Анализ взвешенного графа: Матрица Кирхгофа, Показатель, связанный с количеством Остовных Деревьев, Алгоритм Краскала

Данный ноутбук проводит углубленный анализ взвешенного графа, заданного матрицей смежности. Анализ включает построение матрицы Кирхгофа, расчет показателя, связанного с количеством остовных деревьев (суммы произведений весов ребер по всем остовным деревьям), и поиск минимального остовного дерева (МОД) с использованием алгоритма Краскала. Включены визуализации, подробные математические комментарии и базовые автотесты.

Входные данные: Матрица смежности взвешенного графа.

Матрица смежности для данного варианта:

$$\left(\begin{array}{c}[0,0,6,5,7,0,6,7,3],[0,0,3,0,0,4,3,0,0],[6,3,0,1,0,0,2,2,5],[5,0,1,0,0,4,4,0,0],[7,0,0,0,0,0,6,0,2],\\ [0,4,0,4,0,0,0,5,2],[6,3,2,4,6,0,0,0,2],[7,0,2,0,0,5,0,0,0],[3,0,5,0,2,2,2,0,0]\end{array}\right)$$

```
# Установка необходимых библиотек, если они не установлены
# numpy для работы с матрицами, scipy для определителя, networkx для графовых операций и визуализации, matplotlib для отображени
!pip install numpy scipy networkx matplotlib
!pip install networkx
```

```
Requirement already satisfied: numpy in ./venv/lib/python3.12/site-packages (2.2.6)
Requirement already satisfied: scipy in ./venv/lib/python3.12/site-packages (1.15.3)
Requirement already satisfied: networkx in ./venv/lib/python3.12/site-packages (3.4.2)
Requirement already satisfied: matplotlib in ./venv/lib/python3.12/site-packages (3.10.3)
Requirement already satisfied: contourpy>=1.0.1 in ./venv/lib/python3.12/site-packages (from matplotlib) (1.3.2)
Requirement already satisfied: cycler>=0.10 in ./venv/lib/python3.12/site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in ./venv/lib/python3.12/site-packages (from matplotlib) (4.58.0)
Requirement already satisfied: kiwisolver>=1.3.1 in ./venv/lib/python3.12/site-packages (from matplotlib) (1.4.8)
Requirement already satisfied: packaging>=20.0 in ./venv/lib/python3.12/site-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in ./venv/lib/python3.12/site-packages (from matplotlib) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in ./venv/lib/python3.12/site-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in ./venv/lib/python3.12/site-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in ./venv/lib/python3.12/site-packages (from python-dateutil>=2.7->matplotlib) (1.17
Requirement already satisfied: networkx in ./venv/lib/python3.12/site-packages (3.4.2)
```

```
# Импорт необходимых библиотек
!pip install networkx

import networkx as nx
print(nx.__version__)

import numpy as np
from scipy.linalg import det
import networkx as nx
import matplotlib.pyplot as plt
import collections # Для форматирования разбиения на компоненты
```

```
Requirement already satisfied: networkx in ./venv/lib/python3.12/site-packages (3.4.2)
3.4.2
```

```
# Install necessary libraries (run this only once)
# !pip install numpy scipy networkx matplotlib

# Import required libraries
import numpy as np
from scipy.linalg import det
import networkx as nx
import matplotlib.pyplot as plt
import collections

# Input the new adjacency matrix
adj_matrix = np.array([
    [0,0,6,5,7,0,6,7,3],
    [0,0,3,0,0,4,3,0,0],
    [6,3,0,1,0,0,2,2,5],
    [5,0,1,0,0,4,4,0,0],
    [7,0,0,0,0,0,6,0,2],
    [0,4,0,4,0,0,0,5,2],
    [6,3,2,4,6,0,0,0,2],
    [7,0,2,0,0,5,0,0,0],
```

```
    [3,0,5,0,2,2,2,0,0]
], dtype=int)

num_vertices = adj_matrix.shape[0]

print(f"Matrix size: {num_vertices}x{num_vertices}")
print(adj_matrix)

# Basic validation
assert adj_matrix.shape[0] == adj_matrix.shape[1], "Matrix must be square"
assert np.all(np.diag(adj_matrix) == 0), "Diagonal must be zero"
assert np.all(adj_matrix == adj_matrix.T), "Matrix must be symmetric"
```

```
Matrix size: 9x9
[[0 0 6 5 7 0 6 7 3]
 [0 0 3 0 0 4 3 0 0]
 [6 3 0 1 0 0 2 2 5]
 [5 0 1 0 0 4 4 0 0]
 [7 0 0 0 0 0 6 0 2]
 [0 4 0 4 0 0 0 5 2]
 [6 3 2 4 6 0 0 0 2]
 [7 0 2 0 0 5 0 0 0]
 [3 0 5 0 2 2 2 0 0]]
```

