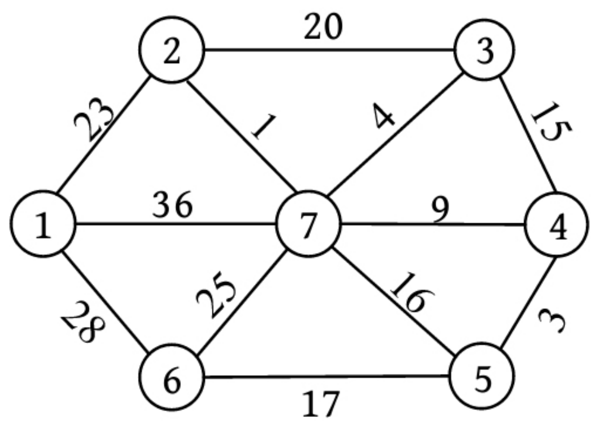
最小生成树

校园网是为学校师生提供资源共享、信息交流和协同工作的计算机网络。如果一所学校包括多个专业学科及部门，则也可以形成多个局域网络，并通过有线或无线方式连接起来。原来的网络系统只局限于以学院、图书馆为单位的局域网，不能完成集中管理及对各种资源的共享，个别院校还远离大学本部，这些情况都严重阻碍了该校的网络化进程。现在需要设计网络电缆布线，将各个单位连通起来，如何设计才能使布线费用最少呢？

可以用无向连通图 G=(V,E)表示通信网络，V 表示节点集，E 表示边集。把各个单位都抽象为图中的节点，把单位之间的通信网络抽象为节点与节点之间的边，边的权值表示布线费用。如果两个节点没有连线，则代表在这两个单位之间不能布线，费用为无穷大，如下图所示。



那么如何设计网络电缆布线，将各个单位连通起来，使布线费用最少呢？对于有 n 个节点的连通图，只需 n-1 条边就可以使这个图连通，在 n-1 条边中要想保证图连通，就必须不包含回路，所以**只需找出 n-1 条权值最小且无回路的边即可**。需要说明以下几个概念。

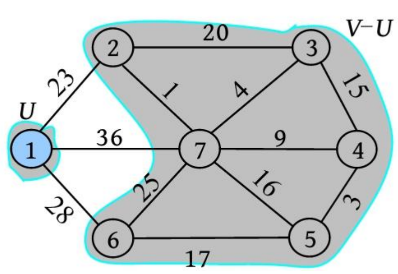
- **子图：**从原图中选中一些由节点和边组成的图，称之为原图的子图。
- **生成子图：**选中一些由边和所有节点组成的图，称之为原图的生成子图。
- **生成树：**如果生成的子图恰好是一棵树，则称之为生成树。
- **最小生成树：**权值之和最小的生成树，称之为最小生成树。

求解最小生成树算法有两种：Prim 算法和 Kruskal 算法。

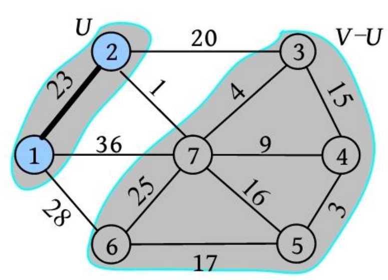
一、 Prim 算法

找出 n-1 条权值最小的边很容易，那么怎么保证无回路呢？如果在一个图中通过深度搜索或广度搜索判断有没有回路，则工作繁重。有一种很好的办法——集合避圈法。在生成树的过程中，我们把已经在生成树中的节点看作一个集合，把剩下的节点看作另一个集合，从连接两个集合的边中选择一条权值最小的边即可。

首先任选一个节点，例如节点 1，把它放在集合 U 中，U={1}，那么剩下的节点即 V-U={2,3,4,5,6,7}，集合 V 是图的所有节点集合，如下图所示。



现在只需看看在连接两个集合（V 和 V-U）的边中，哪一条边的权值最小，把权值最小的边关联的节点加入集合 U 中。从上图可以看出，在连接两个集合的 3 条边中，1-2 的边的权值最小，选中它，把节点 2 加入集合 U 中，U={1,2}，V-U={3,4,5,6,7}，如下图所示。

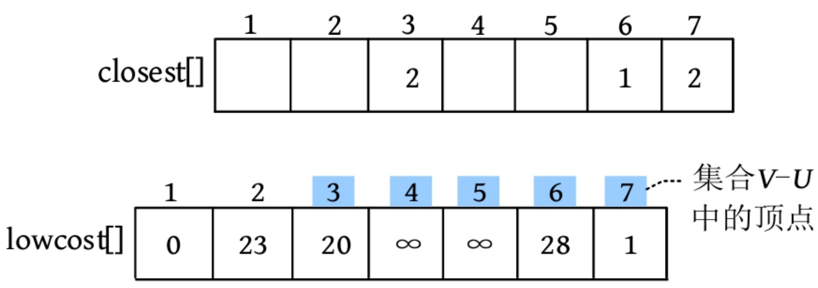


再从连接两个集合（V 和 V−U）的边中选择一条权值最小的边。从上图可以看出，在连接两个集合的 4 条边中，节点 2 到节点 7 的边的权值最小，选中这条边，把节点 7 加入集合 U={1,2,7}中，V−U={3,4,5,6}。

