中 国 矿 业 大 学

2018 级《数据结构与算法分析》课程作业

学生姓名 王茂凯

学　　号 04181425

中国矿业大学信控学院

1. 简述深度优先遍历和广度优先遍历的算法思想，并且给出实现一次遍历的算法，求出图 1 中深度优先遍历和广度优先遍历得到的生成森林（树）。

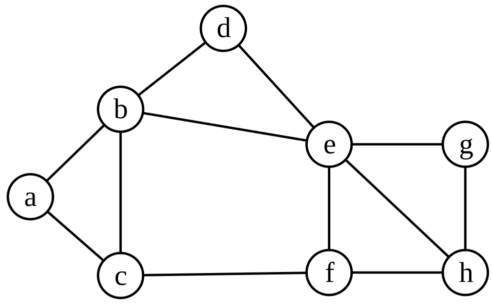
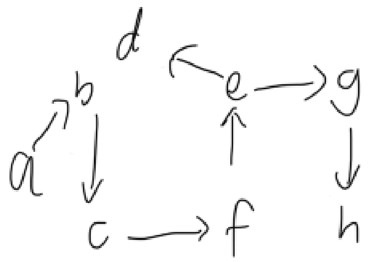


图 1

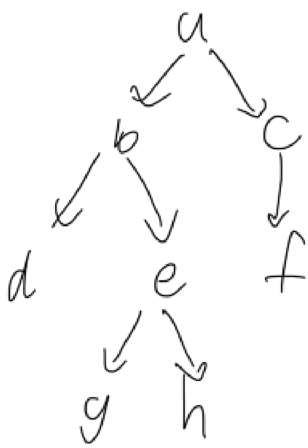
深度优先遍历:使用递归会栈,定义一个访问数组用来标记顶点是否被访问,先将图中顶点初始化为未访问,从图中的某个顶点 v 出发,访问并标记已访问,以此检查 v 的邻接点 w,如果 w 未被访问,则从 w 出发进行递归访问

广度优先遍历:使用队列,定义一个访问数组用来标记顶点是否被访问,从某个顶点出发,一次性访问其所有未被访问的邻接点.先将某个 v 顶点入队,若队不为空,则访问对顶并标记已访问,出队,之后遍历其所有邻接点,将其未被访问的邻接点入队,循环.队列为空时,算法结束.

