第 6 章习题

6.1 有 n 个选手参加的单循环比赛要进行多少场比赛？试用图结构描述。若是主客场制的联赛，又要进行多少场比赛？

【答】

如果把每个选手作为图的一个结点，两个选手的一场竞赛在图中两个结点连线，则 n 个选手的单循环赛就构成了 n 个结点的无向完全图，竞赛的总场次是图的边数，即为 n(n+1)/2。

主客场制联赛则构成有向完全图，比赛总场次为 n(n+1)。

6.2 证明下列命题：

（1）在任意一个有向图中，所有顶点的入度之和与出度之和相等。

【证明】每条有向边提供 1 个入度和 1 个出度，所以入度之和等于出度之和。

（2）任一无向图中各顶点的度的和一定为偶数。

【证明】每条边连接 2 个顶点，提供 2 个度，设图有 n 条边，则顶点度数之和为 2n，一定为偶数。

6.3 一个强连通图中各顶点的度有什么特点？

【答】每个顶点至少一个入度和一个出度。

6.4 证明：有向树中仅有 n-1 条弧。

【证明】有向树中根结点没有入度，其它顶点每个顶点 1 个入度，所以共有 n-1 条变。

6.5 证明树的三个不同定义之间的等价性。

【解】

图论中，树的等价定义很多，这里选择其中三个定义来证明其等价性。

定义一：连通，且边数为 n-1；

定义二：树中，任意两点之间存在唯一路径；

定义三：无回路，且边数为 n-1。

6.6 已知有向图 G 用邻接矩阵存储，设计算法以分别求解顶点 vi 的入度、出度和度。

【解】

假定顶点 vi 的编号为 i，则统计第 i 列所有有效元素数即顶点 vi 的入度 ind；第 i 行的有效元素个数即为顶点的出度 outd；顶点 vi 的度 deg=ind+outd。

为了使算法适用于有向图和网，邻接矩阵中有效元素判定表达式为：

G.AdjMatrix[j][i]>=1 && G.AdjMatrix[j][i]<INF，其中，INF 表示无穷大

【算法描述】

```
void getDegrees(Graph G, int i, int &ind, int &outd)
{
        //ind, outd 分别返回有向图中顶点 i 的入度和出度

    int j;
    ind=0;
    outd=0;
        //求顶点 i 的入度
    for(j=1;j<=G.VerNum;j++)
    {
        if(G.AdjMatrix[j][i]>=1 && G.AdjMatrix[j][i]<INF)   //兼顾有向图和网
            ind++;
    }

    //求顶点 i 的出度
    for(j=1;j<=G.VerNum;j++)
    {
        if(G.AdjMatrix[i][j]>=1 && G.AdjMatrix[i][j]<INF)   //兼顾有向图和网
            outd++;
    }


}
```

顶点的度 deg=ind+outd。

6.7 已知图 G 用邻接矩阵存储，设计算法以分别实现函数 firstAdj（G，v）和 nextAdj(G,v,w)。

【解】

假定图 G 的邻接矩阵为 AdjMatrix[ ][ ]，图的顶点数为 G.verNum，为了兼顾图和网，判定顶点 v、w 邻接（有边）的条件为：AdjMatrix[v][w]>=1 && AdjMatrix[v][w]<INF，其中 INF 为无穷大（程序实现时可用一个很大的数字表示）。

【firstAdj()算法描述】

```
//求顶点 v 的第一个邻接点
int firstAdj(Graph &G,int v)
{
        int w;
        for(w=1;w<=G.VerNum;w++)
        {
         if((G.AdjMatrix[v][w]>=1)     &&
```

```
            (G.AdjMatrix[v][w])<INF)
            return w;       //返回第一个邻接点编号
        }
        return 0;               //未找到邻接点，返回 0
}
```

【nextAdj()算法描述】

```
//求顶点 v 的位于邻接点 w 后的下一个邻接点
int nextAdj(Graph &G,int v,int w)
{
        int k;
        for(k=w+1;k<=G.VerNum;k++)
        {
         if((G.AdjMatrix[v][k]>=1)     &&
            (G.AdjMatrix[v][k])<INF)
            return k;       //返回 v 的位于 w 之后的下一个邻接点 k
        }
        return 0;               //不存在下一个邻接点，返回 0
}
```

