图的问题

5.1 图的应用背景

图结构在计算机中应用广泛，在前面已经对图的基本概念、存储结构和基本操作等进行了介绍，本章介绍关于图的应用的各种算法。图的应用范围广，可以对图进行遍历，逐一访问图的每一个顶点，后面将介绍图的两种遍历算法——广度优先遍历和深度优先遍历。同时也可以利用图的遍历解决数学上常见的八数码问题，求出图的最小生成树，在实际应用中，比如多个场地互相连接，如何以最少路程访问到所有场地节点，这就是图的最小生成树问题，5.3 节将详细介绍图的最小生成树问题。图还可以解决关键路径问题，例如做一项任务，任务分为多个阶段，每个阶段之间需要时间，怎么操作才能使任务完成而且时间最短，这就是关键路径问题。图还有最短路径问题，分为单源和多源最短路径，解决的是从某地到某地有多种路径，求哪一条路径最短的问题，5.5 节将详细介绍这一问题。

5.2 图的遍历问题

5.2.1 什么是图的遍历

图的遍历是图的基本的操作之一。在形式上，图遍历是系统化的检查图的所有边和顶点的步骤。图的遍历是指从图中某一顶点出发，对图中所有顶点访问一次且仅访问一次。图的遍历操作和树的遍历操作类似，但由于图结构本身的复杂度，所以图的遍历操作也比较复杂。在图的遍历中要解决的关键问题有以下几点。

(1) 在图中，没有一个确定的开始顶点，任意一个顶点都可以作为遍历的起始顶点，一般从编号小的顶点开始选取遍历的起始顶点。在线性表中，数据元素在表中的编号就是元素在序列中的位置，因而其编号是唯一的。在树中，将节点按层序编号，由于树具有层次性，因而其层序编号也是唯一的。在图中，任何两个顶点之间都可能存在边，顶点是没有确定的先后次序的，所以，顶点的编号不唯一。

(2) 从某个顶点出发可能到达不了所有其他顶点，例如非连通图，从一个顶点出发，只能访问它所在的连通分量上的所有顶点。那么，如何才能遍历图的所有顶点？解决方案是多次调用从某顶点出发遍历图的算法。

(3) 由于图中可能存在回路，某些顶点可能会被重复访问，那么，如何避免遍历不会因回路而陷入循环？解决方案是：附设访问标志数组 visited[n]。

(4) 在图中，一个顶点可以和其他多个顶点相邻接，当这样的顶点被访问过后，如何选取下一个要访问的顶点？解决方案是：对图进行深度优先遍历和广度(宽度)优先遍历。

5.2.2　图的广度(宽度)优先遍历算法

图的广度优先遍历(BFS)类似于树的层序遍历。图的广度优先搜索的基本思想如下：

(1) 访问顶点 v。

(2) 依次访问 v 的各个未被访问的邻接点 $v_1, v_2, \cdots, v_k$。

(3) 分别从 $v_1, v_2, \cdots, v_k$ 出发依次访问它们未被访问的邻接点，并使“先被访问顶点的邻接点”先于“后被访问顶点的邻接点”被访问。直至图中所有与顶点 v 有路径相通的顶点都被访问到。

图的广度优先遍历使用队列实现，设由 $v_1, v_2, \cdots, v_8$ 共 8 个顶点组成了一个无向图，对这个图进行广度优先遍历的实现步骤如图 5-1 所示。

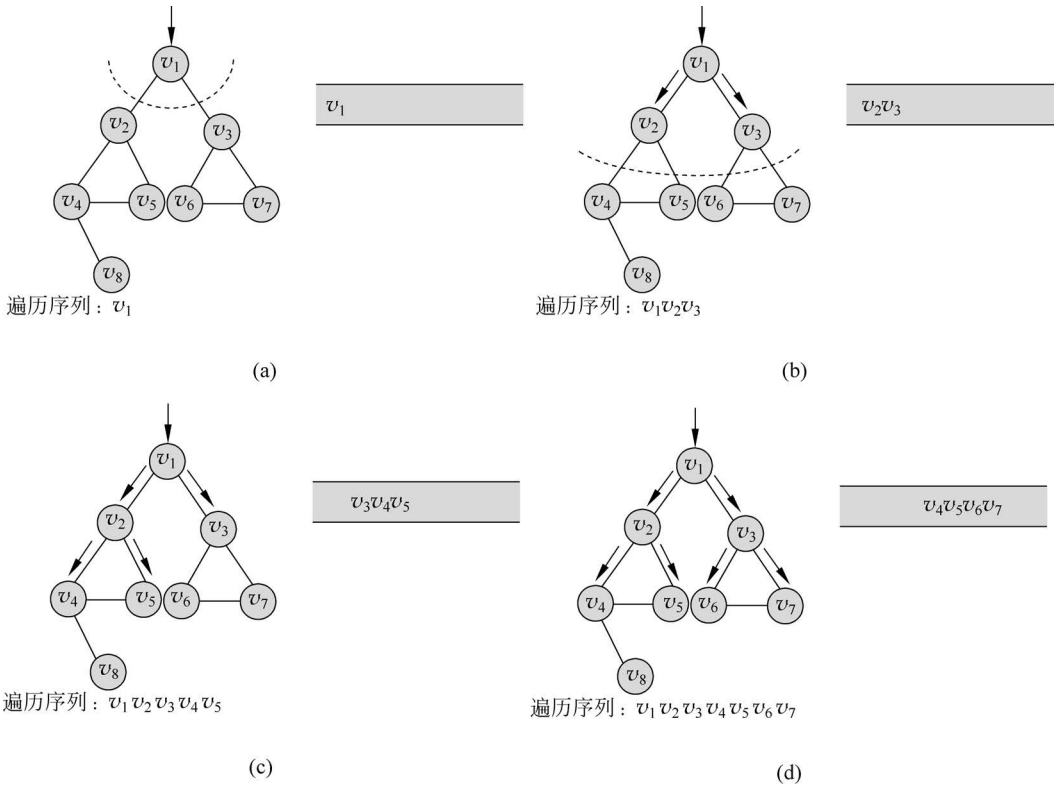
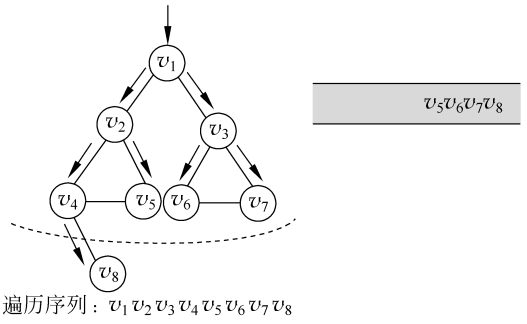


图 5-1　广度优先遍历



(e)

图 5-1 （续）

广度优先遍历算法的伪代码描述如下。

输入：图的每个顶点的编号 v
输出：图的广度优先遍历序列
1. 队列 Q 初始化；
2. 访问顶点 v；修改标志 visited $[v]=1$；顶点 v 入队列 Q；
3. 队列 Q 非空
    3.1 v = 队列 Q 的队头元素出队；
    3.2 w = 顶点 v 的第一个邻接点；
    3.3 w 存在
        3.3.1 如果 w 未被访问，则访问顶点 w；修改标志 visited$[w]$ = 1；顶点 w 入队列 Q；
        3.3.2 w=顶点 v 的下一个邻接点；

广度优先遍历的算法实现如下：

```
def BFTraverse(v):
      front = -1, rear = -1                    # 初始化队列
      print(vertex[v])
visited[v] = 1
Q[++rear] = v                                  # 被访问顶点入队
      while front != rear:                     # 当队列非空时
            w = Q[++front]                     # 将队头元素出队并送到 v 中
            for j in vertex:
                if edge[w][j] == 1 && visited[j] == 0:
                    print(vertex[j])
visited[j] = 1
Q[++rear] = j
```

