# 图论（3）更多图算法

## 一、拓扑排序（Topological Sorting）

### 1.定义

拓扑排序是对有向无环图（DAG）进行排序的一种算法。它将图中的顶点按照一种线性顺序进行排列，使得对于任意的有向边（u，v），顶点u在排序中出现在顶点v的前面

拓扑排序可以用于解决一些依赖关系的问题，例如任务调度、编译顺序等

### 2.典型问题描述：煎松饼

考虑如何制作一批松饼。配方：一个鸡蛋，一勺油，3/4杯牛奶。为了制作松饼，需要加热平底锅，并将所有材料倒入锅中。当出现气泡时候，将松饼反面，并继续煎至底部出现金黄色。在享用松饼之前，还会加热一些枫糖浆。

### 3.DFS实现煎松饼

首先，我们解析好图之后，对图g调用dfs，即根据松饼制作步骤构建深度优先森林；然后再根据结果进行拓扑排序即可

注意：在拓扑排序中，核心在于dfs调用时，我们需要记录每个节点完成解析的时间finish。为此，我们可以在图中增添time属性，以记录当前遍历经过了多少时间。再进行拓扑排序时，为了使得前序步骤在后序步骤前，只需要对finish的时间进行逆序排序即可。

```
import sys

class Graph:
    def __init__(self):
        self.vertices = {}
        self.num_vertices = 0

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1
        new_ertex = Vertex(key)
        self.vertices[key] = new_ertex
        return new_ertex

    def get_vertex(self, n):
        if n in self.vertices:
            return self.vertices[n]
        else:
            return None
```

```
    def __len__(self):
        return self.num_vertices

    def __contains__(self, n):
        return n in self.vertices

    def add_edge(self, f, t, cost=0):
        if f not in self.vertices:
            nv = self.add_vertex(f)
        if t not in self.vertices:
            nv = self.add_vertex(t)
        self.vertices[f].add_neighbor(self.vertices[t], cost)
        #self.vertices[t].add_neighbor(self.vertices[f], cost)

    def getVertices(self):
        return list(self.vertices.keys())

    def __iter__(self):
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, num):
        self.key = num
        self.connectedTo = {}
        self.color = 'white'
        self.distance = sys.maxsize
        self.previous = None
        self.discovery = 0
        self.finish = None

    def __lt__(self, o):
        return self.key < o.key

    def add_neighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def setDiscovery(self, dtime):
        self.discovery = dtime

    def setFinish(self, ftime):
        self.finish = ftime

    def getFinish(self):
        return self.finish

    def getDiscovery(self):
        return self.discovery

    def get_neighbors(self):
        return self.connectedTo.keys()

class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
```

```
        self.time = 0
        self.topologicalList = []

    def dfs(self):
        for aVertex in self:
            aVertex.color = "white"
            aVertex.predecessor = -1
        for aVertex in self:
            if aVertex.color == "white":
                self.dfsvisit(aVertex)

    def dfsvisit(self, startVertex):
        startVertex.color = "gray"
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.get_neighbors():
            if nextVertex.color == "white":
                nextVertex.previous = startVertex
                self.dfsvisit(nextVertex)
        startVertex.color = "black"
        self.time += 1
        startVertex.setFinish(self.time)

    def topologicalSort(self):
        self.dfs()
        temp = list(self.vertices.values())
        temp.sort(key = lambda x: x.getFinish(), reverse = True)
        print([(x.key,x.finish) for x in temp])
        self.topologicalList = [x.key for x in temp]
        return self.topologicalList

# Creating the graph
g = DFSGraph()

g.add_vertex('cup_milk')    # 3/4杯牛奶
g.add_vertex('egg')         # 一个鸡蛋
g.add_vertex('tbl_oil')     # 1勺油

g.add_vertex('heat_griddle')    # 加热平底锅
g.add_vertex('mix_ingredients') # 混合材料——1杯松饼粉
g.add_vertex('pour_batter')   # 倒入1/4杯松饼粉
g.add_vertex('turn_pancake')    # 出现气泡时翻面
g.add_vertex('heat_syrup')  # 加热枫糖浆
g.add_vertex('eat_pancake') # 开始享用

# Adding edges based on dependencies
g.add_edge('cup_milk', 'mix_ingredients')
g.add_edge('mix_ingredients', 'pour_batter')
g.add_edge('pour_batter', 'turn_pancake')
g.add_edge('turn_pancake', 'eat_pancake')

g.add_edge('mix_ingredients', 'heat_syrup')
g.add_edge('heat_syrup', 'eat_pancake')

g.add_edge('heat_griddle', 'pour_batter')
g.add_edge('tbl_oil', 'mix_ingredients')
```

```
g.add_edge('egg', 'mix_ingredients')

# Getting topological sort of the tasks
topo_order = g.topologicalSort()
print("Topological Sort of the Pancake Making Process:")
print(topo_order)

"""
Output:
函数 topologicalSort 中的调试信息
[('heat_griddle', 18), ('tbl_oil', 16), ('egg', 14), ('cup_milk', 12), ('mix_ingredients', 11)
Topological Sort of the Pancake Making Process:
['heat_griddle', 'tbl_oil', 'egg', 'cup_milk', 'cup_mix', 'heat_syrup', 'pour_cup', 'turn_panc
"""
```

