Data Structures

Ch6

图
Graphs

2023 年 11 月 7 日

学而不厭 誨人不倦

6.4 最小生成树

6-4-1　Prim算法

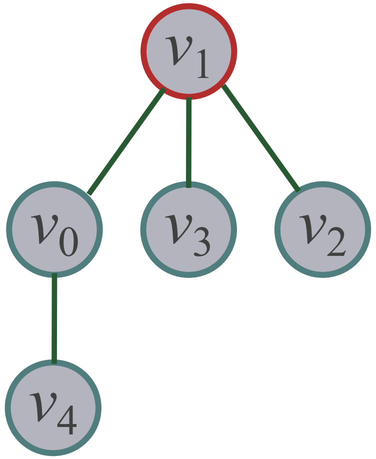
1. 最小生成树定义

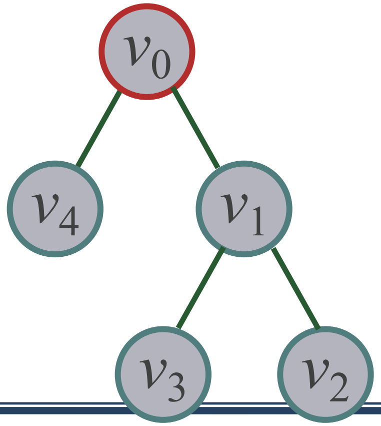
生成树：连通图的生成树是包含全部顶点的一个极小连通子图

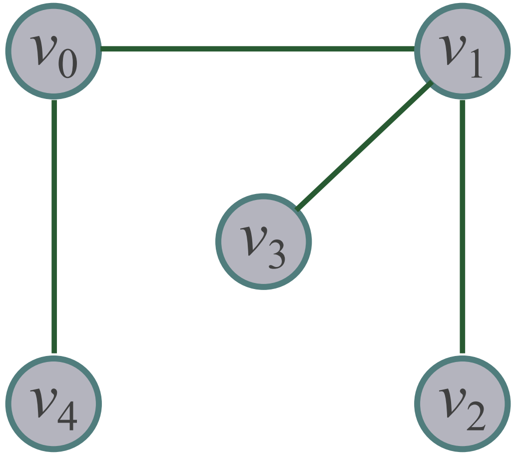
多——构成回路

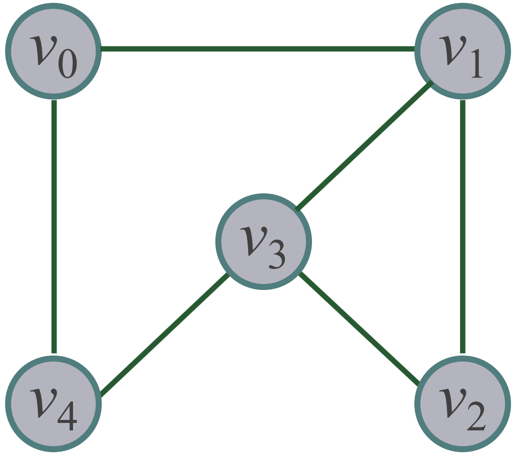
少——不连通

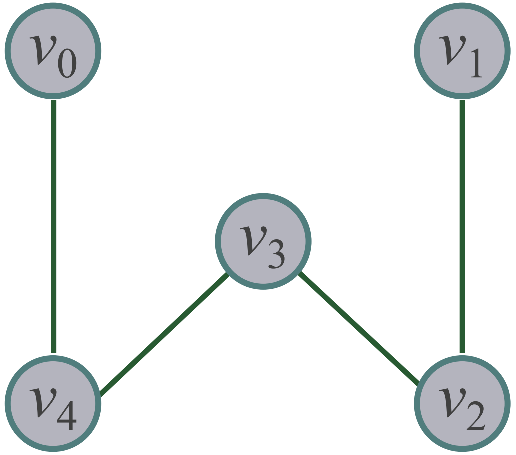
含有n−1条边





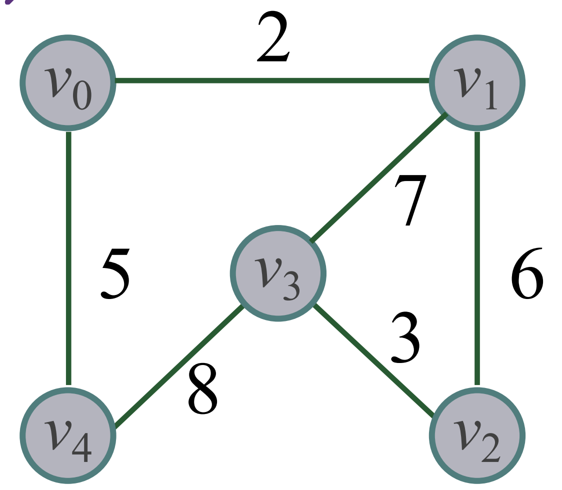


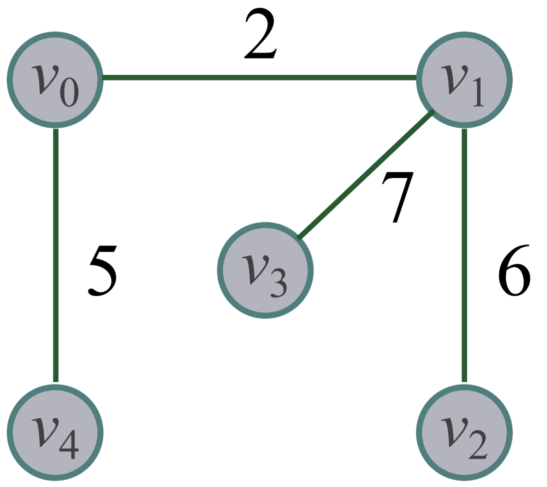




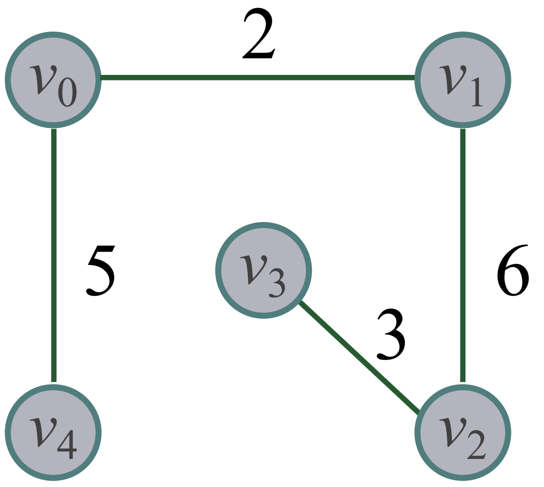
1. 最小生成树定义

- 生成树的代价：在无向连通网中，生成树上各边的权值之和
- 最小生成树：在无向连通网中，代价最小的生成树





生成树的代价： 20



生成树的代价： 16

在n个城市之间建造通信网络，至少要架设n-1条通信线路，而每两个城市之间架设通信线路的造价是不一样的，那么如何设计才能使得总造价最小？

2. Prim算法

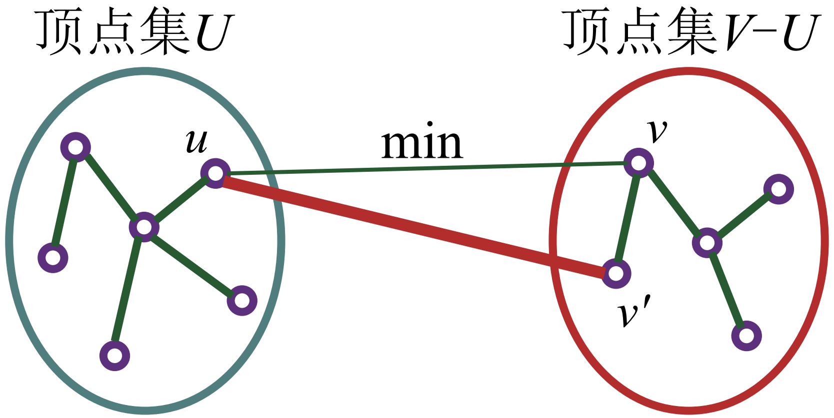
关键：是如何找到连接 U 和 $V\text{-}U$ 的最短边

算法：Prim

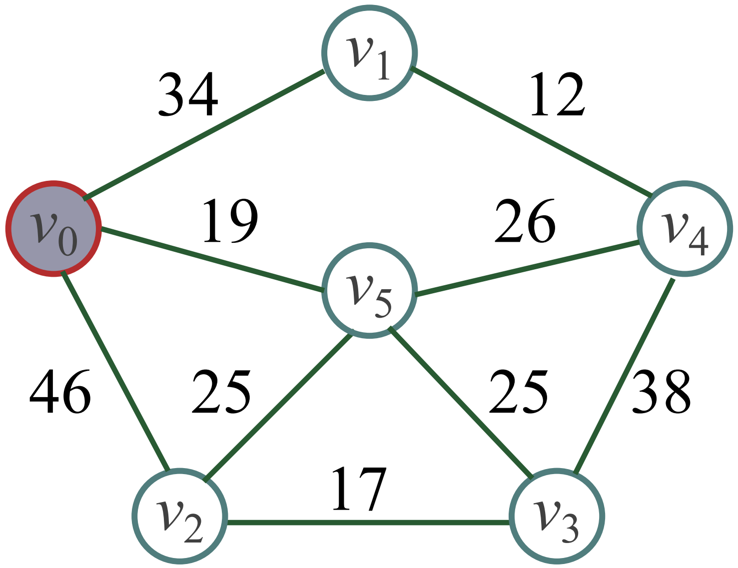
