

Лекция 18. Минимальные оставные деревья (MST): Крускал, Прим.

1. Введение

Минимальное оставное дерево (MST) – это подграф связного взвешенного графа, который:

- Связывает все вершины графа.
- Является деревом (нет циклов).
- Имеет минимальный суммарный вес рёбер.

Примеры применения:

- Проектирование сетей (компьютерных, транспортных).
- Кластеризация данных.
- Приближённые алгоритмы для задачи коммивояжёра.

Основные алгоритмы:

- **Алгоритм Крускала** – основан на сортировке рёбер.
- **Алгоритм Прима** – основан на постепенном добавлении вершин.

2. Алгоритм Крускала

Идея:

1. Отсортировать все рёбра по весу.
2. Последовательно добавлять рёбра с минимальным весом, если они не образуют цикла.

Реализация:

```
from typing import List, Tuple
```

```
class DSU:  
    """Система непересекающихся множеств (Disjoint Set Union)."""  
  
    def __init__(self, n: int):  
        # Изначально каждая вершина - отдельное множество  
        self.parent = list(range(n)) # parent[i] = родитель i-й вершины  
        self.rank = [1] * n # rank[i] = высота дерева с корнем в i  
  
    def find(self, x: int) -> int:  
        """Находит корень множества, содержащего x, с применением сжатия путей."""  
        if self.parent[x] != x:  
            # Рекурсивно поднимаемся к корню и сжимаем путь  
            self.parent[x] = self.find(self.parent[x])  
        return self.parent[x]  
  
    def union(self, x: int, y: int) -> bool:  
        """Объединяет множества, содержащие x и y. Возвращает True, если объединение произошло."""  
        xr, yr = self.find(x), self.find(y)
```

```

# Если корни одинаковые - вершины уже в одном множестве (образуют
цикл)
if xr == yr:
    return False

# Объединяем по рангу (меньшее дерево присоединяется к большему)
if self.rank[xr] < self.rank[yr]:
    self.parent[xr] = yr
elif self.rank[xr] > self.rank[yr]:
    self.parent[yr] = xr
else:
    # Если ранги равны, выбираем произвольно и увеличиваем ранг
    self.parent[yr] = xr
    self.rank[xr] += 1
return True

def kruskal(n: int, edges: List[Tuple[int, int, int]]) -> List[Tuple[int,
int, int]]:
    """
    Находит MST алгоритмом Крускала.
    :param n: количество вершин
    :param edges: список рёбер в формате (вес, u, v)
    :return: список рёбер MST
    """
    # Сортируем рёбра по весу (от меньшего к большему)
    edges.sort()

    # Создаём DSU для отслеживания компонент связности
    dsu = DSU(n)
    mst = [] # Сюда будем складывать рёбра минимального остовного дерева

    # Перебираем все рёбра в порядке возрастания веса
    for w, u, v in edges:
        # Пытаемся объединить компоненты вершин u и v
        # Если объединение успешно - ребро не создаёт цикл, добавляем его в
MST
        if dsu.union(u, v):
            mst.append((w, u, v))

    return mst

# Пример использования
n = 4
edges = [
    (1, 0, 1), # Вес 1, соединяет вершины 0 и 1
    (2, 1, 2), # Вес 2, соединяет вершины 1 и 2
    (3, 2, 3), # Вес 3, соединяет вершины 2 и 3
    (4, 0, 2), # Вес 4, соединяет вершины 0 и 2
    (5, 1, 3) # Вес 5, соединяет вершины 1 и 3
]
mst = kruskal(n, edges)
print("MST (Крускал):", mst)

```

```
# Вывод: [(1, 0, 1), (2, 1, 2), (3, 2, 3)]
# Это означает: берём ребра 0-1 (вес 1), 1-2 (вес 2), 2-3 (вес 3)
Сложность: O(m log m) из-за сортировки ребер.
```

3. Алгоритм Прима

Идея:

1. Начать с произвольной вершины.
2. На каждом шаге добавлять ребро минимального веса, соединяющее текущее дерево с новой вершиной.

Реализация:

```
import heapq
from collections import defaultdict
from typing import List, Tuple

def prim(n: int, graph: List[List[Tuple[int, int]]]) -> List[Tuple[int, int, int]]:
    """
    Находит MST алгоритмом Прима.
    :param n: количество вершин
    :param graph: список смежности: graph[u] = [(v, вес), ...]
    :return: список ребер MST
    """

    visited = [False] * n # Отслеживаем посещённые вершины
    min_heap = [] # Минимальная куча для хранения ребер в формате (вес, u,
    v)
    mst = [] # Сюда складываем ребра MST

    # Начинаем с вершины 0
    visited[0] = True

    # Добавляем все ребра из вершины 0 в кучу
    for v, w in graph[0]:
        heapq.heappush(min_heap, (w, 0, v))

    # Пока куча не пуста и не набрали достаточно ребер для MST
    while min_heap and len(mst) < n - 1:
        # Извлекаем ребро с минимальным весом
        w, u, v = heapq.heappop(min_heap)

        # Если вершина v уже посещена, пропускаем это ребро
        if visited[v]:
            continue

        # Помечаем вершину v как посещённую
        visited[v] = True
        # Добавляем ребро в MST
        mst.append((w, u, v))
```

```

# Добавляем все рёбра из новой вершины v в кучу
for neighbor, weight in graph[v]:
    if not visited[neighbor]:
        heapq.heappush(min_heap, (weight, v, neighbor))

return mst

# Пример использования
n = 4
# Граф представлен в виде списка смежности
graph = [
    [(1, 1), (2, 4)], # Вершина 0: соединена с 1 (вес 1) и с 2 (вес 4)
    [(0, 1), (2, 2), (3, 5)], # Вершина 1: соединена с 0, 2, 3
    [(0, 4), (1, 2), (3, 3)], # Вершина 2: соединена с 0, 1, 3
    [(1, 5), (2, 3)] # Вершина 3: соединена с 1, 2
]
mst = prim(n, graph)
print("MST (Прим):", mst)
# Вывод: [(1, 0, 1), (2, 1, 2), (3, 2, 3)]

```

Сложность: $O(m \log n)$ при использовании кучи.

4. Сравнение алгоритмов

Алгоритм	Сложность	Применение
Крускал	$O(m \log m)$	Лучше для разреженных графов (мало рёбер)
Прим	$O(m \log n)$	Лучше для плотных графов (много рёбер)

Ключевые различия:

- **Крускал** проще в реализации, но требует сортировки всех рёбер.
- **Прим** эффективнее на плотных графах, но требует использования кучи.