6.8 设图 G 用邻接矩阵 A[n+1,n+1]表示，设计算法以判断 G 是否是无向图。

【解】

此图有 n 个顶点，邻接矩阵下标从 1 到 n。

无向图的判定条件：邻接矩阵关于主对角线对称，且边数小于等于 n(n-1)/2。

因为，有向完全图的邻接矩阵也是对称的，但边数将达到：n(n-1)，要排除这种情况。

【算法描述】

```
bool isUdg(Graph G)
{
    int i,j;
    for(i=1;i<=G.VerNum;i++)
        for(j=1;j<=G.VerNum;j++)
        {
            if(G.AdjMatrix[i][j]!=G.AdjMatrix[j][i])
                return false;     //出现一点不对称，不是无向图，返回 false
        }

        //循环结束，邻接矩阵对称，再判定边数是否小于等于 n(n-1)/2
    if(G.ArcNum<=G.VerNum*(G.VerNum-1)/2)
        return true;                //边数小于等于 n(n-1)/2
    else
        return false;
```

```
}
```

6.9 已知图 G 用邻接表存储，设计算法输出其所有边或弧。（假设各表头指针在数组 A[n+1]中）

【解】

算法思想：对顶点表进行循环，对每个顶点遍历其边链表，获取每条边的信息。边用邻接矩阵 E[][]进行保存，E[i][j]=1 表示顶点 i、j 之间有边（弧）。需要注意的是在无向图、网中，边（i,j）和(j,i)是同一条边（弧），只能取其中一条，所以当判定 i 和 j 之间有边时，要看是否 E[j][i]==1，如果已经存在则或略 E[i][j]。

算法中使用的是顶点编号，需要到 G.VerList[]查找顶点元素值。

【算法描述】

```
void getEdge(Graph G,int E[MaxVerNum+1][MaxVerNum+1])
{
        //A[][]为保存边信息的邻接矩阵，0 单元不用
    EdgeNode *p;
    int i;
    for(i=1;i<=G.VerNum;i++)                //对图的顶点表进行循环
    {
        p=G.VerList[i].firstEdge;           //获取顶点 i 的边链表头指针
        while(p)
        {
            if(G.gKind==DG || G.gKind==DN || E[p->adjVer][i]==0)
            {                               //排除无向图网的重复边
                E[i][p->adjVer]=1;          //保存边数据
            }
            p=p->next;
        }
    }
}
```

