**Лабораторная работа №2.**

**Код**

**1.**Класс *DisjointSet*:

class DisjointSet:  
 def \_\_init\_\_(self, n):  
 self.parent = list(range(n))  
 self.rank = [0] \* n

• *DisjointSet* - это класс, реализующий структуру данных "лес непересекающихся множеств". Он используется для отслеживания и объединения компонентов графа.

• *self.parent* - массив, где parent[i] указывает на родителя элемента i. Изначально каждый элемент является своим собственным родителем.

• *self.rank* - массив, который хранит ранг (или глубину) дерева для оптимизации операций объединения.

**2.** Метод *find:*

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

• Метод *find* - находит корень множества, к которому принадлежит элемент u. Если элемент не является своим собственным родителем, метод рекурсивно вызывает себя для его родителя, при этом сжимая путь для повышения эффективности.

**3.** Метод *union:*

def union(self, u, v):  
 root\_u, root\_v = self.find(u), self.find(v)  
 if root\_u != root\_v:  
 if self.rank[root\_u] > self.rank[root\_v]:  
 self.parent[root\_v] = root\_u  
 elif self.rank[root\_u] < self.rank[root\_v]:  
 self.parent[root\_u] = root\_v  
 else:  
 self.parent[root\_v] = root\_u  
 self.rank[root\_u] += 1

• Метод *union* объединяет два множества, содержащие элементы u и v. Сначала он находит корни этих элементов. Если корни разные, то одно дерево присоединяется к другому. При этом учитывается ранг деревьев для минимизации высоты.

**4.** Функция *kraskal:*

def kraskal(vertices, edges):  
 edges.sort(key=lambda x: x[2])  
 disjoint\_set = DisjointSet(vertices)  
 mst = []  
  
 for u, v, weight in edges:  
 if disjoint\_set.find(u) != disjoint\_set.find(v):  
 disjoint\_set.union(u, v)  
 mst.append((u, v, weight))  
  
 return mst

• Функция kraskal реализует алгоритм Краскала для нахождения минимального остовного дерева (MST) графа.

• Сначала ребра сортируются по весу.

• Затем для каждого ребра проверяется, находятся ли его концы в разных множествах. Если да, то они объединяются и ребро добавляется.

**5.** Функция *visualize\_graph:*

def visualize\_graph(edges, filename='graph.dot'):  
 dot\_graph = 'graph G {\n' + ''.join(f' {u} -- {v} [label="{weight}"];\n' for u, v, weight in edges) + '}'  
  
 with open(filename, 'w') as f:  
 f.write(dot\_graph)

• Функция visualize\_graph создает визуализацию графа в формате dot, который может быть использован с инструментами визуализации графов.

• Она принимает список ребер и имя файла для сохранения графа.

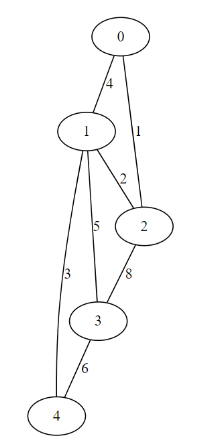
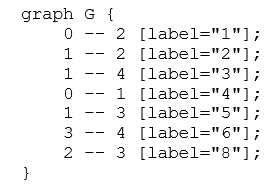
**6.** Основная часть программы:

if \_\_name\_\_ == "\_\_main\_\_":  
 edges = [  
 (0, 1, 4),  
 (0, 2, 1),  
 (1, 2, 2),  
 (1, 3, 5),  
 (2, 3, 8),  
 (3, 4, 6),  
 (1, 4, 3)  
 ]  
  
 vertices = 5  
 mst = kraskal(vertices, edges)  
  
 print("Минимальное остовное дерево:")  
 for u, v, weight in mst:  
 print(f"{u} -- {v} (вес: {weight})")  
  
 visualize\_graph(edges, 'input\_graph.dot')  
 visualize\_graph(mst, 'mst\_graph.dot')

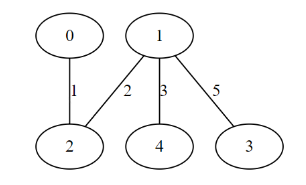
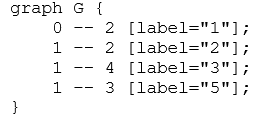
**Тестирование**

**1.**

Входные данные:

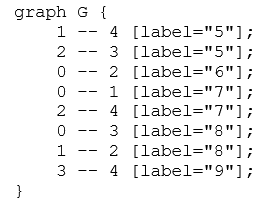
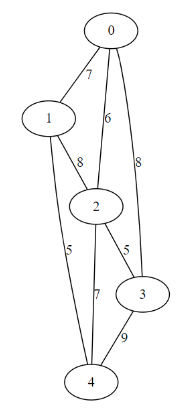


Результат:



**2.**

Входные данные:



Результат:

