Лекция 14. Heavy-Light декомпозиция (HLD), центроидная декомпозиция.

1. Введение

Дерево – одна из самых фундаментальных структур данных в программировании. Многие задачи сводятся к выполнению запросов на деревьях:

- Нахождение минимума, суммы или другой функции на пути между двумя вершинами.
- Изменение значения в вершине или на ребре.
- Поиск наименьшего общего предка (LCA).

Простые алгоритмы часто работают за O(n) на запрос, что неприемлемо для больших деревьев. **Heavy-Light декомпозиция (HLD)** и **центроидная декомпозиция** – это две мощные техники, позволяющие выполнять многие операции за O(log n) или $O(\log^2 n)$.

2. Heavy-Light декомпозиция (HLD)

2.1. Идея метода

HLD разбивает дерево на набор непересекающихся путей – **тяжёлые пути** – такие, что любой путь от корня до листа проходит через O(log n) тяжёлых путей. Это позволяет эффективно применять структуры данных (например, деревья отрезков) к путям дерева.

Ключевые понятия:

- **Тяжёлое ребро** – ребро, ведущее к поддереву с максимальным размером.
- **Лёгкое ребро** – все остальные рёбра.
- **Тяжёлый путь** – максимальный путь, составленный из тяжёлых рёбер.

Свойства:

- Глубина дерева: O(log n), так как при переходе по лёгкому ребру размер поддерева уменьшается как минимум вдвое.
- Любой путь в дереве разбивается на O(log n) тяжёлых путей.

2.2. Построение HLD

Шаги построения:

1. Вычислить размеры поддеревьев (subtree_size).
2. Определить тяжёлых детей для каждой вершины.
3. Построить тяжёлые пути, пронумеровать вершины в порядке обхода.
4. Построить структуру данных (дерево отрезков) на каждом пути.

2.3. Реализация HLD

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


class HLD:
    def __init__(self, n: int):
        self.n = n  # Количество вершин в дереве
        self.graph = defaultdict(list)  # Список смежности для хранения
дерева
        self.parent = [-1] * n  # Массив родителей вершин
        self.depth = [0] * n  # Массив глубин вершин (расстояние от корня)
        self.heavy = [-1] * n  # Массив "тяжелых" детей для каждой вершины
        self.head = [-1] * n  # Массив "голов" путей (начало тяжелого пути
для вершины)
        self.pos = [-1] * n  # Позиция вершины в линейном массиве HLD
        self.cur_pos = 0  # Текущая позиция для нумерации вершин

    def add_edge(self, u: int, v: int):
        """Добавляет неориентированное ребро между вершинами u и v"""
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, v: int, p: int) -> int:
        """
        Первый обход в глубину для вычисления размеров поддеревьев
        и определения тяжелых детей
        """
        size = 1  # Размер текущего поддерева (начинаем с 1 - сама вершина)
        max_subtree = 0  # Максимальный размер поддерева среди детей

        # Обходим всех соседей вершины v
        for to in self.graph[v]:
            if to != p:  # Исключаем обратное движение к родителю
                self.parent[to] = v  # Запоминаем родителя
                self.depth[to] = self.depth[v] + 1  # Увеличиваем глубину
                # Рекурсивно вычисляем размер поддерева ребенка
                subtree_size = self.dfs(to, v)
                size += subtree_size  # Добавляем к общему размеру

                # Если нашли ребенка с большим поддеревом, обновляем тяжелого
ребенка
                if subtree_size > max_subtree:
                    max_subtree = subtree_size
                    self.heavy[v] = to  # Запоминаем тяжелого ребенка
        return size  # Возвращаем размер поддерева с корнем в v

    def decompose(self, v: int, h: int):
        """
        Второй обход в глубину для разбиения дерева на тяжелые пути
        и присвоения позиций в линейном массиве
        """
        self.head[v] = h  # Запоминаем голову пути для вершины v
```

```
        self.pos[v] = self.cur_pos  # Присваиваем позицию в массиве
        self.cur_pos += 1  # Увеличиваем счетчик позиций

        # Если у вершины есть тяжелый ребенок, продолжаем тот же путь
        if self.heavy[v] != -1:
            self.decompose(self.heavy[v], h)  # Голова пути сохраняется

        # Обходим всех остальных детей (легкие ребра)
        for to in self.graph[v]:
            # Пропускаем родителя и тяжелого ребенка (они уже обработаны)
            if to != self.parent[v] and to != self.heavy[v]:
                self.decompose(to, to)  # Начинаем новый тяжелый путь

    def build(self, root: int = 0):
        """Запускает построение HLD из корневой вершины"""
        self.dfs(root, -1)  # Первый DFS для вычисления размеров
        self.decompose(root, root)  # Второй DFS для разбиения на пути

    def query_path(self, u: int, v: int) -> List[Tuple[int, int]]:
        """
        Разбивает путь от u до v на отрезки тяжелых путей
        Возвращает список кортежей (l, r) - отрезков в линейном массиве
        """
        segments = []

        # Поднимаемся от u и v до их общего предка
        while self.head[u] != self.head[v]:
            # Всегда работаем с вершиной, у которой голова глубже
            if self.depth[self.head[u]] < self.depth[self.head[v]]:
                u, v = v, u  # Меняем местами, чтобы u была с более глубокой
головой

            # Добавляем отрезок от головы u до самой u
            segments.append((self.pos[self.head[u]], self.pos[u]))
            # Переходим к родителю головы u
            u = self.parent[self.head[u]]

        # Теперь u и v находятся в одном тяжелом пути
        # Добавляем оставшийся отрезок между u и v
        if self.depth[u] > self.depth[v]:
            u, v = v, u  # Гарантируем, что u будет выше v

        segments.append((self.pos[u], self.pos[v]))
        return segments
```

Пример использования:

```
hld = HLD(6)
edges = [(0, 1), (1, 2), (1, 3), (2, 4), (2, 5)]
for u, v in edges:
    hld.add_edge(u, v)
hld.build(0)

print("Путь от 4 до 5:", hld.query_path(4, 5))
# Вывод: [(4, 4), (2, 3)] - пример для конкретного дерева
```

2.4. Применение HLD

Запросы на пути. Используя структуру данных на отрезках, можно выполнять запросы суммы, минимума и другие за $O(\log^2 n)$.

Модификация рёбер/вершин. Аналогично, обновление за $O(\log n)$ на отрезке.

3. Центроидная декомпозиция

3.1. Идея метода

Центроидная декомпозиция рекурсивно разбивает дерево на центроиды – вершины, удаление которых разбивает дерево на компоненты размера не более n/2. Это позволяет строить дерево центроидов высоты $O(\log n)$.

Определение. Центроид – вершина, при удалении которой все получаемые компоненты имеют размер $\leq n/2$.

Свойства:

- В любом дереве есть хотя бы один центроид.
- Дерево центроидов имеет глубину $O(\log n)$.

3.2. Построение центроидной декомпозиции

Алгоритм:

1. Найти центроид текущего дерева.
2. Удалить центроид, разбить дерево на компоненты.
3. Рекурсивно построить декомпозицию для каждой компоненты.

3.3. Реализация

```python
from collections import defaultdict
from typing import List, Optional


class CentroidDecomposition:
    def __init__(self, n: int):
        self.n = n  # Количество вершин
        self.graph = defaultdict(list)  # Исходное дерево
        self.size = [0] * n  # Размеры поддеревьев для текущего DFS
        self.parent = [-1] * n  # Родители в дереве центроидов
        self.depth = [0] * n  # Глубина в дереве центроидов
        self.centroid_tree = defaultdict(list)  # Дерево центроидов

    def add_edge(self, u: int, v: int):
        """Добавляет неориентированное ребро в исходное дерево"""
        self.graph[u].append(v)
        self.graph[v].append(u)
```

```python
    def dfs_size(self, v: int, p: int) -> int:
        """
        Вычисляет размеры поддеревьев для текущей компоненты
        Игнорирует уже обработанные вершины (те, у которых parent != -1)
        """
        self.size[v] = 1  # Начинаем с размера 1 (сама вершина)

        for to in self.graph[v]:
            # Пропускаем родителя и уже обработанные центроиды
            if to != p and self.parent[to] == -1:
                self.size[v] += self.dfs_size(to, v)  # Рекурсивно добавляем
размеры

        return self.size[v]  # Возвращаем размер поддерева

    def find_centroid(self, v: int, p: int, total: int) -> int:
        """
        Находит центроид в текущей компоненте
        Центроид - вершина, при удалении которой все компоненты имеют размер
<= total/2
        """
        for to in self.graph[v]:
            # Ищем ребенка с размером > total/2 (нарушение свойства
центроида)
            if to != p and self.parent[to] == -1 and self.size[to] > total //
2:
                # Переходим в этого ребенка, так как он может быть центроидом
                return self.find_centroid(to, v, total)
        # Если такого ребенка нет, текущая вершина - центроид
        return v

    def build_centroid_tree(self, v: int, p: int, depth: int) -> int:
        """
        Рекурсивно строит дерево центроидов
        Возвращает корень текущей компоненты в дереве центроидов
        """
        # Вычисляем размеры поддеревьев для текущей компоненты
        total_size = self.dfs_size(v, -1)
        # Находим центроид текущей компоненты
        centroid = self.find_centroid(v, -1, total_size)

        # Устанавливаем родителя центроида в дереве центроидов
        self.parent[centroid] = p
        # Запоминаем глубину центроида
        self.depth[centroid] = depth

        # Обрабатываем все соседние компоненты (после удаления центроида)
        for to in self.graph[centroid]:
            # Если вершина еще не была центроидом
            if self.parent[to] == -1:
                # Рекурсивно строим дерево центроидов для каждой компоненты
                child_centroid = self.build_centroid_tree(to, centroid, depth
+ 1)
```

```
                # Добавляем связь в дереве центроидов
                self.centroid_tree[centroid].append(child_centroid)

        return centroid  # Возвращаем найденный центроид

    def build(self, root: int = 0) -> int:
        """
        Запускает построение центроидной декомпозиции
        Возвращает корень дерева центроидов
        """
        return self.build_centroid_tree(root, -1, 0)
```

Пример использования:

```
cd = CentroidDecomposition(6)
edges = [(0, 1), (1, 2), (1, 3), (2, 4), (2, 5)]
for u, v in edges:
    cd.add_edge(u, v)
root_centroid = cd.build(0)
print("Корень центроидного дерева:", root_centroid)
```

3.4. Применение центроидной декомпозиции

1. **Подсчёт путей с заданным свойством.** Например, число путей длины k.

2. Запросы ближайшей вершины с определённым свойством.

3. Динамические задачи на деревьях.

4. Сравнение HLD и центроидной декомпозиции

Параметр	**HLD**	**Центроидная декомпозиция**
Время построения	O(n)	O(n log n)
Время запроса	$O(\log^2 n)$	O(log n)
Память	O(n)	O(n log n)
Применение	Запросы на путях, модификация	Подсчёт путей, статические запросы
Сложность реализации	Средняя	Средняя

Вывод:

- **HLD** идеально подходит для задач, где нужно часто выполнять запросы на путях.
- **Центроидная декомпозиция** лучше для задач, связанных с подсчётом путей и статических запросов.