Визуализация исходного графа

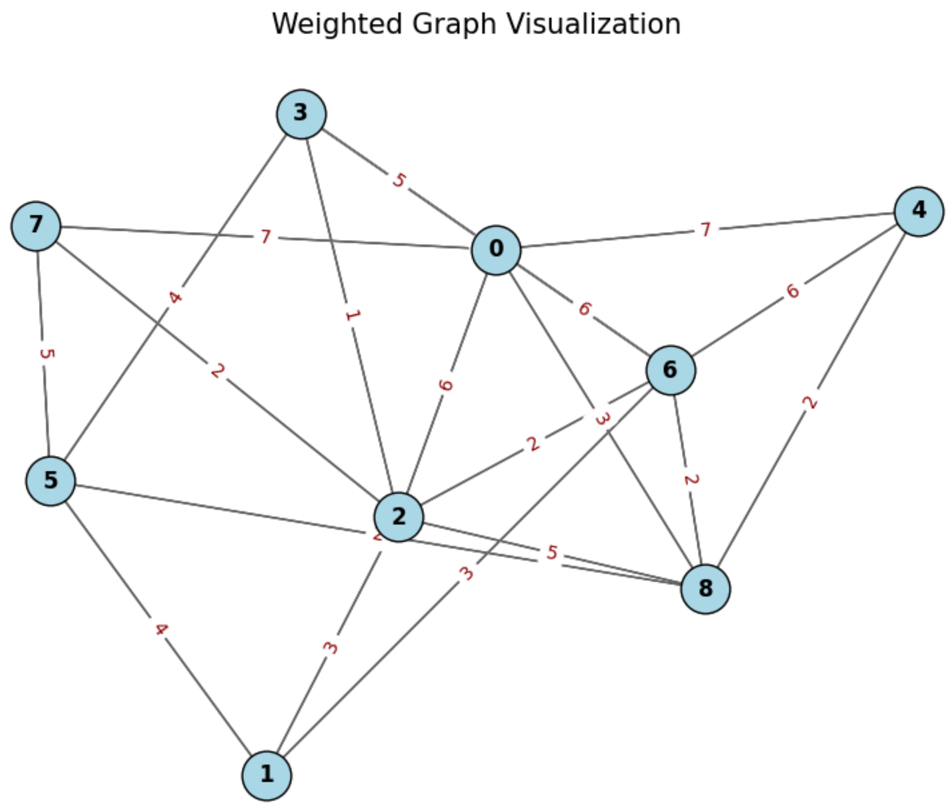
Построим визуализацию заданного взвешенного графа, чтобы лучше понять его структуру. На ребрах указаны их веса.

```
# Create graph from adjacency matrix
G = nx.from_numpy_array(adj_matrix, create_using=nx.Graph)

# Draw the graph
pos = nx.spring_layout(G, seed=42)
plt.figure(figsize=(8, 6))
nx.draw(G, pos, with_labels=True, node_color='lightblue',
        node_size=700, font_weight='bold', edgecolors='black')
nx.draw_networkx_edges(G, pos, edge_color='gray', alpha=0.8)

# Add edge weights
edge_labels = {(u, v): d['weight'] for u, v, d in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='darkred')

plt.title("Weighted Graph Visualization", size=15)
plt.axis('off')
plt.show()
```



Матрица Кирхгофа (Матрица Лапласа)

Матрица Кирхгофа L (или матрица Лапласа) является ключевым инструментом в теории графов, особенно при анализе связности и расчете количества остовных деревьев.

