**《数据结构》实验报告**

**姓名： 杨孟衡 学号： 8002118240**

**班级： 软件工程1809班 专业： 软件工程**

**报告日期： 2019 年 12 月 8 日**

**实验八 图**

**一、问题描述与分析**

1、熟练掌握图的邻接矩阵与邻接表存储方法及其应用；

2、能设计出基于两种遍历算法的相关问题求解，如深度遍历生成树的求解、广度遍历生成树的求解；

3、理解并掌握最小生成树算法的基本思想及其算法方法；

4、理解并掌握最短路径算法的基本思想及其算法方法；

5、理解并掌握拓扑排序算法的基本思想及其算法方法。

2.分析

(1) 基于邻接表求无向图各顶点的度：嵌套循环，外层保证遍历所有结点，内层计算当前结点的度，然后输出度；

(2) 广度优先遍历：首先访问起始点，接着访问与当前结点邻接的顶点，然后以同样的方式访问顶点，为了保证先访问的顶点其邻接点亦先被访问，在搜索过程中可使用队列来保存已访问过的顶点；

(3) 深度优先遍历：对于给定的图，首先将结点中每一个顶点都标记为未访问，然后选取一个源点，将该点标记为已被访问，再递归地用深度优先遍历搜索方法依次搜索该点的所有邻接点；

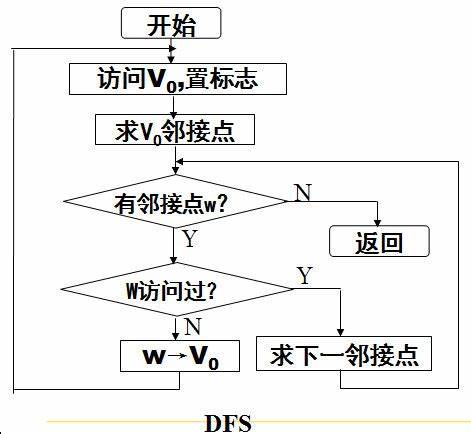
(4) prim算法：参见书上；

(5) Dijkstra算法：参见书上；

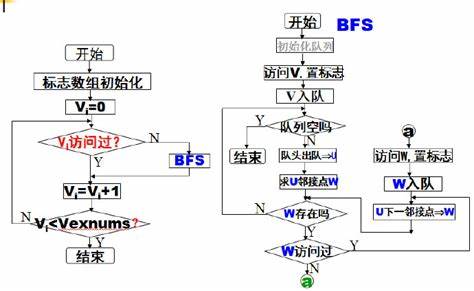
(6)拓扑排序：在有向图中找一个没有前趋的顶点并输出，在图中删除该顶点以及所有从该顶点发出的有向边，反复执行上述分析，直到所有顶点均被输出。

**二、数据结构与算法设计**

深度优先遍历（DFS）:



广度优先遍历（BFS）：



**三、算法复杂度分析**

（1）嵌套循环，算法复杂度为O（n^2）；

（2）嵌套循环，算法复杂度为O（n^2）；

（3）多循环，算法复杂度为O（2n）；

（4）嵌套循环，算法复杂度为O（n^2）；

（5）嵌套循环，算法复杂度为O（n^2）；

（6）嵌套循环，算法复杂度为O（n^2）；

**四、测试计划**

1.编写目的

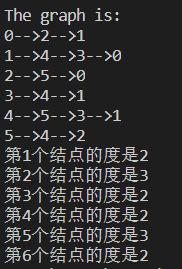
加深对数据结构链表，二叉树的存储结构，递归程序的理解。

2.开发及运行环境Dev c++

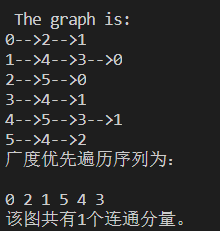
Visual Stdio Code 1.40

3.小项测试截图

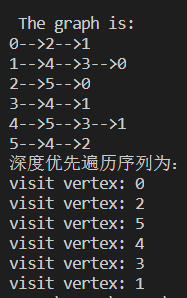
（1）基于邻接表求无向图各顶点的度



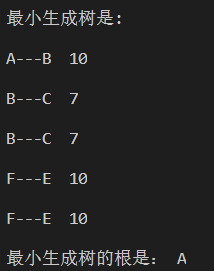
（2）广度优先遍历



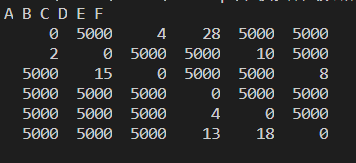
（3）深度优先遍历



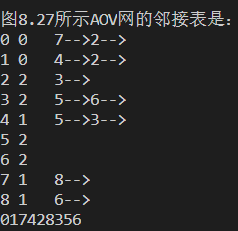
（4）Prim算法



（5）Dijkstra算法



（6）拓扑排序



**五、源程序(仅贴算法函数部分)**

**<lab8\_01>:**

void degree(LinkedGraph g)

{

    EdgeNode\* t;

    int num = g.n;

    int k;

    int i;

    for(i = 0; i < num; i++)

    {

        t = g.adjlist[i].FirstEdge;

        for(k = 1; ;k++)

        {

            if(t->next == NULL)

            {

                break;

            }

            t = t->next;

        }

        printf("第%d个结点的度是%d\n", i + 1, k);

    }

}

**<lab8\_02>:**

void bfs(LinkedGraph g, int i)

{ /\*从顶点i出发广度优先变量图g的连通分量\*/

  int j;

  EdgeNode\* p;

  int queue[M], front, rear;

  front = rear = 0;

  printf("%c ", g.adjlist[i].vertex);

  visited[i] = 1;

  queue[rear++] = i;

  while(rear > front)