输入：无向连通网G=(V，E)

输出：最小生成树T=(U，TE)

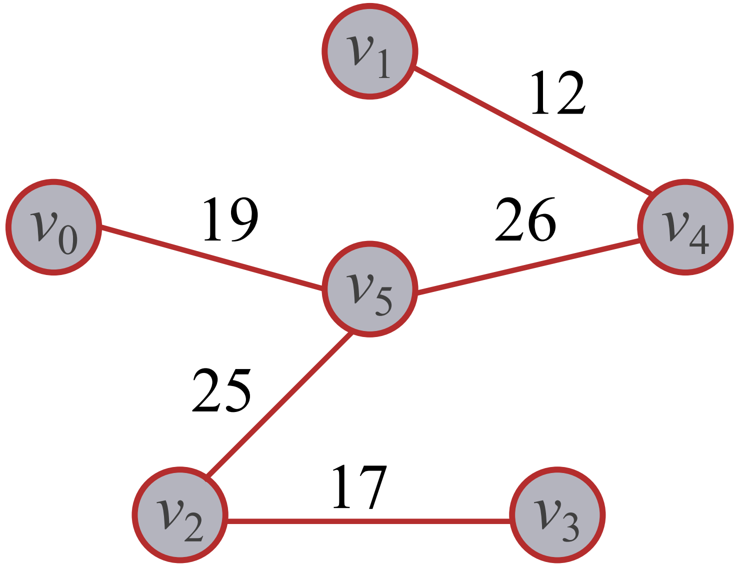
1. 初始化：U = {v}； TE={ }；
2. 重复下述操作直到U = V:
   2.1 在E中寻找最短边(i，j)，且满足i∈U，j∈V-U;
   2.2 U = U + {j}；
   2.3 TE = TE + {(i，j)}；



2. Prim算法



$$\begin{bmatrix} 0 & 34 & 46 & 100 & 100 & 19 \\ 34 & 0 & 100 & 100 & 12 & 100 \\ 46 & 100 & 0 & 17 & 100 & 25 \\ 100 & 100 & 17 & 0 & 38 & 25 \\ 100 & 12 & 100 & 38 & 0 & 26 \\ 19 & 100 & 25 & 25 & 26 & 0 \end{bmatrix}$$



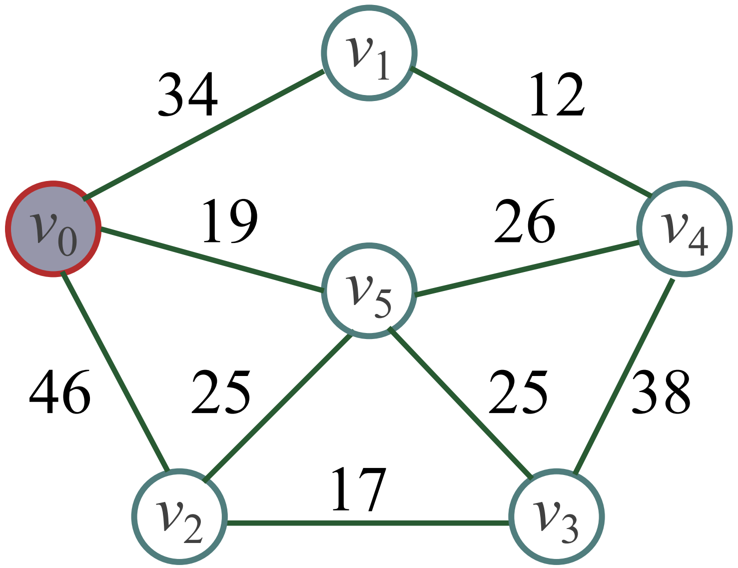
关键：是如何找到连接 U 和 $V\text{-}U$ 的最短边?

U：涂色

$V-U$：尚未涂色

方法：一个顶点涂色、另一个顶点尚未涂色的最短边

2. Prim算法-运行实例

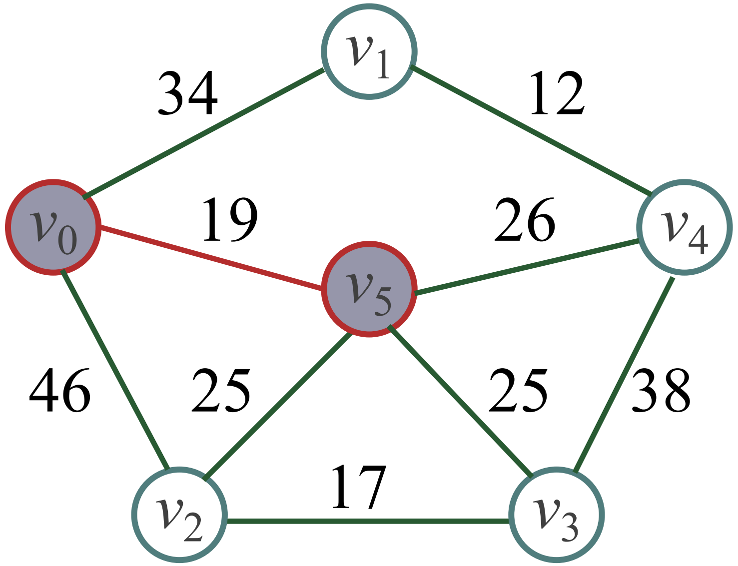


初始化:

$U=\{v_0\}$

$V-U=\{v_1, v_2, v_3, v_4, v_5\}$

cost$=\{(v_0, v_1)34, (v_0, v_2)46, (v_0, v_3)\infty, (v_0, v_4)\infty, \underline{(v_0, v_5)19\}}$