Для взвешенного неориентированного графа с n вершинами она определяется как:

$L = D - A$

где:

- D - **взвешенная матрица степеней** (Weighted Degree Matrix). Это диагональная матрица размера $nimesn$, где каждый диагональный элемент D_{ii} равен сумме весов всех ребер, инцидентных вершине i. $D_{ii} = \sum_{j=0}^{n-1} w_{ij}$, где w_{ij} - вес ребра между вершинами i и j. Внедиагональные элементы D_{ij} при $i \neq j$ равны 0.
- A - **взвешенная матрица смежности** (Weighted Adjacency Matrix). Это матрица размера $nimesn$, где элемент A_{ij} равен весу ребра между вершинами i и j (w_{ij}). Если ребра нет, $w_{ij} = 0$.

Элементы матрицы Кирхгофа L_{ij} вычисляются следующим образом:

- На диагонали ($i = j$): $L_{ii} = D_{ii} - A_{ii} = \sum_{k=0}^{n-1} w_{ik} - 0 = \sum_{k \neq i} w_{ik}$ (сумма весов всех ребер, выходящих из вершины i).
- Вне диагонали ($i \neq j$): $L_{ij} = D_{ij} - A_{ij} = 0 - w_{ij} = -w_{ij}$ (отрицательный вес ребра между i и j, если оно существует, иначе 0).

Важное свойство матрицы Кирхгофа: сумма элементов в любой строке и любом столбце равна нулю. Это свойство может служить проверкой корректности построения матрицы.

```
# Calculate degree matrix
weighted_degrees = np.sum(adj_matrix, axis=1)
degree_matrix = np.diag(weighted_degrees)

print("Degree Matrix:")
print(degree_matrix)

# Calculate Kirchhoff matrix
kirchhoff_matrix = degree_matrix - adj_matrix

print("\nKirchhoff Matrix:")
```

```
print(kirchhoff_matrix)

# Validate Kirchhoff properties
assert np.all(np.isclose(np.sum(kirchhoff_matrix, axis=1), 0)), "Row sums should be zero"
assert np.all(np.isclose(np.sum(kirchhoff_matrix, axis=0), 0)), "Column sums should be zero"
```

```
Degree Matrix:
[[34  0  0  0  0  0  0  0  0]
 [ 0 10  0  0  0  0  0  0  0]
 [ 0  0 19  0  0  0  0  0  0]
 [ 0  0  0 14  0  0  0  0  0]
 [ 0  0  0  0 15  0  0  0  0]
 [ 0  0  0  0  0 15  0  0  0]
 [ 0  0  0  0  0  0 23  0  0]
 [ 0  0  0  0  0  0  0 14  0]
 [ 0  0  0  0  0  0  0  0 14]]

Kirchhoff Matrix:
[[34  0 -6 -5 -7  0 -6 -7 -3]
 [ 0 10 -3  0  0 -4 -3  0  0]
 [-6 -3 19 -1  0  0 -2 -2 -5]
 [-5  0 -1 14  0 -4 -4  0  0]
 [-7  0  0  0 15  0 -6  0 -2]
 [ 0 -4  0 -4  0 15  0 -5 -2]
 [-6 -3 -2 -4 -6  0 23  0 -2]
 [-7  0 -2  0  0 -5  0 14  0]
 [-3  0 -5  0 -2 -2 -2  0 14]]
```

Показатель, связанный с количеством Остовных Деревьев (Матричная Теорема о Деревьях Кирхгофа)

Матричная теорема о деревьях Кирхгофа связывает количество остовных деревьев графа с определителем миноров его матрицы Лапласа.

