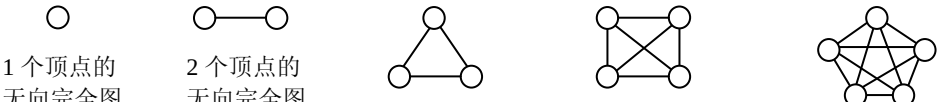
第 8 章 图

8-1 画出 1 个顶点、2 个顶点、3 个顶点、4 个顶点和 5 个顶点的无向完全图。试证明在 n 个顶点的无向完全图中，边的条数为 n(n−1)/2。

【解答】

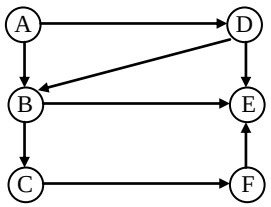


1 个顶点的无向完全图　2 个顶点的无向完全图　3 个顶点的无向完全图　4 个顶点的无向完全图　5 个顶点的无向完全图

【证明】

在有 n 个顶点的无向完全图中，每一个顶点都有一条边与其它某一顶点相连，所以每一个顶点有 n−1 条边与其他 n−1 个顶点相连，总计 n 个顶点有 n(n−1)条边。但在无向图中，顶点 i 到顶点 j 与顶点 j 到顶点 i 是同一条边，所以总共有 n(n−1)/2 条边。

8-2 右边的有向图是强连通的吗？请列出所有的简单路径。



【解答】

判断一个有向图是否强连通，要看从任一顶点出发是否能够回到该顶点。右面的有向图做不到这一点，它不是强连通的有向图。各个顶点自成强连通分量。

所谓简单路径是指该路径上没有重复的顶点。

从顶点 A 出发，到其他的各个顶点的简单路径有 A→B，A→D→B，A→B→C，A→D→B→C，A→D，A→B→E，A→D→E，A→D→B→E，A→B→C→F→E，A→D→B→C→F→E，A→B→C→F，A→D→B→C→F。

从顶点 B 出发，到其他各个顶点的简单路径有 B→C，B→C→F，B→E，B→C→F→E。

从顶点 C 出发，到其他各个顶点的简单路径有 C→F，C→F→E。

从顶点 D 出发，到其他各个顶点的简单路径有 D→B，D→B→C，D→B→C→F，D→E，D→B→E，D→B→C→F→E。

从顶点 E 出发，到其他各个顶点的简单路径无。

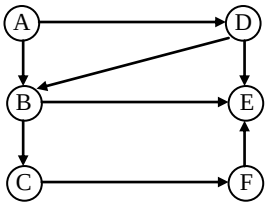
从顶点 F 出发，到其他各个顶点的简单路径有 F→E。

8-3 给出右图的邻接矩阵、邻接表和邻接多重表表示。

【解答】

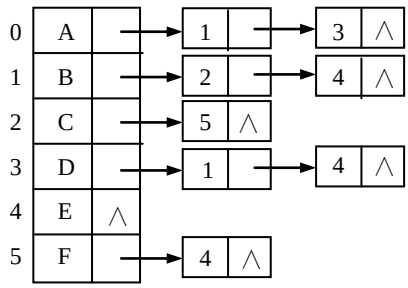
(1) 邻接矩阵

$$\mathrm{Edge}=\begin{bmatrix}0&1&0&1&0&0\\0&0&1&0&1&0\\0&0&0&0&0&1\\0&1&0&0&1&0\\0&0&0&0&0&0\\0&0&0&0&1&0\end{bmatrix}$$

