Остовные деревья

1 Введение

Остовным деревом называется такой ациклический связный подграф связного неориентированного графа, который включает все вершины исходного графа. Сразу отметим, что любое остовное дерево содержит $N-1$ ребро, где N – количество вершин в дереве. Если изначально граф не связный, то вводят понятие остовного леса. Мы для каждой компоненты связности можем найти свое остовное дерево. Тогда **остовный лес** – это множество остовных деревьев для всех компонент связности исходного графа. Количество ребер в остовном лесе будет меньше $N - 1$.

Когда речь идет о взвешенных графах, используют понятие **минимального остовного дерева**. Это такое остовное дерево, сумма весов ребер которого минимальна. Аналогичном образом можно ввести понятие **минимального остовного леса**.

Сразу отметим, что петли и кратные ребра не играют роли построении обычных остовных деревьев, не говоря уже о минимальных, так как образуют циклы. Петли вообще можно не запоминать при считывании. Из кратных ребер стоит оставлять только одно ребро, с наименьшим весом.

2 Алгоритм Прима

2.1 Описание алгоритма

Алгоритм Прима – один из алгоритмов построения минимального остовного дерева. Данный алгоритм очень похож на алгоритм Дейкстры и состоит из следующих шагов:

1. Сначала мы выделим множество S уже посещенных вершин.
2. Изначально в S только одна любая вершина.
3. Далее выполняем $N - 1$ итерацию алгоритма.
4. На каждой итерации выбираем такую вершину u, которая соединена с нашим остовом S ребром наименьшего веса.
5. Добавляем в S вершину u.
6. *Проводим этап релаксации.

Обратите внимание, что этап релаксации может отсутствовать в некоторых реализациях алгоритма. Когда пройдет $N - 1$ итерация, в остове окажется ровно $N - 1$ ребро.

Сразу разберем случай для несвязного графа. Запуск алгоритма из какой-либо вершины обойдет все достижимые из нее вершины. Т.е. за один запуск алгоритм построит минимальное остовное дерево для одной компоненты связности. Тогда, для построения минимального остовного леса необходимо запустить алгоритм для каждой компоненты связности.

2.2 Доказательство корректности

Теорема 1. *Построенный алгоритмом остов является минимальным.*

Доказательство. Обозначим найденный алгоритмом остов как T и минимальный остов как S. Если они не отличаются, то алгоритм нашел минимальный остов. Пусть они отличаются. Тогда найдем первый момент времени, когда в T попало ребро, которого нет в S. Обозначим это ребро за e, его концы – вершины a и b. Также за V обозначим множество уже выбранных в тот момент времени вершин, причем в нем лежит либо только a, либо только b. Т.к. S – остов, то в нем есть путь P из a в b. В P есть ребро g, у которого только один конец лежит V. Т.к. алгоритм выбрал ребро e вместо ребра g, то вес e меньше либо равен весу g. Добавим в S e и удалим g. Докажем, что S – все еще минимальный остов. Во-первых, добавив e, мы замкнули P в цикл. Удалив g мы разорвали цикл, при этом связность не могла нарушиться. Заменив g на e, вес остова точно не увеличился. Т.е. S остался минимальным остовом. Но и не мог уменьшится, так как S изначально был оптимальным. Таким образом мы построили минимальный остов, который содержит ребро e. Будем повторять шаги $2 - 6$, пока T и S отличаются. В результате мы получим минимальный остов S, который совпадет с T. Значит T тоже минимальный остов, ч.т.д. □

2.3 Тривиальная реализация

Разберем реализацию алгоритма для связного графа. Для несвязного графа вам не составит труда модифицировать алгоритм самим. Для примера, храним граф в виде списка смежности. Т.к. граф связный, то стартовать можно с любой вершины. За S обозначим множество посещенных вершин. На выходе мы получим вес минимального остовного дерева и ребра, попавшие в него.

```
if __name__ == "__main__":
    # a - adjacency list
    S = {0}
    edges = []
    weight = 0
```

```
for _ in range(n - 1):
    u = -1
    j = -1
    min_w = float("inf")
    for u in S:
        for v, w in a[u]:
            if v not in S and w < min_w:
                min_w = w
                i = u
                j = v
    edges.append((i, j, min_w))
    weight += min_w
    S.add(j)
print(weight)
print(*edges, sep="\n")
```

В данной реализации этап релаксации отсутствует. Оценим асимптотику. Внешний цикл работает за $O(N)$. Два внутренних цикла перебирают ребра исходящие из вершин множества S. Эти два цикла можно было просто заменить одним циклом, который перебирает все ребра. Такой цикл в реальности работал бы дольше, чем в примере выше, но оба варианта можно оценить, как $O(M)$. Итого, время работы $O(NM)$. В случае плотных графов асимптотика будет $O(N^3)$, что нас совсем не устраивает. Напомню, что граф называет **плотным**, если количество ребер в нем близко к максимальному

$$\binom{N}{2} = \frac{N(N-1)}{2} \approx O(N^2),$$

не считая петель и кратных ребер. Граф, с максимальным числом ребер называется **полным**.