如此下去，直到 U=V 结束，选中的边和所有的节点组成的图就是最小生成树。这就是 **Prim 算法，1957 年由 Robert C.Prim 发现**。那么如何用算法来实现呢？

直观地看图，很容易找出集合 U 到集合 V−U 的边中哪条边的权值是最小的，但是**在程序中如果穷举这些边，再找最小值，则时间复杂度太高，该怎么办呢？**可以通过设置两个数组巧妙地解决这个问题，closest[j]表示集合 V−U 中的节点 j 到集合 U 中的最邻近点，lowcost[j]表示集合 V−U 中的节点 j 到集合 U 中的最邻近点的边值，即边(j,closest[j])的权值。

例如在上图中，节点 7 到集合 U 中的最邻近点是 2，closest[7]=2。节点 7 到最邻近点 2 的边值为 1，即边(2,7)的权值，记为 lowcost[7]=1，如下图所示。



所以只需在集合 V−U 中找到 lowcost[]值最小的节点即可。

1. 算法步骤

（1）初始化。令集合 U={u_0}，u_0∈V，并初始化数组 closest[]、lowcost[]和 s[]。

（2）在集合 V−U 中找 lowcost 值最小的节点 t，即 lowcost[t]=min{lowcost[j]|j∈V−U}，满足该公式的节点 t 就是集合 V−U 中连接集合 U 的最邻近点。

（3）将节点 t 加入集合 U 中。

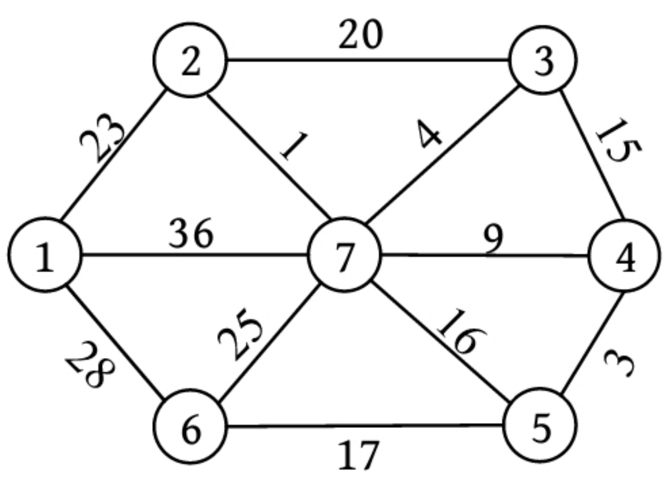
（4）如果集合 V−U 为空，则算法结束，否则转向步骤 5。

（5）对集合 V−U 中的所有节点 j 都更新其 lowcost[]和 closest[]。更新 if(C[t][j]<lowcost [j]){ lowcost[j]=C[t] [j]; closest[j]=t;}，转向步骤 2。

按照上述步骤，最终可以得到一棵权值之和最小的生成树。

2. 图解

图 G（G=(V, E)）是一个无向连通带权图，如下图所示。

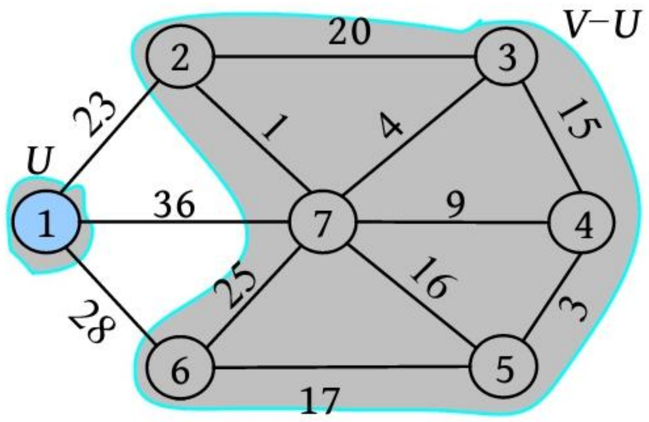


（1）初始化。假设 u_0=1，令集合 U={1}，集合 V−U={2,3,4,5,6,7}，TE={}，s[1]=true，初始化数组 closest[]：除了节点 1，其余节点均为 1，表示集合 V−U 中的节点到集合 U 的最邻近点均为 1。lowcost[]：节点 1 到集合 V−U 中节点的边值。closest[]和 lowcost[]如下图所示。