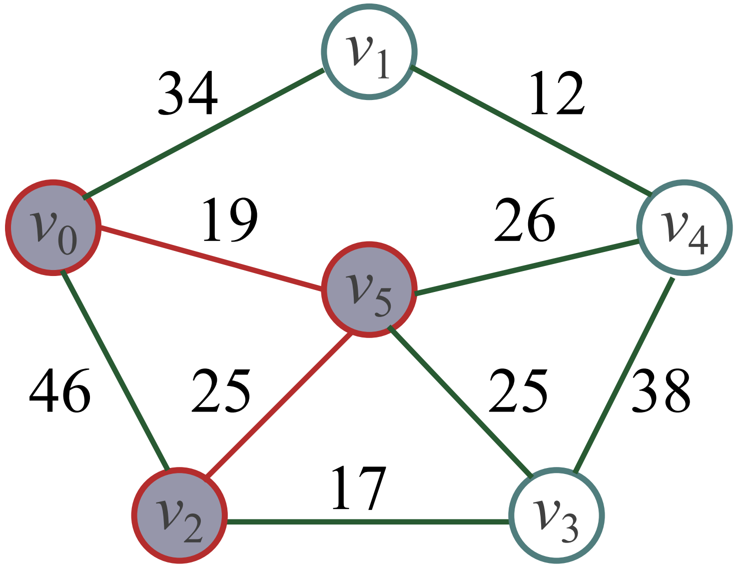
第一次迭代:

$U=\{v_0, v_5\}$

$V-U=\{v_1, v_2, v_3, v_4\}$

cost$=\{(v_0, v_1)34, \underline{(v_5, v_2)25}, (v_5, v_3)25, (v_5, v_4)26\}$

2. Prim算法-运行实例



第二次迭代：

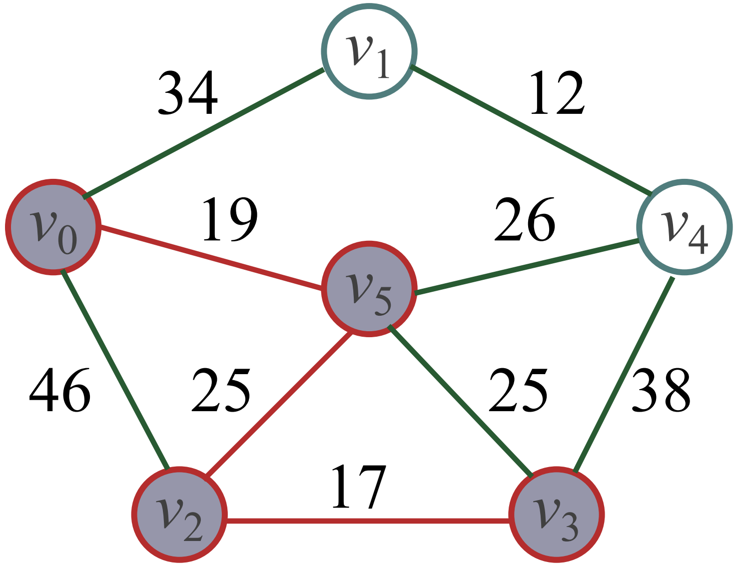
$U=\{v_0, v_5, v_2\}$

$V-U=\{v_1, v_3, v_4\}$

cost $=\{(v_0, v_1)34, (v_2, v_3)17, (v_5, v_4)26\}$

第一次迭代：

cost$=\{(v_0, v_1)34, (v_5, v_3)25, (v_5, v_4)26\}$



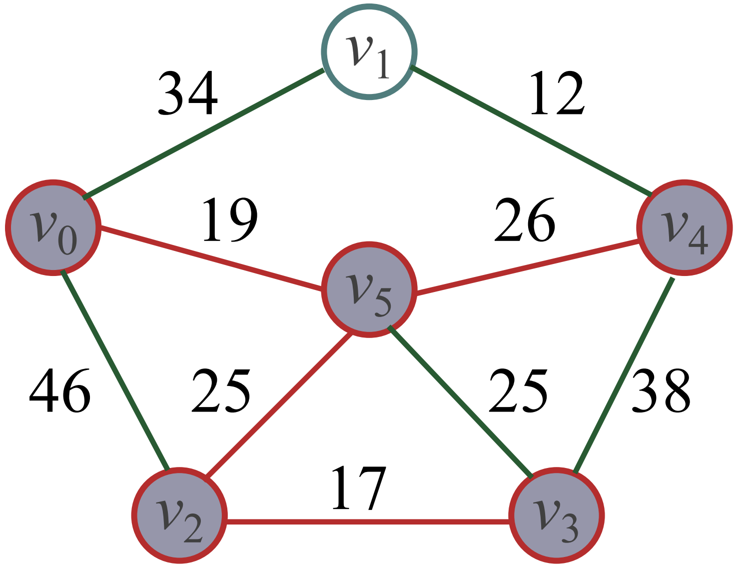
第三次迭代：

$U=\{v_0, v_5, v_2, v_3\}$

$V-U=\{v_1, v_4\}$

cost$=\{(v_0, v_1)34, (v_5, v_4)26\}$

2. Prim算法-运行实例



第四次迭代：

$U=\{v_0, v_5, v_2, v_3, v_4\}$

$V-U=\{v_1\}$

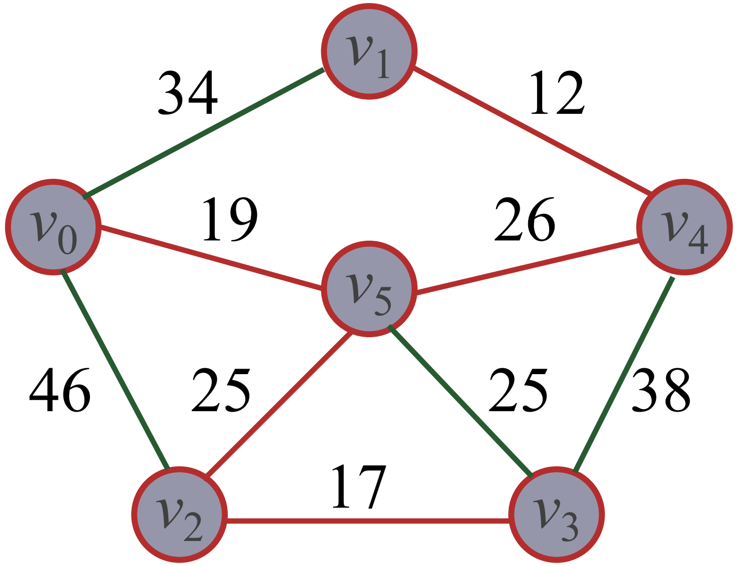
$\text{cost}=\{(v_4, v_1)12\}$

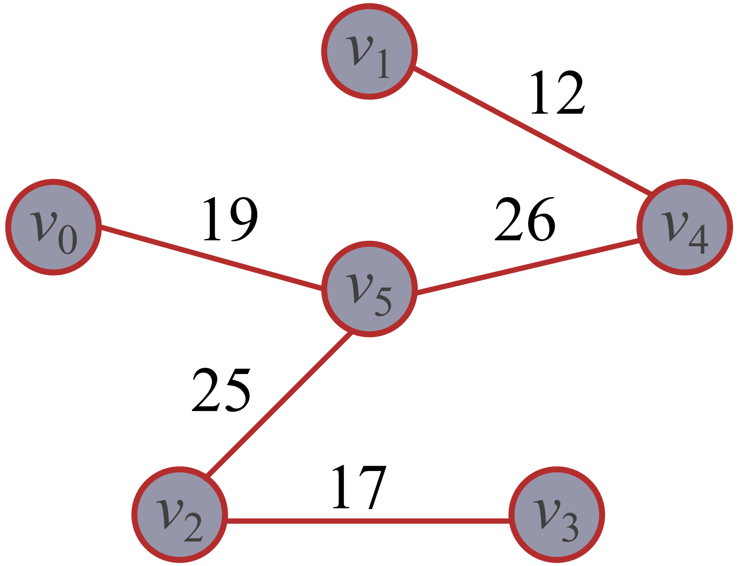
第三次迭代：

$U=\{v_0, v_5, v_2, v_3\}$

$V-U=\{v_1, v_4\}$

$\text{cost}=\{(v_0, v_1)34\}$





3. Prim算法-存储结构

- 图采用什么存储结构呢?