Рассмотрим две оптимизации алгоритма: для плотных и разреженных графов.

2.4 Случай плотных графов

На самом деле оптимизации будут иметь одинаковую идею: мы для каждой вершины u не из S будем поддерживать кратчайшее ребро, соединяющее u с любой вершиной из S. После добавления очередной вершины мы будем пересчитывать эти ребра (этап релаксации). Однако, реализации, а значит и асимптотики, разные.

Пусть T – множество еще непосщененных вершин. В данном варианте мы просто на прямую будем перебирать все вершины из T. Так как, для каждой вершины мы уже знаем кратчайшее ребро, то перебор займет $O(N)$ времени. Итераций алгоритма по прежнему $N-1$. Кроме того, у наспоявился этап релаксации. На этом этапе нам надо рассмотреть только ребра, исходящие из только что добавленной вершины. Суммарно, за

все $N-1$ итераций, этап релаксации рассмотрит каждое ребро не более, чем один раз. Итого, время работы такой реализации $O(N^2+M)$. Поймем, что наш этап релаксации ничего не сломает. Мы всегда поддерживаем кратчайшее ребро для каждой вершины. Ребра из только что добавленной вершины u могут либо вести в другие вершины множества S (нас такие не интересуют), либо вести в множество T. Пусть есть ребро из u в v из T. Если ребро меньше того ребра, что мы поддерживали (отсутствие ребра мы считали, как бесконечно длинное ребро), то мы обновим ребро в нашем списке кратчайших ребер. Иначе мы просто пропустим данное ребро. Таким образом этап релаксации может только уменьшать ребра в нашем списке кратчайших ребер, что нам и надо.

```
if __name__ == "__main__":
    # a - adjacency list
    d = [(-1, -1, float("inf"))] * n  # list of shortest edges
    for v, w in a[0]:
        d[v] = (0, v, w)
    T = {i for i in range(1, n)}
    edges = []
    weight = 0
    for _ in range(n - 1):
        j = -1
        min_w = float("inf")
        for v in T:
            if d[v][2] < min_w:
                min_w = d[v][2]
                j = v
        edges.append(d[j])
        weight += min_w
        T.remove(j)
        for v, w in a[j]:
            if w < d[v][2]:
                d[v] = (j, v, w)
    print(weight)
    print(*edges, sep="\n")
```

2.5 Случай разреженных графов

Предыдущая реализация может быть использована и для разреженных графов, однако для этого случая есть оптимальная реализация. Идея с кратчашими ребрами остается, но реализация основывается на использовании кучи для быстрого выбора кратчайшего ребра. С использованием кучи мы это сделаем за $O(\log N)$. На этапе релаксации мы будем обновлять ребра также за $O(\log N)$. Так как итераций алгоритма $N-1$, то суммарно нам понадобится $O(N \log N)$ для выбора $N-1$ ребра в остов.

Весь этап релаксации теперь работает за $O(M \log N)$. Итого время работы $O((N+M) \log N)$. Учитывая, что порядок M не меньше, чем N, то время работы можно оценить, как $O(M \log N)$.

```
def sift_up(i):
    while i > 0 and heap[(i - 1) // 2][2] > heap[i][2]:
        v2h[heap[i][1]], v2h[heap[(i - 1) // 2][1]] = (i - 1) // 2, i
        heap[i], heap[(i - 1) // 2] = heap[(i - 1) // 2], heap[i]
        i = (i - 1) // 2


def sift_down(i):
    n = len(heap)
    while i * 2 + 1 < n:
        j = i
        if heap[i][2] > heap[i * 2 + 1][2]:
            j = i * 2 + 1
        if i * 2 + 2 < n and heap[j][2] > heap[i * 2 + 2][2]:
            j = i * 2 + 2
        if i == j:
            break
        v2h[heap[i][1]], v2h[heap[j][1]] = j, i
        heap[i], heap[j] = heap[j], heap[i]
        i = j


def extract_min():
    x = heap[0]
    v2h[x[1]] = -1
    tmp = heap.pop()
    if heap:
        heap[0] = tmp
        v2h[heap[0][1]] = 0
        sift_down(0)
    return x


def heapify():
    from math import ceil
    for i in range(ceil(len(heap) / 2), -1, -1):
        sift_down(i)


if __name__ == "__main__":
    # a - adjacency list
    heap = [(i, i, float("inf")) for i in range(1, n)]
```

```
v2h = [-1] + [i-1 for i in range(1, n)]
for v, w in a[0]:
    heap[v2h[v]] = (0, v, w)
heapify()
edges = []
weight = 0
for _ in range(n - 1):
    u, j, w = extract_min()
    edges.append((u, j, w))
    weight += w
    for v, w in a[j]:
        if v2h[v] != -1 and w < heap[v2h[v]][2]:
            heap[v2h[v]] = (j, v, w)
            sift_up(v2h[v])
print(weight)
print(*edges, sep="\n")
```

