作业 4 图型结构及其应用

1. 分别实现无向图的邻接矩阵和邻接表存储结构的建立算法，分析和比较各建立算法的时间复杂度以及存储结构的空间占用情况。

为简化后续程序，本作业将邻接矩阵和邻接表储存在同一结构体中，具体如下：

```
struct graph
{
    int n,e;//图的顶点数与边数
    int edge[N][N] = {0};//邻接矩阵
    int vertex[N];//顶点表
    struct vertexNode vertexlist[N]; //顶点表
};
```

其中，struct vertexNode 为邻接表的表头数组，定义如下：

```
struct vertexNode
{
    int vertax; //数据域
    EdgeNode *phead; //边链表头指针
};

```

EdgeNode 为后续连接的节点，定义为：

```
struct EdgeNode
{
    int adjvex; //邻接点域
    int weight; //边权重
    EdgeNode *next; //指针域
};
```

建立思路为输入顶点个数以及边数后，逐边对邻接矩阵和邻接表进行修改或者插入，具体建立过程如下：

A. 无向图的邻接矩阵的建立

```
void create_matrix(struct graph *G)
{
    cout<<"图的顶点数为：";
    cin>>G->n;
    cout<<"图的边数为：";
    cin>>G->e;
    cout<<"此无向图的顶点为：";
    for(int i = 0; i < G->n; i++)
    {
        G->vertex[i] = i;
        cout<<i<<" ";
    }
    cout<<endl;
    for(int i = 0; i < G->e; i++)
    {
        int m,n,w;//边依附的两个顶点序号m，n；边权值w
```

```
        cout<<"第"<<i+1<<"条边的起止结点以及权值为：";
        cin>>m>>n>>w;
        G->edge[m][n] = w;
        G->edge[n][m] = w;
    }
    show_matrix(G);
}
```

B. **无向图**的邻接表的建立

```
void create_list(struct graph *G)
{
    cout<<"图的顶点数为：";
    cin>>G->n;
    cout<<"图的边数为：";
    cin>>G->e;
    cout<<"此无向图的顶点为：";
    for(int i = 0; i < G->n; i++)
    {
        G->vertex[i] = i;
        cout<<i<<" ";
        G->vertexlist[i].vertax = i;
        G->vertexlist[i].phead = NULL;
    }
    cout<<endl;
    for(int i = 0; i < G->e; i++)
    {
        int head,tail,w;//边依附的两个顶点序号m，n；边权值w
        cout<<"第"<<i+1<<"条边的起止结点以及权值为：";
        cin>>head>>tail>>w;
        EdgeNode *p = new EdgeNode;
        p->adjvex = tail; p->weight = w;
        p->next = G->vertexlist[head].phead;
        G->vertexlist[head].phead = p;
        EdgeNode *q = new EdgeNode;
        q->adjvex = head; q->weight = w;
        q->next = G->vertexlist[tail].phead;
        G->vertexlist[tail].phead = q;
    }
    show_list(G);
}
```

由程序可得，设e为图所含边数，n为顶点数目，无论邻接表或是邻接矩阵的建立时间复杂度均为$O(e)$，但是邻接表的空间复杂度为$O(n)$，而邻接矩阵的空间复杂度为$O(n^2)$。

2. 实现无向图的邻接矩阵和邻接表两种存储结构的相互转换算法。

转换过程和建立过程相似，由邻接表或邻接矩阵读出边信息后再存入对应的矩阵或者表中即可，具体程序如下：

A. 邻接表转换为邻接矩阵

```
void list_to_matrix(struct graph *G, struct graph *g)
{
    for(int i = 0; i < G->n; i++)
        g->vertex[i] = G->vertexlist[i].vertax;
    g->n = G->n;
    g->e = G->e;
    for(int i = 0; i < G->n; i++)
    {
        EdgeNode *p;
        p = G->vertexlist[i].phead;
        while(p != NULL)
        {
            int m = i, n = p->adjvex;
            g->edge[m][n] = p->weight;
            p = p->next;
        }
    }
    cout<<"邻接表转换为邻接矩阵......"<<endl;
    show_matrix(g);
}
```

B. 邻接矩阵转换为邻接表

```
void matrix_to_list(struct graph *G, struct graph *g)
{
    for(int i = 0; i < G->n; i++)
    {
        g->vertexlist[i].vertax = G->vertex[i];
        g->vertexlist[i].phead  = NULL;
    }
    g->n = G->n;
    g->e = G->e;
    for(int i = 0; i < G->n; i++)
        for(int j = 0; j < G->n; j++)
            if(G->edge[i][j] != 0)
            {
                int w = G->edge[i][j];
                EdgeNode *p = new EdgeNode;
                p->adjvex = j; p->weight = w;
                p->next = g->vertexlist[i].phead;
                g->vertexlist[i].phead = p;
            }
    cout<<"邻接矩阵转换为邻接表......"<<endl;
    show_list(g);
}
```

