//重载DShortestPath，在约束下的最短路径

```
 void Graph_List :: DShortestPath(const int v, int **deleted_edge,int *path, int *dist )
{
	int max = 10000;
	int u , k ;
	Edge *p ;
	int n = graphsize;
	int* s = new int[n] ;                                   // 数组s[i]记录i是否被访问过
	for( int i = 0 ; i < n ; i ++ )                 // 数组path, dist, s初始化
	{ path[i] = -1 ; dist[i] = max ; s[i] = 0 ; }
	dist[v] = 0 ;        s[v] = 1 ;                 // 初始顶点v的数组值
	p= Head[v].adjacent;
	u = v ;                                                 // u为即将访问的顶点
	for(int j = 0 ; j < n ; j ++ )                  // 循环(1)
	{
		// 循环(2)：修改u邻接顶点的s[]值、path[]值和dist[]值
		while( p != NULL )
		{
			k = p-> VerAdj ;
			if( s[k] != 1&& deleted_edge[u][k]!=1 && dist[u] + p->cost < dist[k] )
			{
				dist[k] = dist[u] + p->cost ;
				path[k] = u ;
			}
			p = p->link ;
		}
		// 循环(3)：确定即将被访问的顶点u
		int ldist = max;
		for(int    i = 0 ; i < n ; i ++ )
			if( dist[i] > 0 && dist[i] < ldist && s[i] == 0 )
			{ ldist = dist[i];    u = i ; }
			s[u] = 1 ;                              // 访问u
			p = Head[u].adjacent ;                  // p为u的边链的头指针

	}
}
//需要两个辅助函数
int findminlength(int *length,int m) //找到最短路径的存储位置
{
	int minidx=0;
	int min=1000;
	for(int i=0;i<m;i++)
	{
		if(length[i]<min)
		{
			minidx=i;
			min=length[i];
		}
	}
	return minidx;
}
void changepath(int *path,int n,int &l,int *q) //把路径转化为按照节点顺序的数组形式存储
{
	if(path[n-1]==-1)
	{
		cout<<"路径不存在。\n";
	}else
	{
		int i=n-1;
		l=1;//路径长度
		int *p=new int[n];
		p[0]=n-1;
		while(path[i]!=-1)
		{
			p[l]=path[i];
```

```
                i=path[i];
                l=l+1;
            }
            for(int i=l-1;i>=0;i--)
            {
                q[l-i-1]=p[i];
            }
        }
}
void Graph_List::MshortestPath(const int v,int M)
{
        int n = graphsize; //节点个数
        struct Path //节点保存路径的长度以及路径的初始顶点
        {
            int *next;
        };
        struct constraints_edge //保存约束的边，包含或者不包含
        {
            int v1;
            int v2;
            constraints_edge *next;
        };
        struct c_includehead   //约束（包含的边链表）的哨位节点
        {
            int num_con_edges;//约束的边数
            constraints_edge *next;
        };
        struct c_excludehead   //约束（不包含的边链表）的哨位节点
        {
            constraints_edge *next;
        };

        Path Q[100];
        int *length=new int[100]; //保存每一条生成边的长度
        for(int i=0;i<100;i++)
        {
            length[i]=1000;
        }
        int *path_arr=new int[n];//以数组的形式表示路径

        c_includehead included_edge[100]; //保存每一条生成边的约束，存在或者不存在某些边
        c_excludehead excluded_edge[100];

        int m=0;
        int *path=new int[n];

        int *dist=new int[n];
        int **deleted_edge=new int*[n]; //二维数据存储删除的边，删除的边为1，没删除的边为1
        for(int j=0;j<n;j++)   //二维数组初始化
        {
            deleted_edge[j]=new int[n];
            for(int k=0;k<n;k++)
            {
                deleted_edge[j][k]=0;
            }
        }
        DShortestPath(0, deleted_edge,& *path, & *dist );

        Q[m].next=path;
        length[m]=dist[n-1]; //路径长度，是指每条边的花费之和，不是边数
        included_edge[m].next=NULL;
        included_edge[m].num_con_edges=0;
        excluded_edge[m].next=NULL;
        int minindex;//最短路径存储在Q中的位置
        int Edge_num; //最短路径的边数
```

```
//找M条最短路径
for(int i=0;i<M;i++)
{
	int *minpath=new int[n];
	int * minpath1=new int[n]; //节点顺次的方法表示路径
	int **deleted_edge=new int*[n]; //二维数据存储删除的边，删的边为1，没删的边为1
	for(int j=0;j<n;j++)
	{
		deleted_edge[j]=new int[n];
		for(int k=0;k<n;k++)
		{
			deleted_edge[j][k]=0;
		}
	}
	minindex=findminlength(length,m+1);//找到最短路径在Q中的存储位置
	memcpy(minpath, Q[minindex].next, n*sizeof(int));
