数据结构实验报告

姓名：陈卫星
学号：PB21051184
院系：信息科学技术学院

问题描述

实验题目

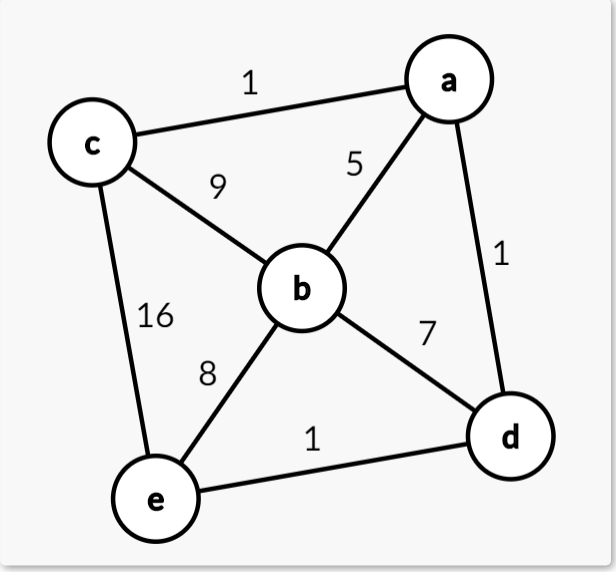
- 图及其应用

实验要求

- 使用邻接矩阵、邻接表作为图的储存结构，实现有向图、无向图、有向网、无向网的创建，图的非递归深度优先遍历、广度优先遍历，$Prim$算法构建最小生成树，单源最短路

测试数据

-



设计过程

图

- 由于分别使用邻接矩阵和邻接表储存的图所需的数据结构、构造方式、遍历方式均不同，但其在各种算法中的使用方式应当相同，故考虑创造一个图的父类、邻接矩阵图子类和邻接表图子类

```
template<class V,class A>class Graph;
public Graph<V,A>template<class V,class A>class ListGraph:public
Graph<V,A>{
    int mxvecnum,mxarcnum;
    int vecnum,arcnum;
    class Arc;
    class Vec{//点类型
        public:
        V val;//点值
        Arc* head;//邻接表头节点
    }*vecs;
    class Arc{//边类型
        public:
        Vec* v;//邻点
        A d;//边值
    };
}
template<class V,class A>class MatrixGraph:public Graph<V,A>{
    int mxvecnum,mxarcnum;
    int vecnum,arcnum;
    V* vecs;//点值
    A* arcs_;//一维化的边值
    A& arcs(int uid,int vid){return arcs_[uid*vecnum+vid];}
    A arcs(int uid,int vid)const{return arcs_[uid*vecnum+vid];}
    int getuid(void* arc)const{return ((A*)arc-arcs_)/vecnum;}
    int getvid(void* arc)const{return ((A*)arc-arcs_)%vecnum;}
}
```

图的创建

- 针对一般无向网，建图方式如下

```
template<class V,class A>void Graph<V,A>::CreateGraph(Graph<V,A>&
G,const string& path){
    FILE* fp=freopen(path.c_str(),"r",stdin);
```

```
    int vn,an;
    cin>>vn>>an;//点数与边数
    G.Init(vn,an);//初始化
    V* vs=new V[vn];
    for(int i=0;i<vn;i++)cin>>vs[i];//点值
    G.InitVec(vn,vs);//图初始化
    V u,v;A d;
    while(an--){
        cin>>u>>v>>d;//建无向边
        G.AddArc(u,v,d);
        G.AddArc(v,u,d);
    }
    fclose(fp);
    fclose(stdin);
    freopen("CON","r",stdin);cin.clear();
    delete[] vs;
}
```

- 若采用邻接表储存

```
template<class V,class A>void ListGraph<V,A>::Init(int vn=0,int
an=0){
    mxvecnum=vecnum=vn;
    mxarcnum=arcnum=an;
    vecs=new Vec[vn];
}
template<class V,class A>int ListGraph<V,A>::GetId(const V&
val)const{//根据点值找点编号
    for(int i=0;i<vecnum;i++)if(vecs[i].val==val)return i;
    ERROR("Can't find the vec!");
}
template<class V,class A>void ListGraph<V,A>::AddArc(const V&
u,const V& v,const A& d){//加边
    Vec *uid=&vecs[GetId(u)],*vid=&vecs[GetId(v)];
    uid->head=new Arc(vid,d,uid->head);
}
```

- 若采用邻接矩阵储存

```
template<class V,class A>void MatrixGraph<V,A>::Init(int vn=0,int
an=0){
    mxvecnum=vecnum=vn;
    mxarcnum=arcnum=an;
    vecs=new V[vn];
    arcs_=new A[vn*vn];
    memset(arcs_,0,sizeof(A)*vn*vn);
```

```
}
template<class V,class A>int MatrixGraph<V,A>::GetId(const V&
val)const{
    for(int i=0;i<vecnum;i++)if(vecs[i]==val)return i;
    ERROR("Can't find the vec!");
}
template<class V,class A>void MatrixGraph<V,A>::AddArc(const V&
u,const V& v,const A& d){
    arcs(GetId(u),GetId(v))=d;
}
```