由程序可知，设e为图所含边数，n为顶点数目，则邻接表转换为邻接矩阵的时间复杂度为$O(n+e)$，邻接矩阵转换为邻接表的时间复杂度为$O(n^2)$。

3. 在上述两种存储结构上，分别实现无向图的深度优先搜索(递归和非递归)和广度优先搜索算法。并以适当的方式存储和展示相应的搜索结果，包括：深度优先或广度优先生成森林（或生成树）、深度优先或广度优先序列和深度优先或广度优先编号。并分析搜索算法的时间复杂度和空间复杂度。

A. 邻接表

1. 深度优先搜索(递归)

```
void DFS_list_recursion(struct graph G)
{
    int i, count = 1;
    for(int j = 0; j < G.n; j++)
        visited[j] = false;
    cout<<"深度优先序列为（递归）：  ";
    int search_tree[N][N] = {0};
    for(int i = 0; i < G.n; i++)
        if(!visited[i]) DFS1(G,i,&count,search_tree);
    cout<<endl;
    cout<<"深度优先编号为（递归）：  "<<endl;
    show_search(dfn);
    cout<<"深度优先生成树的邻接表为："<<endl;
    for(int i = 0; i < G.n; i++)
        for(int j = 0; j < G.n; j++)
            G.edge[i][j] = search_tree[i][j];
    struct graph g;
    matrix_to_list(&G, &g);
    //clear(&g);
}
```

```
void DFS1(struct graph G, int i, int *count, int search_tree[][N])
{
    EdgeNode *p = G.vertexlist[i].phead;
    if(!visited[i])
    {
        cout<<G.vertexlist[i].vertax<<" ";
        dfn[i] = (*count)++;
        visited[i] = true;
    }
    while(p != NULL)
    {
        if(!visited[p->adjvex])
        {
            search_tree[i][p->adjvex] = p->weight;
            search_tree[p->adjvex][i] = p->weight;
            DFS1(G,p->adjvex,count,search_tree);
        }
        p = p->next;
    }
}
```

2. 深度优先搜索(非递归)，基于栈结构

```
void DFS_list(struct graph G)
{
    int i, count = 1;
    stack<int> s;
    for(int j = 0; j < G.n; j++)
        visited[j] = false;
    cout<<"深度优先序列为（非递归）：  ";
    int search_tree[N][N] = {0};
    for(int i = 0; i < G.n; i++)
    {
        if(!visited[i])
        {
            cout<<G.vertexlist[i].vertax<<" ";
            dfn[i] = count++;
            visited[i] = true;
```

```
            s.push(i);
        }

        while(!s.empty())
        {
            EdgeNode *p = G.vertexlist[s.top()].phead;
            int m = G.vertexlist[s.top()].vertax;
            while(p != NULL)
            {
                if(!visited[p->adjvex])
                {
                    search_tree[m][p->adjvex] = p->weight;
                    search_tree[p->adjvex][m] = p->weight;
                    cout<<p->adjvex<<" ";
                    dfn[p->adjvex] = count++;
                    visited[p->adjvex] = true;
                    s.push(p->adjvex);
                    m = G.vertexlist[p->adjvex].vertax;
                    p = G.vertexlist[p->adjvex].phead;
                }
                else p = p->next;
            }
            p = G.vertexlist[s.top()].phead;
            m = G.vertexlist[s.top()].vertax;
            s.pop();
        }
    }
    cout<<endl;
    cout<<"深度优先编号为（非递归）：  "<<endl;
    show_search(dfn);
    cout<<"深度优先生成树的邻接表为："<<endl;
    for(int i = 0; i < G.n; i++)
        for(int j = 0; j < G.n; j++)
            G.edge[i][j] = search_tree[i][j];
    struct graph g;
    matrix_to_list(&G, &g);
    //clear(&g);
```