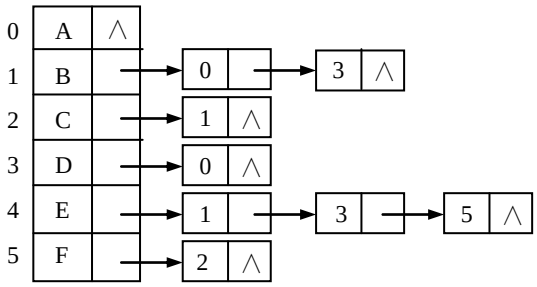


(2) 邻接表

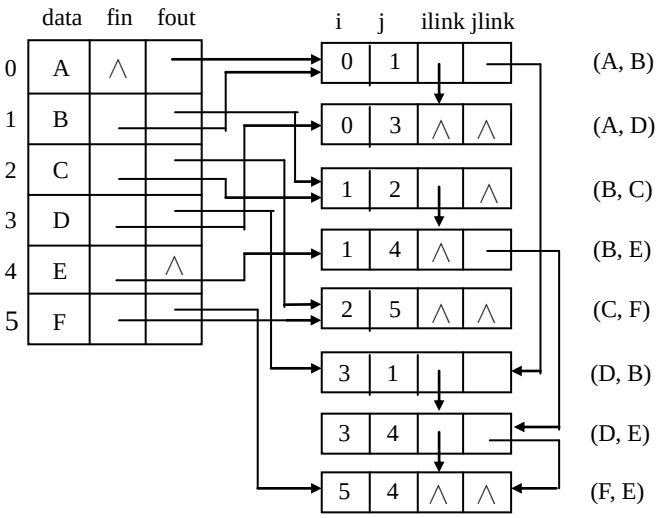
（出边表）



（入边表）



(3) 邻接多重表（十字链表）



8-4 用邻接矩阵表示图时，若图中有 1000 个顶点，1000 条边，则形成的邻接矩阵有多少矩阵元素？有多少非零元素？是否稀疏矩阵？

【解答】

一个图中有 1000 个顶点，其邻接矩阵中的矩阵元素有 $1000^2 = 1000000$ 个。它有 1000 个非零元素（对于有向图）或 2000 个非零元素（对于无向图），因此是稀疏矩阵。

8-5 用邻接矩阵表示图时，矩阵元素的个数与顶点个数是否相关？与边的条数是否相关？

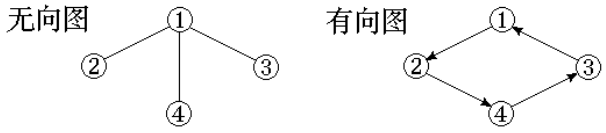
【解答】

用邻接矩阵表示图，矩阵元素的个数是顶点个数的平方，与边的条数无关。矩阵中非零元素的个数与边的条数有关。

8-6 有 n 个顶点的无向连通图至少有多少条边？有 n 个顶点的有向强连通图至少有多少条边？试举例说明。

【解答】

n 个顶点的无向连通图至少有 n−1 条边，n 个顶点的有向强连通图至少有 n 条边。例如：



特例情况是当 n = 1 时，此时至少有 0 条边。

8−7 对于有 n 个顶点的无向图，采用邻接矩阵表示，如何判断以下问题： 图中有多少条边？任意两个顶点 i 和 j 之间是否有边相连？任意一个顶点的度是多少？

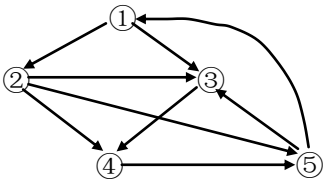
【解答】

用邻接矩阵表示无向图时，因为是对称矩阵，对矩阵的上三角部分或下三角部分检测一遍，统计其中的非零元素个数，就是图中的边数。如果邻接矩阵中 A[i][j] 不为零，说明顶点 i 与顶点 j 之间有边相连。此外统计矩阵第 i 行或第 i 列的非零元素个数，就可得到顶点 i 的度数。

8−8 对于如右图所示的有向图，试写出：

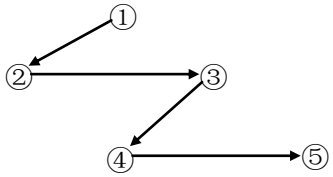
(1) 从顶点①出发进行深度优先搜索所得到的深度优先生成树；

(2) 从顶点②出发进行广度优先搜索所得到的广度优先生成树；

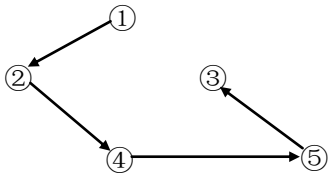


【解答】

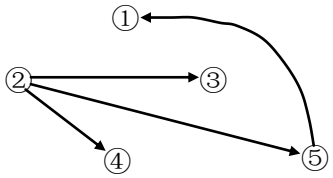
(1) 以顶点①为根的深度优先生成树（不唯一）：② ③ ④ ⑤ ⑥



或



(2) 以顶点②为根的广度优先生成树：



8−9 试扩充深度优先搜索算法，在遍历图的过程中建立生成森林的左子女−右兄弟链表。算法的首部为

void Graph**::**DFS (**const int** v, **int** visited [ ], TreeNode<**int**> * t) 其中，指针 t 指向生成森林上具有图顶点 v 信息的根结点。（提示：在继续按深度方向从根 v 的某一未访问过的邻接顶点 w 向下遍历之前，建立子女结点。但需要判断是作为根的第一个子女还是作为其子女的右兄弟链入生成树。）

【解答】

为建立生成森林，需要先给出建立生成树的算法，然后再在遍历图的过程中，通过一次次地调用这个算法，以建立生成森林。

```
te mplate<Type> void Graph<Type> :: DFS_Tree ( const int v,    int visited [ ],    TreeNode<Type> *t ) {
//从图的顶点 v 出发，深度优先遍历图，建立以 t (已在上层算法中建立)为根的生成树。
    Visited[v] = 1;   int first = 1;   TreeNode<Type> * p, * q;
    int w = GetFirstNeighbor ( v );                          //取第一个邻接顶点
    while ( w != −1 ) {                                      //若邻接顶点存在
```

```
        if ( vosited[w] == 0 ) {                          //且该邻接结点未访问过
            p = new TreeNode<Type> ( GetValue (w) );      //建立新的生成树结点
            if ( first == 1 )                             //若根*t 还未链入任一子女
                { t->setFirstChild ( p );   first = 0; }  //新结点*p 成为根*t 的第一个子女
            else q->setNextSibling ( p );                 //否则新结点*p 成为*q 的下一个兄弟
            q = p;                                        //指针 q 总指示兄弟链最后一个结点
            DFS_Tree ( w, visited, q );                   //从*q 向下建立子树
        }
        w = GetNextNeighbor ( v, w );                     //取顶点 v 排在邻接顶点 w 的下一个邻接顶点
    }
}
```