在广度优先的遍历中每个顶点都要进(出)一次列队且仅仅一下(类似于深度优先遍历的进栈)，对于每一个顶点 v 出列队后，要访问的所有邻接点，时间为 $O(n)$，因此可知广度优先遍历时间复杂度是为 $O(n^2)$或 $O(n+e)$，其中 e 和 v 分别是图的边集和点集。

5.2.3 图的深度优先遍历算法

深度优先遍历(DFS)相当于二叉树的先序遍历。从顶点 v 出发进行深度优先遍历的基本思想是(深度优先遍历是一个递归的过程)：

(1) 访问顶点 v；

（2）从 v 的未被访问的邻接点中选取一个顶点 w，从 w 出发进行深度优先遍历；

（3）重复上述两步，直至访问所有和 v 有路径相通的顶点。

图的深度优先遍历使用栈实现，设由 v_1，v_2，…，v_8 共 8 个顶点组成了一个无向图，对这个图进行深度优先遍历的实现步骤如图 5-2 所示。

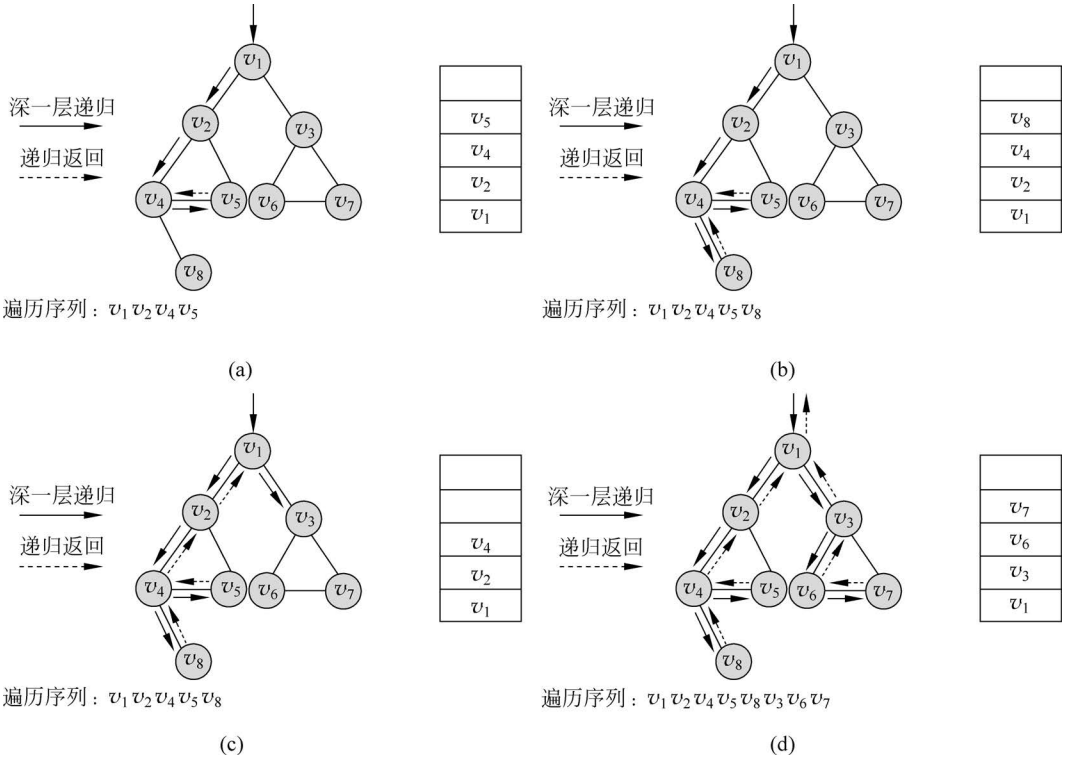


图 5-2　深度优先遍历

深度优先遍历伪代码描述如下：

```
输入：顶点的编号 v
输出：无
1. 访问顶点 v，修改标志 visited[v] = 1；
2. w = 顶点 v 的第一个邻接点；
3. while（w 存在）
    3.1 if（w 未被访问）从顶点 w 出发递归执行该算法；
    3.2 w = 顶点 v 的下一个邻接点。
```

深度优先遍历的算法实现如下：

```
def DFTraverse(v):
    print(vertex[v])
    visited[v] = 1
    for j in vertex:
        if edge[v][j] == 1 && visited[j] == 0:
        DFTraverse( j )
```

在深度优先遍历中使用了递归的方法，该算法的时间复杂度为 $O(n^2)$。

5.2.4 图遍历问题的应用案例

图遍历可以应用在生活中的许多方面。在实际应用中，可以将图遍历应用在求解八数码问题上。什么是八数码问题？八数码问题又称九宫格问题，在一个 3×3 的棋盘上，有 9 个盘位，其中有 8 个棋子和一个空盘位，每个棋子上的数字是 1～8 中的一个数，且棋盘上的数字不能重复，为了方便运算，将空盘位设置为 0，这就组成了 0～8 的整数所组成的 3×3 的棋盘阵列。每个带有数字的棋盘能够移动到空盘位上。问题给定棋盘的初始状态和目标状态，其两种状态示例如图 5-3 所示，需要解决的问题是在给定的初始化状态和目标布局状态下，找到一种能够实现从初始状态到目标状态步骤最少的移动方法。

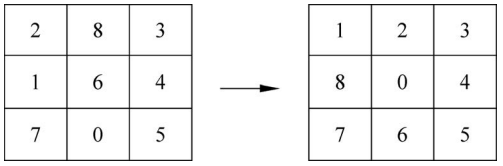


图 5-3　八数码问题初始状体和目标状态

可以将这个棋盘理解为一个图，其中空棋盘(0 位置)和与之相邻的棋盘上的数字存在边，假设设定初始状态为{2,8,3,1,6,4,7,0,5}，目标状态为{1,2,3,8,0,4,7,6,5}，可以使用广度优先遍历和深度优先遍历求解此问题，其问题的求解步骤示例如图 5-4 所示。