6.10 对下列图，分别执行 dfs(1)和 dfs(5)，写出遍历序列，并构造出相应的 dfs 生成树。

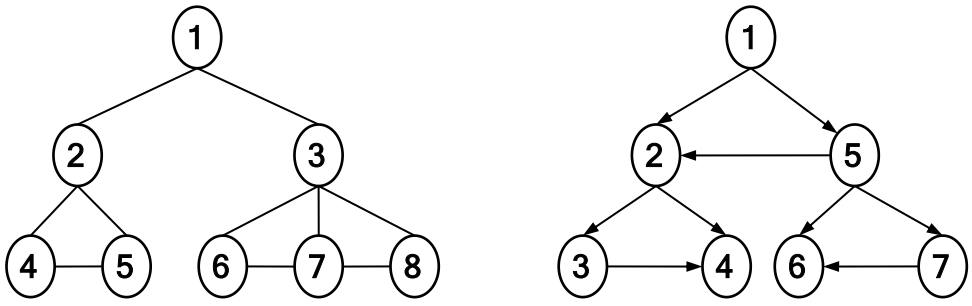

图 6-1　题 6.10 图

【解】

无向图 dfs(1)序列：1、2、4、5、3、6、7、8

dfs(5)序列：5、2、1、3、6、7、8、4

有向图 dfs(1)序列：1、2、3、4、5、6、7

dfs(5)序列：5、2、3、4、6、7

6.11 对图 6-2 的邻接表，不用还原出原图，请执行 dfs(1)，写出遍历序列，并构造出相应的 dfs 生成树。

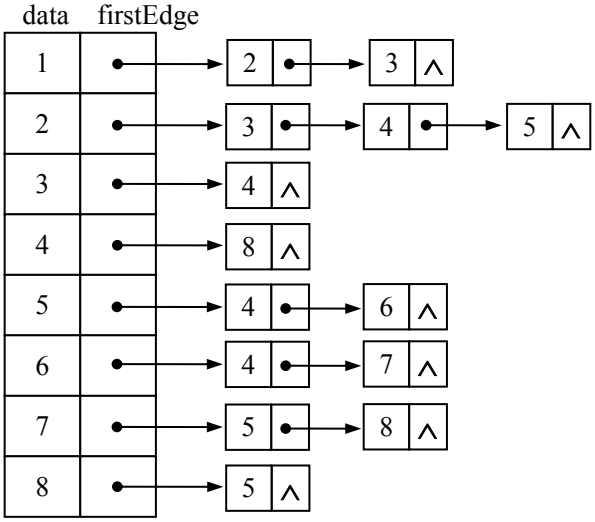

图 6-2 题 6.11 图 G 的邻接表

【解】

DFS(1)遍历序列：12348567。

DFS(1)生成树：

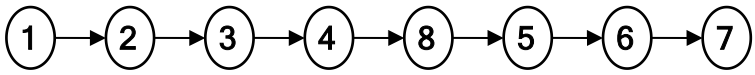

图 6- 题 6.11 解图

6.12 设计算法以判断顶点 v_i 到 v_j 之间是否存在路径？若存在，则返回 TRUE，否则返回 FALSE。

【解】

算法思想：本题可以通过改造联通图的深度优先遍历或广度优先遍历实现，从顶点 vi 开始遍历，如果遍历中访问到顶点 vj 则路径存在，否则遍历完成还没有遇到 j 则不存在路径。

这里的算法针对图的邻接链表实现，对图的邻接矩阵表示实现方法类似。

【改造 DFS 实现】

//当路径不存在时，要等遍历完全部顶点才能知道，且 DFS 是一个递归函数，用函数返回值来判定路径不存在时较为困难，因而我们用一个参数 R 来标记路径是否存在，R=0 路径不存在，R=1 则路径存在。R 可用全局变量，也可用函数参数，下面的实现使用函数参数实现。

```
void hasPath(Graph G,int i,int j, int &R)
{
        //R 标记 i 和 j 之间是否有路径，初始化为 0，有路径时改写为 1.
        //标记 R 也可用全局变量实现。
    EdgeNode *p;
    int w;

    visited[i]=true;     //标记顶点 i 已经访问
    if(i==j)
    {
        R=1;          //标记路径存在
        return;
    }
    p=G.VerList[i].firstEdge;
    w=firstAdj(G,i);
    while(w!=0)
    {
        if(w==j)
        {
            R=1;                          //标记路径存在
            return;
        }
        if(!visited[w])
            hasPath(G,w,j,R);              //递归访问编号为 w 的邻接点
        w=nextAdj(G,i,w);                  //搜索 w 之后的下一个邻接点
    }
}
```

【改造 BFS 实现】

```
//用函数返回值判定是否存在路径。
bool hasPathBFS( Graph G,   int i,   int j )
{
    int v,w,x;          //顶点编号
    EdgeNode *p;   //边结点指针
    seqQueue Q;
    initialQueue(&Q);              //初始化循环顺序队列
    visited[ i ];                  //标记顶点 i 已经访问
```

```
    enQueue(&Q, i);                 //编号为 verID 的顶点入队
    while(!queueEmpty(Q))           //队列不空循环处理顶点
    {
        getFront(Q, v);             //取队头元素到 v，即顶点编号为 v。
        outQueue(&Q,x);             //v 出队
        w=firstAdj(G,v);            //搜索 v（verID）的第一个邻接点，返回到 w
        while(w!=0)
        {
            if(!visited[w])         //w 尚未访问，访问 w，标记，入队
            {
                if(w==j)
                    return true;    //存在路径，返回 true
                visited[w]=true;    //标记顶点 w 已经访问
                enQueue(&Q,w);
            }
            w=nextAdj(G,v,w);       //搜索顶点 v 位于 w 之后的下一个邻接点
        }
    }
    return false;                   //i 和 j 之间不存在路径
}
```

