Сириус
IT-Колледж

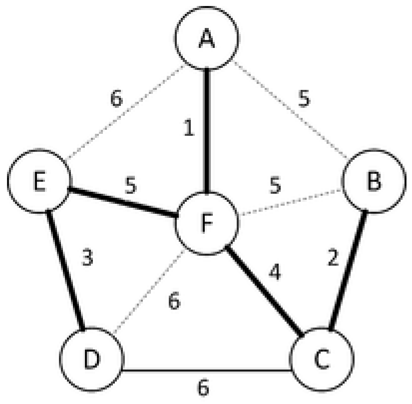
Лекция 9. Минимальное покрывающее дерево

Свойство разреза, жадная стратегия, алгоритм Прима, алгоритм Краскала. Система непересекающихся множеств. Представление множеств с помощью деревьев, две эвристики.

Минимальное покрывающее дерево

Остовное дерево (spanning tree) графа $G = (V, E)$ — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины.

Минимальное остовное дерево (или минимальное покрывающее дерево) в (неориентированном) связном взвешенном графе — это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.



Минимальное покрывающее дерево

Пусть G' — подграф некоторого минимального остовного дерева графа $G = (V, E)$.

Ребро $(u, v) \notin G'$ называется **безопасным (англ. safe edge)**, если при добавлении его в G', $G' \cup (u, v)$ также является подграфом некоторого минимального остовного дерева графа G.

Разрезом (англ. cut) неориентированного графа $G = (V, E)$ называется разбиение V на два непересекающихся подмножества: S и $T = V \setminus S$. Обозначается как $\langle S, T \rangle$.

Ребро $(u, v) \in E$ **пересекает (англ. crosses)** разрез $\langle S, T \rangle$, если один из его концов принадлежит множеству S, а другой — множеству T.

Минимальное покрывающее дерево. Свойство разреза

Теорема: Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$ с весовой функцией $w : E \to R$. 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 (u, v) — ребро минимального веса среди всех ребер, пересекающих разрез $\langle S, T \rangle$. Тогда ребро $e = (u, v)$ является безопасным для G'.

Доказательство: 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, назовем его e'. По условию леммы $w(e) \leqslant w(e')$. 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.

Минимальное покрывающее дерево. Жадная стратегия

Минимум остовных деревьев графа $G = (V, E, W)$ может быть найден применяя процедуру исследования ребер в порядке возрастания их весов. Другими словами, на каждом шаге выбирается новое ребро с наименьшим весом, не образующее циклов с уже выбранными ребрами. Процесс продолжается до тех пор, пока не будет выбрано $|V| - 1$ ребро. Такая процедура называется *жадным алгоритмом*.

Минимальное покрывающее дерево. Жадная стратегия

Для каждой вершины v графа $G = (V, E, W)$ формируются начальные тривиальные компоненты связности $T_i = (V_i, E_i, W)$, где $X_i = x_i, E_i = \varnothing, X_i \cap X_j = \varnothing, i \neq j, X = \bigcup_{i=1}^{|X|} X_i, i = 1, 2, .., |X|$. Компоненты T_i являются деревьями, объединение $T = \bigcup_i T_i$ которых дает начальное приближение строящегося остовного дерева $G_0 = (V, E_0, W)$.

Включение в строящееся остовное дерево G_0 выбранного ребера на очередном шаге *жадного алгоритма* выполняется слиянием $T_i = T_i \cup T_j$ ($V_i = V_i \cup V_j$ и $E_i = E_i \cup E_j$) двух компоненты T_i и T_j, которым принадлежит по вершине нового ребра, и включением самого ребра в объединенное множество $E_i = E_i \cup E_j$ ребер. Процесс роста объединения $T = \bigcup_i T_i$ компоненты к остовному дереву $G_0 = (V, E_0, W)$ продолжается до тех пор, пока не будет включено $|V| - 1$ ребро.

Минимальное покрывающее дерево. Алгоритм Прима

Алгоритм Прима — алгоритм поиска минимального остовного дерева во взвешенном неориентированном связном графе.

Данный алгоритм очень похож на алгоритм Дейкстры. Будем последовательно строить поддерево F ответа в графе G, поддерживая приоритетную очередь Q из вершин $G \setminus F$, в которой ключом для вершины v является $\min\limits_{u \in V(F), uv \in E(G)} w(uv)$ — вес минимального ребра из вершин F в вершины $G \setminus F$. Также для каждой вершины в очереди будем хранить $p(v)$ — вершину u, на которой достигается минимум в определении ключа. Дерево F поддерживается неявно, и его ребра — это пары $(v, p(v))$, где $v \in G \setminus \{r\} \setminus Q$, а r — корень F. Изначально F пусто и значения ключей у всех вершин равны $+\infty$. Выберём произвольную вершину r и присвоим её ключу значение 0. На каждом шаге будем извлекать минимальную вершину v из приоритетной очереди и релаксировать все ребра vu, такие что $u \in Q$, выполняя при этом операцию $decreaseKey$ над очередью и обновление $p(v)$. Ребро $(v, p(v))$ при этом добавляется к ответу.