	changepath(minpath,n,Edge_num,& *minpath1); //把路径按节点的顺序表示出来
	cout<<"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"<<endl;
	cout<<"minindex:"<<minindex<<endl;
	cout<<"第"<<i+1<<"短路径:";
	for (int j = 0;j<Edge_num;j++) //把最短路径换成节点的次序存储
	{
		cout<<minpath1[j]<<",";
	}
	cout<<"        长度："<<length[minindex]<<endl;
	length[minindex]=10000;       //将这条路径的长度设为最大，在后面不再出现
	int include_cost=0;//约束中包含的边的权重
	int include_num=0; //约束中包含的边的数目
	include_num=included_edge[minindex].num_con_edges;
	cout<<"约束条件--包含的边：";
	constraints_edge *E;
	E=included_edge[minindex].next;
	while(E!=NULL) //包含的边
	{
		cout<<"<"<<E->v1<<","<<E->v2<<">"<<",";
		include_cost=include_cost+GetWeight(E->v1,E->v2);
		deleted_edge[E->v1][E->v2]=1;
		E=E->next;
	}
	cout<<"\n";
	cout<<"约束条件--不包含的边：";
	constraints_edge *F;
	F=excluded_edge[minindex].next;
	while(F!=NULL)//不包含的边
	{
		cout<<"<"<<F->v1<<","<<F->v2<<">"<<",";
		deleted_edge[F->v1][F->v2]=1;
		F=F->next;
	}
	cout<<endl;
	int cost_last=0; /*从最短路径的最后一条边开始删除，因为上一次不包含的边下
			一次要包含，所以需要存储删除的边的权重*/
	int cost_last_temp=0;
	int temp_con_edges=0;
	for(int j=Edge_num-1-include_num;j>0;j--)   /*从上一步最短路径的最后一条边（最后
				一条没有在“约束条件--包含的边”中的边）开始删除*/
	{
		m=m+1;
		constraints_edge *excluded_ec=new constraints_edge; //新产生的约束--不包含的边
		constraints_edge *included_ce=new constraints_edge;//新产生的约束--包含的边
		excluded_ec->v1=minpath1[j-1];
		excluded_ec->v2=minpath1[j];
		excluded_ec->next=excluded_edge[minindex].next;;
```

```
                excluded_edge[m].next=excluded_ec;
                cost_last=GetWeight(excluded_ec->v1,excluded_ec->v2)+cost_last;
                deleted_edge[excluded_ec->v1][excluded_ec->v2]=1;
                if(j!=Edge_num-1-include_num)
                {
                        temp_con_edges=temp_con_edges+1;
                        included_ce->v1=minpath1[j];
                        included_ce->v2=minpath1[j+1];
                        included_ce->next= included_edge[minindex].next;
                        included_edge[minindex].next=included_ce;
                        included_edge[m].next= included_edge[minindex].next;
                   included_edge[m].num_con_edges=
                   included_edge[minindex].num_con_edges+temp_con_edges;
                }else{
                        included_edge[m].next= included_edge[minindex].next;
                        included_edge[m].num_con_edges= included_edge[minindex].num_con_edges;
                }
                int *path=new int[n];
                int *dist=new int[n];
                DShortestPath(0,deleted_edge, & *path, & *dist );
                if(path[j]!=-1)
                {
                        int k=n-1;
                        while(k!=minpath1[j])
                        {
                                path[k]=minpath[k];
                                k=minpath[k];
                        }
                }
                Q[m].next=path;
                length[m]=dist[minpath1[j]]+include_cost+cost_last_temp;
                cout<<"----------------------------"<<endl;
                if(length[m]>=10000)
                {
                        cout<<"新产生的路径不存在.."<<endl;
                }else{
                        int *newpath=new int[n];
                        changepath(path,n,Edge_num,& *newpath);
                        cout<<"新产生的路径:";
                        for (int k = 0;k<Edge_num;k++) //把最短路径换成节点的次序存储
                        {
                                cout<<newpath[k]<<",";
                        }
                        cout<<"       长度："<<length[m]<<endl;
                }
                cost_last_temp=cost_last;
        }
        delete[] minpath;
        delete[] minpath1;

    }
}
```