深度优先遍历 a出发



广度优先遍历 a出发




```
#include "head.h"
using T = char;
class mygraph
{
public:
    mygraph(int n = 0, int e = 0, bool f = false) : _n(n),
                                                    _e(e),
                                                    vex(new T[_n]()),
                                                    edge(new int *[_n](
)),
                                                    flag(f),
                                                    visited(new int[_n]
())
    {
        for (int i = 0; i < _n; ++i)
            edge[i] = new int[_n]();
    }
    ~mygraph()
    {
```

```
        if (vex != nullptr)
            delete[] vex;
        for (int i = 0; i < _n; ++i)
            if (edge[i] != nullptr)
            {
                delete []edge[i];
            }
        vex = nullptr;
        if (edge != nullptr)
            delete[] edge;
        edge = nullptr;
        if (visited != nullptr)
            delete[] visited;
        visited = nullptr;
    }
    int findvex(const T &a)
    {
        for (int i = 0; i < _n; ++i)
            if (vex[i] == a)
                return i;
        return -1;
    }
    void getvex(const int &i)
    {
        if (i < 0 || i >= _n)
        {
            cout << "error1" << endl;
            return;
        }
        cout << vex[i] << " ";
    }
    void creategraph() //创建邻接表
    {
        cout << "input vex :" << endl; //输入顶点信息
        for (int i = 0; i < _n; ++i)
            cin >> vex[i];
        cout << "input vex to vex and w(1)" << endl; //输入顶点间关系
        int temp = 0;
        T tempvex1;
        T tempvex2;
        int tempweight = 0;
        int e = _e;
        while (e--)
        {
```

```
            cin >> tempvex1 >> tempvex2 >> tempweight;
            int i = findvex(tempvex1);
            int j = findvex(tempvex2);
            if (i != -1 && j != -1)
            {
                edge[i][j] = tempweight;
                if (!flag) //无向图
                {
                    edge[j][i] = tempweight;
                }
            }
            else
            {
                cout << "input error" << endl;
                e++;
            }
        }
    }
    void print() //以邻接表的形式打印图
    {
        cout << "output the graph:" << endl;
        for (int i = 0; i < _n; ++i)
        {
            for (int j = 0; j < _n; ++j)
                cout << edge[i][j] << " ";
            cout << endl;
        }
    }
    void clear()
    {
        for (int i = 0; i < _n; ++i)
            visited[i] = 0;
    }

private:
    int _n;        //顶点数
    int _e;        //边数
    T *vex;        //顶点数组
    int **edge;    //邻接矩阵
    bool flag;     //是否为有向图,默认 false 无向
    int *visited; //是否被访问过
    friend void dfsam(mygraph &g, const int &v);
    friend void bfsam(mygraph &g, const int &v);
};
```

```
void dfsam(mygraph &g, const int &v) //深度优先遍历,v 顶点开始遍历
{
    if (v < 0 || v >= g._n) //判断 v 是否在顶点范围内
    {
        cout << "error" << endl;
        return;
    }
    cout << g.vex[v] << " "; //输出并标记访问过
    g.visited[v] = 1;
    for (int w = 0; w < g._n; ++w) //递归
    {
        if (g.edge[v][w] != 0 && g.visited[w] == 0)
            dfsam(g, w);
    }
}
void bfsam(mygraph &g, const int &v) //广度优先遍历,v 顶点开始
{
    if (v < 0 || v >= g._n) //判断 v 是否在顶点范围内
    {
        cout << "error" << endl;
        return;
    }
    queue<T> que;        //使用队列
    que.push(g.vex[v]); //将 v 顶点入队
    while (!que.empty())
    {
        T u = que.front();      //取队头
        que.pop();              //出队
        int t = g.findvex(u);   //得到对头顶点的下标
        if (g.visited[t] != 1) //如果没有访问过则访问,并将其邻接点入队
        {
            g.visited[t] = 1; //标记为访问过
            cout << u << " ";
            for (int w = 0; w < g._n; ++w)
            {
                if (g.edge[t][w] != 0)  //存在路径
                    que.push(g.vex[w]); //入队
            }
        }
    }
}
int main()
{
    mygraph test(8, 12, false);
```

```
    test.creategraph();
    test.print();
    cout << "dfs the graph" << endl;
    dfsam(test, 0);
    cout << endl;
    test.clear();
    cout << "bfs the graph" << endl;
    bfsam(test, 0);
    return 0;
}
```

```
input vex :
a b c d e f g h
input vex to vex and w(1)
a b 1
a c 1
b c 1
b d 1
c f 1
b e 1
d e 1
e f 1
e g 1
e h 1
f h 1
g h 1
output the graph:
0 1 1 0 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 1 0 1 0 1 1 1
0 0 1 0 1 0 0 1
0 0 0 0 1 0 0 1
0 0 0 0 1 1 1 0
dfs the graph
a b c f e d g h
bfs the graph
a b c d e f g h
```

2. 分别采用 prim 算法和 kruskal 算法求出图 2 中的最小生成树，简述其算法思想。

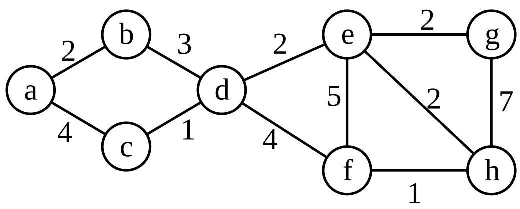
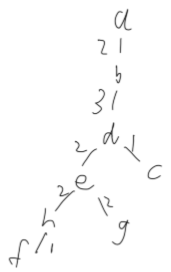


图 2

Prim:

	a	b	c	d	e	f	g	h
closest	a	a	d	b	b	h	e	e
lowcost	0	2	1	3	2	1	2	2

sumcost = 13



Kruskal:

1 $c \overset{1}{-} d$

2 $f \overset{1}{-} h$

3 $a \overset{2}{-} b$

4 $d \overset{2}{-} e$

5 $e \overset{2}{-} g$

6 $e \overset{2}{-} h$

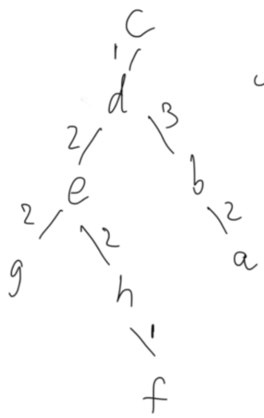
7 $b \overset{3}{-} d$

8 $a \overset{4}{-} c$

9 $d \overset{4}{-} f$

10 $e \overset{5}{-} f$

11 $g \overset{7}{-} h$



sumcost =

13

prim 以顶点为集合,将生成树路径上的顶点加入集合

算法步骤:

- 确定合适的数据结构,邻接矩阵 C,bool 数组是 s[i]=true,说明顶点 i 已加入集合 U,closest[j]表示 V-U 中的顶点 j 到集合 U 中的最近邻近点,lowcost[j]表示 V-U 中顶点 j 到集合 U 中的最邻近点的边值
- 初始化,令集合 U={u0},u0∈V,并初始化数组 closest[],lowcost[]和 s[]
- 在 V-U 集合中找 lowcost 值最小值的顶点 t,即 lowcost[t]=min{lowcost[j] | j∈V-U},满足该公式的顶点 t 就是集合 V-U 中连接集合 U 的最邻近点

- 将顶点 t 加入集合 U
- 如果集合 V-U 为空,算法结束,否则转到下一步
- 对集合 V-U 中的顶点 j,更新其 lowcost[j]和 closest[j],从第三步重复

kurskal 以边为集合,将边按权值排序,边两边的顶点进行合并
算法步骤:
- 初始化,将图 G 的边集数组 E 中的所有边按权值从小到大排序,初始化边集 TE,把每个顶点都初始化为一个孤立的分支
- 在 E 中寻找权值最小的边(i,j)
- 如果顶点 i 和 j 位于两个不同的连通分支,则将边(i,j)加入边集 TE,并执行合并操作,将两个连通分支进行合并,将(i,j)从集合 E 中删除
- 如果选取边数小于 n-1,重复第二步,否则,算法结束

3. 请给出 Dijkstra 算法和 Floyd 算法思想（可用伪代码描述）。

Dijkstra 算法采用的贪心策略是选择特殊路径长度最短的路径,将其连接的 V-S 中的顶点加入集合 S 中,同时更新数组 dist[],一旦 S 包含了所有顶点,dist[]就是从源到所有其它顶点之间的最短路径长度(V 是全部顶点的集合,S 是找到最短路径顶点的集合)
算法步骤:
- 数据结构.设置图的带权邻接矩阵为 G.Edge[][],采用一维数组 dist[i]来记录从源点到 i 顶点的最短路径长度,采用一维数组 p[i]来记录最短路径上 i 顶点的前驱
- 初始化.令集合 S={u},对于集合 V-S 中的所有顶点 x,初始化 dist[i]=G.Edge[u][i],如果源点 u 到顶点 i 右边相连,初始化 p[i]=u,否则 p[i]=-1
- 找最小.在集合 V-S 中依照贪心策略寻找使得 dist[j]具有最小值的顶点 t,则顶点 t 就是集合 V-S 中距离源点 u 最近的顶点(O(n))
- 加入 S 集合.将顶点 t 加入集合 S 中,同时更新 V-S
- 判结束.如果 V-S 为空,算法结束,否则转下一步
- 在第三步中以及找到了源点到 t 的最短路径,那么集合 V-S 中所有与顶点 t 相邻的顶点 j,都可以借助 t 走捷径.如果 dist[j]>dist[t]+G.Edge[t][j],则 dist[j]=dist[t]+G.Edge[t][j],记录顶点 j 的前驱为 t,有 p[j]=t,转第三步

```
void dijkstra(AMGraph &G, int u)
{
    int dist[G.vexnum];
    int p[G.vexnum];
    int flag[G.vexnum];
    for (int i = 0; i < G.vexnum; ++i)
    {
        dist[i] = G.edge[u][i]; //初始化源点 u 到其它各个顶点的最短路径长度
        flag[i] = 0;
        if (dist[i] == inf)
            p[i] = -1; //距离无穷大,u,i 不相邻
        else
```

```
            p[i] = u; //u,i 相邻,i 顶点的前驱 p[i]=u
    }
    dist[u] = 0;
    flag[u] = 1; //将源点 u 放入 S 集合
    for (int i = 0; i < G.vexnum; ++i)
    {
        int temp = inf, t;
        for (int j = 0; j < G.vexnum; ++j) //在 V-S 集合中寻找距离 u 最近的顶
点
        {
            if (!flag[j] && dist[j] < temp)
            {
                t = j;
                temp = dist[j];
            }
        }
        if (t == u)
        {
            cout<<"error"<<endl;
            return;
        }
        flag[t] = 1;                    //将 t 结点加入 S 集合
        for (int j = 0; j < G.vexnum; ++j) //更新与 t 相邻接的顶点到源点 u 的
距离
        {
            if (!flag[j] && G.edge[t][j] < inf)
            {
                if (dist[j] > (dist[t] + G.edge[t][j]))
                {
                    dist[j] = dist[t] + G.edge[t][j]; //更新 dist[j]
                    p[j] = t;                         //更新 j 的前驱顶点
                }
            }
        }
    }
    for (int i = 0; i < G.vexnum; ++i)
    {
        cout << dist[i] << " ";
    }
    cout << endl;
    for (int i = 0; i < G.vexnum; ++i)
    {
        cout << p[i] << " ";
    }
```

```
    cout << endl;
}
```