Для **взвешенного** графа определитель любого $(n-1) \times (n-1)$ минора взвешенной матрицы Лапласа равен **сумме произведений весов ребер по всем различным остовным деревьям** графа.

$\det(L') = \sum_{T \text{ - остовное дерево}} \prod_{(u,v) \in T} w_{uv}$

где L' - минорная матрица Лапласиана L размера $(n-1) \times (n-1)$, полученная удалением одной строки и одного столбца, а w_{uv} - вес ребра (u, v).

Если все веса $w_{uv} = 1$, то определитель равен просто количеству остовных деревьев.

Согласно заданию, мы вычислим определитель минора взвешенной матрицы Кирхгофа, что даст нам сумму произведений весов по всем остовным деревьям.

```
# Calculate minor matrix (remove first row and column)
minor_matrix = kirchhoff_matrix[1:, 1:]

print("Minor Matrix:")
print(minor_matrix)

# Calculate determinant (weighted sum of spanning trees)
det_value = det(minor_matrix)
print(f"\nDeterminant of minor matrix: {det_value:.4f}")
print("(Sum of products of edge weights for all spanning trees)")

# For unweighted version (number of spanning trees)
unweighted_adj = (adj_matrix > 0).astype(int)
unweighted_degrees = np.sum(unweighted_adj, axis=1)
unweighted_laplacian = np.diag(unweighted_degrees) - unweighted_adj
unweighted_minor = unweighted_laplacian[1:, 1:]
num_spanning_trees = round(abs(det(unweighted_minor)))

print(f"\nNumber of spanning trees (unweighted): {num_spanning_trees}")
```

```
Minor Matrix:
[[10 -3  0  0 -4 -3  0  0]
 [-3 19 -1  0  0 -2 -2 -5]
 [ 0 -1 14  0 -4 -4  0  0]
 [ 0  0  0 15  0 -6  0 -2]
 [-4  0 -4  0 15  0 -5 -2]
 [-3 -2 -4 -6  0 23  0 -2]
 [ 0 -2  0  0 -5  0 14  0]
 [ 0 -5  0 -2 -2 -2  0 14]]
```

```
Determinant of minor matrix: 966203840.0000
(Sum of products of edge weights for all spanning trees)

Number of spanning trees (unweighted): 21320
```

Алгоритм Краскала для поиска Минимального Остовного Дерева (МОД)

Алгоритм Краскала - это жадный алгоритм, который находит минимальное остовное дерево во взвешенном неориентированном графе. Он работает путем постепенного добавления ребер в строгом порядке возрастания их веса, избегая создания циклов.

Ключевые идеи:

1. **Сортировка ребер:** Все ребра графа сортируются по возрастанию веса.
2. **Проверка на цикл:** Ребра рассматриваются по одному в отсортированном порядке. Ребро добавляется к МОД только если оно не образует цикл с уже добавленными ребрами.
3. **DSU (Disjoint Set Union):** Для эффективной проверки на циклы и отслеживания компонент связности используется структура данных DSU. Две вершины находятся в одной компоненте, если между ними уже существует путь из добавленных ребер. Добавление ребра между вершинами в одной компоненте создает цикл.
4. **Построение МОД:** Алгоритм продолжается до тех пор, пока не будет добавлено $n-1$ ребро (для связного графа с n вершинами), что достаточно для формирования остовного дерева.