3. 广度优先搜索，基于队列结构

```
void BFS_list(struct graph G)
{
    int i, count = 1;
    queue<int> q;
    for(int j = 0; j < G.n; j++)
        visited[j] = false;
    cout<<"广度优先序列为：  ";
    int search_tree[N][N] = {0};
    for(int i = 0; i < G.n; i++)
    {
        if(!visited[i])
        {
            cout<<G.vertexlist[i].vertax<<" ";
            bfn[i] = count++;
            visited[i] = true;
            q.push(i);
        }
        while(!q.empty())
        {
            int i = q.front();
            q.pop();
            EdgeNode *p = G.vertexlist[i].phead;
            while(p != NULL)
            {
                if(!visited[p->adjvex])
                {
                    search_tree[i][p->adjvex] = p->weight;
```

```
                    search_tree[p->adjvex][i] = p->weight;
                    cout<<p->adjvex<<" ";
                    bfn[p->adjvex] = count++;
                    visited[p->adjvex] = true;
                    q.push(p->adjvex);
                }
                p = p->next;
            }
        }
    }
    cout<<endl;
    cout<<"广度优先编号为： "<<endl;
    show_search(bfn);
    cout<<"广度优先生成树的邻接表为："<<endl;
    for(int i = 0; i < G.n; i++)
        for(int j = 0; j < G.n; j++)
            G.edge[i][j] = search_tree[i][j];
    struct graph g;
    matrix_to_list(&G, &g);
    //clear(&g);
}
```

由程序可得，设n为顶点数，e为边数，则对邻接表进行深度（广度）优先搜索的时间复杂度为$O(n+e)$，对每个顶点都访问一次的时间复杂度为$O(n)$，而图的遍历过程是对每个顶点通过边查找邻接顶点的过程，这一过程开销为$O(e)$, 而空间复杂度为$O(n)$，这是因为最坏的情况下所有的顶点进入到栈或队列当中。

B. 邻接矩阵

1. 深度优先搜索(递归)

```
void DFS_matrix_recursion(struct graph G)
{
    int i, count = 1;
    for(int j = 0; j < G.n; j++)
        visited[j] = false;
    cout<<"深度优先序列为（递归）： ";
    int search_tree[N][N] = {0};
    for(int i = 0; i < G.n; i++)
        if(!visited[i]) DFS2(G,i,&count,search_tree);
    cout<<endl;
    cout<<"深度优先编号为（递归）： "<<endl;
    show_search(dfn);
    cout<<"深度优先生成树矩阵为："<<endl;
    for(int i = 0; i < G.n; i++)
    {
        for(int j = 0; j < G.n; j++)
            cout<<search_tree[i][j]<<" ";
        cout<<endl;
    }
}
```

```cpp
void DFS2(struct graph G, int i, int *count, int search_tree[][N])
{
    if(!visited[i])
    {
        cout<<G.vertex[i]<<" ";
        dfn[i] = (*count)++;
        visited[i] = true;
    }
    for(int j = 0; j < G.n; j++)
    {
        if((G.edge[i][j] != 0) && (!visited[j]))
        {
            search_tree[i][j] = G.edge[i][j];
            search_tree[j][i] = G.edge[i][j];
            DFS2(G,j,count,search_tree);
        }

    }
}
```

2. 度优先搜索(非递归)，基于栈结构

```cpp
void DFS_matrix(struct graph G)
{
    int i, count = 1;
    stack<int> s;
    for(int j = 0; j < G.n; j++)
        visited[j] = false;
    cout<<"深度优先序列为（非递归）：  ";
    int search_tree[N][N] = {0};
    for(int i = 0; i < G.n; i++)
    {
        if(!visited[i])
        {
            cout<<G.vertex[i]<<" ";
            dfn[i] = count++;
            visited[i] = true;
            s.push(i);
        }
        int j = 0, f = i;
        while(!s.empty())
        {
            while((j >= 0) && (j < G.n))
            {
                if((G.edge[f][j] != 0) && (!visited[j]))
                {
                    search_tree[f][j] = G.edge[f][j];
                    search_tree[j][f] = G.edge[f][j];
                    cout<<G.vertex[j]<<" ";
                    dfn[j] = count++;
                    visited[j] = true;
                    s.push(j);
                    f = j; j = 0;
                }
                else j++;
            }
            f = s.top();s.pop();
            j = 0;
```

```
        }
    }
    cout<<endl;
    cout<<"深度优先编号为（非递归）：  "<<endl;
    show_search(dfn);
    cout<<"深度优先生成树矩阵为: "<<endl;
    for(int i = 0; i < G.n; i++)
    {
        for(int j = 0; j < G.n; j++)
            cout<<search_tree[i][j]<<" ";
        cout<<endl;
    }
}
```