  {

    j = queue[front++];

    p = g.adjlist[j].FirstEdge;

    while(p)

    {

      if(visited[p->adjvex] == 0)

      {

        printf("%c ", g.adjlist[p->adjvex].vertex);

        queue[rear++] = p->adjvex;

        visited[p->adjvex] = 1;

      }

      p = p->next;

    }

  }

}

**<lab8\_03>:**

void dfs(LinkedGraph g,int i)

{   /\*从顶点i开始深度优先遍历图的连通分量\*/

    EdgeNode \*p;

    printf("visit vertex: %c \n",g.adjlist[i].vertex);/\*访问顶点i\*/

    visited[i]=1;

    p=g.adjlist[i].FirstEdge;

    while (p)                 /\*从p的邻接点出发进行深度优先搜索\*/

    {

            if (visited[p->adjvex]==0)

                dfs(g,p->adjvex);

            p=p->next;

    }

}

/\*函数功能：深度优先遍历图

  函数参数：图的邻接表g

\*/

void DfsTraverse(LinkedGraph g)

{  int i;

   for (i=0;i<g.n;i++)

       visited[i]=0;     /\*初始化标志数组\*/

   for (i=0;i<g.n;i++)

       if (!visited[i])  /\*vi未访问过\*/

            dfs(g,i);

 }

**<lab8\_04>:**

void prim(Mgraph g, edge tree[M-1])

{  edge x;

   int d,min,j,k,s,v;

   /\* 建立初始入选点，并初始化生成树边集tree\*/

  for (v=1;v<=g.n-1;v++)

  {

      tree[v-1].beg=0;

      tree[v-1].en=v;

      tree[v-1].length=g.edges[0][v];

  }

  /\*依次求当前(第k条）最小两栖边，并加入TE\*/

  for (k=0;k<=g.n-3;k++)

  {

      min=tree[k].length;

      s=k;

      for (j=k+1;j<=g.n-2;j++)

            if (tree[j].length<min)

            {

                    min=tree[j].length;

                    s=j;

            }

        v=tree[s].en;

        x=tree[s];

        tree[s]=tree[k];

        /\*由于新顶点v的加入，修改两栖边的基本信息\*/

        for (j=k+1;j<=g.n-2;j++)

        {

            d=g.edges[v][tree[j].en];

            if (d<tree[j].length)

            {

                tree[j].length=d;

                tree[j].beg=v;

            }

        }

  }

   /\*输出最小生成树\*/

    printf("\n最小生成树是:\n");/\*输出最小生成树\*/

    for (j=0;j<=g.n-2;j++)

        printf("\n%c---%c  %d\n",g.vexs[tree[j].beg],g.vexs[tree[j].en],tree[j].length);

    printf("\n最小生成树的根是： %c\n", g.vexs[0]);

 }

**<lab8\_05>:**

void dijkstra(Mgraph g,int v0,path p,dist d)

 { boolean final[M]; /\*表示当前元素是否已求出最短路径\*/

   int i,k,j,v,min,x;

   /\*  第1步  初始化集合S与距离向量d \*/

    for (v=0;v<g.n;v++)

    {

        final[v]=FALSE;

        d[v]=g.edges[v0][v];

        if (d[v]<FINITY &&d[v]!=0)

                p[v]=v0; else p[v]=-1;

    }

    final[v0]=TRUE;

    d[v0]=0;

   /\* 第2步  依次找出n-1个结点加入S中   \*/

    for (i=1;i<g.n;i++)

    {

        min=FINITY;

        for (k=0;k<g.n;k++)

            if (!final[k] && d[k]<min)

            {

                v=k;

                min=d[k];

            }

            if (min==FINITY)     return ;

            final[v]=TRUE;

    /\*第3步 修改S与V-S中各结点的距离\*/

    for (k=0;k<g.n;++k)

            if (!final[k] && (min+g.edges[v][k]<d[k]))

            {

                d[k]=min+g.edges[v][k];

                p[k]=v;

            }

    }

}

/\*函数功能：输出有向图的最短路径

函数参数：邻接矩阵g；路径向量p；距离向量d

\*/

void print\_gpd(Mgraph g,path p,dist d)

 {

   int st[M],i,pre,top=-1;

   for (i=0;i<g.n;i++)

    { printf("\nDistancd: %7d , path:" ,d[i]);

      st[++top]=i;

      pre=p[i];

      while (pre!=-1)   /\*从第i个顶点开始向前搜索最短路径上的顶点\*/

        { st[++top]=pre;

          pre=p[pre];

         }

      while (top>0)

     printf("%2d",st[top--]);

    }

 }

**<lab8\_06>:**

int TopSort(AovGraph g)

 {int k=0,i,j,v, flag[M];

   int queue[M];  /\*队列\*/

   int h=0,t=0;

   edgenode\* p;

   for (i=0;i<g.n;i++)  flag[i]=0;  /\*访问标记初始化\*/

    /\*先将所有入度为0的结点进队\*/

    /\*将程序补充完整\*/

  for (i=0;i<g.n;i++)

        if (g.adjlist[i].id==0)

        {

            queue[t++]=i;

            flag[i]=1;

        }

   while (h<t)

   {

       v=queue[h++];

       printf("%c",g.adjlist[v].vertex);

       k++;

       p=g.adjlist[v].FirstEdge;

       while (p)

       {

           j=p->adjvex;

           g.adjlist[j].id--;

           if (g.adjlist[j].id==0 && flag[j]==0)

           {  queue[t++]=j;

               flag[j]=1;

           }

           p=p->next;

       }

   }

    return k;  //返回输出的顶点个数

 }