Ниже приведена реализация DSU и пошаговое выполнение алгоритма Краскала для данного графа, с описанием добавляемых ребер и изменения текущего разбиения вершин на компоненты связности.

```python
class DSU:
    def __init__(self, n):
        # Изначально каждая вершина (от 0 до n-1) находится в своем собственном множестве.
        # parent[i] хранит родителя вершины i. Если parent[i] == i, i является корнем множества.
        self.parent = list(range(n))
        # rank[i] используется для оптимизации операции union (объединение по рангу).
        # Ранг - это верхняя граница высоты дерева. Объединение по рангу помогает сохранять
        # деревья относительно плоскими, что улучшает производительность find и union.
        self.rank = [0] * n

    # Операция find с сжатием путей (path compression)
    # Находит корень (представителя) множества, содержащего элемент i.
    # Во время поиска, делает все узлы на пути от i до корня прямыми потомками корня,
    # что ускоряет будущие операции find для этих узлов.
    def find(self, i):
        if self.parent[i] == i:
            return i
        # Рекурсивно находим корень и устанавливаем его как непосредственного родителя i
        self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    # Операция union по рангу
    # Объединяет два множества, содержащие элементы i и j.
    # Сначала находит корни множеств, содержащих i и j.
    # Если корни разные, объединяет множества (подвешивает дерево с меньшим рангом к корню дерева с большим рангом).
    # Если ранги равны, одно дерево становится дочерним к другому, и ранг нового корня увеличивается.
    # Возвращает True, если объединение произошло (т.е., i и j были в разных множествах), иначе False.
    def union(self, i, j):
        root_i = self.find(i) # Находим корень множества i
        root_j = self.find(j) # Находим корень множества j

        # Если корни разные, значит элементы i и j находятся в разных компонентах связности.
        # Добавление ребра (i, j) не создаст цикл, и компоненты можно объединить.
        if root_i != root_j:
            # Объединяем по рангу: дерево с меньшим рангом присоединяем к дереву с большим рангом
            if self.rank[root_i] < self.rank[root_j]:
                self.parent[root_i] = root_j
            elif self.rank[root_i] > self.rank[root_j]:
                self.parent[root_j] = root_i
            else:
                # Если ранги равны, можно выбрать любой корень, например root_i
                self.parent[root_j] = root_i
                # И увеличить ранг нового корня
                self.rank[root_i] += 1
            return True # Объединение успешно выполнено
        return False # Элементы уже в одном множестве, объединение не требуется (ребро создаст цикл)

    # Метод для получения текущего разбиения вершин на компоненты связности
```

```
    # Полезно для отслеживания состояния DSU в процессе работы алгоритма.
    def get_partitions(self):
        # Находим корни для всех вершин. Это также выполняет сжатие путей.
        roots = [self.find(i) for i in range(len(self.parent))]

        # Группируем вершины по их корням. collections.defaultdict удобен для этого.
        partitions_dict = collections.defaultdict(list)
        for i, root in enumerate(roots):
            partitions_dict[root].append(i)

        # Форматируем вывод для удобства чтения:
        # 1. Сортируем вершины внутри каждого раздела.
        # 2. Сортируем сами разделы на основе первой вершины в разделе (для стабильного порядка вывода).
        formatted_partitions = []
        # Получаем список пар (корень, список_вершин) и сортируем по корню
        sorted_items = sorted(partitions_dict.items())

        for root, vertices in sorted_items:
             # Сортируем вершины внутри каждого раздела
             formatted_partitions.append(sorted(vertices))

        return formatted_partitions

# --- Подготовка ребер для алгоритма Краскала ---

# Извлекаем все ребра из матрицы смежности вместе с их весами.
# Поскольку граф неориентированный, матрица симметрична. Каждое ребро (i, j) с весом w
# появляется как A[i, j] = w и A[j, i] = w. Чтобы избежать дублирования, просматриваем
# только верхний треугольник матрицы (i < j) и добавляем ребро, только если вес > 0.
# Храним ребра как кортежи (вес, вершина1, вершина2).
edges = []
for i in range(num_vertices):
    for j in range(i + 1, num_vertices): # Просматриваем только элементы выше главной диагонали
        if adj_matrix[i, j] != 0:
            edges.append((adj_matrix[i, j], i, j))

# Сортируем список ребер по весу в порядке неубывания - это ключевой шаг алгоритма Краскала (жадный подход)
edges.sort()

print("Список ребер графа, отсортированный по весу (вес, вершина_u, вершина_v):")
for edge in edges:
    print(edge)

# --- Выполнение алгоритма Краскала ---

# Инициализируем структуру DSU для всех вершин графа.
# Каждая вершина изначально является отдельной компонентой связности.
dsu = DSU(num_vertices)

# Список для хранения ребер, которые войдут в Минимальное Остовное Дерево (МОД)
mst_edges = []

print("\n" + "="*40)
print("--- Выполнение алгоритма Краскала --- ")
print(f"Начальное разбиение вершин на компоненты: {dsu.get_partitions()}")
print("="*40)

step_counter = 1

# Перебираем ребра в порядке сортировки по весу
for weight, u, v in edges:
    # Для текущего ребра (u, v) с весом 'weight', проверяем, находятся ли вершины u и v
    # в одной и той же компоненте связности, используя операцию find() DSU.
    root_u = dsu.find(u) # Находим корень компоненты вершины u
    root_v = dsu.find(v) # Находим корень компоненты вершины v

    print(f"\nШаг {step_counter}:")
    print(f"  Рассматриваемое ребро: ({u}, {v}) с весом {weight}")
    print(f"  Корни компонент вершин {u} и {v}: {root_u} и {root_v}.")

    # Если корни разные, значит вершины u и v находятся в разных компонентах.
    # Добавление ребра (u, v) не создает цикл с уже выбранными ребрами.
    if root_u != root_v:
        # Ребро (u, v) является безопасным ребром (по свойству среза) и включается в МОД.
        mst_edges.append((u, v, weight)) # Добавляем ребро и его вес в список ребер МОД

        # Объединяем компоненты, содержащие вершины u и v, используя операцию union().
```

```
        # Теперь эти вершины и их компоненты считаются одной большой компонентой.
        dsu.union(u, v)

        print(f"  Вершины в разных компонентах. Ребро ({u}, {v}) добавляется к МОД.")
        print(f"  Произведено объединение компонент {root_u} и {root_v}.")

        # Если количество ребер в МОД достигло N-1 (где N - число вершин),
        # мы нашли остовное дерево. Для связного графа это минимальное остовное дерево.
        if len(mst_edges) == num_vertices - 1:
            print("  МОД найдено (добавлено N-1 ребро). Завершение алгоритма.")
            break # Алгоритм Краскала завершен
    else:
        # Если root_u == root_v, вершины u и v уже находятся в одной компоненте связности.
        # Добавление ребра (u, v) создало бы цикл с уже выбранными ребрами МОД.
        # Такие ребра игнорируются алгоритмом Краскала.
        print(f"  Вершины в одной компоненте. Ребро ({u}, {v}) игнорируется (создаст цикл).")

    # Выводим текущее разбиение вершин на компоненты после обработки ребра
    print(f"  Текущее разбиение вершин: {dsu.get_partitions()}")

    step_counter += 1

print("\n" + "="*40)
print("--- Алгоритм Краскала завершен ---")
print("="*40)

# --- Автотест: Проверка количества ребер в найденном МОД ---
# Для связного графа с N вершинами МОД должно содержать N-1 ребро.
# Если найдено меньше ребер, возможно исходный граф несвязный.
assert len(mst_edges) == num_vertices - 1, f"Ошибка: Ожидалось {num_vertices - 1} ребер в МОД для связного графа, найдено {len(ms
print(f"\nПроверка: Найдено {len(mst_edges)} ребер в МОД. Для {num_vertices} вершин ожидается {num_vertices - 1}. OK (предполагае
```

```
  Вершины в разных компонентах. Ребро (2, 3) добавляется к МОД.
  Произведено объединение компонент 2 и 3.
  Текущее разбиение вершин: [[0], [1], [2, 3], [4], [5], [6], [7], [8]]

Шаг 2:
  Рассматриваемое ребро: (2, 6) с весом 2
  Корни компонент вершин 2 и 6: 2 и 6.
  Вершины в разных компонентах. Ребро (2, 6) добавляется к МОД.
  Произведено объединение компонент 2 и 6.
  Текущее разбиение вершин: [[0], [1], [2, 3, 6], [4], [5], [7], [8]]

Шаг 3:
  Рассматриваемое ребро: (2, 7) с весом 2
  Корни компонент вершин 2 и 7: 2 и 7.
  Вершины в разных компонентах. Ребро (2, 7) добавляется к МОД.
  Произведено объединение компонент 2 и 7.
  Текущее разбиение вершин: [[0], [1], [2, 3, 6, 7], [4], [5], [8]]

Шаг 4:
```

```
  Вершины в разных компонентах. Ребро (1, 2) добавляется к МОД.
  Произведено объединение компонент 1 и 2.
  МОД найдено (добавлено N-1 ребро). Завершение алгоритма.

========================================
--- Алгоритм Краскала завершен ---
========================================

Проверка: Найдено 8 ребер в МОД. Для 9 вершин ожидается 8. OK (предполагается связный граф)
```

```
dsu = DSU(num_vertices)
mst_edges = []
total_weight = 0

print("\nKruskal's Algorithm Steps:")
for weight, u, v in edges:
    if dsu.find(u) != dsu.find(v):
        dsu.union(u, v)
        mst_edges.append((u, v, weight))
        total_weight += weight
        print(f"Added edge ({u}, {v}) with weight {weight}")
        if len(mst_edges) == num_vertices - 1:
            break

print("\nMST Edges:")
for u, v, weight in mst_edges:
    print(f"({u}, {v}) - {weight}")

print(f"\nTotal MST Weight: {total_weight}")
```

```
Kruskal's Algorithm Steps:
Added edge (2, 3) with weight 1
Added edge (2, 6) with weight 2
Added edge (2, 7) with weight 2
Added edge (4, 8) with weight 2
Added edge (5, 8) with weight 2
Added edge (6, 8) with weight 2
Added edge (0, 8) with weight 3
Added edge (1, 2) with weight 3

MST Edges:
(2, 3) - 1
(2, 6) - 2
(2, 7) - 2
(4, 8) - 2
(5, 8) - 2
(6, 8) - 2
(0, 8) - 3
(1, 2) - 3

Total MST Weight: 17
```