3. 广度优先搜索，基于队列结构

```
void BFS_matrix(struct graph G)
{
    int i, count = 1;
    queue<int> q;
    for(int j = 0; j < G.n; j++)
        visited[j] = false;
    cout<<"广度优先序列为:  ";
    int search_tree[N][N] = {0};
    for(int i = 0; i < G.n; i++)
    {
        if(!visited[i])
        {
            cout<<G.vertex[i]<<" ";
            bfn[i] = count++;
            visited[i] = true;
            q.push(i);
        }
        while(!q.empty())
        {
            int j = 0, f = q.front();
            q.pop();
            while((j >= 0) && (j < G.n))
            {
                if((G.edge[f][j] != 0) && (!visited[j]))
                {
                    search_tree[j][f] = G.edge[f][j];
                    search_tree[f][j] = G.edge[f][j];
                    cout<<G.vertex[j]<<" ";
                    bfn[j] = count++;
                    visited[j] = true;
                    q.push(j);
                }
                j++;
            }
        }
    }
```

由程序可得，则对邻接表进行深度（广度）优先搜索的时间复杂度为$O(n^2)$，因为需要对矩阵中每个位置进行遍历，而空间复杂度为$O(n)$，这是因为最坏的情况下所有的顶点进入到栈或队列当中。

4. 对于无向图，采用“邻接表”存储结构，设计和实现计算每个顶点度的算法，并分析其时间复杂度。

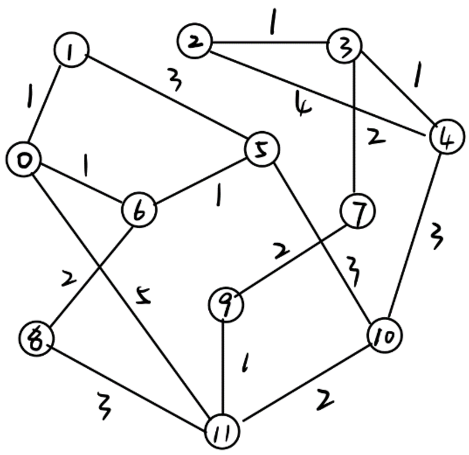
只需对每个顶点之后连接的顶点数进行遍历即可，具体程序如下：

```
void degree(struct graph G, int D[])
{
    for(int i = 0; i < G.n; i++)
    {
        D[i] = 0;
        EdgeNode *p;
        p = G.vertexlist[i].phead;
        while(p != NULL)
        {
            D[i]++;
            p = p->next;
        }
    }
    for(int i = 0; i < G.n; i++)
        cout<<"第"<<i<<"个顶点的度为："<<D[i]<<endl;
}
```

由程序可得，设n为顶点数，e为边数，则对邻接表计算度数的时间复杂度为$O(n+e)$，空间复杂度为$O(1)$

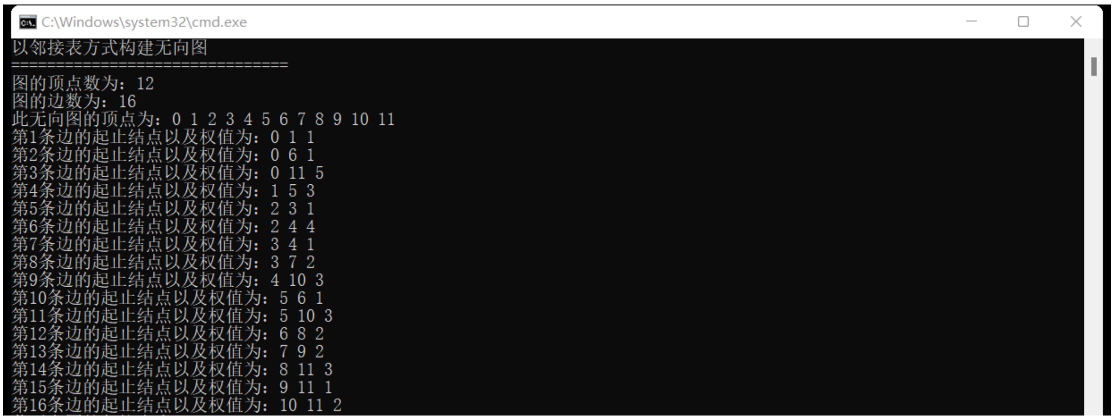
5. 以适当的方式输入图的顶点和边，并显示相应的结果。要求顶点不少于 10 个，边不少于 13 个。

如图所示，图中共有 12 个顶点，16 条边，包含权值等信息。为方便观察，以下分为两个程序，一个展示以邻接表形式储存图以及遍历，另外一种以邻接矩阵形式储存图以及遍历。