3 Алгоритм Краскала

3.1 Описание алгоритма

Алгоритм Краскала является типичным жадным алгоритмом. Он состоит из следующих шагов:

1. Сначала отсортируем ребра в порядке возрастания их весов.
2. Каждая вершина изначально находится в своем множестве.
3. На каждом шаге мы выбираем ребро наименьшего веса, концы которого находятся в разных множествах.
4. Объединяем эти множества.

В конце мы получим $N-1$ ребро, которые объединят все вершины в одно множество. Данные ребра будут образовывать минимальный остов. Доказательство этого алгоритма тут не приводится, т.к. это потребует введение ряда определений и теорем.

Особенность этого алгоритма в том, что он не требует модификации для построения минимального остовного леса. Так как на каждой итерации алгоритм просто объединяет два дерева, то в случае несвязного графа просто останется несколько необъединенных деревьев. Каждое дерево будет минимальным остовом в своей компоненте связности.

3.2 Наивная реализация

При реализации алгоритма Краскала удобней всего использовать список ребер, так как нам придется сортировать ребра. Для каждой верши-

ны я поддерживаю цвет, изначально все цвета уникальны. При объединении мы просто перекрашиваем одно множество в цвет другого.

```
def get_key(x):
    return x[2]


if __name__ == "__main__":
    # a - edge list
    a.sort(key=get_key)
    tree_id = [i for i in range(n)]
    edges = []
    weight = 0
    for u, v, w in a:
        if tree_id[u] != tree_id[v]:
            weight += w
            edges.append((u, v, w))
            j = tree_id[v]
            for i in range(n):
                if tree_id[i] == j:
                    tree_id[i] = tree_id[u]
    print(weight)
    print(*edges, sep="\n")
```

Оценим асимптотику. Внешний цикл проходит по всем ребрам, что требует $O(M)$ времени. Однако мы выполним операцию объединения ровно $N - 1$ раз. Само объединение работает за $O(N)$. И не забываем про сортировку. Итого, время работы алгоритма $O(M \log M + N^2)$, что достаточно долго.

3.3 С использованием СНМ

Как вы могли заметить, операция объединения достаточно долгая. Ключ к оптимизации алгоритма лежит в оптимизации объединения. Для этого можно использовать **систему непересекающихся множеств** (СНМ) с эвристиками. Здесь не будет приводиться описание данной структуры. Основное, что надо помнить, это то, что время работы операций в такой структуре является почти константным и может быть опущено. Используя СНМ, мы сможем объединять два множество за почти константное время. Время, необходимое на проверку, что две вершины принадлежат разным множествам, увеличится с $O(1)$ до почти константного, но как было сказано ранее, мы это не учитываем. Итого, время работы будет $O(M \log M + N) = O(M \log M)$.

```
def get_key(x):
    return x[2]
```

```python
def make_set(x):
    return {
        "parent": x,
        "rank": 0
    }


def find_set(x):
    if dsu[x]["parent"] == x:
        return x
    dsu[x]["parent"] = find_set(dsu[x]["parent"])
    return dsu[x]["parent"]


def union_sets(x, y):
    x = find_set(x)
    y = find_set(y)
    if x != y:
        if dsu[x]["rank"] < dsu[y]["rank"]:
            x, y = y, x
        dsu[y]["parent"] = x
        if dsu[x]["rank"] == dsu[y]["rank"]:
            dsu[x]["rank"] += 1


if __name__ == "__main__":
    # a - edge list
    a.sort(key=get_key)
    dsu = [make_set(i) for i in range(n)]
    edges = []
    weight = 0
    for u, v, w in a:
        if find_set(u) != find_set(v):
            weight += w
            edges.append((u, v, w))
            union_sets(u, v)
    print(weight)
    print(*edges, sep="\n")
```