Лекция 5. Система непересекающихся множеств (СНМ).

1. Введение и основные понятия

Система непересекающихся множеств (СНМ), также известная в англоязычной литературе как **Union-Find** или **Disjoint Set Union (DSU)**, – это структура данных, предназначенная для работы с коллекцией непересекающихся динамических множеств.

Каждое множество идентифицируется своим **представителем** (или **лидером**). Обычно это один из элементов множества, выбранный по определённому правилу (например, корень дерева).

СНМ поддерживает две основные операции:

1. find(x) – определяет, к какому множеству принадлежит элемент x. Возвращает представителя этого множества.

2. union(x, y) – объединяет множества, содержащие элементы x и y, в одно новое множество.

Типичные применения СНМ:

- Построение минимального остовного дерева (алгоритм Краскала).
- Проверка связности графа.
- Поиск связных компонент в графе.
- Алгоритмы кластеризации.
- Отслеживание связности в динамически изменяющихся графах.

2. Наивная реализация и её проблема

Простейшая идея – представить каждое множество в виде **дерева**. Корень дерева является его представителем. Все элементы множества хранятся в этом дереве, и каждый элемент, кроме корня, имеет ссылку на своего родителя (parent). У корня родитель ссылается на самого себя.

Изначально имеется n элементов, каждый из которых образует отдельное множество (дерево из одного узла).

```
class NaiveDSU:
    def __init__(self, n: int):
        # Изначально каждый элемент - корень своего дерева.
        # parent[i] = i для всех i.
        self.parent = list(range(n))

    def find(self, x: int) -> int:
        """Рекурсивно поднимается по родителям до корня."""
        if self.parent[x] == x:
            return x
        return self.find(self.parent[x])

    def union(self, x: int, y: int) -> None:
        """Находит корни деревьев для x и y и делает один корень родителем
другого."""
        root_x = self.find(x)
        root_y = self.find(y)
```

```
        if root_x != root_y:
            # Просто подвешиваем дерево root_x к корню root_y.
            self.parent[root_x] = root_y
```

Проблема: в наивной реализации операция union может легко создавать **вырожденные деревья** (вытянутые в «бамбук»). В таком дереве операция find будет вынуждена пройти весь путь до корня, что в худшем случае требует **O(n)** времени. При последовательности из m операций общая сложность может достичь **O(m * n)**, что неэффективно для больших n и m.

3. Эвристики для оптимизации

Для ускорения работы СНМ применяются две мощные эвристики.

3.1. Ранговая эвристика (Union by Rank)

Идея: при объединении двух деревьев всегда **подвешиваем дерево меньшей высоты к корню дерева большей высоты**. Это предотвращает вырождение дерева в длинную цепочку.

Для реализации заведём массив rank, где rank[x] хранит **верхнюю оценку высоты** дерева с корнем в x. Изначально ранг каждого элемента равен 0.

```
class DSUWithRank:
    def __init__(self, n: int):
        self.parent = list(range(n))
        self.rank = [0] * n  # Изначально высота каждого дерева ~ 0.

    def find(self, x: int) -> int:
        if self.parent[x] == x:
            return x
        return self.find(self.parent[x])

    def union(self, x: int, y: int) -> None:
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:
            return  # Уже в одном множестве.

        # Решаем, какое дерево к какому подвешивать.
        if self.rank[root_x] < self.rank[root_y]:
            # Дерево root_x "меньше". Подвешиваем его к root_y.
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            # Дерево root_y "меньше". Подвешиваем его к root_x.
            self.parent[root_y] = root_x
        else:
            # Ранги равны. Неважно, какое к какому подвесить, но ранг нового
корня увеличится на 1.
            self.parent[root_y] = root_x
            self.rank[root_x] += 1
```

Утверждение: при использовании ранговой эвристики высота любого дерева не превышает **O(log n)**.

Доказательство (интуитивное): рассмотрим дерево с корнем, имеющим ранг r. Индукцией можно показать, что такое дерево содержит как минимум 2^r элементов. Поскольку всего элементов n, максимально возможный ранг (а значит, и высота) ограничен $\log_2 n$. Следовательно, операция find в худшем случае выполняется за **O(log n)**.