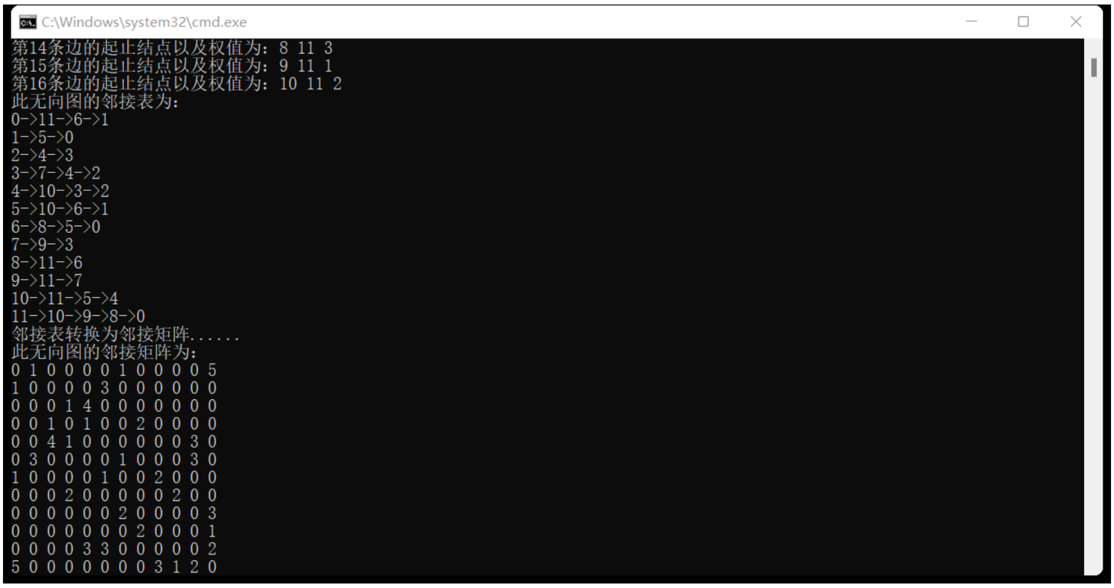
A. 以邻接表方式构建无向图：

1. 输入



```
C:\Windows\system32\cmd.exe
以邻接表方式构建无向图
==============================
图的顶点数为: 12
图的边数为: 16
此无向图的顶点为: 0 1 2 3 4 5 6 7 8 9 10 11
第1条边的起止结点以及权值为: 0 1 1
第2条边的起止结点以及权值为: 0 6 1
第3条边的起止结点以及权值为: 0 11 5
第4条边的起止结点以及权值为: 1 5 3
第5条边的起止结点以及权值为: 2 3 1
第6条边的起止结点以及权值为: 2 4 4
第7条边的起止结点以及权值为: 3 4 1
第8条边的起止结点以及权值为: 3 7 2
第9条边的起止结点以及权值为: 4 10 3
第10条边的起止结点以及权值为: 5 6 1
第11条边的起止结点以及权值为: 5 10 3
第12条边的起止结点以及权值为: 6 8 2
第13条边的起止结点以及权值为: 7 9 2
第14条边的起止结点以及权值为: 8 11 3
第15条边的起止结点以及权值为: 9 11 1
第16条边的起止结点以及权值为: 10 11 2
```