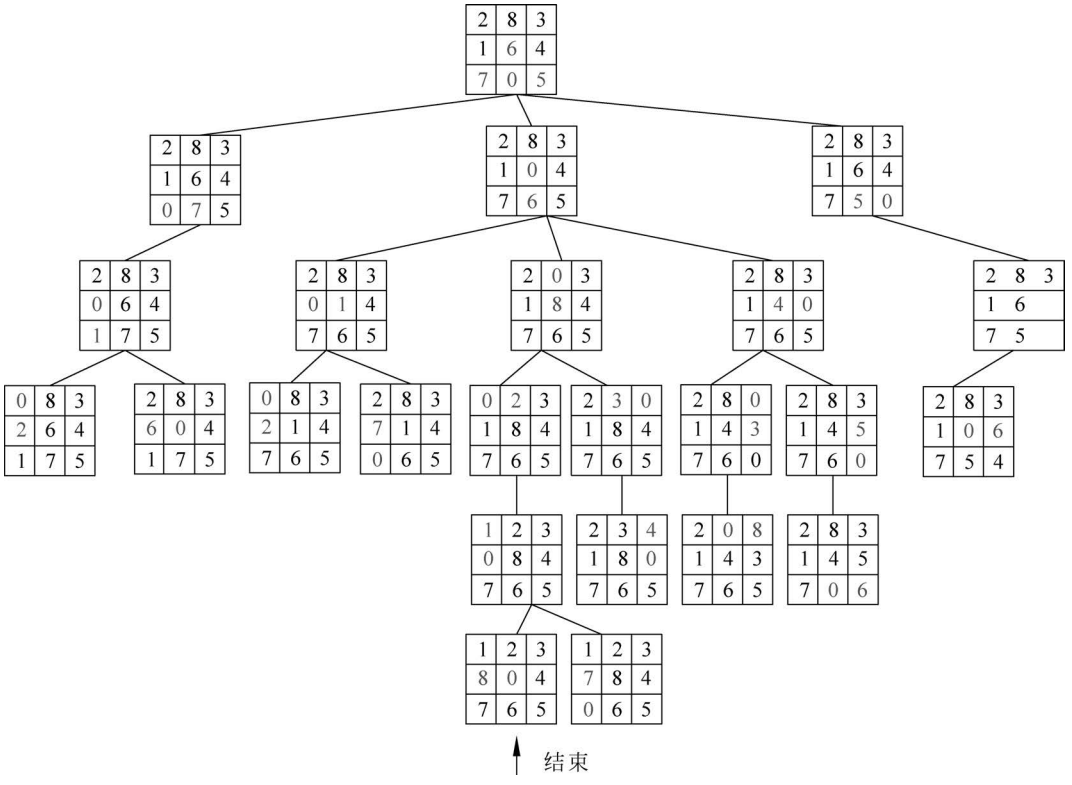


图 5-4　八数码问题示例

可以通过树的基本知识得到八数码问题的数字移动步骤。

5.3 最小生成树问题

5.3.1 什么是最小生成树

5.1节曾提到，图可以通过某些算法生成树的结构，本节将介绍最小生成树的概念和性质。

设$G=(V,E)$是一个有权值的无向连通网，由图G的所有顶点E组成的树称为图的生成树，生成树上各边的权值之和称为该生成树的代价。在图G的所有生成树中，代价最小的生成树称为最小生成树(MST)。最小生成树的概念可以应用到生活中大量的实际问题中。如在n个城市之间建造通信网络，由图的性质可知，至少要架设$n-1$条通信线路，但是每两个城市之间架设通信线路的价格是不一样的，如何设计才能使得总造价最低就是一个最小生成树问题。

最小生成树的性质是：假设$G=(V,E)$是一个带有权值的无向连通网，U是顶点集V的一个非空子集。若(U,V)是一条具有最小权值的边，其中$u\in U$，$v\in V-U$，则必定存在一棵包含边(U,V)的最小生成树。

通过相关算法可以计算出G的最小生成树。最小生成树的算法主要有两种，分别是普里姆(Prim)算法和克鲁斯卡尔(Kruskal)算法，它们是利用MST性质构造最小生成树的两个经典算法。

5.3.2 克鲁斯卡尔算法

克鲁斯卡尔算法的基本思想是：设无向连通网为$G=(V,E)$，令G的最小生成树为$T=(U,\mathrm{TE})$，其初态为$U=V$，$\mathrm{TE}=\{\ \}$，然后，按照边的权值由小到大的顺序，考查G的边集E中的各条边。若被考查的边的两个顶点属于T的两个不同的连通分量，则将此边作为最小生成树的边加入到T中，同时把两个连通分量连接为一个连通分量；若被考查边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路；如此下去，当T中的连通分量个数为1时，此连通分量便为G的一棵最小生成树。

通过一个例子熟悉该算法的执行过程。设有一个无向连通图G，G中的顶点有$\{A,B,C,D,E,F\}$，其初始形状和克鲁斯卡尔算法生成最小生成树的执行过程如图5-5所示。

克鲁斯卡尔算法的伪代码描述如下：

```
1.初始化:U=V;TE={ };
2.重复下述操作直到 T 中的连通分量个数为 1:
    2.1 在 E 中寻找最短边(u,v);
    2.2 如果顶点 u,v 位于 T 的两个不同连通分量,则
        (1)将边(u,v)并入 TE;
        (2)将这两个连通分量合为一个;
    2.3 标记边(u,v),使得(u,v)不参加后续最短边的选取。
```

克鲁斯卡尔算法实现如下：

```
def Kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)
```

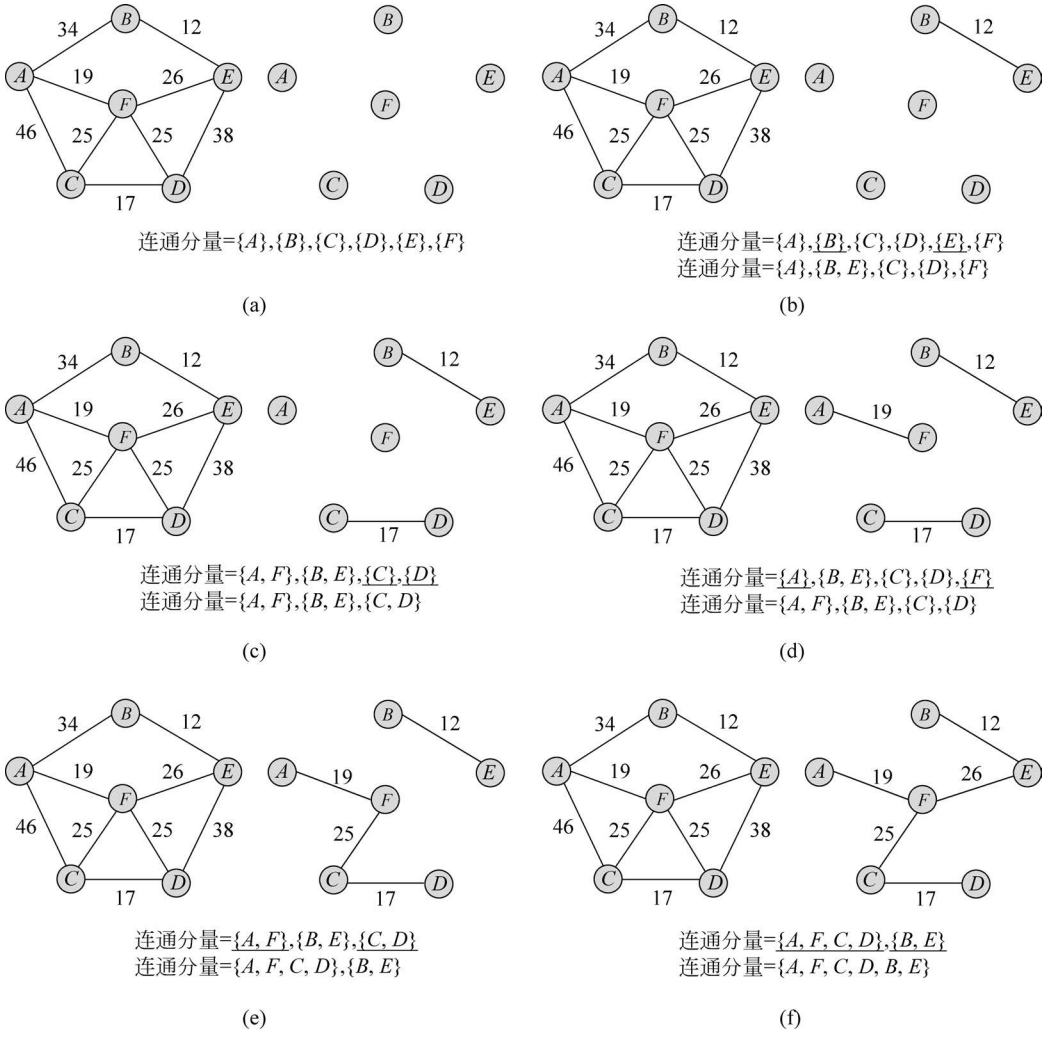


图 5-5　克鲁斯卡尔算法案例

```
mst = set()
edges = list(graph['edges'])
edges.sort()
for edge in edges:
    weight, v1, v2 = edge
    if find(v1) != find(v2):
        merge(v1 , v2)
        mst.add(edge)
return mst
```