Минимальное покрывающее дерево. Алгоритм Прима

```
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printMST(self, parent):
        print("Ребро \tВес")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t",
                self.graph[i][parent[i]])

    def minKey(self, key, mstSet):
        min = float('inf')

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index
```

```
    def primMST(self):
        key = [float('inf')] * self.V
        parent = [None] * self.V
        key[0] = 0
        mstSet = [False] * self.V

        parent[0] = -1

        for cout in range(self.V):
            u = self.minKey(key, mstSet)
            mstSet[u] = True
            for v in range(self.V):
                if self.graph[u][v] > 0 \
                and mstSet[v] == False \
                and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)
```

Минимальное покрывающее дерево. Алгоритм Краскала

Алгоритм Краскала (Крускала) — алгоритм поиска минимального остовного дерева во взвешенном неориентированном связном графе.

Будем последовательно строить подграф F графа G ("растущий лес"), пытаясь на каждом шаге достроить F до некоторого MST. Начнем с того, что включим в F все вершины графа G. Теперь будем обходить множество $E(G)$ в порядке неубывания весов ребер. Если очередное ребро e соединяет вершины одной компоненты связности F, то добавление его в остов приведет к возникновению цикла в этой компоненте связности. В таком случае, очевидно, e не может быть включено в F. Иначе e соединяет разные компоненты связности F, тогда существует $\langle S, T \rangle$ разрез такой, что одна из компонент связности составляет одну его часть, а оставшаяся часть графа — вторую. Тогда e — минимальное ребро, пересекающее этот разрез. Значит, из леммы о безопасном ребре следует, что e является безопасным, поэтому добавим это ребро в F. На последнем шаге ребро соединит две оставшиеся компоненты связности, полученный подграф будет минимальным остовным деревом графа G. Для проверки возможности добавления ребра используется система непересекающихся множеств.

Сириус
IT-Колледж

Минимальное покрывающее дерево. Алгоритм Краскала

```python
class Graph:

    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] != i:
            parent[i] = self.find(parent,
                                  parent[i])
        return parent[i]

    def union(self, parent, rank, x, y):
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x
        else:
            parent[y] = x
            rank[x] += 1
```

```python
    def KruskalMST(self):
        result = []
        i = 0
        e = 0
        self.graph = sorted(self.graph,
                            key=lambda item: item[2])

        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)

        minimumCost = 0
        print("Ребра в MST")
        for u, v, weight in result:
            minimumCost += weight
            print(f"{u} -- {v} == {weight}")
        print("MST", minimumCost)
```

Различия в скорости работы алгоритмов Прима и Крускала

Хотя оба алгоритма работают за $O(M \log N)$, существуют константные различия в скорости их работы. На разреженных графах (количество рёбер примерно равно количеству вершин) быстрее работает алгоритм Крускала, а на насыщенных (количество рёбер примерно равно квадрату количеству вершин) - алгоритм Прима (при использовании матрицы смежности).

На практике чаще используется алгоритм Крускала.

Система непересекающихся множеств

Система непересекающихся множеств (disjoint-set или union–find data structure) — структура данных, которая позволяет администрировать множество элементов, разбитое на непересекающиеся подмножества. При этом каждому подмножеству назначается его представитель — элемент этого подмножества. Абстрактная структура данных определяется множеством трёх операций: $\{\mathrm{Union}, \mathrm{Find}, \mathrm{MakeSet}\}$.

Система непересекающихся множеств

- $\mathrm{make\_set}(x)$ — **добавляет** новый элемент x, помещая его в новое множество, состоящее из одного него.
- $\mathrm{union\_sets}(x, y)$ — **объединяет** два указанных множества (множество, в котором находится элемент x, и множество, в котором находится элемент y).
- $\mathrm{find\_set}(x)$ — **возвращает, в каком множестве** находится указанный элемент x. На самом деле при этом возвращается один из элементов множества (называемый **представителем** или **лидером (leader)**). Этот представитель выбирается в каждом множестве самой структурой данных (и может меняться с течением времени, а именно, после вызовов $\mathrm{union\_sets}()$).

Например, если вызов $\mathrm{find\_set}()$ для каких-то двух элементов вернул одно и то же значение, то это означает, что эти элементы находятся в одном и том же множестве, а в противном случае — в разных множествах.