需要不断读取任意两个顶点之间边的权值

⇩

图采用邻接矩阵存储

- 如何存储候选最短边集（连接U和$V-U$的候选最短边）？

例如：$\{(v_0, v_1)34, (v_0, v_2)46, (v_0, v_3)\infty, (v_0, v_4)\infty, (v_0, v_5)19\}$

数组adjvex[n]：表示候选最短边的邻接点
数组lowcost[n]：表示候选最短边的权值

$$\begin{cases} adjvex[i] = j \\ lowcost[i] = w \end{cases}$$

含义是：候选最短边（i, j）的权值为w，其中i∈V-U，j∈U

3. Prim算法-存储结构

初始时，lowcost[v] = 0，表示将顶点v加入集合U中；

adjvex[i] = v，lowcost[i] = edge[v][i]（$0 \le i \le n$-1）

例如： $\{(v_0, v_0)0, (v_0, v_1)34, (v_0, v_2)46, (v_0, v_3)\infty, (v_0, v_4)\infty, (v_0, v_5)19\}$

	0	1	2	3	4	5
adjvex[n] =	0	0	0	0	0	0
lowcost[n] =	0	34	46	∞	∞	19

3. Prim算法-存储结构

每一次迭代，设数组lowcost[n]中的最小权值是lowcost [j]，则

令lowcost[j] = 0，表示将顶点 *j* 加入集合 *U* 中；

由于顶点 *j* 从集合 *V*-*U* 进入集合 *U*，候选最短边集发生变化，需要更新：

$$\begin{cases} \text{lowcost[i]} = \min\{\text{lowcost[i], edge[i][j]}\} \\ \text{adjvex[i]} = \text{j}\ （如果\text{edge[i][j]} < \text{lowcost[i]}） \end{cases} \quad (0 \le i \le n\text{-}1)$$

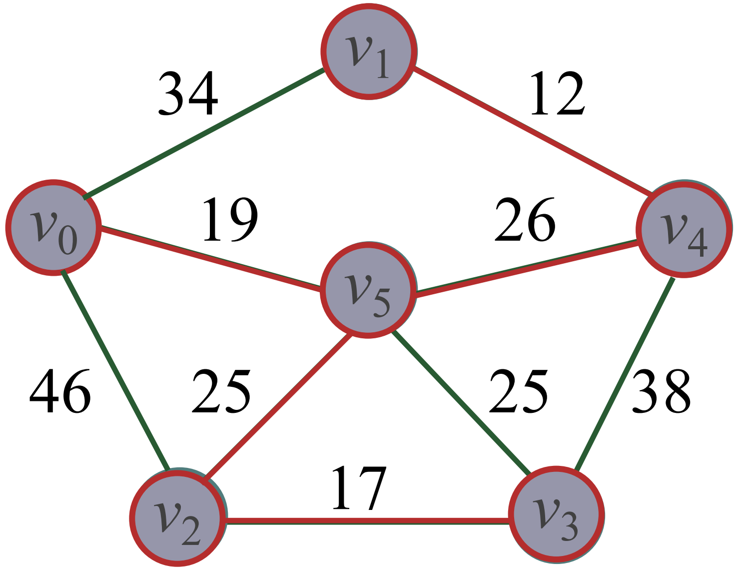
例如： $\{(v_0, v_0)0, (v_0, v_1)34, (v_0, v_2)46, (v_0, v_3)\infty, (v_0, v_4)\infty, (v_0, v_5)19\}$

例如： $\{(v_0, v_0)0, (v_0, v_1)34, (v_5, v_2)25, (v_5, v_3)25, (v_5, v_4)26, (v_0, v_5)0\}$

	0	1	2	3	4	5
adjvex[n] =	0	0	0	0	0	0
lowcost[n] =	0	34	46	∞	∞	19

	0	1	2	3	4	5
adjvex[n] =	0	0	5	5	5	0
lowcost[n] =	0	34	25	25	26	0

最小生成树-Prim算法



	v_0	v_1	v_2	v_3	v_4	v_5
v_0	0	34	46	100	100	19
v_1	34	0	100	100	12	100
v_2	46	100	0	17	100	25
v_3	100	100	17	0	38	25
v_4	100	12	100	38	0	26
v_5	19	100	25	25	26	0

	0	1	2	3	4	5	
adjvex[n] =	0	0	0	0	0	**0**	
lowcost[n] =	0	34	46	∞	∞	**19**	$(v_0,v_5)19$
adjvex[n] =	0	0	**5**	5	5	0	
lowcost[n] =	0	34	**25**	25	26	0	$(v_5,v_2)25$
adjvex[n] =	0	0	5	**2**	5	0	
lowcost[n] =	0	34	0	**17**	26	0	$(v_2,v_3)17$
adjvex[n] =	0	0	5	2	**5**	0	
lowcost[n] =	0	34	0	0	**26**	0	$(v_5,v_4)26$
adjvex[n] =	0	**4**	5	2	5	0	
lowcost[n] =	0	0	0	0	0	0	$(v_4,v_1)12$

6.4 最小生成树

4. Prim算法-实现

```
void Prim(int v)
{
    int i, j, k, adjvex[MaxSize], lowcost[MaxSize];
    for (i = 0; i < vertexNum; i++)
    {
        lowcost[i] = edge[v][i];  adjvex[i] = v;
    }
    lowcost[v] = 0;
    for (k = 1; k < vertexNum; k++)
    {
        j = MinEdge(lowcost, vertexNum)
        cout << j << adjvex[j] << lowcost[j];    lowcost[j] = 0;
        for (i = 0; i < vertexNum; i++)
            if (edge[i][j] < lowcost[i]) {
                lowcost[i] = edge[i][j];  adjvex[i] = j;
            }
    }
}
```