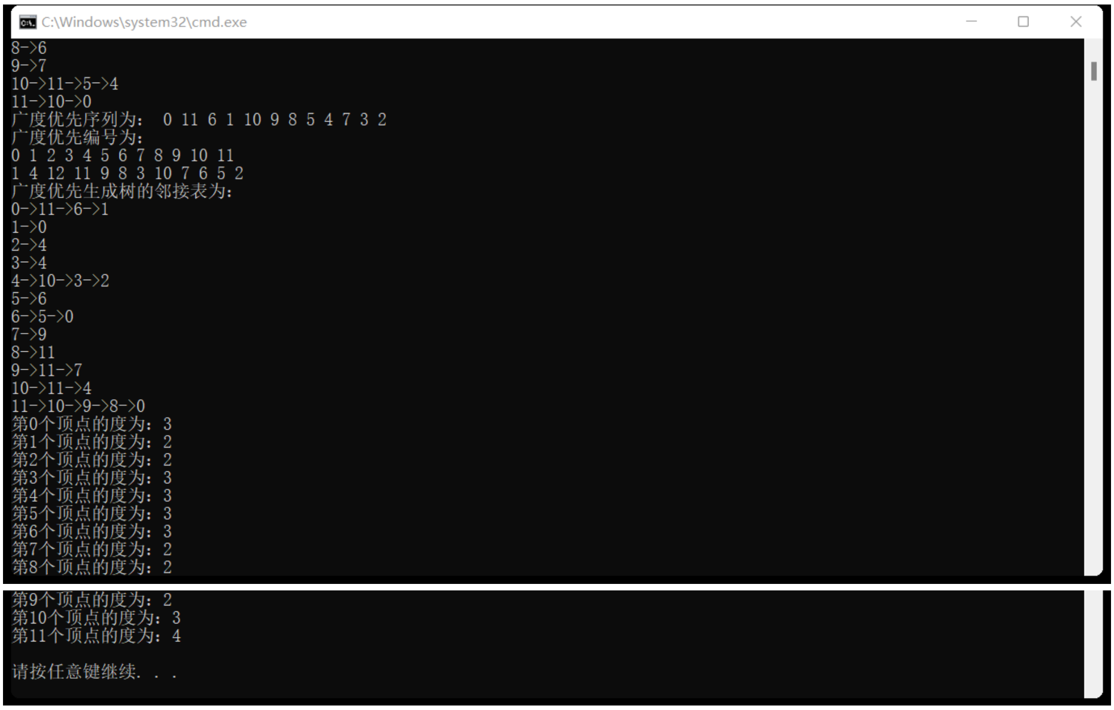
2. 显示以及转换



```
C:\Windows\system32\cmd.exe
第14条边的起止结点以及权值为: 8 11 3
第15条边的起止结点以及权值为: 9 11 1
第16条边的起止结点以及权值为: 10 11 2
此无向图的邻接表为:
0->11->6->1
1->5->0
2->4->3
3->7->4->2
4->10->3->2
5->10->6->1
6->8->5->0
7->9->3
8->11->6
9->11->7
10->11->5->4
11->10->9->8->0
邻接表转换为邻接矩阵......
此无向图的邻接矩阵为:
0 1 0 0 0 0 1 0 0 0 0 5
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 1 0 0 2 0 0 0 0
0 0 4 1 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 3 0
1 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 3 0 0 0 0 0 2
5 0 0 0 0 0 0 0 3 1 2 0
```

3. 搜索以及顶点度数

```
C:\Windows\system32\cmd.exe
深度优先序列为（递归）： 0 11 10 5 6 8 1 4 3 7 9 2
深度优先编号为（递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 7 12 9 8 4 5 10 6 11 3 2
深度优先生成树的邻接表为:
0->11
1->5
2->3
3->7->4->2
4->10->3
5->10->6->1
6->8->5
7->9->3
8->6
9->7
10->11->5->4
11->10->0
深度优先序列为（非递归）： 0 11 10 5 6 8 1 4 3 7 9 2
深度优先编号为（非递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 7 12 9 8 4 5 10 6 11 3 2
深度优先生成树的邻接表为:
0->11
1->5
2->3
3->7->4->2
4->10->3
5->10->6->1
6->8->5
7->9->3
```



```
8->6
9->7
10->11->5->4
11->10->0
广度优先序列为:  0 11 6 1 10 9 8 5 4 7 3 2
广度优先编号为:
0 1 2 3 4 5 6 7 8 9 10 11
1 4 12 11 9 8 3 10 7 6 5 2
广度优先生成树的邻接表为:
0->11->6->1
1->0
2->4
3->4
4->10->3->2
5->6
6->5->0
7->9
8->11
9->11->7
10->11->4
11->10->9->8->0
第0个顶点的度为: 3
第1个顶点的度为: 2
第2个顶点的度为: 2
第3个顶点的度为: 3
第4个顶点的度为: 3
第5个顶点的度为: 3
第6个顶点的度为: 3
第7个顶点的度为: 2
第8个顶点的度为: 2
第9个顶点的度为: 2
第10个顶点的度为: 3
第11个顶点的度为: 4

请按任意键继续. . .
```