图的遍历

- 由于邻接表和邻接矩阵储存结构存在巨大差异，且邻接表中的边类型被保护在 `ListGraph` 内，外界无法访问，故考虑使用 `void*` 类型指针 `arc` 分别指向两种储存结构的边。

```
V u;//顶点
void* arc=NULL;
while(G.GetNextArc(u,arc)){
    V v=G.GetVec(arc);//邻接点
    A d=G.GetArc(arc);//边值
}
```

- 对于邻接表， `arc` 即对应 `Arc*` 类型

```
template<class V,class A>void* ListGraph<V,A>::GetNextArc(const V&
u,void*& arc)const{
    if(!arc)return (void*)(arc=vecs[GetId(u)].head);//指向u的首条边
    return arc=((Arc*)arc)->next;
}
template<class V,class A>V ListGraph<V,A>::GetVec(void* arc)const{
    if(!arc)ERROR("The arc is wrong!");
    return ((Arc*)arc)->v->val;
}
template<class V,class A>A ListGraph<V,A>::GetArc(void* arc)const{
    if(!arc)ERROR("The arc is wrong!");
    return ((Arc*)arc)->d;
}
```

- 对于邻接矩阵，`arc`即对应一维化的`arcs_`数组，为`A*`类型

```
template<class V,class A>void* MatrixGraph<V,A>::GetNextArc(const
V& u,void*& arc)const{
    int uid=arc?getuid(arc):GetId(u),vid=arc?getvid(arc):-1;
    while(++vid<vecnum&&!arcs(uid,vid));//找到arc后的首条邻边
    return (void*)(arc=vid==vecnum?NULL:(arcs_+uid*vecnum+vid));
}
template<class V,class A>V MatrixGraph<V,A>::GetVec(void*
arc)const{
    if(!arc)ERROR("The arc is wrong!");
    return vecs[getvid(arc)];
}
template<class V,class A>A MatrixGraph<V,A>::GetArc(void*
arc)const{
    if(!arc)ERROR("The arc is wrong!");
    return *(A*)arc;
}
```

非递归深搜

```
template<class V,class A>void dfs(const MyGraph::Graph<V,A>&
G,const V& root){
    bool* vis=new bool[G.Vecnum()];
    memset(vis,0,sizeof(bool)*G.Vecnum());
    struct Tmp{
        int uid;
        void* arc;
        Tmp():arc(NULL){}
        Tmp(int uid,void* arc=NULL):uid(uid),arc(arc){}
    };
    Stack<Tmp>stk;
    int rootid=G.GetId(root);
    stk.push(Tmp(rootid));
    vis[rootid]=1;
    puts("dfs:");
    while(!stk.empty()){
        Tmp cur=stk.top();
        int uid=stk.top().uid;
        void*& arc=stk.top().arc;
        if(G.GetNextArc(G[uid],arc)){
            int vid=G.GetId(G.GetVec(arc));
            if(!vis[vid]){
```

```
                cout<<G[uid]<<"--->"<<G[vid]<<" dis="
<<G.GetArc(arc)<<endl;
                vis[vid]=1;
                stk.push(Tmp(vid));
            }
        }else stk.pop();
    }
    puts("");
    delete[] vis;
}
```

广搜

```
template<class V,class A>void bfs(const MyGraph::Graph<V,A>&
G,const V& root){
    bool* vis=new bool[G.Vecnum()];
    memset(vis,0,sizeof(bool)*G.Vecnum());
    vis[G.GetId(root)]=1;
    Queue<V>q;
    q.push(root);
    puts("bfs:");
    while(!q.empty()){
        V cur=q.front();q.pop();
        void* arc=NULL;
        while(G.GetNextArc(cur,arc)){
            V v=G.GetVec(arc);
            if(!vis[G.GetId(v)]){
                cout<<cur<<"---->"<<v<<" dis="<<G.GetArc(arc)
<<endl;
                vis[G.GetId(v)]=1;
                q.push(v);
            }
        }
    }
    puts("");
    delete[] vis;
}
```