6.13 **设计算法**以判断无向图 G 是否是连通的，若连通，返回 TRUE，否则返回 FALSE。

【解】

算法思想：改造图的通用深度优先遍历或广度优先遍历算法实现，先指定任一顶点遍历一个连通分量，再对剩下的顶点进行遍历，同时计数遍历的连通分量数，如连通分量数为 1 则为连通图。以下算法改造通用 DFS 实现，用 BFS 实现类似。

【算法描述】

```
bool isConnGrp(Graph G)
{
    int vID;        //顶点编号
    int conNum;     //连通分量数
    for(vID=1;vID<=G.VerNum;vID++)   //访问标记数组初始化
        visited[vID]=false;

    DFS(G,1);       //从顶点 1 开始遍历第一个连通分量
    conNum=1;       //连通分量数置为 1
    for(vID=2;vID<=G.VerNum;vID++)   //依次遍历图中其它的连通分量
    {
        if(!visited[vID])
        {
            conNum++;       //连通分量数加 1
```

```
                DFS(G,vID);
            }
        }

        if(conNum==1)
            return true;            //连通图
        else
            return false;
    }
```

6.14 设 G 是无向图，设计算法求出 G 中的边数。（假设图 G 分别采用邻接矩阵、邻接表以及不考虑具体存储形式，而是通过调用前面所述函数来求邻接点）

【解】

算法思想：用双重循环求解，第一层循环对图中每个顶点进行循环；第二层循环求出当前顶点的所有邻接点，每求出一个邻接点，边的计数加 1。对无向图，总计数除 2 即为边数；如果是有向图，计数结果即为边数。

【邻接链表表示算法】

```
int edgeNum(Graph G)
{
    EdgeNode *p;
    int i;
    int num=0;                          //计数边数
    for(i=1;i<=G.VerNum;i++)            //对顶点表进行循环
    {
        p=G.VerList[i].firstEdge;       //取顶点 i 的边链表头指针
        while(p)                        //内层循环求 i 的邻接点，边数计数
        {
            num++;
            p=p->next;
        }
    }
    return num/2;                       //因为是无向图，所以除 2
}
```

【通用算法】

```
//使用 firstAdj()和 nextAdj()函数来求邻接点，算法对邻接矩阵和邻接表表示皆使用。
int edgeNum1(Graph &G)
{
    int i,w;
    int num=0;
    for(i=1;i<=G.VerNum;i++)        //对所有顶点循环
```

```
    {
        w=firstAdj(G,i);
        while(w!=0)                    //求当前顶点的所有邻接点，进行边计数
        {
            num++;
            w=nextAdj(G,i,w);
        }
    }
    return num/2;                      //因为是无向图，所以除 2
}
```

【邻接矩阵表示求边数算法】

```
//对邻接矩阵行、列进行双重循环，对矩阵中有效的元素进行计数。
```

6.15 设 G 是无向图，设计算法以判断 G 是否是一棵树，若是树，则返回 TRUE，否则返回 FALSE；

【解】

算法思想：如果无向图 G 连通，且边数=顶点数-1，则 G 是一棵无向树。利用这个判据，计数图 G 的边数 eNum，利用通用遍历算法，计数图 G 的连通分量数 conNum。如果满足：eNum==G.VerNum-1 && conNum==1，则 G 是有向树，否则不是。