4. 深度优先树以及广度优先树的图示

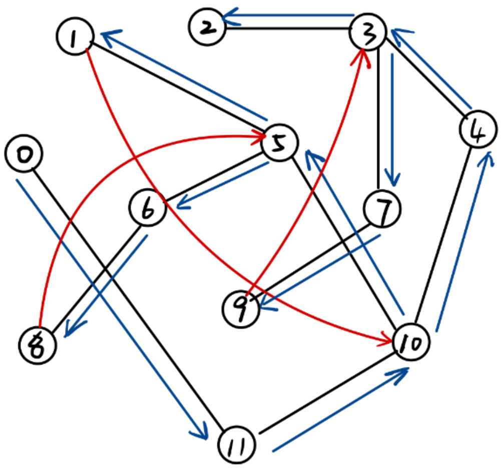


Figure 1 **深度优先树**

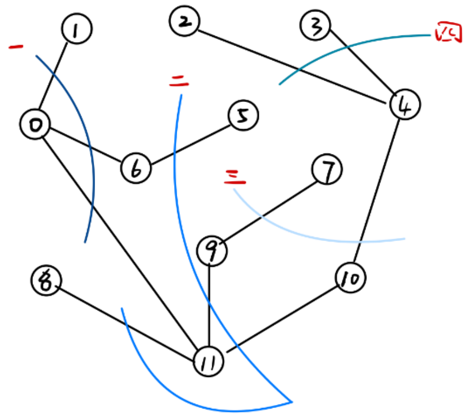


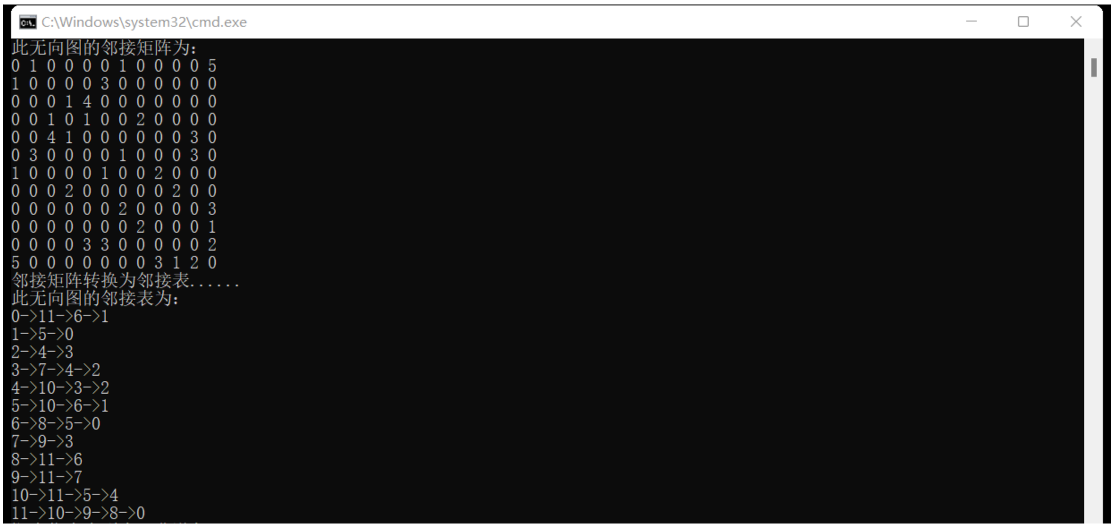
Figure 2 **广度优先树**

B. 以邻接矩阵方式构建无向图：

1. 输入

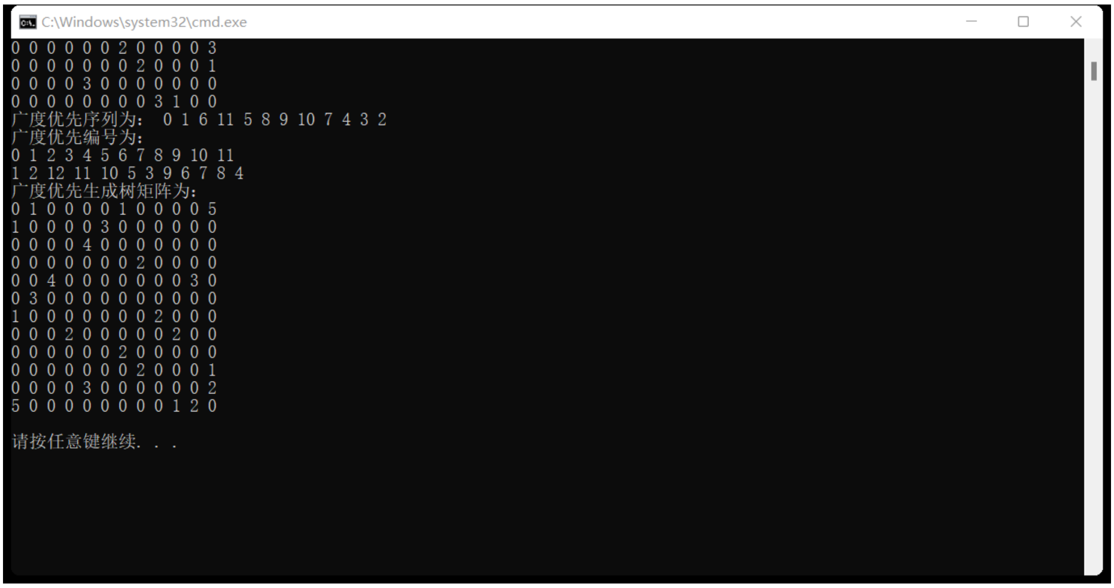

```
C:\Windows\system32\cmd.exe
以邻接矩阵方式构建无向图
===============================
图的顶点数为：12
图的边数为：16
此无向图的顶点为：0 1 2 3 4 5 6 7 8 9 10 11
第1条边的起止结点以及权值为：0 1 1
第2条边的起止结点以及权值为：0 6 1
第3条边的起止结点以及权值为：0 11 5
第4条边的起止结点以及权值为：1 5 3
第5条边的起止结点以及权值为：2 3 1
第6条边的起止结点以及权值为：2 4 4
第7条边的起止结点以及权值为：3 4 1
第8条边的起止结点以及权值为：3 7 2
第9条边的起止结点以及权值为：4 10 3
第10条边的起止结点以及权值为：5 6 1
第11条边的起止结点以及权值为：5 10 3
第12条边的起止结点以及权值为：6 8 2
第13条边的起止结点以及权值为：7 9 2
第14条边的起止结点以及权值为：8 11 3
第15条边的起止结点以及权值为：9 11 1
第16条边的起止结点以及权值为：10 11 2
```