Визуализация Минимального Остовного Дерева (МОД)

Теперь визуализируем исходный граф, выделив другим цветом и толщиной ребра, которые вошли в состав найденного Минимального Остовного Дерева. Это помогает наглядно увидеть структуру МОД.

```
G_mst = nx.from_numpy_array(adj_matrix, create_using=nx.Graph)
pos = nx.spring_layout(G_mst, seed=42)

mst_edge_set = set(tuple(sorted((u, v))) for u, v, w in mst_edges)

edge_colors = []
edge_widths = []
for u, v in G_mst.edges():
    if tuple(sorted((u, v))) in mst_edge_set:
        edge_colors.append('red')
        edge_widths.append(3)
    else:
        edge_colors.append('gray')
        edge_widths.append(1)

plt.figure(figsize=(8, 6))
nx.draw(G_mst, pos, with_labels=True, node_color='lightblue',
        node_size=700, font_weight='bold', edgecolors='black')
nx.draw_networkx_edges(G_mst, pos, edge_color=edge_colors, width=edge_widths)
```

```
edge_labels = {(u, v): d['weight'] for u, v, d in G_mst.edges(data=True)}
nx.draw_networkx_edge_labels(G_mst, pos, edge_labels=edge_labels, font_color='darkred')

plt.title("Graph with Minimum Spanning Tree", size=15)
plt.axis('off')
plt.show()
```