## 4.Kahn算法：基于BFS的拓扑排序算法

### 4.1 基本思想与步骤

Kahn算法的基本思想是通过不断移除图中的入度为0的顶点，将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除，具体步骤如下：

**（1）初始化一个队列，用于存储当前入度为0的顶点**

**（2）遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中**

**（3）不断从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度-1。如果某个邻居的入度减为0，则将其加入到队列中**

**（4）重复步骤3，直到队列为空**

### 4.2 Kahn算法进行拓扑排序的代码实现：

```
from collections import deque, defaultdict
def topological_sort(graph):
    indegree = defaultdict(int)
    #defaultdict(type) 可以为不存在的key生成一个默认值，避免keyError异常
    result = []
    queue = deque()
    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)
    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)
        for v in graph[u]:
```

```
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None
# 示例调用代码
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['F'],
    'E': ['F'],
    'F': []
}
sorted_vertices = topological_sort(graph)
if sorted_vertices:
    print("Topological sort order:", sorted_vertices)
else:
    print("The graph contains a cycle.")
```

注：当存在环时，生成的result不会涵盖全部graph的节点

### 4.3 使用Kahn算法实现煎松饼

```
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)

    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)

        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
```

```
    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None

# 示例调用代码
graph = {
    'cup_milk': ['mix_ingredients'],
    'mix_ingredients': ['pour_batter', 'heat_syrup'],
    'pour_batter': ['turn_pancake'],
    'turn_pancake': ['eat_pancake'],
    'heat_syrup': ['eat_pancake'],
    'heat_griddle': ['pour_batter'],
    'tbl_oil': ['mix_ingredients'],
    'egg': ['mix_ingredients'],
    'eat_pancake': []
}
sorted_vertices = topological_sort(graph)
if sorted_vertices:
    print("Topological sort order:", sorted_vertices)
else:
    print("The graph contains a cycle.")
```

# 二、强连通单元（SCCs）：Kosaraju算法

## 1.基本定义1：强连通（单元）

强连通是指一个有向图（Directed Graph）中，任意两点v1、v2之间均存在v1到v2的路径及v2到v1的路径

对于图G，强连通单元（strongly connected component,SCC）C为最大的顶点子集C，对于每一对顶点v，w∈C，都有一条从v到w的路径和一条从w到v的路径

定义强连通单元之后，抽象意义上就可以把一个强连通单元看成单个顶点，从而将图简化

## 2.图的转置G^T

图G的转置图G^T，指的是所有的边都与图G的边反向。这意味着，如果在图G中有一条由A到B的边，那么在G^T中就会有一条从B到A的边

容易证明，图及其转置的强连通单元个数是相等的

## 3.Kosaraju算法（2DFS）

Kosaraju算法是一种用于在有向图中寻找强连通分量SCC的算法，其基于DFS和图的转置操作

### 3.1 基本步骤

第一次DFS：在第一次DFS中，对图进行标准的深度优先搜索，并用栈记录顶点完成搜索的顺序

反向图：对原图取转置，得到G^T

第二次DFS：在第二次DFS中，我们遍历的顶点次序按照stack的逆序（直接弹出node），对反向图进行DFS

Kosaraju证明，在第二次DFS中每次搜索得到的连通分量，即为一组强连通分量SCC

### 3.2 python代码实现

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
```

```
    print(scc)
```

### 3.3 时间复杂度分析

主程序即为两次dfs，因此时间复杂度为O(V+E)

## 4. 补充：Tarjan算法

Tarjan算法与Kosaraju算法相似，均用于寻找图的强连通分量，但是两者对于SCC的计算方法并不相同

Tarjan算法的底层逻辑依旧是用DFS来遍历图。其基本思想是，在DFS中维护一个栈来记录已经访问过的顶点，并为每个顶点分配一个“搜索次序”（DFS编号）和一个“最低链接值”

搜索次序表示顶点被首次访问的次序，最低链接值表示从当前顶点出发经过一系列边能到达的最早的顶点的搜索次序

### 4.1 算法步骤

1. 从图中选择一个未访问的顶点进行DFS
2. 为当前顶点分配一个搜索次序和最低链接值，并将其入栈
3. 对当前顶点的每个未访问过的邻接顶点进行DFS
4. 在递归回溯的过程中，更新当前顶点的最低链接值，使其指向当前顶点和其邻接顶点之间较小的搜索次序
5. 如果当前顶点的最低链接值等于其自身的搜索次序，那么将从当前顶点开始的栈中的所有顶点弹出，并将它们构成一个强连通分量

### 4.2 算法时间复杂度：O（V+E）

### 4.3 Tarjan算法的实现

```
def tarjan(graph):
    def dfs(node):
        nonlocal index, stack, indices, low_link, on_stack, sccs
        #nonlocal用于在嵌套函数中修改外层函数的局部变量
        index += 1
        indices[node] = index
        low_link[node] = index
        stack.append(node)
        on_stack[node] = True

        for neighbor in graph[node]:
            if indices[neighbor] == 0:  # Neighbor not visited yet
                dfs(neighbor)
                low_link[node] = min(low_link[node], low_link[neighbor])
            elif on_stack[neighbor]:  # Neighbor is in the current SCC
                low_link[node] = min(low_link[node], indices[neighbor])

        if indices[node] == low_link[node]:
            scc = []
            while True:
                top = stack.pop()
```

```
                on_stack[top] = False
                scc.append(top)
                if top == node:
                    break
            sccs.append(scc)

    index = 0
    stack = []
    indices = [0] * len(graph)
    low_link = [0] * len(graph)
    on_stack = [False] * len(graph)
    sccs = []

    for node in range(len(graph)):
        if indices[node] == 0:
            dfs(node)

    return sccs

# Example
graph = [[1],        # Node 0 points to Node 1
         [2, 4],     # Node 1 points to Node 2 and Node 4
         [3, 5],     # Node 2 points to Node 3 and Node 5
         [0, 6],     # Node 3 points to Node 0 and Node 6
         [5],        # Node 4 points to Node 5
         [4],        # Node 5 points to Node 4
         [7],        # Node 6 points to Node 7
         [5, 6]]     # Node 7 points to Node 5 and Node 6

sccs = tarjan(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

"""
Strongly Connected Components:
[4, 5]
[7, 6]
[3, 2, 1, 0]
"""
```

# 三、最短路径算法（Shortest Paths）：从Dijkstra算法谈起

单源最短路径问题广泛应用在互联网的运作机制中。抽象而言，我们要解决的问题是为给定信息找到权重最小的路径。在无权重的情况下，BFS算法可以帮助我们找到最短路径（无权图）。然而，当存在权重时，我们就需要使用Dijkstra算法（有权图）

Dijkstra算法是一种有权图的最短路径算法，它通过贪心策略逐步确定从起始顶点到所有其他顶点的最短加权路径。具体而言，Dijkstra算法使用优先队列（通常是最小堆）来保存待访问的顶点，并按照顶点到起始顶点的距离进行排序。它根据路径长度来决定下一个要访问的顶点，从而保证每次都是选择最短路径的顶点进行访问。

特别注意：Dijkstra算法只适用于边的权重均为正的情况

## 1.Dijkstra算法的具体步骤

在实践中，我们通常使用heapq包来实现Dijkstra算法的完整python代码。这个实现包括了图的类表示、顶点类，以及Dijkstra算法的具体逻辑

1. 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为0，其他节点的距离为无穷大（sys可以帮助实现）
2. 选择一个未访问的节点中距离最小的节点作为当前节点
3. 更新当前节点的邻居节点的距离，如果通过当前节点到达邻居节点的路径比已知最短路径更短，则更新最短路径
4. 标记当前节点为已访问
5. 重复上述步骤，直到所有节点都被访问或者所有节点的最短路径都被确定

## 2.代码实现

该实现代码关键的地方在于，没有采用heapify，而是从初始顶点开始，每弹出一个顶点，都遍历neighbor，在新路径距离小于neighbor原有路径距离的情况下（此时必然neighbor未完成访问），就更新该结点的权重以及pred，并将该值输入heap中

另外，这种输入没有删去原先二叉堆中的冗余结点。冗余结点距离更长，一定会在这个结点最短路径弹出后面，为此，只要确定visited当中存在这个结点，则不用进行后续操作