	1	2	3	4	5	6	7
closest[]		1	1	1	1	1	1

	1	2	3	4	5	6	7
lowcost[]	0	23	∞	∞	∞	28	36

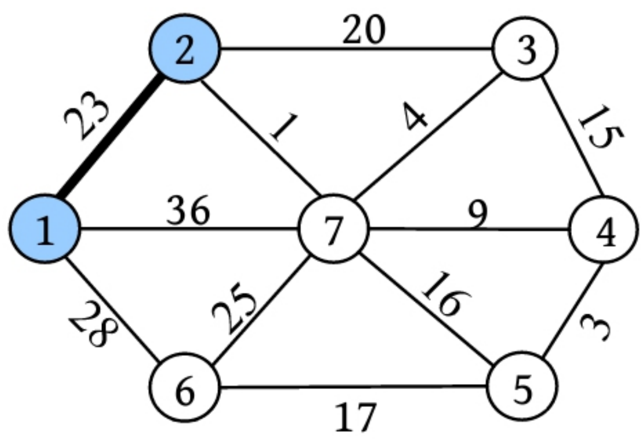
初始化后如下图所示。



（2）找 lowcost 最小的节点。在集合 V−U={2,3,4,5,6,7}中，依照贪心策略寻找集合 V−U 中 lowcost 最小的节点 t。找到的最小值为 23，对应的节点 t=2，如下图所示。

	1	2	3	4	5	6	7
lowcost[]		23	∞	∞	∞	28	36

选中的边和节点如下图所示。



（3）加入集合 U 中。将节点 t 加入集合 U 中，U={1,2}，同时更新 V−U={3,4,5,6,7}。

（4）更新。对 t 在集合 V−U 中的每一个邻接点 j，都可以借助 t 更新。节点 2 的邻接点是节点 3 和节点 7：

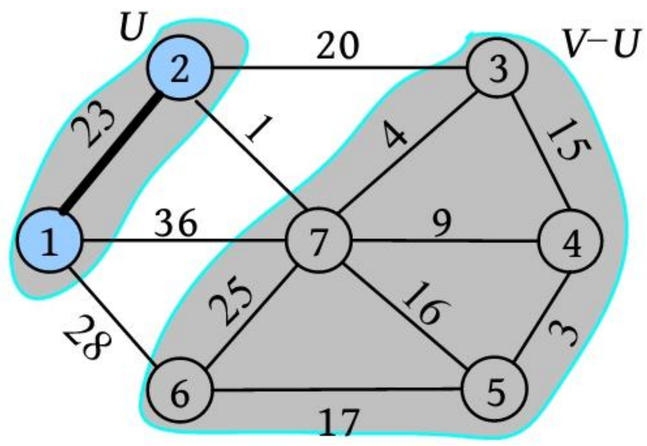
- C[2][3]=20<lowcost[3]=∞，更新最邻近距离 lowcost[3]=20，最邻近点 closest[3]=2；
- C[2][7]=1<lowcost[7]=36，更新最邻近距离 lowcost[7]=1，最邻近点 closest[7]=2。

更新后的 closest[]和 lowcost[]数组如下图所示。

	1	2	3	4	5	6	7
closest[]		1	2	1	1	1	2

	1	2	3	4	5	6	7
lowcost[]	0	23	20	∞	∞	28	1

更新后的集合如下图所示。

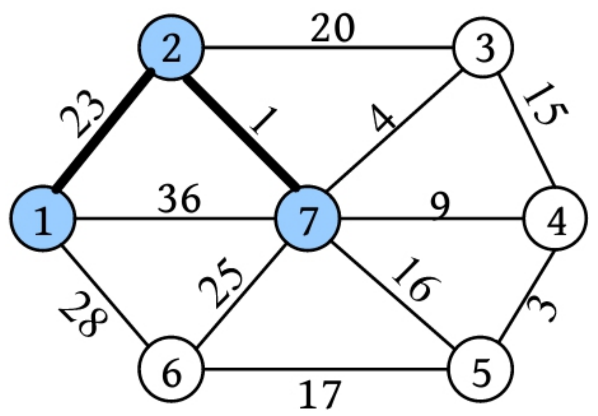


closest[j]和 lowcost[j]分别表示集合 V−U 中节点 j 到集合 U 的最邻近节点和最邻近距离。节点 3 到集合 U 的最邻近点为 2 ,最邻近距离为 20；节点 4、5 到集合 U 的最邻近点仍为初始化状态 1，最邻近距离为∞；节点 6 到集合 U 的最邻近点为 1，最邻近距离为 28；节点 7 到集合 U 的最邻近点为 2，最邻近距离为 1。

（5）找 lowcost 最小的节点。在集合 V−U={3,4,5,6,7}中，依照贪心策略寻找集合 V−U 中 lowcost 最小的节点 t，找到的最小值为 1，对应的节点 t=7，如下图所示。