$O(n)$

$O(n)$

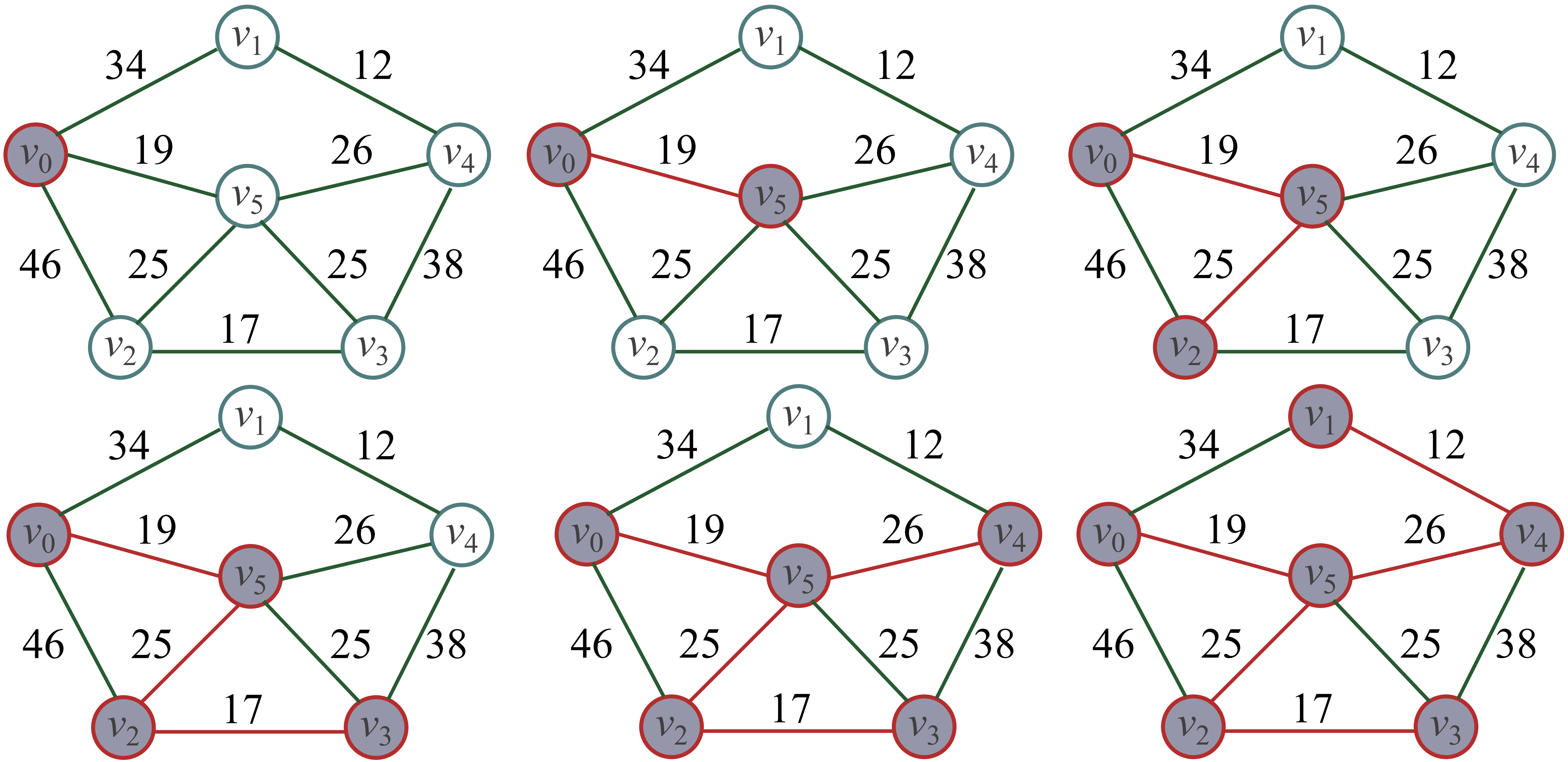
$O(n)$

时间复杂度? ⇨ $O(n^2)$

最小生成树-Prim算法

```
int adjvex[MaxSize], lowcost[MaxSize];
```

借助邻接矩阵，候选最短边的邻接点adjvex，候选最短边的权值lowcost，依次加入新顶点，直到全部结点都加入。



6.4 最小生成树

6-4-2　Kruskal算法

1. Kruskal算法思路

Prim算法的关键是什么?

找到连接 U 和 $V-U$ 的最短边 $\begin{cases} \text{最短边} \\ \text{顶点分别位于 } U \text{ 和 } V-U \text{ 中} \end{cases}$

- Prim算法：先构造满足条件的候选最短边集，再查找最短边
- Kruskal算法：先查找最短边，再判断是否满足条件

1. Kruskal算法思路

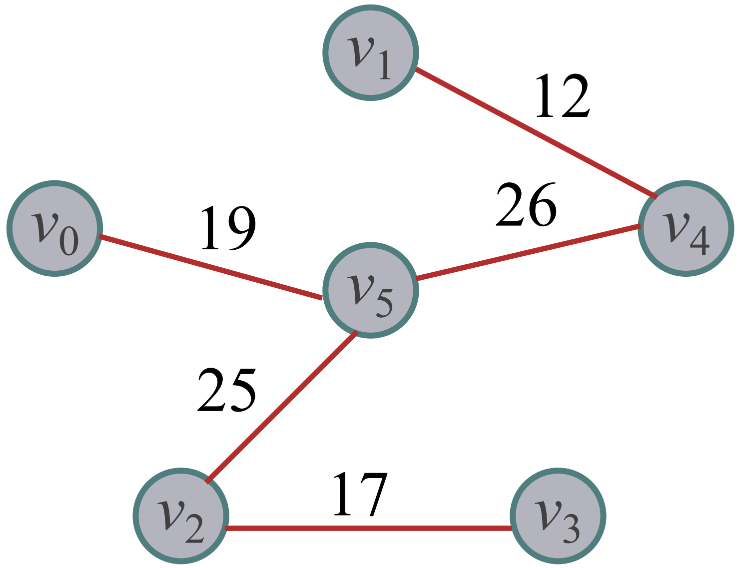
算法：Kruskal算法

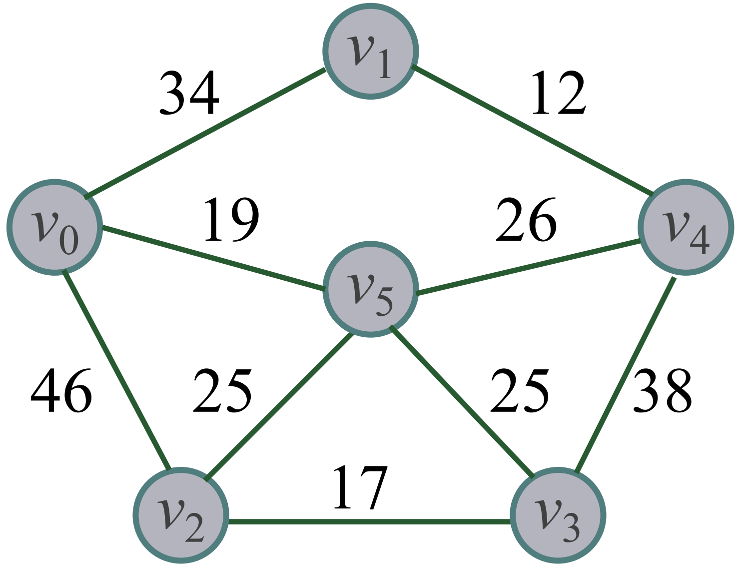
输入：无向连通网G=(V，E)

输出：最小生成树T=(U，TE)

1. 初始化：U=V；TE={ }；
2. 重复下述操作直到所有顶点位于一个连通分量：
   2.1 在E中选取最短边(u，v);
   2.2 如果顶点 u、v 位于两个连通分量，则
      2.2.1 将边 (u，v) 并入TE；
      2.2.2 将这两个连通分量合成一个连通分量；
   2.3 在 E 中标记边 (u，v)，使得 (u，v) 不参加后续最短边的选取；

1. Kruskal算法思路





初始化：连通分量 = $\{v_0\}, \{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}$

第一次迭代：连通分量 = $\{v_0\}, \{v_1, v_4\}, \{v_2\}, \{v_3\}, \{v_5\}$

第二次迭代：连通分量 = $\{v_0\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_5\}$

第三次迭代：连通分量 = $\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$

第四次迭代：连通分量 = $\{v_0, v_2, v_3, v_5\}, \{v_1, v_4\}$

第五次迭代：连通分量 = $\{v_0, v_2, v_3, v_5, v_1, v_4\}$

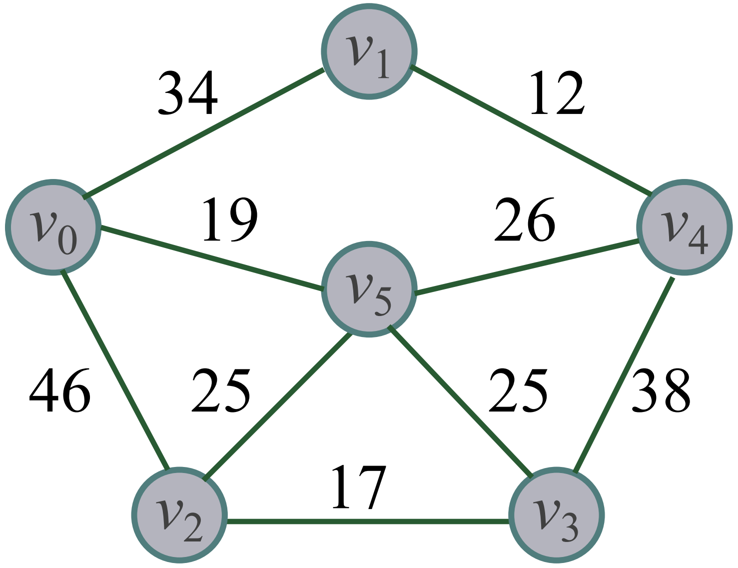
2. Kruskal算法存储结构

图采用什么存储结构呢？ ⇨ **边集数组表示法**

算法：Kruskal算法
输入：无向连通网G=(V，E)
输出：最小生成树T=(U，TE)
1. 初始化：U=V；TE={ }；
2. 重复下述操作直到所有顶点位于一个连通分量：
  2.1 在 E 中选取最短边(u，v)；
  2.2 如果顶点 u、v 位于两个连通分量，则
    2.2.1 将边 (u，v) 并入TE；
    2.2.2 将这两个连通分量合成一个连通分量；
  2.3 在 E 中标记边 (u，v)，使得 (u，v) 不参加后续最短边的选取；