```
import heapq
import sys

class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}
        self.distance = sys.maxsize
        self.pred = None

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def getConnections(self):
        return self.connectedTo.keys()

    def getWeight(self, nbr):
        return self.connectedTo[nbr]

    def __lt__(self, other):
        return self.distance < other.distance

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0
```

```
    def addVertex(self, key):
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        self.numVertices += 1
        return newVertex

    def getVertex(self, n):
        return self.vertList.get(n)

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            self.addVertex(f)
        if t not in self.vertList:
            self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)

def dijkstra(start):
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()

    while pq:
        currentDist, currentVert = heapq.heappop(pq)     # 当一个顶点的最短路径确定后（也就是这个顶点
                                                         # 从优先队列中被弹出时），它的最短路径不会再
        if currentVert in visited:
            continue
        visited.add(currentVert)

        for nextVert in currentVert.getConnections():
            newDist = currentDist + currentVert.getWeight(nextVert)
            if newDist < nextVert.distance:
                nextVert.distance = newDist
                nextVert.pred = currentVert
                heapq.heappush(pq, (newDist, nextVert))

# 创建图和边
g = Graph()
g.addEdge('A', 'B', 4)
g.addEdge('A', 'C', 2)
g.addEdge('C', 'B', 1)
g.addEdge('B', 'D', 2)
g.addEdge('C', 'D', 5)
g.addEdge('D', 'E', 3)
g.addEdge('E', 'F', 1)
g.addEdge('D', 'F', 6)

# 执行 Dijkstra 算法
print("Shortest Path Tree:")
dijkstra(g, g.getVertex('A'))

# 输出最短路径树的顶点及其距离
for vertex in g.vertList.values():
    print(f"Vertex: {vertex.id}, Distance: {vertex.distance}")
```

```
# 输出最短路径到每个顶点
def printPath(vert):
    if vert.pred:
        printPath(vert.pred)
        print(" -> ", end="")
    print(vert.id, end="")

print("\nPaths from Start Vertex 'A':")
for vertex in g.vertList.values():
    print(f"Path to {vertex.id}: ", end="")
    printPath(vertex)
    print(", Distance: ", vertex.distance)

"""
Shortest Path Tree:
Vertex: A, Distance: 0
Vertex: B, Distance: 3
Vertex: C, Distance: 2
Vertex: D, Distance: 5
Vertex: E, Distance: 8
Vertex: F, Distance: 9