*Prim*算法求最小生成树

- 由于边值类型未知，无法赋一个统一的极大值，故考虑使用 `vis` 数组表示相应的 `key` 值是否已赋值

```
namespace Prim{
    template<class T>T* getMinElem(T* begin,T* end,bool* acce){//
查询acce值为真的最小元素
        while(begin!=end&&!*acce)begin++,acce++;
        if(begin==end)return begin;
        T* res=begin;
        while(++begin!=end)if(*++acce&&*begin<*res)res=begin;
        return res;
    }
    template<class V,class A>void Prim(const MyGraph::Graph<V,A>&
G){
        puts("Prim:");
        A *key=new A[G.Vecnum()+1];
        int *fa=new int[G.Vecnum()];
        bool *acce=new bool[G.Vecnum()],*vis=new bool[G.Vecnum()];
        memset(acce,1,sizeof(bool)*G.Vecnum());
        memset(vis,0,sizeof(bool)*G.Vecnum());
        A ans=0;
        key[0]=0;
        vis[0]=1;
        for(int i=0;i<G.Vecnum();i++){
            int uid=getMinElem(key,key+G.Vecnum(),acce)-key;
            acce[uid]=0;
            if(i){
                cout<<"connect "
                    <<G[fa[uid]]
                    <<" and "
                    <<G[uid]
                    <<" dis="
                    <<key[uid]<<endl;
                ans+=key[uid];
            }
            void* arc=NULL;
            while(G.GetNextArc(G[uid],arc)){
                int vid=G.GetId(G.GetVec(arc));
                A w=G.GetArc(arc);
                if(!vis[vid]||w<key[vid]){
                    vis[vid]=1;
                    key[vid]=w;
                    fa[vid]=uid;
                }
            }
        }
        cout<<"total val:"<<ans<<'\n'<<endl;
        delete[] key;
```

```
        delete[] fa;
        delete[] vis;
        delete[] acce;
    }
}
```

*Dijkstra*算法求单源最短路（堆优化）

```
namespace Dijkstra{
    template<class V,class A>void PrintPath(const
MyGraph::Graph<V,A>& G,const int* fa,int uid){
        if(uid==-1)return;
        PrintPath(G,fa,fa[uid]);
        cout<<G[uid]<<' ';
    }
    template<class V,class A>void Dijkstra(const
MyGraph::Graph<V,A>& G,const V& root){
        cout<<"Shortest pathes root="<<root<<endl;
        bool* vis=new bool[G.Vecnum()];
        A* dis=new A[G.Vecnum()];
        int* fa=new int[G.Vecnum()];
        struct node{
            int uid;
            A d;
            node(){}
            node(int uid,const A& d):uid(uid),d(d){}
            bool operator <(const node& _)const{return d>_.d;}
        };
        Heap<node>h;
        memset(vis,0,sizeof(bool)*G.Vecnum());
        int rootid=G.GetId(root);
        h.push(node(rootid,dis[rootid]=0));
        fa[rootid]=-1;
        vis[rootid]=1;
        while(!h.empty()){
            node cur=h.top();h.pop();
            int uid=cur.uid;
            if(vis[uid]&&dis[uid]<cur.d)continue;
            if(dis[uid]){
                PrintPath(G,fa,uid);
                cout<<"dis="<<dis[uid]<<endl;
            }
            void* arc=NULL;
            while(G.GetNextArc(G[uid],arc)){
                int vid=G.GetId(G.GetVec(arc));
```

```
                A w=G.GetArc(arc);
                if(!vis[vid]||dis[uid]+w<dis[vid]){
                    vis[vid]=1;
                    dis[vid]=dis[uid]+w;
                    fa[vid]=uid;
                    h.push(node(vid,dis[vid]));
                }
            }
        }
        delete[] vis;
        delete[] dis;
        delete[] fa;
    }
}
```

调试分析

算法时空复杂度分析：

- 由于在外部算法中顶点均储存为点值类型，而在图的储存结构里为编号，故几乎每次调用图的函数都会需要调用 `int GetId(const V&)const` 函数来查找顶点编号，并产生额外的$O(\nu(G))$的复杂度
- 此时可以考虑利用二分查找或 `map` 实现 `GetId` 函数，将额外的复杂度降为$O(\log_2 \nu(G))$

调试结果

- 测试数据为：
    - ```
      5 8
      a b c d e
      a b 5
      c a 1
      a d 1
      b e 8
      c b 9
      b d 7
      e c 16
      d e 1
      c
      ```
    - 其中图从文件 `data.in` 输入
- 运行结果为：
      ```

- 


```
420                 }
421             }
422         }
423         delete[] vis;
424         delete[] dis;
425         delete[] fa;
426     }
427 }
428 int main(){
429     string s="data.in";
430     MyGraph::ListGraph<char,int>lG;
431     MyGraph::CreateGraph(lG,s);
432     MyGraph::MatrixGraph<char,int>mG;
433     MyGraph::CreateGraph(mG,s);
434     puts("Done Create Graph");
435 
436     char root;
437     cin>>root;
438     dfs(lG,root);
439     dfs(mG,root);
440 
441     bfs(lG,root);
442     bfs(mG,root);
443 
444     Prim::Prim(lG);
445     Prim::Prim(mG);
446 
447     Dijkstra::Dijkstra(lG,root);
448     Dijkstra::Dijkstra(mG,root);
449 }
450 /*
451 5 8
452 a b c d e
453 a b 5
454 c a 1
455 a d 1
456 b e 8
```

附录

原程序文件清单：

```
bits/stdc++.h
图.cpp
data.in
```