克鲁斯卡尔算法的时间复杂度为 $O(|e|\log|v|)$，其中 e 和 v 分别是图的边集和点集。

5.3.3　普里姆算法

普里姆算法的基本思想是：设 $G=(V,E)$是具有 n 个顶点的连通网，$T=(U,\mathrm{TE})$是 G 的最小生成树，T 的初始状态为 $U=\{v_0\}(v_0\in V)$，$\mathrm{TE}=\{\ \}$，重复执行下述操作：在所有 $u\in U, v\in V-U$ 的边中找一条代价最小的边(u,v)并入集合 TE，同时 v 并入 U，直至 $U=$

V。此时 TE 中必须有 $n-1$ 条边，则 T 就是一棵最小生成树。

通过一个例子熟悉一下该算法的执行过程。设有一个无向连通图 G，G 中的顶点有 $\{A,B,C,D,E,F\}$，其初始形状和普里姆算法生成最小生成树的执行过程如图 5-6 所示。

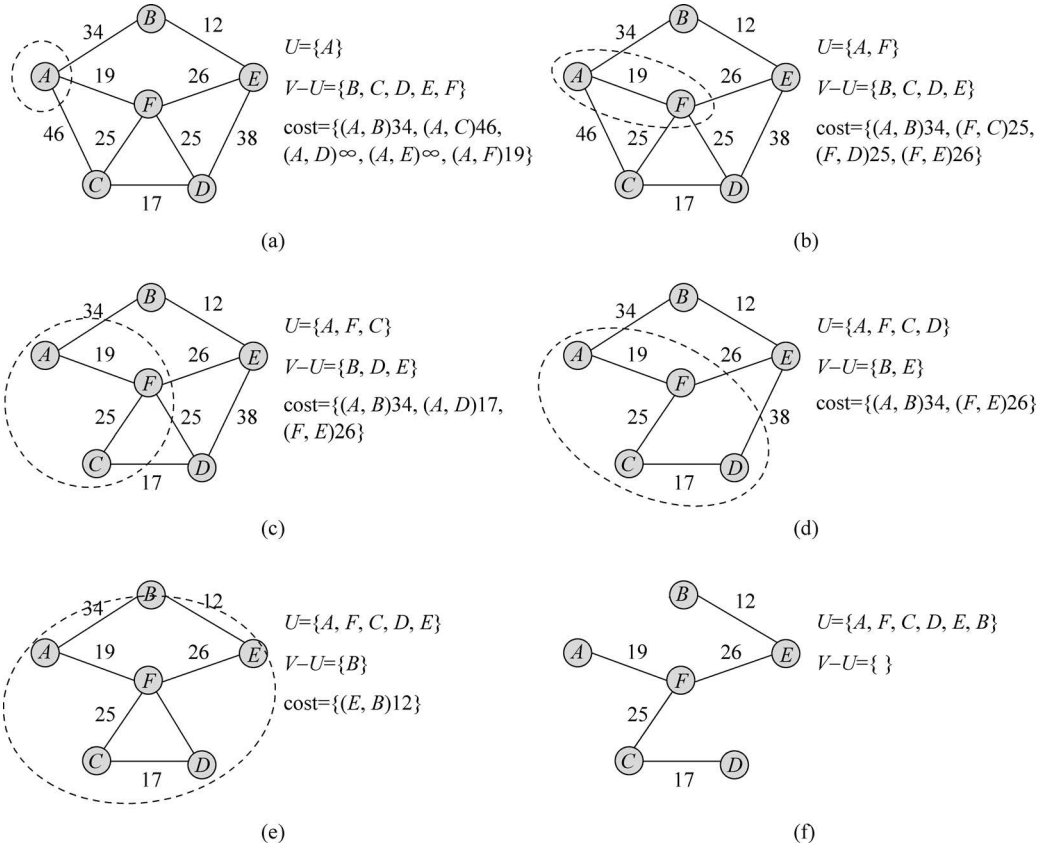


图 5-6　普里姆算法案例

普里姆算法的伪代码描述如下：

1. 初始化：$U=\{v_0\}$；TE=$\{\ \}$；
2. 重复下述操作直到 $U=V$：
   2.1 在 E 中寻找最短边(u,v)，且满足 $u\in U, v\in V-U$；
   2.2 $U=U+\{v\}$；
   2.3 TE = TE + $\{(u,v)\}$；关键是如何找到连接 U 和 $V-U$ 的最短边。

普里姆算法的算法实现如下：

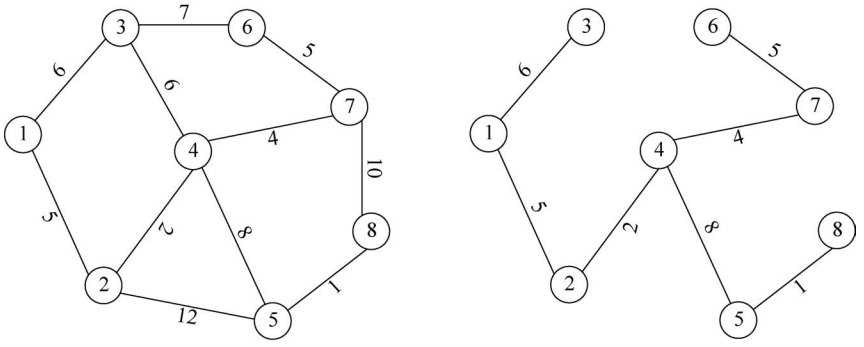
```
def Prim(vertexs, edges, start_node):
    adjacent_vertex = defaultdict(list)
    for v1, v2, length in edges:
        adjacent_vertex[v1].append((length, v1, v2))
        adjacent_vertex[v2].append((length, v2, v1))
    mst = []
    closed = set(start_node)
    adjacent_vertexs_edges = adjacent_vertex[start_node]
    heapify(adjacent_vertexs_edges)
    while adjacent_vertexs_edges:
```

```
            w, v1, v2 = heappop(adjacent_vertexs_edges)
            if v2 not in closed:
                closed.add(v2)
                mst.append((v1, v2, w))
                for next_vertex in adjacent_vertex[v2]:
                    if next_vertex[2] not in closed:
                        heappush(adjacent_vertexs_edges, next_vertex)
    return mst
```

普里姆算法的时间复杂度与网中的边数无关。可通过邻接矩阵图表示的简易实现，找到所有最小权边共需的运行时间。若使用简单的二叉堆与邻接表来表示，则普里姆算法的运行时间则可缩短为 $O(e\log v)$，其中 e 为连通图的边数，v 为顶点数。如果使用较为复杂的斐波那契堆，则可将运行时间进一步缩短为 $O(e+v\log v)$，这在连通图足够密集时，可较显著地提高运行速度。读者可以尝试实现这些结构的普利姆算法。

5.3.4 最小生成树问题的应用案例

最小生成树（MST）是图论中的基本问题，在实际中应用广泛，在数学建模中也经常出现。路线设计、道路规划、官网布局、公交路线、网络设计，都可以转化为最小生成树问题，如要求总线路长度最短、材料最少、成本最低、耗时最短。下面介绍一个使用最小生成树的案例。某市区有 7 个小区需要铺设天然气管道，各小区的位置及可能的管道路线、费用如图所示，要求设计一个管道铺设路线，使天然气能输送到各小区，且铺设管道的总费用最低。小区位置和之间线路费用及使用算法实现的总费用最小的铺设方案如图 5-7 所示。