3.2. Эвристика сжатия путей (Path Compression)

Идея: при выполнении операции find(x) мы проходим от x до корня. Давайте **«переподвесим»** сам элемент x и все элементы на этом пути непосредственно к корню. Это значительно упростит будущие запросы для этих элементов.

Реализуется небольшом изменением в методе find:

```
class DSUWithPathCompression:
    def __init__(self, n: int):
        self.parent = list(range(n))

    def find(self, x: int) -> int:
        if self.parent[x] != x:
            # Рекурсивно ищем корень и сразу же делаем его родителем для x.
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x: int, y: int) -> None:
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            self.parent[root_x] = root_y
```

Эта эвристика радикально «сплющивает» деревья. После нескольких операций find дерево превращается в практически плоскую структуру, где большинство элементов ссылаются напрямую на корень.

4. Оптимальная реализация: комбинация эвристик

Максимальная эффективность достигается при одновременном использовании **обеих эвристик**: ранговой и сжатия путей.

```
class DSU:
    """Оптимальная реализация СНМ с эвристиками ранга и сжатия путей."""

    def __init__(self, n: int):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x: int) -> int:
        """Возвращает корень дерева, содержащего x, со сжатием пути."""
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
```

```
    def union(self, x: int, y: int) -> None:
        """Объединяет множества, содержащие x и y, используя ранг."""
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:
            return

        # Ранговая эвристика: меньшее дерево подвешиваем к большему.
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            # Если ранги равны, неважно, кого к кому подвесить, но ранг
нового корня увеличится.
            self.parent[root_y] = root_x
            self.rank[root_x] += 1

    def is_connected(self, x: int, y: int) -> bool:
        """Проверяет, принадлежат ли x и y одному множеству."""
        return self.find(x) == self.find(y)
```

5. Анализ сложности

Амортизированный анализ показывает, что эта структура данных чрезвычайно эффективна.

- **Только ранговая эвристика:** время работы операции **O(log n)**.
- **Только эвристика сжатия путей:** время работы также примерно **O(log n)** для реалистичных значений n.
- **Комбинация обеих эвристик:** время работы на операцию становится **почти постоянным**. Точнее, оно ограничено **обратной функцией Аккермана α(n)**.

Функция Аккермана A(m, n) растёт чрезвычайно быстро. Её обратная функция α(n) растёт, наоборот, очень медленно. Для всех мыслимых на практике значений n $\alpha(n) < 5$.

Таким образом, на практике можно считать, что каждая операция find и union выполняется в **среднем за время, близкое к O(1)**.

Неформальное объяснение: эвристика сжатия путей делает деревья очень плоскими, а трудные случаи, которые могли бы привести к длинным путям, «разруливаются» ранговой эвристикой на этапе union. Чем больше операций выполняется, тем «лучше» становится структура деревьев, ускоряя последующие запросы.

6. Пример использования

Рассмотрим классическую задачу: подсчёт компонент связности в графе.

```
def count_connected_components(n: int, edges: list[tuple[int, int]]) -> int:
    dsu = DSU(n)
```

```
    for u, v in edges:
        dsu.union(u, v)

    # Количество компонент связности равно количеству уникальных корней.
    unique_roots = {dsu.find(i) for i in range(n)}
    return len(unique_roots)


# Пример
if __name__ == "__main__":
    # Граф с 5 вершинами и рёбрами: 0-1, 1-2, 3-4
    n = 5
    edges = [(0, 1), (1, 2), (3, 4)]
    num_components = count_connected_components(n, edges)
    print(f"Количество компонент связности: {num_components}")  # Вывод: 2
```

7. Итог

Реализация	**find (худший случай)**	**union (худший случай)**	**Примечания**
Наивная	O(n)	O(n)	Быстро вырождается
С рангом	O(log n)	O(log n)	Хороший баланс
Со сжатием путей	O(α(n))	O(α(n))	Практически константа