2. Kruskal算法存储结构

图采用用什么存储结构呢? ⇨ **边集数组表示法**



下标:	0	1	2	3	4	5
	v_0	v_1	v_2	v_3	v_4	v_5

from	1	2	0	2	3	4	0	3	0
to	4	3	5	5	5	5	1	4	2
weight	12	17	19	25	25	26	34	38	46

2. Kruskal算法存储结构

图采用什么存储结构呢? ⇨ **边集数组表示法**

```
const int MaxVertex = 10;
const int MaxEdge = 100;
template <typename DataType>
class EdgeGraph
{
public:
    EdgeGraph(DataType a[ ], int n, int e);
    ~EdgeGraph( );
    void Kruskal( );
private:
    int FindRoot(int parent[ ], int v)
    DataType vertex[MaxVertex];
    EdgeType edge[MaxEdge];
    int vertexNum, edgeNum;
} ;
```

下标：	0	1	2	3	4	5
	v_0	v_1	v_2	v_3	v_4	v_5

```
struct EdgeType
{
    int from, to, weight;
};
```

from	1	2	0	2	3	4	3	0	0
to	4	3	5	5	5	5	4	1	2
weight	12	17	19	25	25	26	38	34	46

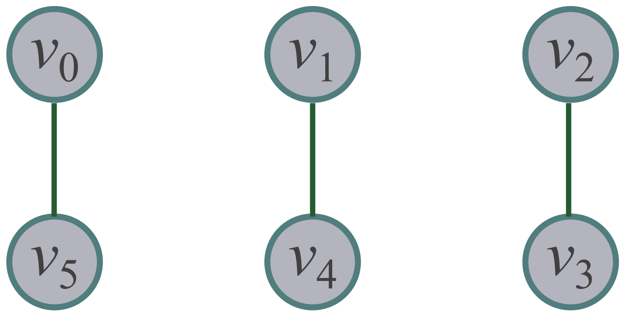
2. Kruskal算法存储结构

如何存储连通分量呢？ ⇨ **并查集**

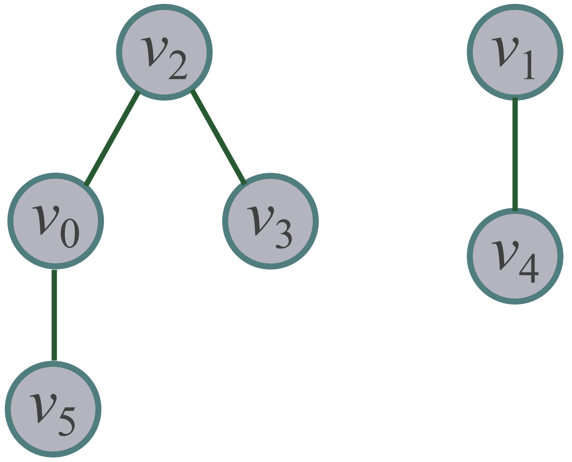
并查集：集合中的元素组织成树的形式：

（1）查找两个元素是否属于同一集合：所在树的根结点是否相同

（2）合并两个集合——将一个集合的根结点作为另一个集合根结点的孩子



$\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$



$\{v_0, v_5, v_2, v_3\}, \{v_1, v_4\}$

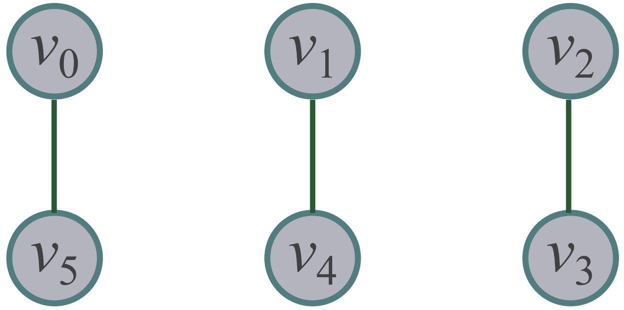
2. Kruskal算法存储结构

如何存储并查集呢? ⇨ **双亲表示法** ⇨ parent[n]

并查集：集合中的元素组织成树的形式：

（1）查找两个元素是否属于同一集合：所在树的根结点是否相同

（2）合并两个集合——将一个集合的根结点作为另一个集合根结点的孩子



$\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$

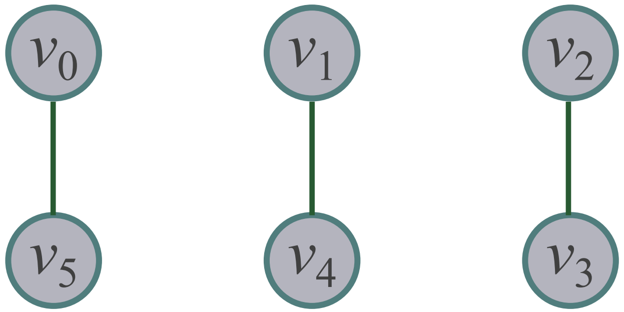
下标：	0	1	2	3	4	5
vertex	v_0	v_1	v_2	v_3	v_4	v_5
parent	−1	−1	−1	2	1	0

3. Kruskal算法实现

如何判断两个顶点是否位于同一个连通分量呢？ 例如，边 (v_2, v_5)？

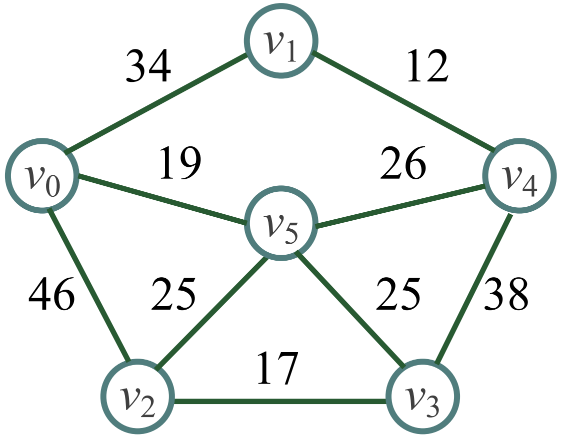
```
vex1 = FindRoot(parent, i);
vex2 = FindRoot(parent, j);
    if (vex1 != vex2) {

    }
```