(a) 原始　　(b) 生成的最小生成树

图 5-7　最小生成树案例

其代码实现如下：

```
import numpy as np
import matplotlib.pyplot as plt                    # 导入 Matplotlib 工具包
import networkx as nx                              # 导入 NetworkX 工具包
G1 = nx.Graph()                                    # 创建:空的无向图
G1.add_weighted_edges_from([(1,2,5),(1,3,6),(2,4,2),(2,5,12),(3,4,6),
                (3,6,7),(4,5,8),(4,7,4),(5,8,1),(6,7,5),(7,8,10)])   #向图中添加多条
#赋权边: (node1,node2,weight)
T = nx.minimum_spanning_tree(G1)                   # 返回包括最小生成树的图
print(T.nodes)                                     # 最小生成树的顶点
print(T.edges)                                     # 最小生成树的边
print(sorted(T.edges))                             # 排序后的最小生成树的边
```

```
print(sorted(T.edges(data = True)))          # data = True 表示返回值包括边的权值

mst1 = nx.tree.minimum_spanning_edges(G1, algorithm = "kruskal")
#返回最小生成树的边
print(list(mst1))                            # 最小生成树的边
mst2 = nx.tree.minimum_spanning_edges(G1, algorithm = "prim",data = False)
# data = False 表示返回值不带权
print(list(mst2))
# 绘图
pos = {1:(1,5),2:(3,1),3:(3,9),4:(5,5),5:(7,1),6:(6,9),7:(8,7),8:(9,4)}
# 指定顶点位置
nx.draw(G1, pos, with_labels = True, node_color = 'c', alpha = 0.8)   # 绘制无向图
labels = nx.get_edge_attributes(G1,'weight')
nx.draw_networkx_edge_labels(G1,pos,edge_labels = labels, font_color = 'm')
# 显示边的权值
nx.draw_networkx_edges(G1,pos,edgelist = T.edges,edge_color = 'b',width = 4)
# 设置指定边的颜色
plt.show()
```

此代码使用了 Python 附加的 NetworkX 工具包和 Matplotlib 工具包创建图和计算最小生成树，其最小生成树代码已经写入相关模块，感兴趣的读者可以查阅分析相关数据，深入探讨相关算法的使用。

5.4 关键路径问题

5.4.1 AOV 网与拓扑排序

在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系。这种以有向图顶点表示活动的网称为 AOV 网。

AOV 网具有如下特点：

(1) AOV 网中的弧表示活动之间存在的某种制约关系；

(2) AOV 网中不能出现回路。

设 $G=(V,E)$是一个具有 n 个顶点的有向图，V 中的顶点序列 $v_1,v_2,\cdots,v_n$ 称为一个拓扑序列，当且仅当满足下列条件：若从顶点 v_i 到 v_j 有一条路径，则在顶点序列中顶点 v_i 必在顶点 v_j 之前。拓扑序列使得 AOV 网中所有应存在的前驱和后继关系都能得到满足。对一个有向图构造拓扑序列的过程称为拓扑排序。

5.4.2 拓扑排序算法

对 AOV 网进行拓扑排序的基本思想如下：

(1) 从 AOV 网中选择一个没有前驱的顶点并且输出。

(2) 从 AOV 网中删去该顶点，并且删去所有以该顶点为尾的弧。

(3) 重复上述两步，直到全部顶点都被输出，或 AOV 网中不存在没有前驱的顶点。

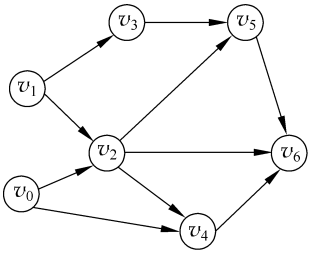
AOV 网拓扑排序后的结果有两种：

(1) AOV 网中全部顶点都被输出，这说明 AOV 网中不存在回路。

(2) AOV 网中顶点未被全部输出，剩余的顶点均不存在没有前驱的顶点，这说明 AOV

网中存在回路。

图 5-8 给出了一个 AOV 网及其拓扑序列。可以看出，对于任何一项工程中各个活动的安排，必须按拓扑序列中的顺序进行才是可行的，并且一个 AOV 网的拓扑序列可能不唯一。拓扑排序的第一个顶点为入度为 0 的点。



(a) 一个AOV网

拓扑序列1：$v_0v_1v_2v_3v_4v_5v_6$
拓扑序列2：$v_0v_1v_3v_2v_4v_5v_6$
拓扑序列3：$v_0v_1v_3v_2v_5v_4v_6$
拓扑序列4：$v_1v_3v_0v_2v_4v_5v_6$
拓扑序列5：$v_1v_3v_0v_2v_5v_4v_6$

(b) 拓扑序列

图 5-8　拓扑排序案例

拓扑排序的伪代码描述如下：

1. 栈 S 初始化；累加器 count 初始化；
2. 扫描顶点表，将没有前驱的顶点压栈；
3. 当栈 S 非空时循环
    3.1 v_j =退出栈顶元素；输出 v_j；累加器加 1；
    3.2 将顶点 v_j 的各个邻接点的入度减 1；
    3.3 将新的入度为 0 的顶点入栈；
4. if (count < vertexNum)，则输出有回路信息。

拓扑排序算法实现如下：

```
def toposort(graph):
    in_degrees = dict((u,0) for u in graph)              #初始化所有顶点入度为0
    vertex_num = len(in_degrees)
    for u in graph:
        for v in graph[u]:
            in_degrees[v] += 1                           #计算每个顶点的入度
    Q = [u for u in in_degrees if in_degrees[u] == 0]    #筛选入度为0的顶点
    Seq = []
    while Q:
        u = Q.pop()                                      #退出栈顶元素
        Seq.append(u)
        for v in graph[u]:
            in_degrees[v] = 1                            #移除其所有指向
            if in_degrees[v] == 0:
                Q.append(v)                              #再次筛选入度为0的顶点入栈
if len(Seq) == vertex_num:
#如果循环结束后存在非0入度的顶点，则说明图中有环，不存在拓扑排序
        return Seq
    else:
        print("there's a circle.")
```

如果 AOV 网络有 n 个顶点、e 条边，那么在拓扑排序的过程中，搜索入度为 0 的顶点所需的时间是 $O(n)$。在正常情况下，每个顶点进一次栈，出一次栈，所需时间 $O(n)$。每个顶点入度减 1 的运算共执行了 e 次。所以总的时间复杂度为 $O(n+e)$。

5.4.3 AOE 网与关键路径

在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，边上的权值表示活动的持续时间，将这样的有向图称为边表示活动的网，简称 AOE 网。AOE 网中没有入边的顶点称为始点(或源点)，没有出边的顶点称为终点(或汇点)。

AOE 网有两个性质：

(1) 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始；

(2) 只有在进入某顶点的各活动都结束后，该顶点所代表的事件才能发生。

图 5-9 所示是一个 AOE 网，事件 v_4 表示活动 a_3 和 a_4 已经结束，活动 a_6 和 a_7 可以开始。

如果用 AOE 网来表示一项工程，那么，仅仅考虑各个活动之间的优先关系还不够，更多的是关心整个工程完成的最短时间是多少，哪些活动的延期将会影响整个工程的进度，加速这些活动是否会提高整个工程的效率。