【算法描述】

```
bool isUDTree(Graph G)
{
    int eNum=0;      //计数无向图的边数
    int conNum=0;    //计数图的连通分量数
    int i, w;
    for(i=1;i<=G.VerNum;i++)
    {
        w=firstAdj(G,i);
        while(w!=0)
        {
            eNum++;                //计数边数
            w=nextAdj(G,i,w);
        }
    }
    eNum=eNum/2;       //因为是无向图，所以除 2

    for(i=1;i<=G.VerNum;i++)   //访问标记数组初始化
        visited[i]=false;

    for(i=1;i<=G.VerNum;i++)   //依次遍历图的连通分量，计数连通分量数
```

```
    {
        if(!visited[i])
        {
            conNum++;
            BFS(G,i);              //调用一种遍历，也可调用 DFS(G,i) 完成
        }
    }

    if(eNum==G.VerNum-1 && conNum==1)
        return 1;              //是无向树，连通，且边数=顶点数-1
    else
        return 0;              //不是无向树
}
```

6.16 设 G 是有向图，设计算法以判断 G 是否是一棵以 v_0 为根的有向树，若是返回 TRUE，否则返回 FALSE；

【解】

解法一：用于 6.15 题相同的方法，从指定起点开始可以访问所有顶点，即连通分量数为 1，计数图的边数为 n-1 条，n 为图的顶点数，则图为有向树。

解法二：从 v0 开始，连通分量数为 1，且 v0 的入度为 0，其它顶点的入度皆为 1.

6.17 在图 G 分别采用邻接矩阵和邻接表存储时，分析深度遍历算法的时间复杂度。

【解】

邻接矩阵存储，DFS 时间复杂度为 $O(n^2)$。搜索第 i 顶点的邻接点时，需要检查邻接矩阵第 i 行的所有元素，对每个顶点都要这样处理，事实上相当于检查了 n*n 矩阵的每个元素，所以时间复杂度为 $O(n^2)$。

邻接表存储，DFS 时间复杂度为 O(n+e)。搜索顶点 i 的邻接点，即需要搜索 i 的边链表各个结点。虽然各个顶点边链表长度不同，但搜索总次数取决于图的边数 e（无向图和网为 2e），所以时间复杂度为：O(n+e)。

6.18 设连通图用邻接表 A 表示，设计算法以产生 dfs（1）的 dfs 生成树，并存储到邻接矩阵 B 中。

【解】

```
void spanningTreeLinkedList(Graph G, int B[][],int v)
{
    int i,w;
    visited[v]=true;
    w=firstAdj(G,v);
    while(w!=0)
    {
        if(!visited[w])
```

```
            {
                B[v][w]=1;              //v、w 之间有边（弧），且 w 未访问
                spanningTreeLinkedList(G,B,w);              //递归访问编号为 w 的邻接点
            }
            w=nextAdj(G,v,w);           //搜索 w 之后的下一个邻接点
        }
}
```

6.19 在图 G 分别采用邻接矩阵和邻接表存储时，分析广度遍历算法的时间复杂度。

【解】

邻接矩阵存储，DFS 时间复杂度为 $O(n^2)$。搜索第 i 顶点的邻接点时，需要检查邻接矩阵第 i 行的所有元素，对每个顶点都要这样处理，事实上相当于检查了 n*n 矩阵的每个元素，所以时间复杂度为 $O(n^2)$。

邻接表存储，DFS 时间复杂度为 O(n+e)。搜索顶点 i 的邻接点，即需要搜索 i 的边链表各个结点。虽然各个顶点边链表长度不同，但搜索总次数取决于图的边数 e（无向图和网为 2e），所以时间复杂度为：O(n+e)。

6.20 设计算法以求解从 v_i 到 v_j 之间的最短路径。（每条边的长度为 1）

【解】

解法一：改造 BFS 算法，已 vi 为起点，访问顶点遇到 vj 时，得到 vi 到 vj 的最短路径。需要注意的是要利用栈，或专门设计的结构保存 vj 到 vi 的路径。