	1	2	3	4	5	6	7
lowcost[]	0	23	20	∞	∞	28	1

选中的边和节点如下图所示：



（6）加入集合 U 中。将节点 t 加入集合 U 中，U={1,2,7}，同时更新 V−U={3,4,5,6}。

（7）更新。对 t 在集合 V−U 中的每一个邻接点 j，都可以借 t 更新。节点 7 在集合 V−U 中的邻接点是节点 3、4、5、6：

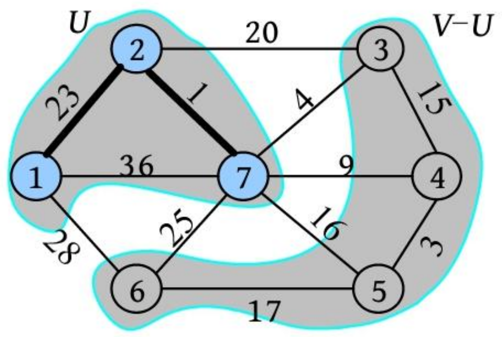
- C[7][3]=4<lowcost[3]=20，更新最邻近距离 lowcost[3]=4，最邻近点 closest[3]=7；
- C[7][4]=9<lowcost[4]=∞，更新最邻近距离 lowcost[4]=9，最邻近点 closest[4]=7；
- C[7][5]=16<lowcost[5]=∞，更新最邻近距离 lowcost[5]=16，最邻近点 closest[5]=7；
- C[7][6]=25<lowcost[6]=28，更新最邻近距离 lowcost[6]=25，最邻近点 closest[6]=7。

更新后的 closest[]和 lowcost[]数组如下图所示：

	1	2	3	4	5	6	7
closest[]		1	7	7	7	7	2

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	16	25	1

更新后的集合如下图所示：

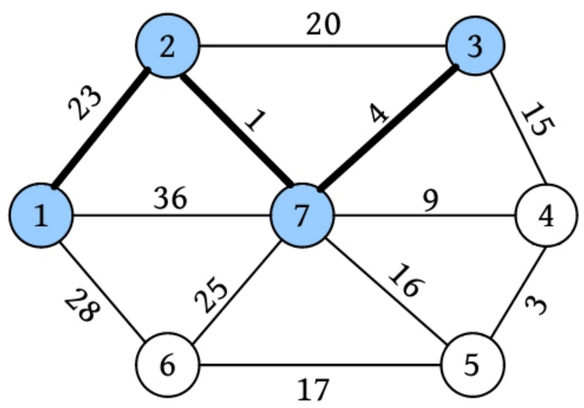


节点 3 到集合 U 的最邻近点为 7，最邻近距离为 4；节点 4 到集合 U 的最邻近点为 7，最邻近距离为 9；节点 5 到集合 U 的最邻近点为 7，最邻近距离为 16；节点 6 到集合 U 的最邻近点为 7，最邻近距离为 25。

（8）找 lowcost 最小的节点。在集合 V−U={3,4,5,6}中，依照贪心策略寻找集合 V−U 中 lowcost 最小的节点 t，找到的最小值为 4，对应的节点 t=3，如下图所示。

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	16	25	1

选中的边和节点如下图所示：



（9）加入集合 U 中。将节点 t 加入集合 U 中，U ={1,2,3,7}，同时更新 V−U={4,5,6}。

（10）更新。对 t 在集合 V−U 中的每一个邻接点 j，都可以借助 t 更新。节点 3 在集合 V−U 中的邻接点是节点 4：C[3][4]=15>lowcost[4]=9，不更新；closest[j]和 lowcost[j]数组不改变。

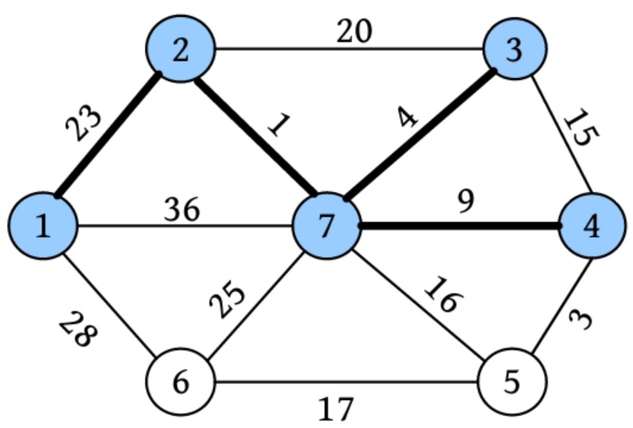
更新后的集合如下图所示。

节点 4 到集合 U 的最邻近点为 7，最邻近距离为 9；节点 5 到集合 U 的最邻近点为 7，最邻近距离为 16；节点 6 到集合 U 的最邻近点为 7，最邻近距离为 25。

（11）找 lowcost 最小的节点。在集合 V−U={4,5,6}中，依照贪心策略寻找集合 V−U 中 lowcost 最小的节点 t，找到的最小值为 9，对应的节点 t=4，如下图所示。

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	16	25	1

选中的边和节点如下图所示。



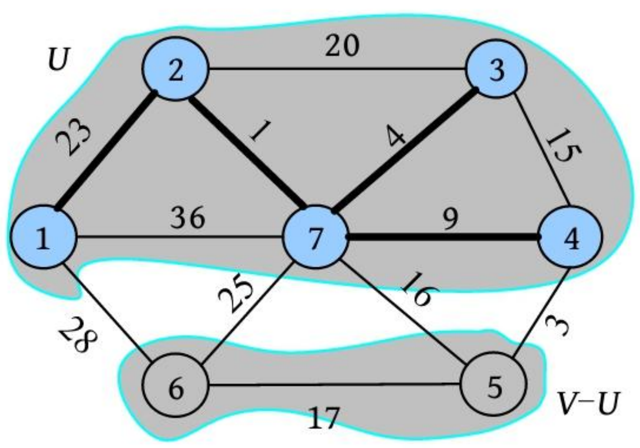
（12）加入集合 U 中。将节点 t 加入集合 U 中，U ={1,2,3,4,7}，同时更新 V−U={5,6}。