Paths from Start Vertex 'A':
Path to A: A, Distance:  0
Path to B: A -> C -> B, Distance:  3
Path to C: A -> C, Distance:  2
Path to D: A -> C -> B -> D, Distance:  5
Path to E: A -> C -> B -> D -> E, Distance:  8
Path to F: A -> C -> B -> D -> E -> F, Distance:  9
"""
```

## 3.时间复杂度分析

在不适用任何队列优化的情况下，Dijkstra算法的时间复杂度为O(V^2)，这是因为对于每个节点；在使用优先队列（如）最小堆来操作时，可以将时间复杂度优化到O((V+E)logV)

## 4.补充1：Bellman-Ford，SPFA算法

图论中，除了Dijkstra算法外，Bellman-Ford算法也可以用于求解最短路径问题。如果图中不存在负权边，并且只需要求解单源最短路径，Dijkstra算法可以胜任；如果图中存在负权边或者需要检测负权回路，可以使用Bellman-Ford算法

### 4.1 基本定义：Bellman-Ford算法

Bellman-Ford算法用于解决单源最短路径问题。与Dijkstra算法不同，它可以处理带有负权边的图。算法的基本思想是通过松弛操作逐步更新节点的最短路径估计值，直到收敛的最终结果，具体步骤如下：

1. 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为0，其他节点的距离为无穷大；

2. 进行V-1次循环（V是图中的节点数），每次循环对所有边进行松弛操作：如果节点u到节点v的路径经过节点u的距离加上边（u，v）的权重比当前已知的最短路径更短，则更新最短路径
3. 检查是否存在负权回路。如果在V-1次循环中，仍然可以通过松弛操作更新最短路径，说明存在负权回路，因此无法确定最短路径（具体做法是遍历所有的边，如果发现如2中可以更新distance的情况，则说明存在负权回路，无法确定最短路径）

**4.2 时间复杂度：O(V*E)**

**4.3 优化：SPFA算法（Shortest Path Faster Algorithm）**

SPFA算法（最短路径快速算法）是对Bellman-Ford算法的优化，通过引入一个队列来避免对所有边进行松弛操作，从而提高算法效率

注：SPFA算法无法处理存在负权环路的情况，但适用于稀疏图和存在负权边的情况

基本步骤：

1. 初始化源节点的最短距离为0，其他节点的最短距离为无穷大
2. 将源节点加入队列中
3. 循环执行以下步骤直到队列为空：

（1）从队列中取出一个节点作为当前节点；

（2）遍历当前节点的所有邻接节点：如果经过当前节点到达该邻接节点的路径比当前记录的最短路径更短，则更新最短路径。在该邻接节点不在当前队列中的情况下，将该邻接节点加入队列中

4. 当队列为空时，算法结束，所有节点的最短路径已计算出来

## 5.补充2：多源最短路径Floyd-Warshall算法

求解所有顶点之间的最短路径可以使用Floyd-Warshall算法，它是一种多源最短路径算法。其基本思想是通过一个二维数组来存储任意两个顶点之间的最短距离。初始时，这个数组包含图中各个顶点之间的直接边的权重（包括自己到自己），对于不直接相连的顶点，权重为无穷大。然后，通过迭代更新这个数组，逐步求得所有顶点之间的最短路径。这本质是一种动态规划的算法

Floyd-Warshall算法可以处理负权边和负权回路

**5.1 具体步骤**

1. 初始化一个二维数组dist，用于存储任意两个顶点之间的最短距离，并更新直接边的权重
2. 对于每个顶点k，在更新dist数组时，考虑顶点k作为中间节点的情况。遍历所有的顶点对（i，j），如果通过顶点k可以使得从顶点i到顶点j的路径变短，则更新dist[i][j]为更小的值
3. 重复上述步骤，迭代更新。最终dist存储的就是所有顶点之间的最短路径

**5.2 时间复杂度：O(V^3)**

**5.3 代码实现**

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]

    for k in range(n): #一定要先遍历k
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

# 四、最小生成树（MSTs）

## （一）Prim算法：用于找到连接所有顶点的最小生成树

**1.基本介绍**

Prim算法与BFS算法也有一些相似之处，均为图的遍历算法，也都从一个起始顶点开始逐步扩展，并以某种方式记录已经访问过的顶点。

然而，Prim算法是一种用于解决最小生成树MST问题的贪心算法，其会逐步构建一个包含所有顶点的树，并且使得树的**边权重之和**最小：Prim算法通过选择具有最小权重的边来扩展生成树，并且只考虑与当前生成树相邻的顶点

Prim算法常用于解决互联网广播服务等需要高效把信息传递给所有人的方法，本质上是贪婪算法

**2.定义：最小生成树MST（Minimum Spanning Tree）**

对于图G = (V,E)，最小生成树MST是E的无环子集，在连接V中的所有顶点的同时，使得T中边集合的权重之和最小（每条路径按照路径上的给定权重加权）

**3.代码实现**

```
import sys
import heapq

class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}
        self.distance = sys.maxsize
        self.pred = None
```

```
    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def getConnections(self):
        return self.connectedTo.keys()

    def getWeight(self, nbr):
        return self.connectedTo[nbr]

    def __lt__(self, other):
        return self.distance < other.distance

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        self.numVertices += 1
        return newVertex

    def getVertex(self, n):
        return self.vertList.get(n)