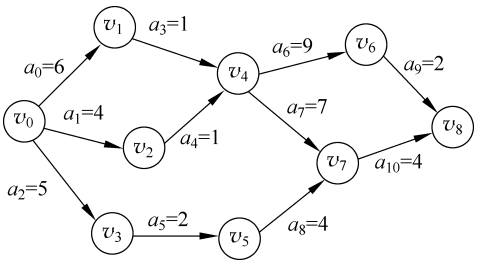


图 5-9 AOE 网举例

由于 AOE 网中的某些活动能够同时进行，故完成整个工程所必须花费的时间应该为始点到终点的最大路径长度(这里的路径长度是指该路径上的各个活动所持续的时间之和)。具有最大路径长度的路径称为关键路径(critical path)，关键路径上的活动称为关键活动(critical activity)。关键路径长度是整个工程所需的最短工期。也就是说，要缩短整个工期，必须加快关键活动的进度。

利用 AOE 网进行工程管理时需解决的主要问题如下：

(1) 完成整个工程至少需要多少时间？

(2) 为缩短完成工程所需的时间，应当加快哪些活动？

关键路径可能不只一条，重要的是找到关键活动，即不按期完成就会影响整个工程完成的活动。首先计算以下与关键活动有关的量。

1. 事件的最早发生时间

事件的最早发生时间 ve[k]是指从始点开始到顶点 v_k 的最大路径长度，这个长度决定了所有从顶点 v_k 发出的活动能够开工的最早时间：

$$\mathrm{ve}[1]=0 \tag{5-1}$$

$$\mathrm{ve}[k]=\max\{\mathrm{ve}[j]+\mathrm{len}<v_j,v_k>\}(<v_j,v_k>\in p[k]) \tag{5-2}$$

其中，$p[k]$表示所有到达 v_k 的有向边的集合。

例如，采用图 5-9 所示的 AOE 网计算事件的最早发生时间，如图 5-10 所示。

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
ve[k]	0	6	4	5	7	7	16	14	18

图 5-10 最早发生时间

2. 事件的最迟发生时间

事件的最迟发生时间 $v1[k]$是指在不推迟整个工期的前提下，事件 v_k 允许的最晚发生时间：

$$vl[n]=ve[n] \tag{5-3}$$

$$vl[k]=\min\{vl[j]-\text{len}<v_k,v_j>\}(<v_k,v_j>\in s[k]) \tag{5-4}$$

其中，$s[k]$为所有从 v_k 发出的有向边的集合。

例如，采用如图 5-9 所示的 AOE 网计算事件的最迟发生时间，如图 5-11 所示。

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
ve[k]	0	6	4	5	7	7	16	14	18
vl[k]	0	6	6	8	7	10	16	14	18

图 5-11　最迟发生时间

3. 活动的最早开始时间

若活动 a_i 是由弧$<v_k,v_j>$表示，则活动 a_i 的最早开始时间 $e[i]$应等于事件 v_k 的最早发生时间，则有

$$e[i]=ve[k] \tag{5-5}$$

例如，采用如图 5-9 所示的 AOE 网计算活动的最早开始时间如图 5-12 所示。

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
e[i]	0	0	0	6	4	5	7	7	7	16	14
l[i]	0	2	3	6	6	8	7	7	10	16	14

图 5-12　活动最早开始时间和最晚开始时间

4. 活动的最晚开始时间

若 a_i 由弧$<v_k,v_j>$表示，则 a_i 的最晚开始时间 $l[i]$要保证事件 v_j 的最晚发生时间不拖后，则有：

$$l[k]=vl[j]-\text{len}<v_k,v_j> \tag{5-6}$$

5.4.4　关键路径问题的应用案例

本节了解一个关键路径问题的实际应用。假定有一建筑工程，包括筹备、签合同、付款、施工做预置件和验收几个步骤，其 AOE 网如图 5-13 所示。引出以下几个问题。

（1）每个活动持续多少时间？

（2）完成整个工程至少需要多少时间？

（3）哪些活动是关键活动？

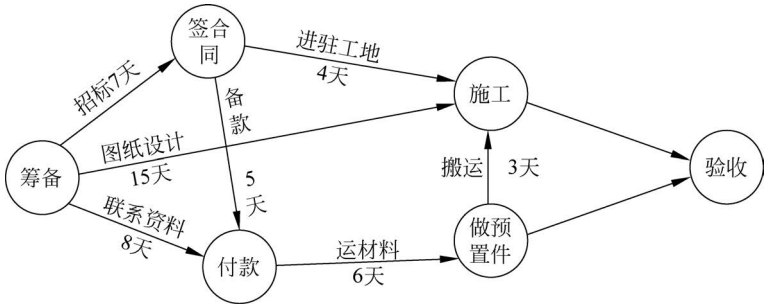


图 5-13　关键路径问题案例 AOE 网

案例的程序实现如下：

```
from queue import Queue,LifoQueue,PriorityQueue
n,m,s,t = map(int,input().split())
G = [[] for i in range(n + 1)]
E = [[] for i in range(n + 1)]
TE = [0 for i in range(n + 1)]
TL = [2100000000 for i in range(n + 1)]
TL[0] = 0
In = [0 for i in range(n + 1)]
Out = [0 for i in range(n + 1)]
que = Queue(maxsize = 0)
que.put(s)
que1 = Queue(maxsize = 0)
que1.put(t)
ans = []
tmp = [0 for i in range(n + 1)]
tmp[0] = s
def dfs(x,step = 1):
    if x == t:
        ag = []
        for i in range(0,step,1):
            ag.append(tmp[i])
        ans.append(ag)
        return
    for (u,w) in G[x]:
        if TE[u] == TL[u] and TE[u] == TE[x] + w:
            tmp[step] = u
            dfs(u,step + 1)
for i in range(m):
    u,v,w = map(int,input().split())
    G[u].append((v,w))
    In[v] = In[v] + 1
    E[v].append((u,w))
    Out[u] = Out[u] + 1

while not que.empty():
    x = que.get()
    for (u,w) in G[x]:
        if TE[u]< TE[x] + w:
            TE[u] = TE[x] + w
        In[u] = In[u] - 1
        if In[u] == 0:
            que.put(u)
TL[t] = TE[t]
while not que1.empty():
    x = que1.get()
    for (u,w) in E[x]:
        if TL[u]> TL[x] - w:
            TL[u] = TL[x] - w
        Out[u] = Out[u] - 1
        if Out[u] == 0:
            que1.put(u)
dfs(s)
print("Dis = %d" % (TE[t]))
for i in range(1, n + 1):
```

```
        print("Node",i, end = "")
        print(": TE = %3d" % (TE[i]),sep = "",end = " ")
    print(" TL = %3d" % (TL[i]),sep = "",end = " ")
        print(" TL - TE = ",TL[i] - TE[i],sep = "")
    print(sorted(ans,key = len))
```

程序的输入内容为：第一行是4个由空格隔开的整数n(节点个数)、m(边数)、s(源点)和t(终点)。此后的m行，每行3个正整数u、v、w表示一条从节点u到节点v的长度为w的边。

输出内容为：第一行输出关键路径的长度；第二行到第 $n+1$ 行输出每一个顶点的TETE、TLTL和缓冲时间；最后一行输出所有的关键路径，完成求解案例的基本问题。

5.5 单源与多源最短路径问题

5.5.1 什么是单源与多源最短路径

在非网图中，最短路径是指两顶点之间经历的边数最少的路径。路径上的第一个顶点称为源点，最后一个顶点称为终点。在网图中，最短路径是指两顶点之间经历的边上权值之和最小的路径。

最短路径问题是图的又一个比较典型的应用问题。例如，给定某公路网的 n 个城市以及这些城市之间相通公路的距离，能否找到城市A到城市B之间一条距离最近的通路呢？如果将城市用顶点表示，城市间的公路用边表示，公路的长度作为边的权值，那么这个问题可归结为在网图中，求顶点 A 到顶点 B 的所有路径中，边的权值之和最少的那条路径。