（13）更新。对 t 在集合 V−U 中的每一个邻接点 j，都可以借助 t 更新。节点 4 在集合 V−U 中的邻接点是节点 5：C[4][5]=3<lowcost[5]=16，更新最邻近距离 lowcost[5]=3，最邻近点 closest[5]=4；更新后的 closest[]和 lowcost[]数组如下图所示。

	1	2	3	4	5	6	7
closest[]		1	7	7	4	7	2

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	3	25	1

更新后的集合如下图所示。

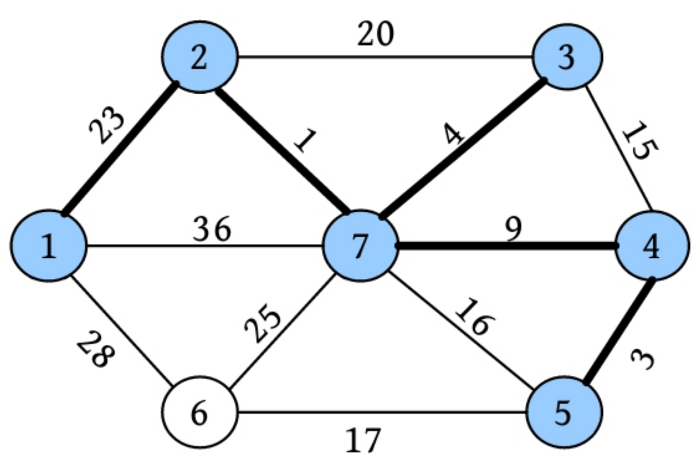


节点 5 到集合 U 的最邻近点为 4，最邻近距离为 3；节点 6 到集合 U 的最邻近点为 7，最邻近距离为 25。

（14）找最小。在集合 V−U={5,6}中，依照贪心策略寻找集合 V−U 中 lowcost 最小的节点 t，找到的最小值为 3，对应的节点 t=5，如下图所示。

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	3	25	1

选中的边和节点如下图所示。



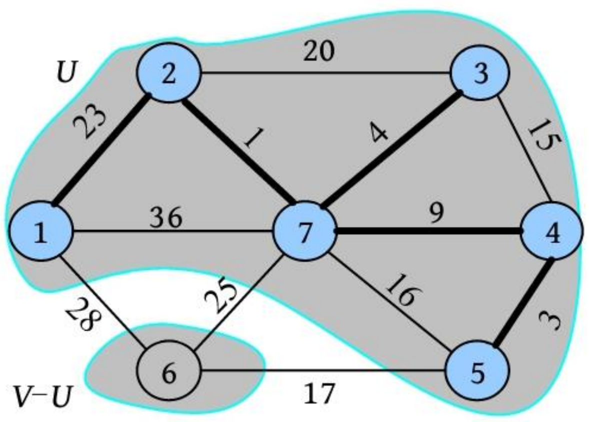
（15）加入集合 U 中。将节点 t 加入集合 U 中，U={1,2,3,4,5,7}，同时更新 V−U={6}。

（16）更新。对节点 t 在集合 V−U 中的每一个邻接点 j，都可以借助 t 更新。节点 5 在集合 V−U 中的邻接点是节点 6：C[5][6]=17<lowcost[6]=25，更新最邻近距离 lowcost[6]=17，最邻近点 closest[6]=5；更新后的 closest[]和 lowcost[]数组如下图所示。

	1	2	3	4	5	6	7
closest[]		1	7	7	4	5	2

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	3	17	1

更新后的集合如下图所示。

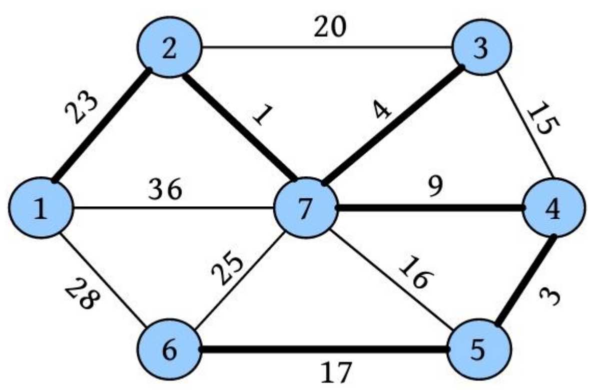


节点 6 到集合 U 的最邻近点为 5，最邻近距离为 17。

（17）找 lowcost 最小的节点。在集合 V−U={6}中，依照贪心策略寻找集合 V−U 中 lowcost 最小的节点 t，找到的最小值为 17，对应的节点 t=6。