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            self.addVertex(f)
        if t not in self.vertList:
            self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)
        self.vertList[t].addNeighbor(self.vertList[f], cost)

def prim(graph, start):
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()

    while pq:
        currentDist, currentVert = heapq.heappop(pq)
        if currentVert in visited:
            continue
        visited.add(currentVert)

        for nextVert in currentVert.getConnections():
            weight = currentVert.getWeight(nextVert)
            if nextVert not in visited and weight < nextVert.distance:
                nextVert.distance = weight
                nextVert.pred = currentVert
                heapq.heappush(pq, (weight, nextVert))

# 创建图和边
g = Graph()
```

```
g.addEdge('A', 'B', 4)
g.addEdge('A', 'C', 3)
g.addEdge('C', 'B', 1)
g.addEdge('C', 'D', 2)
g.addEdge('D', 'B', 5)
g.addEdge('D', 'E', 6)

# 执行 Prim 算法
print("Minimum Spanning Tree:")
prim(g, g.getVertex('A'))

# 输出最小生成树的边
for vertex in g.vertList.values():
    if vertex.pred:
        print(f"{vertex.pred.id} -> {vertex.id} Weight:{vertex.distance}")

"""
Minimum Spanning Tree:
C -> B Weight:1
A -> C Weight:3
C -> D Weight:2
D -> E Weight:6
"""
```

本质上，prim算法寻找的是下一个与MST中任意顶点相距最近的顶点，而Dijkstra算法寻找的是下一个离给定顶点（单源，或者理解为根节点）最近的顶点

## （二）Kruskal算法

Kruskal算法是一种类似于并查集的解决MST问题的贪心算法。它通过不断选择具有最小权重的边，并确保选择的边不形成环，最终构建出一个包含所有顶点的最小生成树MST

在Kruskal算法中，通常会使用并查集来维护图中顶点的连通性信息。当选择一条边时，通过并查集判断该边的两个端点是否属于同一个连通分量，以避免形成环

注：Kruskal算法一般应用于**无向图**中

**1.Kruskal算法的基本步骤**

1. 将图中的所有边按照权重从小到大进行排序
2. 初始化一个空的边集，用于存储MST的边
3. 重复以下步骤，直到边集中的边数等于顶点数减1或者所有边都已经考虑完毕：

（1）选择排序后的边集中权重最小的边 （2）如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中

4. 返回最小生成树的边集作为结果

**2.代码实现**

```
class DisjointSet:
    def __init__(self, num_vertices):
        self.parent = list(range(num_vertices))
        self.rank = [0] * num_vertices

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

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            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:
                self.parent[root_x] = root_y
                self.rank[root_y] += 1


def kruskal(graph):
    num_vertices = len(graph)
    edges = []

    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))

    # 按照权重排序
    edges.sort(key=lambda x: x[2])

    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)

    # 构建最小生成树的边集
    minimum_spanning_tree = []

    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))

    return minimum_spanning_tree
```

**3.时间复杂度：单纯的Kruskal算法（不包括边解析）为O(ElogE)**

并查集操作的时间复杂度近似为O(E)

# 五、题目练习

## （一）sy403：先导课程

**1.问题描述**

现有n门课程（假设课程编号为从 `0` 到 `n-1` ），课程之间有依赖关系，即可能存在两门课程，必须学完其中一门才能学另一门。现在给出个依赖关系，问能否把所有课程都学完。

注：能同时学习多门课程时总是先学习编号最小的课程。

**输入**

第一行两个整数n、m，分别表示顶点数、边数；

接下来m行，每行两个整数u、v，表示一条边的起点和终点的编号。数据保证不会有重边。

**输出**

如果能学完所有课程，那么输出一行 `Yes` ，然后在第二行输出学习课程编号的顺序，编号之间用空格隔开，行末不允许有多余的空格；如果不能学完所有课程，那么输出一行 `No` ，然后在第二行输出不能学习的课程门数。

**2.代码实现（拓扑排序）**

```
from collections import defaultdict

def courseSchedule(n, edges):
    graph = defaultdict(list)
    indegree = [0] * n
    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    queue = [i for i in range(n) if indegree[i] == 0]
    queue.sort()
    result = []

    while queue:
        u = queue.pop(0)
        result.append(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
        queue.sort()

    if len(result) == n:
        return "Yes", result
    else:
        return "No", n - len(result)
```