Представление множеств с помощью деревьев

Множества элементов будем хранить в виде **деревьев**: одно дерево соответствует одному множеству. Корень дерева — это представитель (лидер) множества.

При реализации это означает, что мы заводим массив parent, в котором для каждого элемента мы храним ссылку на его предка в дереве. Для корней деревьев будем считать, что их предок — они сами (т.е. ссылка зацикливается в этом месте).

Представление множеств с помощью деревьев. Наивная реализация

Вся информация о множествах элементов хранится с помощью массива parent.

Чтобы создать новый элемент (операция $\mathrm{make\_set}(v)$), мы просто создаём дерево с корнем в вершине v, отмечая, что её предок — это она сама.

Представление множеств с помощью деревьев. Наивная реализация

Чтобы объединить два множества (операция $\mathrm{union\_sets}(a, b)$), сначала найдём лидеров множества, в котором находится a, и множества, в котором находится b. Если лидеры совпали, то ничего не делаем — это значит, что множества и так уже были объединены. В противном случае можно указать, что предок вершины b равен a (или наоборот) — тем самым присоединив одно дерево к другому.

```
def union(parent, rank, i, j):
    irep = find(parent, i)
    jrep = find(parent, j)
    parent[irep] = jrep
```

Представление множеств с помощью деревьев. Наивная реализация

Реализация операции поиска лидера $(\mathrm{find\_set}(v))$ проста: поднимаемся по предкам от вершины v, пока не дойдём до корня, т.е. пока ссылка на предка не ведёт в себя. Эту операцию удобнее реализовать рекурсивно .

```
def find(i):
    if (parent[i] == i):
        return i
    else:
        return find(parent[i])
```

Представление множеств с помощью деревьев. Наивная реализация

Впрочем, такая реализация системы непересекающихся множеств весьма неэффективна. Легко построить пример, когда после нескольких объединений множеств получится ситуация, что множество — это дерево, выродившееся в длинную цепочку. В результате каждый вызов $\mathrm{find\_set}()$ будет работать на таком тесте за время порядка глубины дерева, т.е. за $O(n)$.

Представление множеств с помощью деревьев. Эвристика сжатия пути

Эта эвристика предназначена для ускорения работы $\mathrm{find\_set}()$.

Она заключается в том, что когда после вызова $\mathrm{find\_set}(v)$ мы найдём искомого лидера p множества, то запомним, что у вершины v и всех пройденных по пути вершин — именно этот лидер p. Проще всего это сделать, перенаправив их $\mathrm{parent}[]$ на эту вершину p.

Таким образом, у массива предков $\mathrm{parent}[]$ смысл несколько меняется: теперь это сжатый массив предков, т.е. для каждой вершины там может храниться не непосредственный предок, а предок предка, предок предка предка, и т.д.

С другой стороны, понятно, что нельзя сделать, чтобы эти указатели parent всегда указывали на лидера: иначе при выполнении операции $\mathrm{union\_sets}()$ пришлось бы обновлять лидеров у $O(n)$ элементов.

Таким образом, к массиву $\mathrm{parent}[]$ следует подходить именно как к массиву предков, возможно, частично сжатому.

Представление множеств с помощью деревьев. Эвристика объединения по рангу

Рассмотрим другую эвристику, которая сама по себе способна ускорить время работы алгоритма, а в сочетании с эвристикой сжатия путей и вовсе способна достигнуть практически константного времени работы на один запрос в среднем.

Эта эвристика заключается в небольшом изменении работы $\mathrm{union\_sets}$: если в наивной реализации то, какое дерево будет присоединено к какому, определяется случайно, то теперь мы будем это делать на основе рангов.

Есть два варианта ранговой эвристики: в одном варианте рангом дерева называется количество вершин в нём, в другом — глубина дерева (точнее, верхняя граница на глубину дерева, поскольку при совместном применении эвристики сжатия путей реальная глубина дерева может уменьшаться).

В обоих вариантах суть эвристики одна и та же: при выполнении $\mathrm{union\_sets}$ будем присоединять дерево с меньшим рангом к дереву с большим рангом.

Представление множеств с помощью деревьев. Объединение эвристик

Сириус
IT-Колледж

```
class DisjSet:
    def __init__(self, n):
        self.rank = [1] * n
        self.parent = [i for i in range(n)]

    def find(self, x):
        if (self.parent[x] != x):
            self.parent[x] =
                self.find(self.parent[x])
        return self.parent[x]
```

```
    def Union(self, x, y):
        xset = self.find(x)
        yset = self.find(y)
        if xset == yset:
            return

        if self.rank[xset] < self.rank[yset]:
            self.parent[xset] = yset
        elif self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
        else:
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + 1
```