2. 显示以及转换


```
C:\Windows\system32\cmd.exe
此无向图的邻接矩阵为:
0 1 0 0 0 0 1 0 0 0 0 5
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 1 0 0 2 0 0 0 0
0 0 4 1 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 3 0
1 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 3 0 0 0 0 0 2
5 0 0 0 0 0 0 0 3 1 2 0
邻接矩阵转换为邻接表......
此无向图的邻接表为:
0->11->6->1
1->5->0
2->4->3
3->7->4->2
4->10->3->2
5->10->6->1
6->8->5->0
7->9->3
8->11->6
9->11->7
10->11->5->4
11->10->9->8->0
```

3. 搜索


```
C:\Windows\system32\cmd.exe
深度优先序列为（非递归）：  0 1 5 6 8 11 9 7 3 2 4 10
深度优先编号为（非递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 2 10 9 11 3 4 8 5 7 12 6
深度优先生成树矩阵为:
0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 0 0 0 2 0 0 0 0
0 0 4 0 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 3 1 0 0
深度优先序列为（递归）：  0 1 5 6 8 11 9 7 3 2 4 10
深度优先编号为（递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 2 10 9 11 3 4 8 5 7 12 6
深度优先生成树矩阵为:
0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 0 0 0 2 0 0 0 0
0 0 4 0 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
```



```
C:\Windows\system32\cmd.exe
0 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 3 1 0 0
广度优先序列为:  0 1 6 11 5 8 9 10 7 4 3 2
广度优先编号为:
0 1 2 3 4 5 6 7 8 9 10 11
1 2 12 11 10 5 3 9 6 7 8 4
广度优先生成树矩阵为:
0 1 0 0 0 0 1 0 0 0 0 5
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 0 4 0 0 0 0 0 0 0
0 0 0 0 0 0 0 2 0 0 0 0
0 0 4 0 0 0 0 0 0 0 3 0
0 3 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 0 0 0 0 0 0 2
5 0 0 0 0 0 0 0 0 1 2 0

请按任意键继续. . .
```

4. 深度优先树以及广度优先树的图示

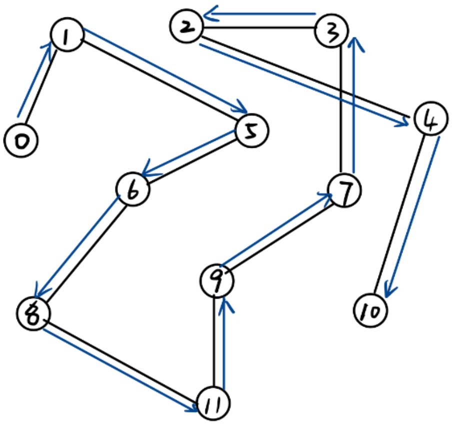


Figure 3 深度优先树

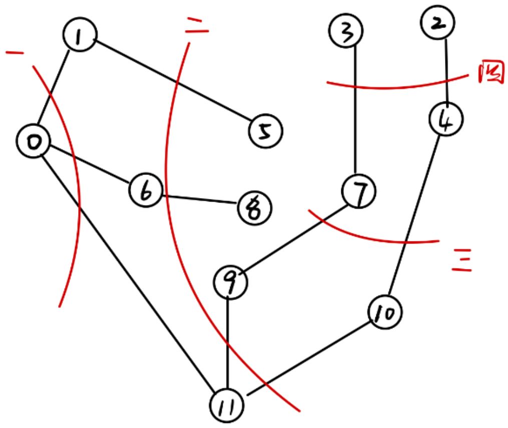


Figure 4 广度优先树