```
n, m = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]
res, courses = courseSchedule(n, edges)
print(res)
if res == "Yes":
    print(*courses)
else:
    print(courses)
```

该代码使用拓扑排序中的Kahn算法解决问题。为了保证拓扑排序保证顺序，每次循环后都对queue进行排序，以保证每次弹出的都是度为0的最小节点

## （二）sy386：最短距离

**1.问题描述**

现有一个共n个顶点（代表城市）、m条边（代表道路）的无向图（假设顶点编号为从 `0` 到 `n-1` ），每条边有各自的边权，代表两个城市之间的距离。求从s号城市出发到达t号城市的最短距离。

**输入**

第一行四个整数n、m、s、t，分别表示顶点数、边数、起始编号、终点编号；

接下来m行，每行三个整数u、v、w，表示一条边的两个端点的编号及边权距离。数据保证不会有重边。

**输出**

输出一个整数，表示最短距离。如果无法到达，那么输出 `-1` 。

**2.代码实现**

采用Dijkstra算法即可

```
import heapq

def dijkstra(n, edges, s, t):
    graph = [[] for _ in range(n)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    pq = [(0, s)]  # (distance, node)
    visited = set()
    distances = [float('inf')] * n
    distances[s] = 0

    while pq:
        dist, node = heapq.heappop(pq)
        if node == t:
            return dist
```

```
            if node in visited:
                continue
            visited.add(node)
            for neighbor, weight in graph[node]:
                if neighbor not in visited:
                    new_dist = dist + weight
                    if new_dist < distances[neighbor]:
                        distances[neighbor] = new_dist
                        heapq.heappush(pq, (new_dist, neighbor))
    return -1

# Read input
n, m, s, t = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]

# Solve the problem and print the result
result = dijkstra(n, edges, s, t)
print(result)
```

## （三）sy396：最小生成树-Prim算法

**1.问题描述**

现有一个共n个顶点、m条边的无向图（假设顶点编号为从 `0` 到 `n-1` ），每条边有各自的边权。在图中寻找一棵树，使得这棵树包含图上所有顶点、所有边都是图上的边，且树上所有边的边权之和最小。使用Prim算法求出这个边权之和的最小值。

**输入**

第一行两个整数n、m，分别表示顶点数、边数；

接下来m行，每行三个整数u、v、w，表示一条边的两个端点的编号及边权距离。数据保证不会有重边。

**输出**

输出一个整数，表示最小的边权之和。如果不存在这样的树，那么输出 `-1` 。

**2.代码实现**

```
import heapq

def prim(graph, n):
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, vertex)
    min_spanning_tree_cost = 0

    while min_heap:
        weight, vertex = heapq.heappop(min_heap)

        if visited[vertex]:
            continue
```

```
            visited[vertex] = True
            min_spanning_tree_cost += weight

            for neighbor, neighbor_weight in graph[vertex]:
                if not visited[neighbor]:
                    heapq.heappush(min_heap, (neighbor_weight, neighbor))

        return min_spanning_tree_cost if all(visited) else -1

    def main():
        n, m = map(int, input().split())
        graph = [[] for _ in range(n)]

        for _ in range(m):
            u, v, w = map(int, input().split())
            graph[u].append((v, w))
            graph[v].append((u, w))

        min_spanning_tree_cost = prim(graph, n)
        print(min_spanning_tree_cost)

    if __name__ == "__main__":
        main()
```

## （四）最小生成树-Kruskal算法

**1.问题描述**

现有一个共n个顶点、m条边的无向图（假设顶点编号为从 0 到 n-1 ），每条边有各自的边权。在图中寻找一棵树，使得这棵树包含图上所有顶点、所有边都是图上的边，且树上所有边的边权之和最小。使用Kruskal算法求出这个边权之和的最小值。

**输入**

第一行两个整数n、m，分别表示顶点数、边数；

接下来m行，每行三个整数u、v、w，表示一条边的两个端点的编号及边权。数据保证不会有重边。

**输出**

输出一个整数，表示最小的边权之和。 如果不存在这样的树，那么输出 -1 。

**2.代码实现**

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

        def find(self, x):
```

```
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[px] = py
            if self.rank[px] == self.rank[py]:
                self.rank[py] += 1

def kruskal(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])
    res = 0
    for u, v, w in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            res += w
    if len(set(uf.find(i) for i in range(n))) > 1:
        return -1
    return res

n, m = map(int, input().split())
edges = []
for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((u, v, w))
print(kruskal(n, edges))
```