解法二：Dijkstra 算法，直接可以获得 vi 到 vj 的最短路径。

解法三：Floyd 算法，直接可以获得 vi 到 vj 的最短路径。

6.21 设计算法以求解距离 v_0 最远的一个顶点。

【解】

对这个问题，最容易想到的可能是利用 Dijkstra 算法，从起点开始每次取权值最大的点，但 Dijkstra 算法具有最优子结构，而求最长路径则没有这个特性，所以不能使用。另一个容易想到的是利用 Dijkstra 算法，把每个边的权值加负号，然后求最短路径，对于原来的权值即最长路径，但是，Dijkstra 算法，变得权值不能为负数，所以这种方法也是不可行的。

解法一：如果图中没有负权值回路，则可以利用 Floyd 算法求两点之间的最长路径，改造求最短路径的方法，当选择某个顶点为跳点时，路径变得更长时，接受这个跳点。这种方法允许出现负权值。事实上这是线性规划发求两点之间的最长路径。

解法二：如果图为有向无环图（DAG），则可以利用求关键路径方法来求源点和汇点之间的最长路径。

解法三：对于一般图，可以通过改造 Bellman-Ford 算法来求最长路径。

6.22 设计算法以求解二叉树 T 中层次最小的一个叶子结点的值。

【解】

层次遍历（广度优先），首先遇到的叶子即层次最小的叶子。

6.23 分别用 prim 算法和 Kruskal 算法求解下图的最小生成树，标注出中间求解过程的各状态。

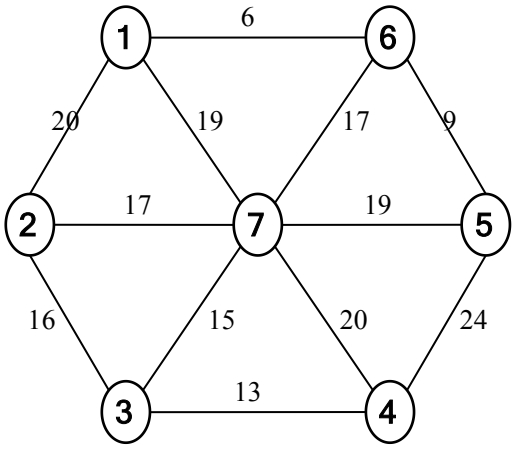

图 6-2　题 6.23 图

【解】

Prim 和 Kruskal 最小生成树相同，如下图：

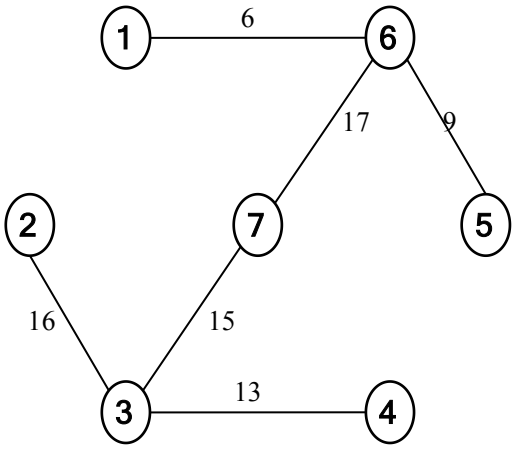

图 6-2　Prim 和 Kruskal 最小生成树

6.24 在图分别采用邻接矩阵和邻接表存储时，prim 算法的时间复杂度是否一致？为什么？

【解】

Prim 算法时间性能对邻接表和邻接矩阵存储是一样的，皆为 $O(n^2)$。

Prim 算法的执行时间取决于 for 循环的时间，在这个 for 循环中调用了 2 个函数 GetMinEdge( )和　UpdateTE()，它们的时间复杂度为 O(n)，所以，Prim 算法的时间复杂度为：$O(n^2)$。

且执行时间与连通网 N 的边数无关，适合于求解边数相对较多（较稠密）的网的生成

树。

6.25 在实现 Kruskal 算法时，如何判断某边和已选边是否构成回路？

【解】

给每个连通分量一个编号，同一分量中每个顶点的连通分量号相同。如果一条边的两个顶点的连通分量编号相同，则会和已选边形成回路；如果边的两个顶点的连通分量号不同，则不会形成回路。