给定一个带权有向图 $G=(V,E)$，指定图 G 中的某一个顶点的 V 为源点，求出从 V 到其他各顶点之间的最短路径，这个问题称为单源点最短路径问题。如果指定图 G 的多个点，求出从多个点到其他顶点的最短路径问题就称为多源最短路径。

通常使用迪杰斯特拉(Dijkstra)算法寻找单源最短路径，使用一种改进的弗洛伊德(Floyd)算法来实现求解多源最短路径。

5.5.2 迪杰斯特拉算法

1959年，迪杰斯特拉(Dijkstra)提出了一个按路径长度递增的次序产生最短路径的算法，是从一个顶点到其余各顶点的求最短路径的算法。迪杰斯特拉算法的基本思想是：设置一个集合 S 存放已经找到最短路径的顶点，S 的初始状态只包含源点 v，对 $v_i \in V-S$，从源点 v 到 v_i 的有向边为最短路径。以后每求得一条最短路径 $v,v_1,v_2,\cdots,v_k$，就将 v_k 加入集合 S 中，并将路径 $v,v_1,v_2,\cdots,v_k,v_i$ 与原来的假设相比较，取路径长度较小者为最短路径。重复上述过程，直到集合 V 中全部顶点加入到集合 S 中。

假设给定一个有向图 G 的顶点集合 $S=\{A,B,C,D,E\}$，查找图 G 的每一个顶点到其他节点的最短路径流程如图5-14所示。得到顶点 A 到 B 的最短路径为 (A,B)，顶点 A 到 C 的最短路径为 (A,D,C)，顶点 A 到 D 的最短路径为 (A,D)，顶点 A 到 E 的最短路径为 (A,D,C,E)。

迪杰斯特拉算法求单源最短路径的伪代码描述如下：

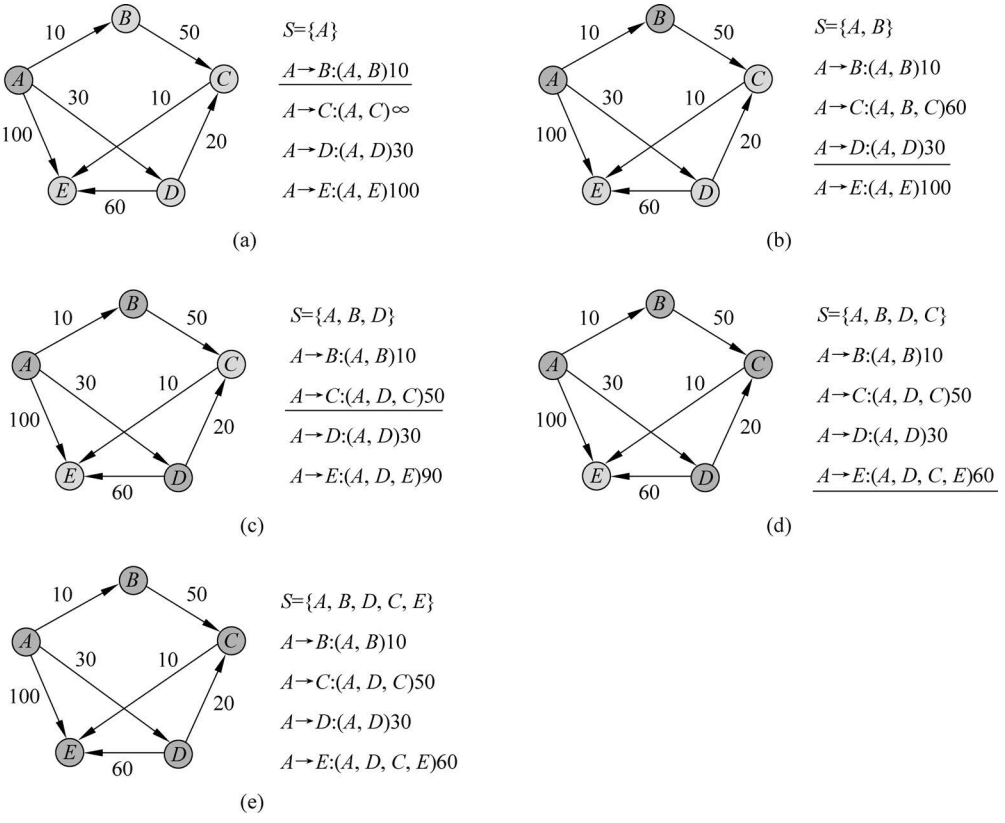


图 5-14 迪杰斯特拉算法执行流程案例

v:源点
S:已经生成最短路径的终点
$w<v, v_i>$:从顶点 v 到顶点 v_i 的权值
dist(v, v_i):表示从顶点 v 到顶点 v_i 的最短路径长度
算法:迪杰斯特拉算法
输入:有向网图 $G=(V,E)$,源点 v
输出:从 v 到其他所有顶点的最短路径
1. 初始化:集合 $S = \{v\}$;dist(v, v_i) $= w<v, v_i>$, ($i=1,2,\cdots,n$);
2. 重复下述操作:直到 $S = V$
2.1 dist(v, v_k) $=$ min{dist(v, v_j), ($j=1,2,\cdots,n$)};
2.2 $S = S + \{v_k\}$;
2.3 dist(v, v_j)=min{dist(v, v_j), dist(v, v_k) $+ w<v_k, v_j>$};

迪杰斯特拉算法求单源最短路径的代码实现如下:

```
def dijkstra(s):
    distance[s] = 0
    while True:
        # v在这里相当于是一个哨兵,对包含起点s做统一处理!
        v = -1
        # 从未使用过的顶点中选择一个距离最小的顶点
        for u in range(V):
            if not used[u] and (v == -1 or distance[u] < distance[v]):
                v = u
```

```
if v == -1:
    # 说明所有顶点都维护到 S 中了!
    break
# 将选定的顶点加入到 S 中, 同时进行距离更新
used[v] = True
for u in range(V):
    distance[u] = min(distance[u], distance[v] + cost[v][u])
```

迪杰斯特拉算法的思想上是很简单的，但在实现上是非常麻烦的。但求单源最短路径没有更好的办法。迪杰斯特拉算法的时间复杂度为 $O(n^2)$。

5.5.3 弗洛伊德算法

弗洛伊德(Floyd)算法又称为插点法，是一种利用动态规划的思想寻找给定的加权图中多源点之间最短路径的算法，与迪杰斯特拉算法类似。这个算法所要求解的问题是：已知一个各边权值都大于 0 的带权有向图，对任意两个顶点 $v_i \neq v_j$，要求求出 v_i 与 v_j 之间的最短路径和最短路径长度。

弗洛伊德算法的基本思想是：递推产生一个 n 阶仿真序列 $A^{(-1)}, A^{(0)}, \cdots, A^{(k)}, \cdots, A^{(n)}$，其中，$A^{(k)}[i][j]$表示从顶点 v_i 到顶点 v_j 的路径长度，k 表示绕行第 k 个顶点的运算步骤。初始时，对于任意两个顶点 v_i 和 v_j，若它们之间存在边，则以此边上的权值作为它们之间的最短路径长度；若它们之间不存在有向边，则以∞作为它们之间的最短路径长度。以后逐步尝试在原路径中加入顶点 $k(k=0,1,\cdots,n-1)$ 作为中间顶点。若增加中间顶点后，得到的路径比原来的路径长度减少了，则以此新路径代替原路径。