下一个算法用于建立以左子女-右兄弟链表为存储表示的生成森林。

```
template<Type> void Graph<Type> :: DFS_Forest ( Tree<Type> & T ) {
//从图的顶点 v 出发，深度优先遍历图，建立以左子女-右兄弟链表表示的生成森林 T。
    T.root = NULL;   int n = NumberOfVertices ( );          //顶点个数
    TreeNode<Type> * p, * q;
    int * visited = new int [ n ];                          //建立访问标记数组
    for ( int v = 0; v < n; v++ ) visited[v] = 0;
    for ( v = 0; v < n; v++ )                               //逐个顶点检测
        if ( visited[v] == 0 ) {                            //若尚未访问过
            p = new TreeNode<Type> ( GetValue ( v ) );      //建立新结点*p
            if ( T.root == NULL ) T.root = p;               //原来是空的生成森林，新结点成为根
            else q-> setNextSibling ( p );                  //否则新结点*p 成为*q 的下一个兄弟
            q = p;
            DFS_Tree ( v, visited, p );                     //建立以*p 为根的生成树
        }
}
```

8-10 用邻接表表示图时，顶点个数设为 n，边的条数设为 e，在邻接表上执行有关图的遍历操作时，时间代价是 O(n*e)？还是 O(n+e)？或者是 O(max(n,e))？

【解答】

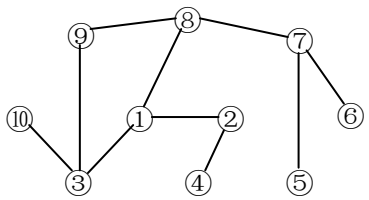
在邻接表上执行图的遍历操作时，需要对邻接表中所有的边链表中的结点访问一次，还需要对所有的顶点访问一次，所以时间代价是 O(n+e)。

8-11 右图是一个连通图，请画出

(1) 以顶点①为根的深度优先生成树；

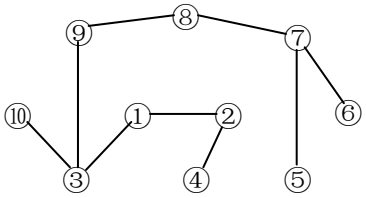
(2) 如果有关节点，请找出所有的关节点。

(3) 如果想把该连通图变成重连通图，至少在图中加几条边？如何加？



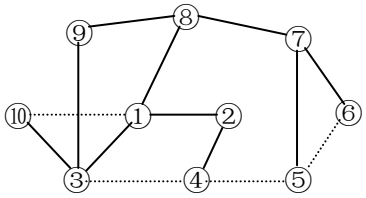
【解答】

(1) 以顶点①为根的深度优先生成树：



(2) 关节点为 ①, ②, ③, ⑦, ⑧

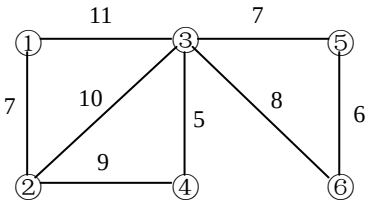
(3) 至少加四条边 (1, 10), (3, 4), (4, 5), (5, 6)。从③的子孙结点⑩到③的祖先结点①引一条边，从②的子孙结点④到根①的另一分支③引一条边，并将⑦的子孙结点⑤、⑥与结点④连结起来，可使其变为重连通图。



8−12 试证明在一个有 n 个顶点的完全图中，生成树的数目至少有 $2^{n-1}-1$。

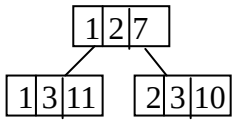
【证明】略

8−13 编写一个完整的程序，首先定义堆和并查集的结构类型和相关操作，再定义 Kruskal 求连通网络的最小生成树算法的实现。并以右图为例，写出求解过程中堆、并查集和最小生成树的变化。

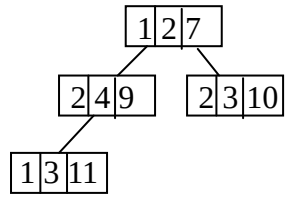


【解答】

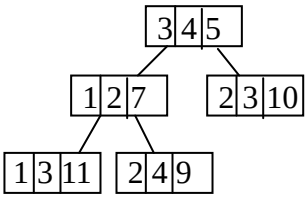
求解过程的第一步是对所有的边，按其权值大小建堆：



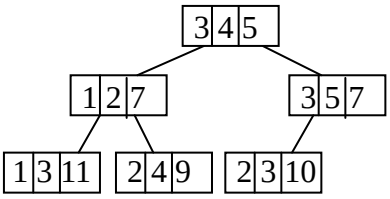
加(1, 2), (1, 3), (2,3)