算法初始化时，生成树中只含 n 个顶点，没有边，即具有 n 个连通分量。我们就用顶点编号作为初始的连通分量号。每选择一条边就会将 2 个连通分量连接为一个分量，这时我们要将一个连通分量上所有顶点的连通分量号改写为另一个分量的编号。成功选择 n-1 条边后，所有分量连接为一个连通分量，即最小生成树。

如果往小编号方向合并，最后生成树的连通分量编号为 1；如果往大号方向合并，最后生成树的连通分量编号为 n。

6.26 对下列 AOV 网，完成如下操作：

（1）按拓扑排序方法进行拓扑排序，写出中间各步的入度数组和栈的状态值，并写出拓扑序列。

（2）写出左图所示 AOV 网的所有的拓扑序列。

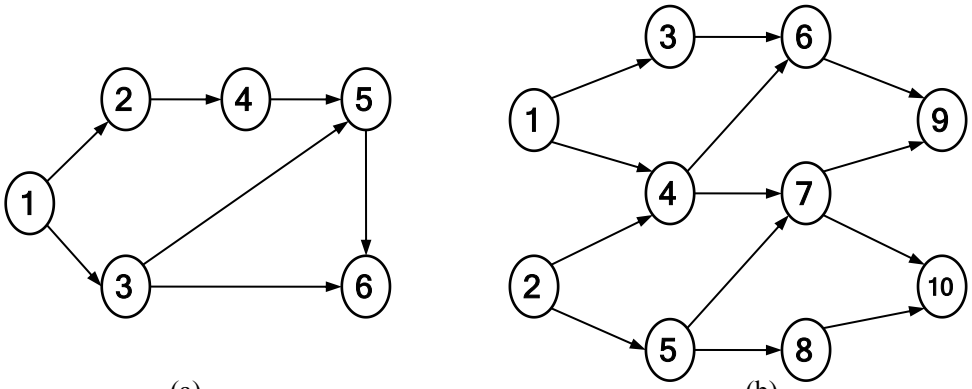

图 6-3　题 6.26 图

【解】

(a)的拓扑序列

123456，124356，132456

(b)的拓扑序列

1、2、3、4、5、6、7、8、9、10 等多个。

6.27 分析在图分别采用邻接矩阵和邻接表存储时的拓扑排序算法的时间复杂度。

【解】

由算法可知，整个算法要循环 n 次以输出每个顶点，其中在每一次循环中，每个顶点无需搜索。在输出每个顶点后，要对其所有的邻接（后继）顶点的入度减 1，因此，搜索其邻接顶点是花费时间最多的部分，并且所需的时间与深度优先搜索遍历算法类似，取决于存储结构：

① 若采用邻接矩阵存储图，算法的时间复杂度为 $O(n^2)$。

② 若采用邻接表存储图，算法的时间复杂度为 O(n+e)。

6.28 对下面的图，求出从顶点 1 到其余各顶点的最短路径。

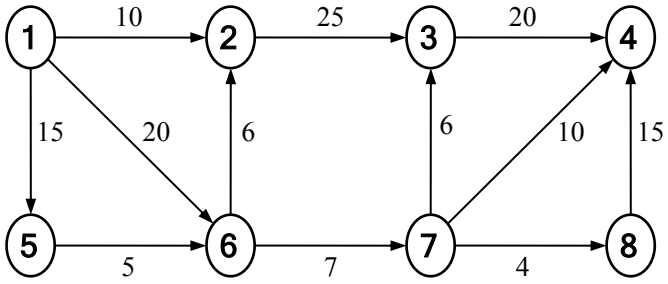

图 6-4　题 6.28 图

【解】

1-2：10；

1-5：15；

1-6：20；

1-6-7：27；

1-6-7-3：33；

1-6-7-8：31；

1-6-7-4：37.

6.29 分析在图分别采用邻接矩阵和邻接表存储时，求最短路径的 Dijkstra 算法的时间复杂度。

【解】

Dijkstra 算法对图的邻接矩阵和邻接表存储时间复杂度皆为 $O(n^2)$。