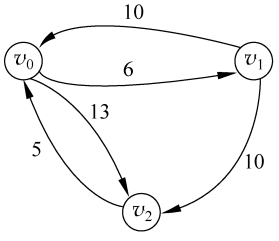
下面通过一个案例了解弗洛伊德算法的执行过程。图 5-15 所示为带权有向图 G 及其邻接矩阵。应用弗洛伊德算法求所有顶点之间的最短路径长度的过程如表 5-1 所示。算法执行过程的说明如下。

(1) 初始化：方阵 $A^{-1}[i][j]=\text{arcs}[i][j]$。

(2) 第一轮：将 v_0 作为中间顶点，对于所有顶点对$\{i,j\}$，如果有 $A^{-1}[i][j]>A^{-1}[i][0]+A^{-1}[0][j]$，则将 $A^{-1}[i][j]$更新为 $A^{-1}[i][0]+A^{-1}[0][j]$。有 $A^{-1}[2][1]>A^{-1}[2][0]+A^{-1}[0][1]=11$，更新 $A^{-1}[2][1]=11$，更新后的方阵标记为 $\boldsymbol{A}^0$。

(3) 第二轮：将 v_1 作为中间顶点，继续检测全部顶点对$\{i,j\}$。有 $A^0[0][2]>A^0[0][1]+A^0[1][2]=10$，更新 $A^0[0][2]=10$，更新后的方阵标记为 $\boldsymbol{A}^1$。

(4) 第三轮：将 v_2 作为中间顶点，继续检测全部顶点对$\{i,j\}$。有 $A^1[1][0]>A^1[1][2]+A^1[2][0]=9$，更新 $A^1[1][0]=9$，更新后的方阵标记为 $\boldsymbol{A}^2$。此时 $\boldsymbol{A}^2$ 中保存的就是任意顶点对的最短路径长度。



(a) 有向图G

$$\begin{bmatrix} 0 & 6 & 13 \\ 10 & 0 & 4 \\ 5 & \infty & 0 \end{bmatrix}$$

(b) 有向图G的邻接矩阵

图 5-15　带权有向图 G 及其邻接矩阵

表 5-1 弗洛伊德(Floyd)算法的执行过程

A	A^{-1}			A^0			A^1			A^2		
	v_0	v_1	v_2	v_0	v_1	v_2	v_0	v_1	v_2	v_0	v_1	v_2
v_0	0	6	13	0	6	13	0	6	**10**	0	6	10
v_1	10	0	4	10	0	4	10	0	4	**9**	0	4
v_2	5	∞	0	5	**11**	0	5	11	0	5	11	0

弗洛伊德算法求多源最短路径的伪代码描述如下：

dp(v,v_i)：表示从顶点 v 到顶点 v_i 的最短路径长度
算法：弗洛伊德算法
输入：有向网图 $G=(V,E)$
输出：从 v 到顶点 v_i 其他所有顶点的最短路径矩阵
1. 初始化：dp(v,v_i)=w<v,v_i>($i=1,2,\cdots,n$)；
2. 循环重复执行下述操作，直到执行到最短路径矩阵最后
2.1 如果 dp[i][k]+dp[j][k]< dp[i][j]；
2.2 执行 dp[i][j]=dp[i][k]+dp[j][k]替换最短路径。

弗洛伊德算法求多源最短路径的代码实现如下：

```
def Floyd(dis):
    nums_vertex = len(dis[0])
    for k in range(nums_vertex):
        for i in range(nums_vertex):
            for j in range(nums_vertex):
                if dis[i][j] > dis[i][k] + dis[k][j]:
                    dis[i][j] = dis[i][k] + dis[k][j]
    return dis
```

弗洛伊德算法的实现比较简单，其算法的时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$。

5.5.4 最短路径问题的应用案例

本节介绍最短路径问题的一个实际应用。假设某公司有 6 个办公楼，其中 0 号办公楼到 2、4、5 号办公楼的文件传输时间分别为 10s、30s、100s，由于 0 号办公楼到 1 号和 3 号办公楼没有铺设线路不能进行文件传输；1 号只能向 2 号楼传输文件，时间是 5s，2 号楼只能向 3 号楼传输文件，时间是 50s；3 号楼只能向 5 号楼传输文件，时间是 10s；4 号楼能向 3 号楼和 5 号楼传输文件，时间分别是 20s 和 50s，试求 0 号楼到 3 号楼的最短文件传输时间。

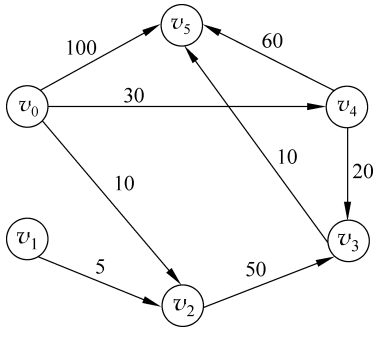


图 5-16 最短路径案例有向图

通过分析可以将案例抽象为求一个带权有向图的最短路径问题，其转化的无向图如图 5-16 所示。

此案例的迪杰斯特拉算法代码实现如下：

```
#!/bin/python
# - * - coding:utf-8 - * -
def dijkstra(graph, startIndex, path, cost, max):
    """
```

```
    求解各节点最短路径,获取 path,和 cost 数组,
    path[i] 表示 vi 节点的前驱节点索引,一直追溯到源点。
    cost[i] 表示 vi 节点的花费
    """
    lenth = len(graph)
    v = [0] * lenth
    # 初始化 path,cost,V
    for i in range(lenth):
        if i == startIndex:
            v[startIndex] = 1
        else:
            cost[i] = graph[startIndex][i]
            path[i] = (startIndex if (cost[i] < max) else -1)
    # print v, cost, path
    for i in range(1, lenth):
        minCost = max
        curNode = -1
        for w in range(lenth):
            if v[w] == 0 and cost[w] < minCost:
                minCost = cost[w]
                curNode = w
        # for 获取最小权值的节点
        if curNode == -1: break
        # 剩下都是不可通行的节点,跳出循环
        v[curNode] = 1
        for w in range(lenth):
            if v[w] == 0 and (graph[curNode][w] + cost[curNode] < cost[w]):
                cost[w] = graph[curNode][w] + cost[curNode]          # 更新权值
                path[w] = curNode                                    # 更新路径
        # for 更新其他节点的权值(距离)和路径
    return path

if __name__ == '__main__':
    max = 2147483647
    graph = [
        [max, max, 10, max, 30, 100],
        [max, max, 5, max, max, max],
        [max, max, max, 50, max, max],
        [max, max, max, max, max, 10],
        [max, max, max, 20, max, 60],
        [max, max, max, max, max, max],
        ]
    path = [0] * 6
    cost = [0] * 6
    print dijkstra(graph, 0, path, cost, max)
```

5.6 作业与思考题

1. 普里姆算法的时间复杂度是什么？适用于求什么样图的最小生成树？克鲁斯卡尔算法的时间复杂度是什么？适用于求什么样图的最小生成树？

2. 有向图 G 可拓扑排序的判别条件是什么？

3. 设有向图有 n 个顶点和 e 条边，进行拓扑排序时，时间复杂度是什么？

4. AOE 网为边表示活动的网，是一个带权的什么图，其长度最长的路径称为什么？

5. 在 AOE 网中，从源点到汇点路径上各活动时间总和最长的路径称为什么？

6. AOV 网中，节点表示什么？边表示什么？

7. AOE 网中，节点表示什么？边表示什么？

8. 求最短路径的迪杰斯特拉算法的时间复杂度是什么？

9. 求从某源点到其余各顶点的迪杰斯特拉算法在图的顶点数为 10，用邻接矩阵表示图时计算时间约为 10ms，则在图的顶点数为 40，计算时间约为多少？

10. 迪杰斯特拉最短路径算法从源点到其余各顶点的最短路径的路径长度按什么样的次序依次产生？