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	3	17	1

选中的边和节点如下图所示。



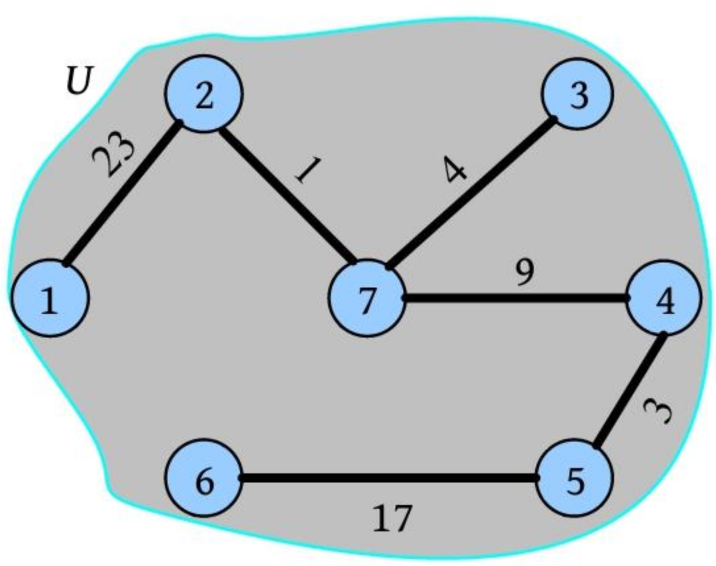
（18）加入集合 U 中。将节点 t 加入集合 U 中，U ={1,2,3,4,5,6,7}，同时更新 V−U={}。

（19）更新。对 t 在集合 V−U 中的每一个邻接点 j，都可以借 t 更新。节点 6 在集合 V−U 中无邻接点，因为 V−U={}。更新后的 closest[]和 lowcost[]数组如下图所示。

	1	2	3	4	5	6	7
closest[]		1	7	7	4	5	2

	1	2	3	4	5	6	7
lowcost[]	0	23	4	9	3	17	1

（20）得到的最小生成树如下图所示。最小生成树的权值之和为 57，即把 lowcost[]数组中的值加起来。



3. 算法实现

```
void Prim(int n){
    s[1]=true; //初始时，在集合 U 中只有一个元素，即节点 1
    for(int i=1;i<=n;i++){//①初始化
        if(i!=1){
            lowcost[i]=c[1][i];
            closest[i]=1;
            s[i]=false;
        }
        else
            lowcost[i]=0;
    }
    for(int i=1;i<n;i++){ //②在集合 V-U 中寻找距离集合 U 最近的节点 t
        int temp=INF;
        int t=1;
        for(int j=1;j<=n;j++){//③在集合 V-U 中寻找距离集合 U 最近的节点 t
            if((!s[j])&&(lowcost[j]<temp)){
                t=j;
                temp=lowcost[j];
            }
        }
        if(t==1)
            break;//找不到 t，跳出循环
        s[t]=true;//否则，将 t 加入集合 U 中
        for(int j=1;j<=n;j++){ //④更新 lowcost 和 closest
            if((!s[j])&&(c[t][j]<lowcost[j])){
                lowcost[j]=c[t][j];
                closest[j]=t;
            }
        }
    }
}
```

4. 算法分析

时间复杂度：在 Prim(int n, int u0, int c[N][N])算法中，共有 4 个 for 语句，①for 语句的执行次数为 n；②在 for 语句里面嵌套了两个 for 语句③、④，它们的执行次数均为 n，对算法的运行时间贡献最大，当外层循环标号为 1 时，③、④for 语句在内层循环的控制下均执行 n 次，外层循环②从 1 ~ n，因此，该语句的执行次数为 n^2，时间复杂度为 $O(n^2)$。

空间复杂度：算法所需要的辅助空间包含 lowcost[]、closest[]和 s[]，空间复杂度为 O(n)。

二、 Kruskal 算法

构造最小生成树还有一种算法，即 Kruskal 算法：设图 G（G=(V,E)）是无向连通带权图，V={1,2,…,n}；设最小生成树 T=(V,TE)，该树的初始状态为只有 n 个节点而无边的非连通图 T=(V,{})，Kruskal 算法将这 n 个节点看成 n 个孤立的连通分支。它首先将所有的边都按权值从小到大排序，然后只要在 T 中选的边数不到 n−1，就做这样的**贪心选择**：在边集 E 中选取权值最小的边(i,j)，如果将边(i,j)加入集合 TE 中不产生回路（圈），则将边(i,j)加入边集 TE 中，即用边(i,j)将这两个连通分支合并连接成一个连通分支；否则继续选择下一条最短边。把边(i,j)从集合 E 中删去，继续上面的贪心选择，直到 T 中的所有节点都在同一个连通分支上为止。此时，选取的 n−1 条边恰好构成图 G 的一棵最小生成树 T。

那么，怎样判断加入某条边后图 T 会不会出现回路呢？该算法对于手工计算十分方便，因为肉眼可以很容易看出挑选哪些边能够避免回路（避圈法），但计算机程序需要一种机制进行判断。Kruskal 算法用了一种非常聪明的方法，就是**运用集合避圈**：如果所选择加入的边的起点和终点都在 T 的集合中，就可以断定会形成回路（圈）。这其实就是前面提到的“避圈法”：**边的两个节点不能属于同一个集合。**