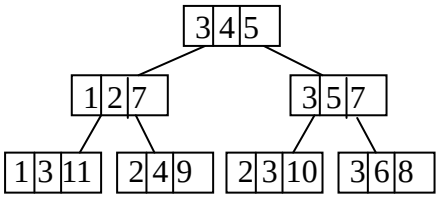
加(2, 4)



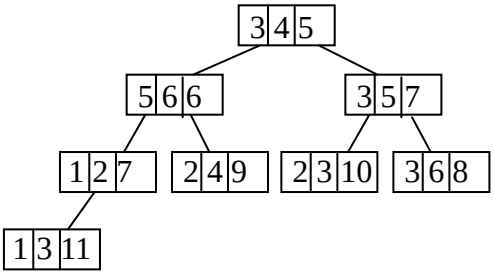
加(3, 4)



加(3, 5)

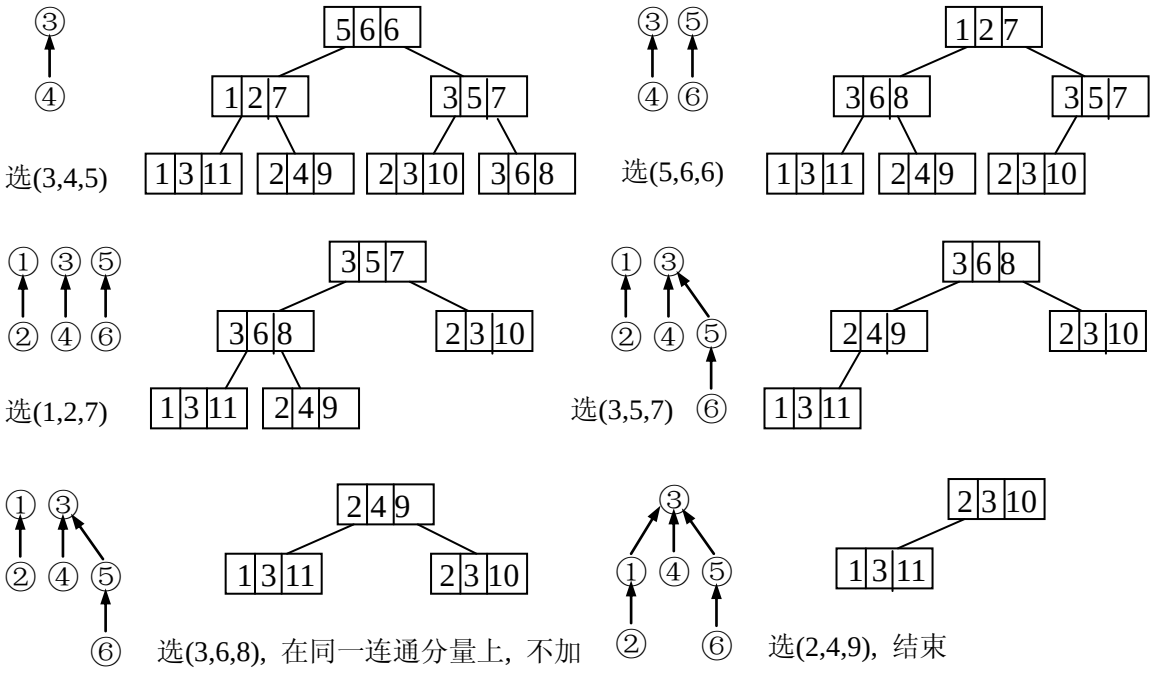


加(3, 6)

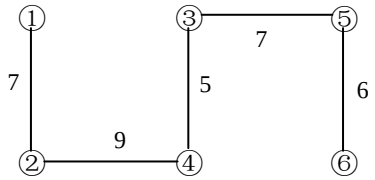


加(5, 6)

