图论4

图的三种存储方法

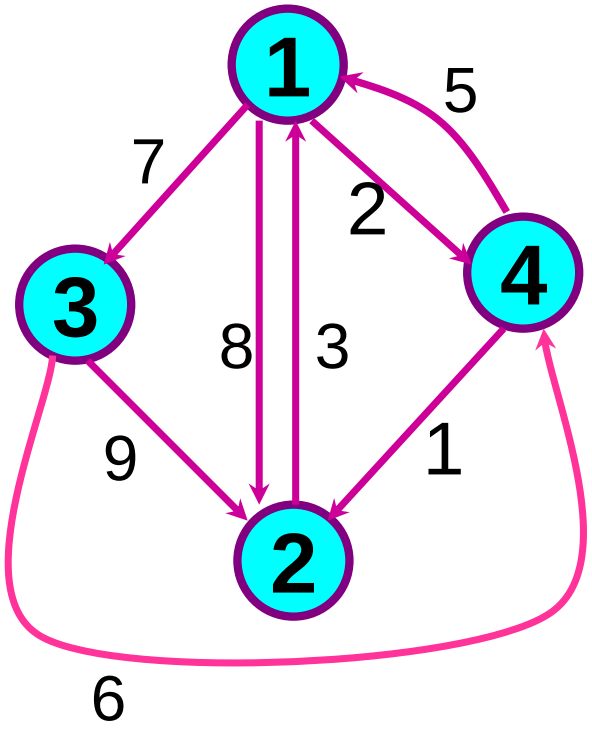
方法一：邻接矩阵

图的存储方法1：邻接矩阵

输入：
4 8
1 3 7
1 2 8
1 4 2
3 4 6
3 2 9
……

	1	2	3	4
1	0	8	7	2
2	3	0	∞	∞
3	∞	9	0	∞
4	5	1	∞	0

int Map[5][5];




```
cin>>n>>m;
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(i!=j)Map[i][j]=inf;else Map[i][j]=0;
for(i=1;i<=m;i++)//总共m条边
{
        cin>>x>>y>>z;
        Map[x][y]=z;
}
```

找出k号点能直接到达的点

```
for(i=1;i<=n;i++)
    if(Map[k][i]!=inf)
    {
            //k能直接到i
    }
```

二.vector

vector，实际上就是个动态数组。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能,但在中间插入慢。它里面存储的元素可以是任意类型。

#include <vector>
常用函数：
push_back(e) - 在数组尾部添加一个元素e,数组长度自动+1
pop_back() - 删除数组最后一个元素,但无返回值,数组长度自动-1
front() - 得到数组第一个元素
back()-得到数组最后一个元素
size()-数组中元素的个数
empty()-判断数组是否为空
clear()-清空整个数组

vector相当于是一个动态数组，它也可进行下标操作。

```
#include<vector>
using namespace std;
vector<int> vt;
int main()
{
        int n,x;
        cin>>n;
        for(int i=1;i<=n;i++)
        {
                cin>>x;
                vt.push_back(x);
        }
        for(int i=0;i<vt.size();i++)cout<<vt[i]<<" ";   //输出1 3 5 7 9
        cout<<endl;
        vt[3]=100;              //可用下标来修改已存在的元素
        for(i=vt.size()-1;i>=0;i--)cout<<vt[i]<<" ";    //输出?        9 100 5 3 1
}
```

输入：
5
1 3 5 7 9

vector用"[ ]"来随机访问已经存在的元素

vector可以方便表示一个邻接表。

```
#include<vector>
using namespace std;
vector<int> G[10];          //申明了一个vector数组，相当于有10个vector
int main()
{
    int i,n,m,x,y,k;
    cin>>n>>m;
    for(i=1;i<=m;i++)
    {
            cin>>x>>y;
            G[x].push_back(y);
    }
    cin>>k;    //输出跟k号点相连的点的编号
    for(i=0;i<G[k].size();i++)cout<<G[k][i]<<" ";
    return 0;
}
```

输入格式：
第一行，两个整数n和m，分别代表有向图中点和边的数量
接下来m行，每行两个整数x，y 表示一条边从x出发指向y
接下来一行，一个整数k
输出k能直接到达的点的编号