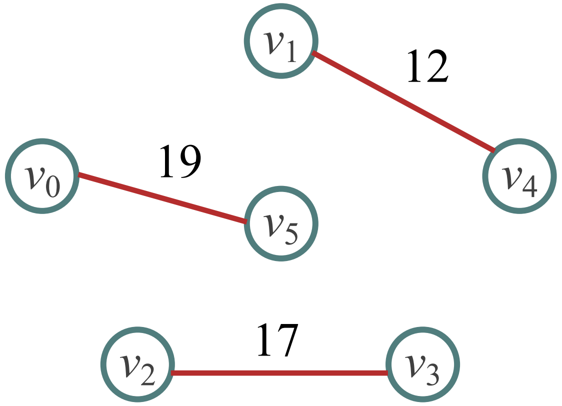
```
int FindRoot(int parent[ ], int v)
{
    int t = v;
    while (parent[t] > -1)
        t = parent[t];
    return t;
}
```



$\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$

下标：	0	1	2	3	4	5
parent	−1	−1	−1	2	1	0



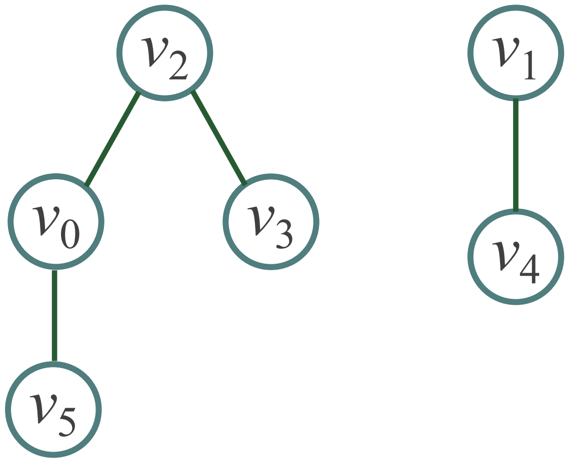


3. Kruskal算法实现

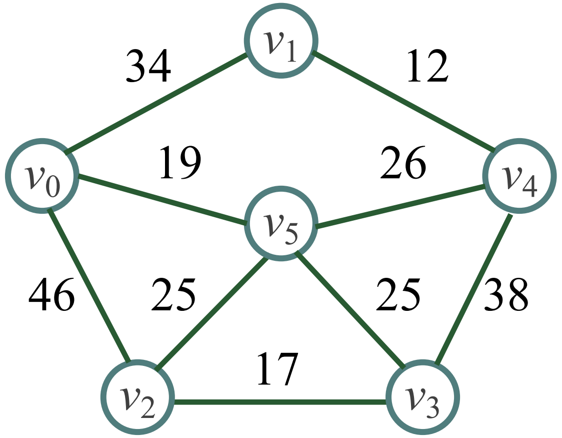
如何合并两个连通分量呢?

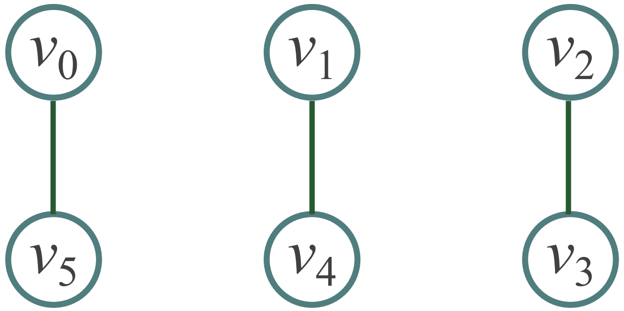
```
vex1 = FindRoot(parent, i);
vex2 = FindRoot(parent, j);
   if (vex1 != vex2) {
      parent[vex2] = vex1;
   }
```

例如，边 (v_2, v_5)?



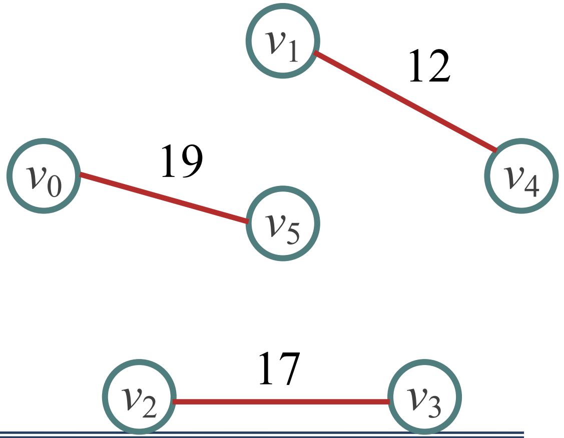
$\{v_0, v_5, v_2, v_3\}, \{v_1, v_4\}$



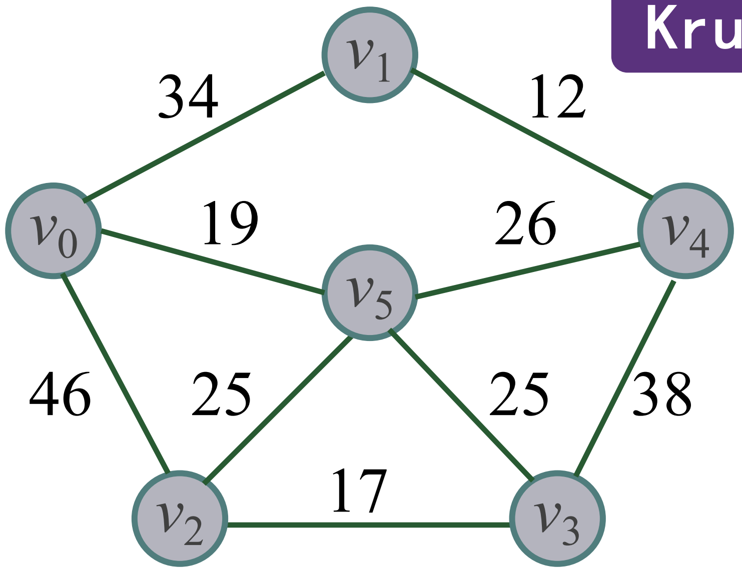


$\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$

下标：	0	1	2	3	4	5
parent	$\cancel{-1}$ 2	−1	−1	2	1	0



Kruskal算法实现



i:	0	1	2	3	4	5	6	7	8
from	1	2	0	2	3	4	3	0	0
to	4	3	5	5	5	5	4	1	2
weight	12	17	19	25	25	26	38	34	46

下标：	0	1	2	3	4	5
vertex	v_0	v_1	v_2	v_3	v_4	v_5
parent	−1	−1	−1	-1	-1	-1

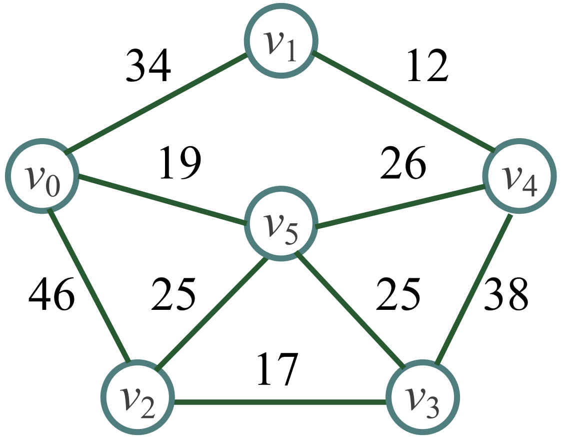
```
for (num = 0, i = 0; num < vertexNum-1; i++)
  {
        vex1 = FindRoot(parent, edge[i].from);
        vex2 = FindRoot(parent, edge[i].to);
        if (vex1 != vex2)
       {
              cout << edge[i].from << edge[i].to << edge[i].weight;
              parent[vex2] = vex1;
              num++;
        }
  }
```