Floyd 算法可以求解任意两个顶点的最短路径,又称为了插点法,其核心算法是在顶点 i 到顶点 j 之间,插入顶点 k,看是否能缩短 i 和 j 之间距离

算法步骤:

- 数据结构.设置图的带权连接矩阵为 G.Edge[][],采用两个辅助数组,最短距离数组 dist[i][j],记录从 i 到 j 顶点的最短路径长度,前驱数组 p[i][j],记录从 i 到 j 顶点的最短路径上 i 顶点的前驱
- 初始化.初始化 dist[i][j]=G.Edge[i][j],如果顶点 i 到顶点 j 有边相连,初始化 p[i][j]=i,否则 p[i][j]=-1
- 插点.在 i,j 之间插入顶点 k,看能否缩短 i 和 j 之间的距离(松弛操作).如果 dist[i][j]>dist[i][k]+dist[k][j],则 dist[i][j]=dist[i][k]+dist[k][j],记录顶点 j 的前驱为:p[i][j]=p[k][j]

```
void floyd(AMGraph &G)
{
    int dist[G.vexnum][G.vexnum]; //记录顶点间距离数组
    int p[G.vexnum][G.vexnum];     //记录前驱顶点数组
    for (int i = 0; i < G.vexnum; ++i)
    {
        for (int j = 0; j < G.vexnum; ++j)
        {
            dist[i][j] = G.edge[i][j];
            if (dist[i][j] < inf && i != j)
                p[i][j] = i; //如果 i 和 j 之间有弧,则将 j 的前驱置位 i
            else
                p[i][j] = -1; //如果 i 和 j 之间无弧,则将 j 的前驱置为-1
        }
    }
    for (int k = 0; k < G.vexnum; ++k)
    {
        for (int i = 0; i < G.vexnum; ++i)
        {
            for (int j = 0; j < G.vexnum; ++j)
            {
                //i 经 k 到 j 的最短路径
                if (dist[i][k] + dist[k][j] < dist[i][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j]; //更新
dist[i][j]
                    p[i][j] = p[k][j];                    //更新 j 的前驱
为 k
                }
            }
```

```
        }
    }
    for (int i = 0; i < G.vexnum; ++i)
    {
        for (int j = 0; j < G.vexnum; ++j)
            cout << dist[i][j] << " ";
        cout << endl;
    }
    cout << endl;
    for (int i = 0; i < G.vexnum; ++i)
    {
        for (int j = 0; j < G.vexnum; ++j)
            cout << p[i][j] << " ";
        cout << endl;
    }
    cout << endl;
}
```