求解过程中并查集与堆的变化：



最后得到的生成树如下



0	1	2	3	4	5	6
	3	1	−6	3	3	5

并查集的存储表示

完整的程序如下:

```
#include <iostream.h>
template <class Type> class MinHeap {
public:
    enum { MaxHeapSize = 50 };
    MinHeap ( int Maxsize = MaxHeapSize );
    MinHeap ( Type Array[ ], int n );
    void Insert ( const Type &ele );
    void RemoveMin ( Type &Min );
    void Output ();
private:
    void FilterDown ( int start, int end );
    void FilterUp ( int end );
    Type    *pHeap;
    int HMaxSize;
    int CurrentSize;
};

class UFSets {
public:
```

```
    enum { MaxUnionSize = 50 };
    UFSets ( int MaxSize = MaxUnionSize );
    ~UFSets () { delete [ ] m_pParent; }
    void Union ( int Root1, int Root2 );
    int Find ( int x );
private:
    int m_iSize;
    int *m_pParent;
};

class Graph {
public:
    enum { MaxVerticesNum = 50 };
    Graph( int Vertices = 0) { CurrentVertices = Vertices; InitGraph(); }
    void InitGraph ();
    void Kruskal ();
    int GetVerticesNum () { return CurrentVertices; }
private:
    int Edge[MaxVerticesNum][MaxVerticesNum];
    int CurrentVertices;
};

class GraphEdge {
public:
    int head, tail;
    int cost;
    int operator <= ( GraphEdge &ed );
};

GraphEdge :: operator <= ( GraphEdge &ed ) {
    return this->cost <= ed.cost;
}

UFSets :: UFSets ( int MaxSize ) {
    m_iSize = MaxSize;
    m_pParent = new int[m_iSize];
    for ( int i = 0; i < m_iSize; i++ ) m_pParent[i] = -1;
}

void UFSets :: Union ( int Root1, int Root2 ) {
    m_pParent[Root2] = Root1;
}
```

```
int UFSets :: Find ( int x ) {
    while ( m_pParent[x] >= 0 ) x = m_pParent[x];
    return x;
}

template <class Type> MinHeap<Type> :: MinHeap ( int Maxsize ) {
    HMaxSize = Maxsize;
    pHeap = new Type[HMaxSize];
    CurrentSize = -1;
}

template <class Type> MinHeap<Type> :: MinHeap ( Type Array[], int n ) {
    HMaxSize = ( n < MaxHeapSize ) ? MaxHeapSize : n;
    pHeap = new Type[HMaxSize];
    for ( int i = 0; i < n; i++ ) pHeap[i] = Array[i];
    CurrentSize = n-1;
    int iPos = ( CurrentSize - 1 ) / 2;
    while ( iPos >= 0 ) {
        FilterDown ( iPos, CurrentSize );
        iPos--;
    }
}

template <class Type> void MinHeap<Type> :: FilterDown ( int start, int end ) {
    int i = start, j = 2 * start + 1;
    Type Temp = pHeap[i];
    while ( j <= end ) {
        if ( j < end && pHeap[j+1] <= pHeap[j] ) j++;
        if ( Temp <= pHeap[j] ) break;
        pHeap[i] = pHeap[j];
        i = j; j = 2 * j + 1;
    }
    pHeap[i] = Temp;
}

template <class Type> void MinHeap<Type> :: FilterUp ( int end ) {
    int i = end, j = ( end - 1 ) / 2;
    Type Temp = pHeap[i];
    while ( i > 0 ) {
        if ( pHeap[j] <= Temp ) break;
        pHeap[i] = pHeap[j];
        i = j; j = ( j - 1 ) / 2;
    }
```

```
    pHeap[i] = Temp;
}

template <class Type> void MinHeap<Type> :: Insert ( const Type &ele ) {
    CurrentSize++;
    if ( CurrentSize == HMaxSize ) return;
    pHeap[CurrentSize] = ele;
    FilterUp ( CurrentSize );
}

template <class Type> void MinHeap<Type> :: RemoveMin ( Type &Min ) {
    if ( CurrentSize < 0 )      return;
    Min = pHeap[0];
    pHeap[0] = pHeap[CurrentSize--];
    FilterDown ( 0, CurrentSize );
}

template <class Type> void MinHeap<Type> :: Output ( ) {
    for ( int i = 0; i <= CurrentSize; i++ ) cout << pHeap[i] << " ";
    cout << endl;
}

void Graph :: InitGraph( ) {
    Edge[0][0] = -1;   Edge[0][1] = 28;   Edge[0][2] = -1;   Edge[0][3] = -1;   Edge[0][4] = -1;   Edge[0][5] = 10;
    Edge[0][6] = -1;
    Edge[1][1] = -1;   Edge[1][2] = 16;   Edge[1][3] = -1;   Edge[1][4] = -1;   Edge[1][5] = -1;   Edge[1][6] = 14;
    Edge[2][2] = -1;   Edge[2][3] = 12;   Edge[2][4] = -1;   Edge[2][5] = -1;   Edge[2][6] = -1;
    Edge[3][3] = -1;   Edge[3][4] = 22;   Edge[3][5] = -1;   Edge[3][6] = 18;
    Edge[4][4] = -1;   Edge[4][5] = 25;   Edge[4][6] = 24;
    Edge[5][5] = -1;   Edge[5][6] = -1;
    Edge[6][6] = -1;
    for ( int i = 1; i < 6; i++ )
        for ( int j = 0; j < i; j ++ ) Edge[i][j] = Edge[j][i];
}

void Graph :: Kruskal( ) {
    GraphEdge e;
    int VerticesNum = GetVerticesNum ( );
    int i, j, count;
    MinHeap<GraphEdge> heap ( VerticesNum *VerticesNum );
    UFSets set ( VerticesNum );
    for ( i = 0; i < VerticesNum; i++ )
        for ( j = i + 1; j < VerticesNum; j++ )
```

