Proyecto de AIS

Диапозитива 2:

"Алгоритм был предложен Джозефом Краскалом в 1956 году и основан на теории графов. Цель алгоритма Краскала — найти минимальное остовное дерево, подмножество рёбер, которое соединяет все вершины графа наиболее эффективным способом, без образования циклов. Он использует жадный подход для итеративного выбора рёбер с наименьшим весом."

Диапозитива 3:

"Алгоритм Краскала работает в три основных этапа. Сначала сортируются все рёбра графа по весу. Затем используется структура данных Union-Find, чтобы определить, вызовет ли добавление ребра цикл в дереве. Если цикл не образуется, ребро добавляется в минимальное остовное дерево. Этот процесс продолжается до тех пор, пока все вершины не будут соединены."

Диапозитива 4 hasta acabar de explicar el codigo:

"Реализация на Java включает четыре основные класса: Kruskal, который содержит логику для применения алгоритма; Graph, который управляет представлением графа; UnionFind, который важен для предотвращения циклов в процессе построения минимального остовного дерева (MST); и Edge, который представляет каждое ребро графа, с его начальной и конечной вершинами и весом. Класс Edge также реализует метод для сравнения рёбер по весу. Кроме того, проект генерирует случайные графы и измеряет время выполнения каждой итерации."

**Пример с 5 узлами**:

Предположим, у нас есть узлы 0, 1, 2, 3 и 4:

* Изначально каждый узел является своим собственным корнем, и ранг каждого равен 0:  
  parent = [0, 1, 2, 3, 4], rank = [0, 0, 0, 0, 0].

1. **Union(0, 1)**:  
   Объединяем узлы 0 и 1. Узел 1 указывает на 0. Ранг 0 увеличивается, так как теперь его дерево стало выше:  
   parent = [0, 0, 2, 3, 4], rank = [1, 0, 0, 0, 0].
2. **Union(2, 3)**:  
   Объединяем узлы 2 и 3. Узел 3 указывает на 2. Ранг 2 увеличивается:  
   parent = [0, 0, 2, 2, 4], rank = [1, 0, 1, 0, 0].
3. **Union(1, 2)**:  
   Объединяем узлы 1 и 2. Корень 1 — это 0, а корень 2 — это 2. Так как ранг 0 меньше, чем ранг 2, объединяем 1 с 2. Ранг 2 увеличивается:  
   parent = [0, 0, 0, 2, 4], rank = [1, 0, 2, 0, 0].
4. **Union(3, 4)**:  
   Объединяем узлы 3 и 4. Узел 4 указывает на 2. Ранг 2 не изменяется, так как его высота остаётся той же:  
   parent = [0, 0, 0, 0, 0], rank = [1, 0, 2, 0, 0].

В конце все узлы объединены в одну компоненту с корнем 0, и ранг корня 0 равен 2.

В этом примере показано, как алгоритм использует **сжатие пути** и **union by rank** для эффективного объединения компонентов.

1. Класс Edge:

 **Описание**: Класс Edge представляет ребро графа. Каждое ребро содержит начальную вершину (src), конечную вершину (dest) и вес (weight). Этот класс реализует интерфейс Comparable<Edge>, чтобы рёбра можно было отсортировать по весу, используя метод compareTo.

 **Метод compareTo**: Этот метод сравнивает веса двух рёбер и позволяет им автоматически сортироваться в коллекциях, требующих порядка, например, при выполнении алгоритма Краскала.

1. Класс GraphGenerator:

 **Описание**: Этот класс генерирует случайные графы и сохраняет их в текстовых файлах. Количество вершин варьируется от 100 до 10 000 с шагом 200. Для каждого графа генерируется случайное количество рёбер (в зависимости от числа вершин), и каждому ребру присваивается случайный вес от 1 до 1000. Графы сохраняются в файлы с именами типа graph\_100.txt, graph\_200.txt и так далее.

 **Метод main**: В этом методе создается файл для каждого числа вершин. Затем для каждого файла генерируются случайные рёбра и записываются в файл. После завершения процесса выводится сообщение о том, что файл был успешно сгенерирован.

1. Класс KruskalAlgorithm

 **Описание**: Этот класс реализует алгоритм Краскала для нахождения минимального остовного дерева (MST) графа. Он читает файлы, сгенерированные классом GraphGenerator, которые содержат графы, и затем выполняет алгоритм Краскала для каждого графа.

 **Метод main**: Этот метод читает файлы, которые содержат графы, обрабатывает их и измеряет время, необходимое для выполнения алгоритма Краскала. После выполнения алгоритма он выводит время выполнения, количество итераций и общий вес минимального остовного дерева.

 **Метод kruskal**: Этот метод реализует основную логику алгоритма Краскала. Сначала рёбра сортируются по весу, затем используется структура данных Union-Find для добавления рёбер в минимальное остовное дерево (MST), только если они не образуют цикл. Алгоритм завершится, когда в MST будет n-1 рёбер для графа с n вершинами.

Диапозитива 5

**Содержание:**

* Зависимость времени от размера графа: показывает, как увеличивается время выполнения по мере роста размера графа.
* Зависимость числа итераций от размера графа: демонстрирует, как изменяется количество итераций в зависимости от размера графа.

**Текст для чтения:**  
"На этих графиках можно увидеть, как время выполнения и количество итераций изменяются по мере увеличения размера графа. Как и ожидалось, время выполнения растет логарифмически, что соответствует сложности O(E log E), поскольку количество рёбер увеличивается пропорционально размеру графа. Однако важно помнить, что накладные расходы виртуальной машины Java (JVM) могут вносить дополнительную вариативность во время выполнения, чего может не быть в языках, ближе работающих к оборудованию."

Диапозитива 6

Алгоритм Краскала имеет временную сложность O(E log E), где основная нагрузка связана с сортировкой рёбер. Операции Union-Find выполняются почти за постоянное время и мало влияют на общую сложность.  
Пространственная сложность может достигать O(n²) при использовании матрицы смежности.  
Экспериментальные графики подтверждают теоретические ожидания: время выполнения растёт с увеличением числа рёбер.  
Следует учитывать накладные расходы JVM, вызывающие небольшие отклонения.  
В целом, алгоритм Краскала остаётся эффективным для построения минимальных остовных деревьев, особенно в разреженных графах.

Диапозитива 7

Алгоритм Краскала легко реализовать и он эффективен для разрежённых графов. Однако на плотных графах его производительность может снизиться из-за затратной сортировки рёбер. Также, использование матрицы смежности увеличивает потребление памяти. Накладные расходы Java, такие как сборка мусора и выполнение кода в JVM, упрощают разработку, но могут снизить производительность по сравнению с языками, компилируемыми в машинный код, особенно для задач с высокой производительностью