (добавив это ребро в остов, и удалив самое тяжелое ребро из цикла). Если же это условие не выполнилось ни для одного ребра, то все эти рёбра не улучшают вес остова при их добавлении.

Ваша задача понять, почему свойства верные!

9 Ссылки

1. Алгоритм Прима (осторожно, Прим за $O(m \log n)$ не рабочий)
2. Алгоритм Крускала
3. СНМ
4. Алгоритм Борувки