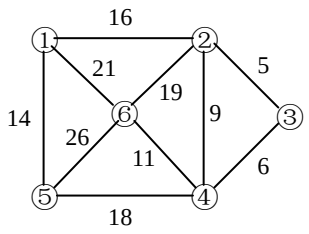
```
            if ( Edge[i][j] > 0 )    {
                e.head = i; e.tail = j; e.cost = Edge[i][j];
                heap.Insert ( e );
            }
        count = 1;
        while ( count < VerticesNum ) {
            heap.RemoveMin ( e );
            i = set.Find ( e.head );
            j = set.Find ( e.tail );
            if ( i != j ) {
                set.Union ( i, j );
                count++;
                cout << "( " << e.head << ", " << e.tail << ", " <<     e.cost << " )" << endl;
            }
        }
    }
```

8-14 利用 Dijkstra 算法的思想，设计一个求最小生成树的算法。

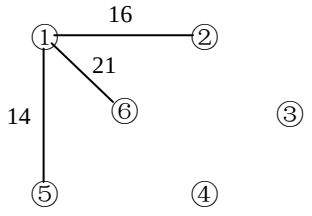
【解答】

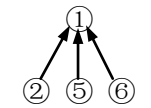
计算连通网络的最小生成树的 Dijkstra 算法可描述如下：将连通网络中所有的边以方便的次序逐步加入到初始为空的生成树的边集合 T 中。每次选择并加入一条边时，需要判断它是否会与先前加入 T 的边构成回路。如果构成了回路，则从这个回路中将权值最大的边退选。

下面以邻接矩阵作为连通网络的存储表示，并以并查集作为判断是否出现回路的工具，分析算法的执行过程。

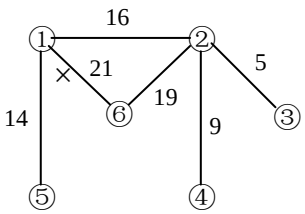


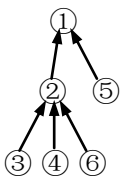
$$\text{Edge} = \begin{array}{c} \begin{matrix} ① & ② & ③ & ④ & ⑤ & ⑥ \end{matrix} \\ \begin{pmatrix} 0 & 16 & \infty & \infty & 14 & 21 \\ - & 0 & 5 & 9 & \infty & 19 \\ - & - & 0 & 6 & \infty & \infty \\ - & - & - & 0 & 18 & 11 \\ - & - & - & - & 0 & 26 \\ - & - & - & - & - & 0 \end{pmatrix} \end{array} \begin{matrix} ① \\ ② \\ ③ \\ ④ \\ ⑤ \\ ⑥ \end{matrix}$$