1. 算法步骤

（1）初始化。将所有边都按权值从小到大排序，将每个节点的集合号都初始化为自身编号。

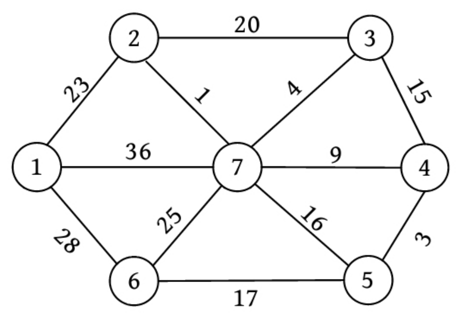
（2）按排序后的顺序选择权值最小的边(u,v)。

（3）如果节点 u 和 v 属于两个不同的连通分支，则将边(u,v)加入边集 TE 中，并将两个连通分支合并。

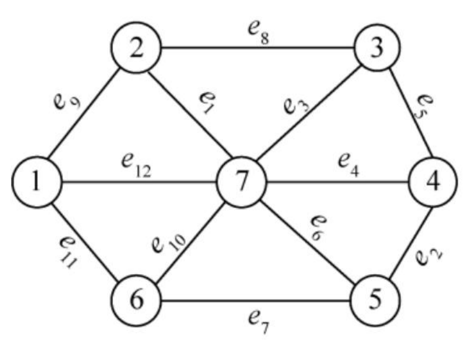
（4）如果选取的边数小于 n−1，则转向步骤 2，否则算法结束。

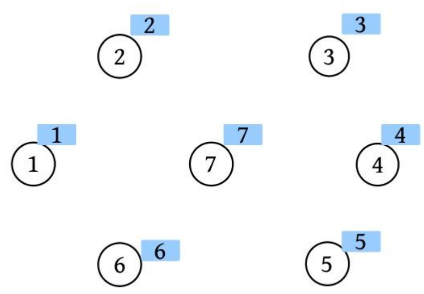
2. 图解

设图 G（G =(V, E)）是无向连通带权图，如下图所示。



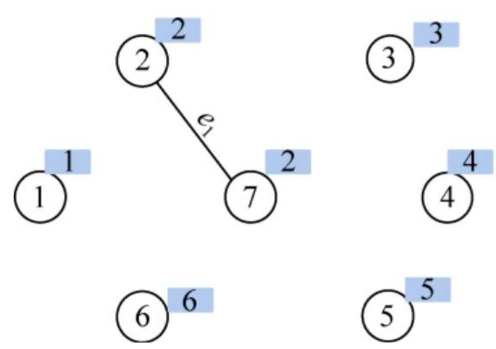
（1）初始化。将所有边都按权值从小到大排序，如下图所示。将每个节点都初始化为一个孤立的分支，即一个节点对应一个集合，集合号为该节点的序号，如下图所示。





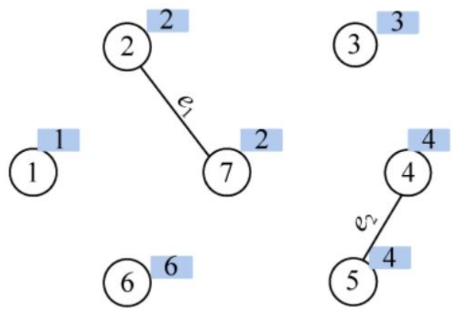
（2）找最小。在 E 中寻找权值最小的边 e_1(2,7)，边值为 1。

（3）合并。节点 2 和节点 7 的集合号不同，即属于两个不同的连通分支，将边(2,7)加入边集 TE 中，执行合并操作，将两个连通分支的所有节点都合并为一个集合；假设把小的集合号赋值给大的集合号，以下均做如此处理，那么将节点 7 的集合号也改为 2，如下图所示。



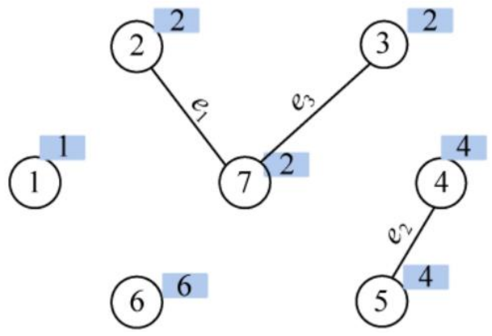
（4）找最小。在 E 中寻找权值最小的边 e_2(4,5)，边值为 3。

（5）合并。节点 4 和节点 5 的集合号不同，即属于两个不同的连通分支，将边(4,5)加入边集 TE 中，执行合并操作，将两个连通分支的所有节点都合并为一个集合，将节点 5 的集合号也改为 4，如下图所示。