样例输入：

```
4 6
1 3
2 3
1 4
2 1
3 1
2 4
2
```

样例输出？

3 1 4

上面是用下标来访问的，也可迭代器的形式：

```
vector<int>::iterator it;
for(it=G[k].begin();it!=G[k].end();it++)
cout<<*it<<" ";
```

```
struct edge{
        int End;
        int Len;
    };

vector<edge> G[10];
int main()
{
    int i,n,m,x,y,z,k;
    cin>>n>>m;
    for(i=1;i<=m;i++)
    {
        cin>>x>>y>>z;
        G[x].push_back(edge(y,z));
    }
    cin>>k;
    for(i=0;i<G[k].size();i++)
        cout<<G[k][i].End<<" "<<G[k][i].Len<<endl;
}
```

输入格式：
第一行，两个整数n和m，分别代表有向图中点和边的数量
接下来m行，每行两个整数x, y, z表示一条边从x出发指向y, **长度为z**
接下来一行，一个整数k
输出k能直接到达的点的编号，及对应边长
样例输入：

```
4 6
1 3 5
2 3 7
1 4 1
2 1 3
3 1 9
2 4 11
2
```

样例输出？

```
3 7
1 3
4 11
```

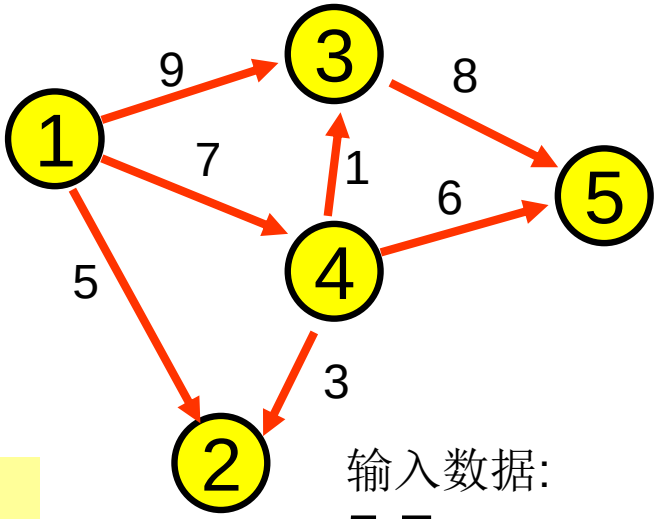
方法三：边存储

图的存储：存边

```
#define maxm 1000000
#define maxn  10000
int Next[maxm],end[maxm],len[maxm],last[maxn];
//Next[i]表示跟第i条边有相同起点的上一条边的编号
//last[x]表示以x为起点的边最新一条的边的编号
```

```
cin>>n>>m
for(i=1;i<=m;i++)
{
   cin>>x>>y>>z;
   end[i]=y;
   len[i]=z;
   Next[i]=last[x];
   last[x]=i;
}
```

```
//讨论与x相关的边或点，例如输出从点x出发的边
t=last[x];
while(t!=0)
{
    cout<<x<<"  "<<end[t]<<"  "<<len[t]<<endl;
    t=Next[t];
}
```



输入数据:

```
5 7
4 2 3
1 4 7
4 5 6
3 5 8
4 3 1
1 2 5
1 3 9
```

查询4出发的边和指向点

	1	2	3	4	5	6	7
len[ ]	3	7	6	8	1	5	9
end[ ]	2	4	5	5	3	2	3
Next[ ]	0	0	1	0	3	2	6
last[ ]	267		4	1 35			

图的存储：存边

```
#define maxm 100000
#define maxn  50000
int Next[maxm],end[maxm],len[maxm],last[maxn];
//Next[i]表示跟第i条边有相同起点的边最近出现的位置
//last[x]表示以x为起点的边最新出现的位置
cin>n>>m
for(i=1;i<=m;i++)
{
   cin>>x>>y>>z;
   end[i]=y;
   len[i]=z;
   Next[i]=last[x];
   last[x]=i;
}
```

```
//讨论与x相关的边或点，例如输出从点x出发的边
t=last[x];
while(t!=0)
{
    cout<<x<<"  "<<end[t]<<"  "<<len[t]<<endl;
    t=Next[t];
}
```

用边存储改进SPFA

```
void SPFA(int s)
……
for(i=1;i<=n;i++)dis[i]=inf;
q.push(s);   f[s]=true;    dis[s]=0;
while(q.empty()==false)
{
    x=q.front();  q.pop();  f[x]=false;
    t=last[x];
    while(t!=0)
   {
        y=end[t];
        if (dis[x]+len[t]<dis[y])
        {
            dis[y]=dis[x]+len[t];
            if(f[y]==false)
            {
                f[y]=true;
                q.push(y);
            }
        }
        t=Next[t];
    }
}
```

采用链式存储可大大减少讨论的次数

USACO 3.2.6