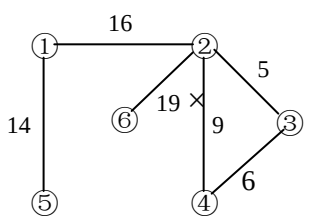


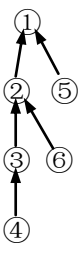


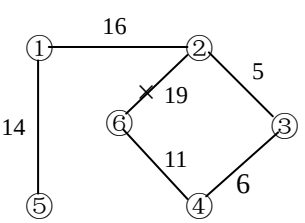
并查集，表明4个结点在同一连通分量上

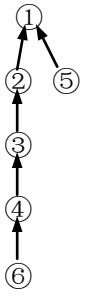




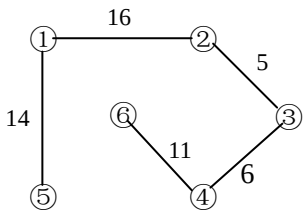








最终得到的最小生成树为



实现算法的过程为

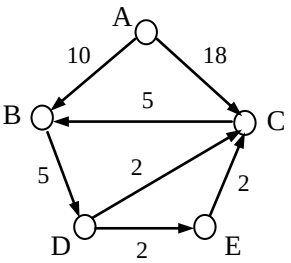
```
const int MaxNum = 10000;
void Graph :: Dijkstra ( ) {
    GraphEdge e;
    int VerticesNum = GetVerticesNum ( );
    int i, j, p, q, k;
    int disJoint[VerticesNum];                          //并查集
    for ( i = 0; i < VerticesNum; i++ ) disJoint[i] = −1;      //并查集初始化
    for ( i = 0; i < VerticesNum−1; i++ )                      //检查所有的边
        for ( j = i + 1; j < VerticesNum; j++ )
            if ( Edge[i][j] < MaxNum )        {                 //边存在
                p = i;    q = j;                                //判结点 i 与 j 是否在同一连通分量上
                while ( disJoint[p] >= 0 ) p = disJoint[p];
                while ( disJoint[q] >= 0 ) p = disJoint[q];
                if ( p != q ) disJoint[j] = i;                  // i 与 j 不在同一连通分量上, 连通之
            } else {                                            // i 与 j 在同一连通分量上
                p = i;                                          //寻找离结点 i 与 j 最近的祖先结点
                while ( disJoint[p] >= 0 ) {                    //每变动一个 p, 就对 q 到根的路径检测一遍
                    q = j;
                    while ( disJoint[q] >= 0 && disJoint[q] == disJoint[p] )
                        q = disJoint[q];
                    if ( disJoint[q] == disJoint[p] ) break;
                    else p = disJoint[p];
                }
                k = disJoint[p];                                //结点 k 是 i 和 j 的最近的共同祖先
                p = i;    q = disJoint[p];    max = −MaxNum;    //从 i 到 k 找权值最大的边(s1, s2)
                while ( q <= k ) {
                    if ( Edge[q][p] > max ) { max = Edge[q][p];    s1 = p;    s2 = q; }
                    p =q;    q = disJoint[p];
                }
                p = j;    q = disJoint[p];    max = −MaxNum;    //从 j 到 k 找权值最大的边(t1, t2)
                while ( q <= k ) {
                    if ( Edge[q][p] > max ) { max = Edge[q][p];    t1 = p;    t2 = q; }
                    p =q;    q = disJoint[p];
                }
                max = Edge[i][j];    k1 = i;    k2 = j;
                if ( max < Edge[s1][s2] ) { max = Edge[s1][s2];    k1 = s1;    k2 = s2; }
```

```
                if ( max < Edge[t1][t2] ) { max = Edge[t1][t2];   k1 = t1;   k2 = t2; }
                if ( max != Edge[i][j] ) {              //当 Edge[i][j] == max 时边不改
                    if ( disJoint[k1] == k2 ) disJoint[k1] = −1;
                    else disJoint[k2] = −1;             //删除权值最大的边
                    disJoint[j] = i;                    //加入新的边
                    Edge[j][i] = − Edge[j][i];
                }
            }
    }
```

8−15 以右图为例，按 Dijkstra 算法计算得到的从顶点①(A)到其它各个顶点的最短路径和最短路径长度。

【解答】

源点	终点	最短路径				最短路径长度			
A	B	(A,B)	(A,B)	(A,B)	(A,B)	10	10	10	10
	C	(A,C)	(A,C)	(A,C)	(A,C)	18	18	18	18
	D	—	(A,B,D)	(A,B,D)	(A,B,D)	∞	15	15	15
	E	—	—	(A,B,D,E)	(A,B,D,E)	∞	∞	17	17



8−16 在以下假设下，重写 Dijkstra 算法：

(1) 用邻接表表示带权有向图 G，其中每个边结点有 3 个域：邻接顶点 vertex，边上的权值 length 和边链表的链接指针 link。

(2) 用集合 T = V(G) − S 代替 S (已找到最短路径的顶点集合)，利用链表来表示集合 T。

试比较新算法与原来的算法，计算时间是快了还是慢了，给出定量的比较。

【解答】

(1) 用邻接表表示的带权有向图的类定义：

```
const int DefaultSize = 10;                     //缺省顶点个数
class Graph;                                    //图的前视类定义
struct Edge {                                   //边的定义
friend class Graph;
   int vertex;                                  //边的另一顶点位置
   float length;                                //边上的权值
   Edge *link;                                  //下一条边链指针
   Edge ( ) { }                                 //构造函数
   Edge ( int num, float wh ) : vertex (num), length (wh), link (NULL) { }      //构造函数
   int operator < ( const Edge & E ) const { return length != E.length; }      //判边上权值小否
}

struct Vertex {                                 //顶点的定义
friend class Graph;
   char data;                                   //顶点的名字
   Edge *adj;                                   //边链表的头指针
}
```

```
class Graph {                                  //图的类定义
private:
   Vertex *NodeTable;                          //顶点表 (各边链表的头结点)
   int NumVertices;                            //当前顶点个数
   int NumEdges;                               //当前边数
   int GetVertexPos ( const Type vertex );     //给出顶点 vertex 在图中的位置
public:
   Graph ( int sz );                           //构造函数
   ~Graph ( );                                 //析构函数
   int NumberOfVertices ( ) { return NumVertices; }      //返回图的顶点数
   int NumberOfEdges ( ) { return NumEdges; }            //返回图的边数
   char GetValue ( int i )                     //取位置为 i 的顶点中的值
      { return i >= 0 && i < NumVertices ? NodeTable[i].data : ' '; }
   float GetWeight ( int v1, int v2 );         //返回边(v1, v2)上的权值
   int GetFirstNeighbor ( int v );             //取顶点 v 的第一个邻接顶点
   int GetNextNeighbor ( int v, int w );       //取顶点 v 的邻接顶点 w 的下一个邻接顶点
}
```

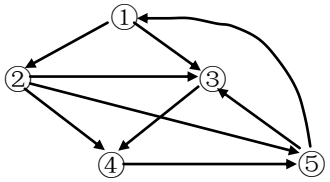
(2) 用集合 T = V(G) − S 代替 S (已找到最短路径的顶点集合)，利用链表来表示集合 T。