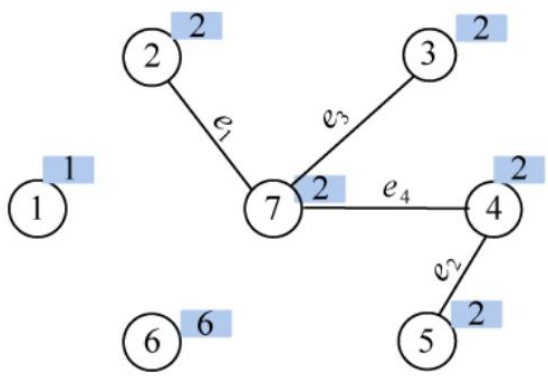
（6）找最小。在 E 中寻找权值最小的边 e_3(3,7)，边值为 4。

（7）合并。节点 3 和节点 7 的集合号不同，即属于两个不同的连通分支，将边(3,7)加入边集 TE 中，执行合并操作，将两个连通分支的所有节点都合并为一个集合，将节点 3 的集合号也改为 2，如下图所示。



（8）找最小。在 E 中寻找权值最小的边 e_4(4,7)，边值为 9。

（9）合并。节点 4 和节点 7 的集合号不同，即属于两个不同的连通分支，将边(4,7)加入边集 TE 中，执行合并操作，将两个连通分支的所有节点都合并为一个集合，将节点 4、5 的集合号都改为 2，如下图所示。



（10）找最小。在 E 中寻找权值最小的边 e_5(3,4)，边值为 15。

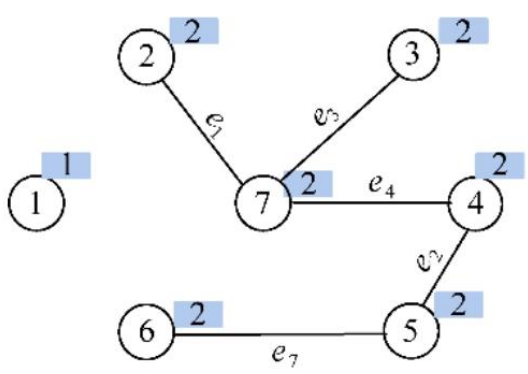
（11）合并。节点 3 和节点 4 的集合号相同，属于同一连通分支，不能选择，否则会形成回路。

（12）找最小。在 E 中寻找权值最小的边 e_6(5,7)，边值为 16。

（13）合并。节点 5 和节点 7 的集合号相同，属于同一连通分支，不能选择，否则会形成回路。

（14）找最小。在 E 中寻找权值最小的边 e_7(5,6)，边值为 17。

（15）合并。节点 5 和节点 6 集合号不同，即属于两个不同的连通分支，将边(5,6)加入边集 TE 中，执行合并操作，将两个连通分支的所有节点都合并为一个集合，将节点 6 的集合号改为 2，如下图所示。

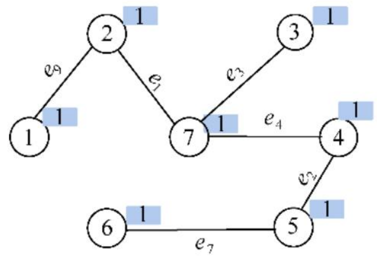


（16）找最小。在 E 中寻找权值最小的边 e_8(2,3)，边值为 20。

（17）合并。节点 2 和节点 3 的集合号相同，属于同一连通分支，不能选择，否则会形成回路。

（18）找最小。在 E 中寻找权值最小的边 $e_9(1,2)$，边值为 23。

（19）合并。节点 1 和节点 2 的集合号不同，即属于两个不同的连通分支，将边(1,2)加入边集 TE 中，执行合并操作，将两个连通分支的所有节点都合并为一个集合，将节点 2、3、4、5、6、7 的集合号都改为 1，如下图所示。



（20）选中的各边和所有的节点就是最小生成树，各边权值之和就是最小生成树的代价。

3. 算法代码

```
struct Edge{//边集数组
   int u,v,w;
}e[N*N];

bool cmp(Edge x, Edge y) {//排序优先级，边权从小到大
   return x.w<y.w;
}

void Init(int n){//初始化集合号为自身
   for(int i=1;i<=n;i++)
      fa[i]=i;
}

int Merge(int a,int b){//合并
   int p=fa[a];
   int q=fa[b];
   if(p==q) return 0;
   for(int i=1;i<=n;i++){//检查所有节点，把集合号是 q 的都改为 p
      if(fa[i]==q)
         fa[i]=p;//将 a 的集合号赋值给 b
   }
   return 1;
}
int Kruskal(int n){//求最小生成树
   int ans=0;
   sort(e,e+m,cmp);
   for(int i=0;i<m;i++)
      if(Merge(e[i].u,e[i].v)){
         ans+=e[i].w;
         n--;
         if(n==1)//共执行 n-1 次合并，在 n=1 时算法结束
            return ans;
      }
   return 0;
}
```

4. 算法分析

时间复杂度：在该算法中需要对边进行排序，若使用快速排序，则算法的时间复杂度为 O(mlogm)。而合并集合需要 n−1 次合并，每次合并的时间复杂度都为 O(n)，合并集合的时间复杂度为 $O(n^2)$。总的时间复杂度为 O(mlogm)。

如果使用并查集优化合并操作，则每次合并的时间复杂度都为 O(logn)。

空间复杂度：辅助空间包括一些变量和集合号数组 fa[]，空间复杂度为 O(n)。