8-17 试证明：对于一个无向图 G = (V, E)，若 G 中各顶点的度均大于或等于 2，则 G 中必有回路。

【解答】

反证法：对于一个无向图 G=（V，E），若 G 中各顶点的度均大于或等于 2，则 G 中没有回路。此时从某一个顶点出发，应能按拓扑有序的顺序遍历图中所有顶点。但当遍历到该顶点的另一邻接顶点时，又可能回到该顶点，没有回路的假设不成立。

8-18 设有一个有向图存储在邻接表中。试设计一个算法，按深度优先搜索策略对其进行拓扑排序。并以右图为例检验你的算法的正确性。



【解答】

(1) 利用题 8-16 定义的邻接表结构。

增加两个辅助数组和一个工作变量：

- 记录各顶点入度 **int** indegree[NumVertices]。
- 记录各顶点访问顺序 **int** visited[NumVertices]，初始时让 visited[i] = 0, i = 1, 2, …, NumVertices。
- 访问计数 **int** count，初始时为 0。

(2) 拓扑排序算法

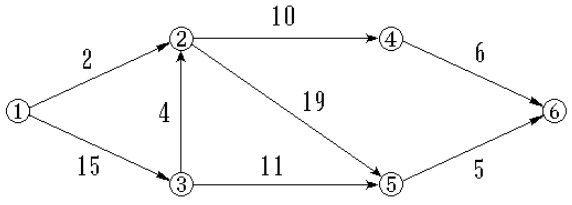
```
void Graph :: dfs ( int visited[ ], int indegree[ ], int v, int & count ) {
   count++;   visited[v] = count;
   cout << NodeTable[v].data << endl;
   Edge *p = NodeTable[v].adj;
   while ( p != NULL ) {
      int w = p->vertex;
      indegree[w]--;
      if ( visited[w] == 0 && indegree[w] == 0 ) dfs ( visited, indegree, w, count );
```

```
            p = p->link;
        }
    }
```

主程序

```
    int i, j;    Edge *p;    float w;
    cin >> NumVertices;
    int * visited = new int[NumVertices+1];
    int * indegree = new int[NumVertices+1];
    for ( i = 1; i <= NumVertices; i++ ) {
        NodeTable[i].adj = NULL;    cin >> NodeTable[i].data;    cout << endl;
        visited[i] = 0;    indegree[i] = 0;
    }
    int count = 0;
    cin >> i >> j >> w;    cout << endl;
    while ( i != 0 && j != 0 ) {
        p = new Edge ( j, w );
        if ( p == NULL ) { cout << "存储分配失败！" << endl;    exit(1); }
        indegree[j]++;
        p->link = NodeTable[i].adj;    NodeTable[i].adj = p;
        NumEdges++;
        cin >> i >> j >> w;    cout << endl;
    }
    for ( i = 1; i <= NumVertices; i++ )
       if ( visited[i] == 0 && indegree[i] == 0 ) dfs ( visited, indegree, i, count );
    if ( count < NumVertices ) cout << "排序失败!" << endl;
    else cout << "排序成功!" << endl;
    delete [ ] visited;    delete [ ] indegree;
```

8-19 试对右图所示的 AOE 网络，解答下列问题。



(1) 这个工程最早可能在什么时间结束。

(2) 求每个事件的最早开始时间 Ve[i]和最迟开始时间 Vl[I]。

(3) 求每个活动的最早开始时间 e( )和最迟开始时间 l( )。

(4) 确定哪些活动是关键活动。画出由所有关键活动构成的图，指出哪些活动加速可使整个工程提前完成。

【解答】

按拓扑有序的顺序计算各个顶点的最早可能开始时间 Ve 和最迟允许开始时间 Vl。然后再计算各个活动的最早可能开始时间 e 和最迟允许开始时间 l，根据 l - e = 0? 来确定关键活动，从而确定关键路径。

	1	2	3	4	5	6
Ve	0	19	15	29	38	43
Vl	0	19	15	37	38	43

	<1, 2>	<1, 3>	<3, 2>	<2, 4>	<2, 5>	<3, 5>	<4, 6>	<5, 6>
e	0	0	15	19	19	15	29	38
l	17	0	15	27	19	27	37	38
l−e	17	0	0	8	0	12	8	0

此工程最早完成时间为 43。关键路径为<1, 3><3, 2><2, 5><5, 6>

8−20 若 AOE 网络的每一项活动都是关键活动。令 G 是将该网络的边去掉方向和权后得到的无向图。

(1) 如果图中有一条边处于从开始顶点到完成顶点的每一条路径上，则仅加速该边表示的活动就能减少整个工程的工期。这样的边称为桥(bridge)。证明若从连通图中删去桥，将把图分割成两个连通分量。

(2) 编写一个时间复杂度为 O(n+e)的使用邻接表表示的算法，判断连通图 G 中是否有桥，若有。输出这样的桥。