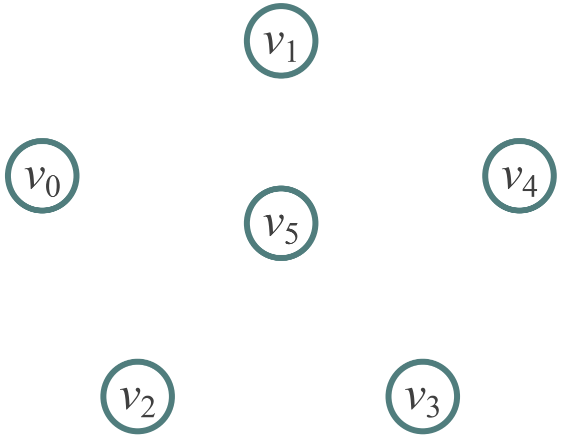
```
int FindRoot(int parent[ ], int v)
{
   int t = v;
   while (parent[t] > -1)
      t = parent[t];
   return t;
}
```

3. Kruskal算法实现

下标 V / parent	0 v_0	1 v_1	2 v_2	3 v_3	4 v_4	5 v_5	最短边	说明
parent	-1	-1	-1	-1	-1	-1		初始化 $\{v_0\}\{v_1\}\{v_2\}\{v_3\}\{v_4\}\{v_5\}$

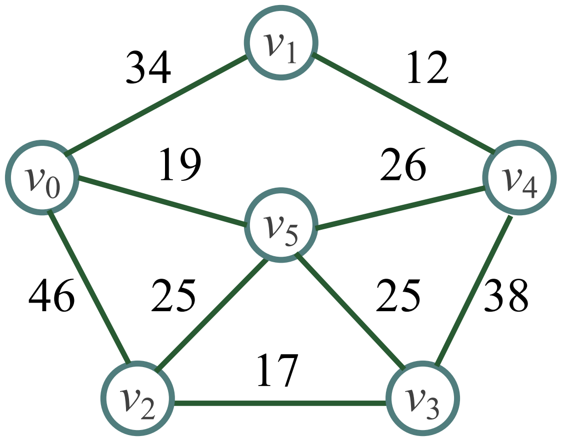
```
void Kruskal( )
{
    int i, num = 0, vex1, vex2;
    for (i = 0; i < vertexNum; i++)
        parent[i] = -1;
}
```



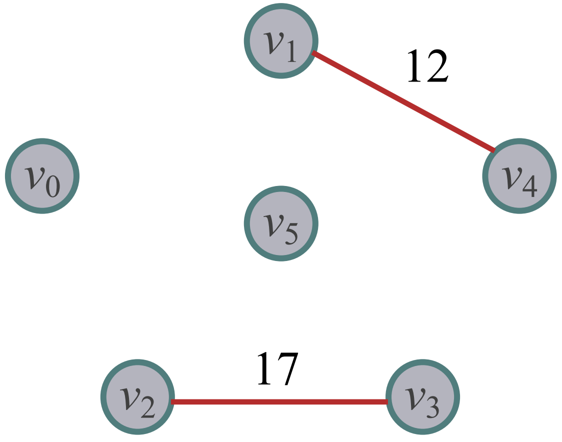


3. Kruskal算法实现

下标 / parent V	0 v_0	1 v_1	2 v_2	3 v_3	4 v_4	5 v_5	最短边	说明
parent	-1	-1	-1	-1	$\cancel{-1}$	-1	$(v_4,v_1)12$	初始化 $\{v_0\}\{v_1\}\{v_2\}\{v_3\}\{v_4\}\{v_5\}$
parent	-1	-1	-1	$\cancel{-1}$	1	-1	$(v_2,v_3)17$	vex1=1, vex2=4, parent[4]=1 $\{v_0\}\{v_1,v_4\}\{v_2\}\{v_3\}\{v_5\}$
parent	-1	-1	-1	2	1	-1		vex1=2, vex2=3, parent[3]=2 $\{v_0\}\{v_1,v_4\}\{v_2,v_3\}\{v_5\}$

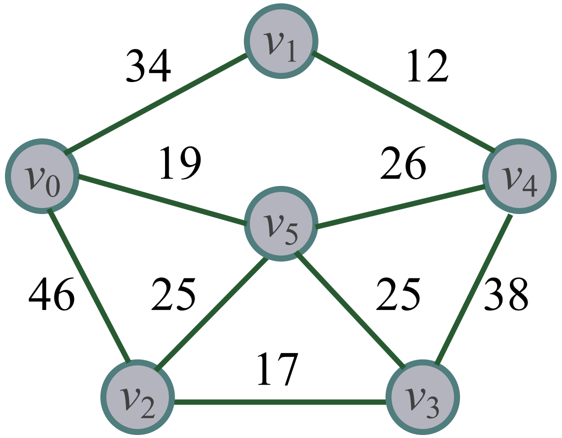


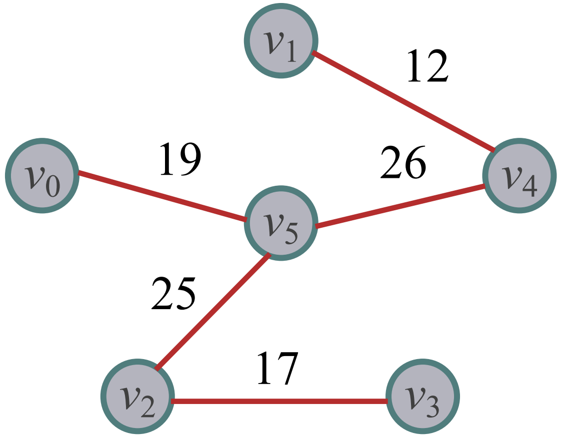

```
for (num = 0, i = 0; num < vertexNum-1; i++)
{
    vex1 = FindRoot(parent, edge[i].from);
    vex2 = FindRoot(parent, edge[i].to);
    if (vex1 != vex2) {
        cout << edge[i].from << edge[i].to << edge[i].weight;
        parent[vex2] = vex1;   num++;
    }
}
```



3. Kruskal算法实现

下标 / parent V	0 v_0	1 v_1	2 v_2	3 v_3	4 v_4	5 v_5	最短边	说明
parent	-1	-1	-1	-1	$\cancel{-1}$	-1	$(v_4,v_1)12$	初始化 $\{v_0\}\{v_1\}\{v_2\}\{v_3\}\{v_4\}\{v_5\}$
parent	-1	-1	-1	$\cancel{-1}$	1	-1	$(v_2,v_3)17$	vex1=1, vex2=4, parent[4]=1 $\{v_0\}\{v_1,v_4\}\{v_2\}\{v_3\}\{v_5\}$
parent	-1	-1	-1	2	1	$\cancel{-1}$	$(v_0,v_5)19$	vex1=2, vex2=3, parent[3]=2 $\{v_0\}\{v_1,v_4\}\{v_2,v_3\}\{v_5\}$
parent	$\cancel{-1}$	-1	-1	2	1	0	$(v_2,v_5)25$	vex1=0, vex2=5, parent[5]=0 $\{v_0,v_5\}\{v_1,v_4\}\{v_2,v_3\}$
parent	2	-1	$\cancel{-1}$	2	1	0	$(v_3,v_5)25$	vex1=2, vex2=0, parent[0]=2 $\{v_0,v_5\}\{v_1,v_4\}\{v_2,v_3\}$
parent							$(v_4,v_5)26$	vex1=2, vex2=2 在一个连通分量中
parent	2	-1	1	2	1	0		vex1=1, vex2=2, parent[2]=1 $\{v_0,v_5,v_1,v_4,v_2,v_3\}$





3. Kruskal算法实现

```
void Kruskal( )
{
    int i, num = 0, vex1, vex2;
    for (i = 0; i < vertexNum; i++)
        parent[i] = -1;
    for (num = 0, i = 0; num < vertexNum-1; i++)
    {
        vex1 = FindRoot(parent, edge[i].from);
        vex2 = FindRoot(parent, edge[i].to);
        if (vex1 != vex2) {
            cout << edge[i].from << edge[i].to << edge[i].weight;
            parent[vex2] = vex1;
            num++;
        }
    }
}
```

$O(n)$

$O(\log_2 n)$

$O(e)$

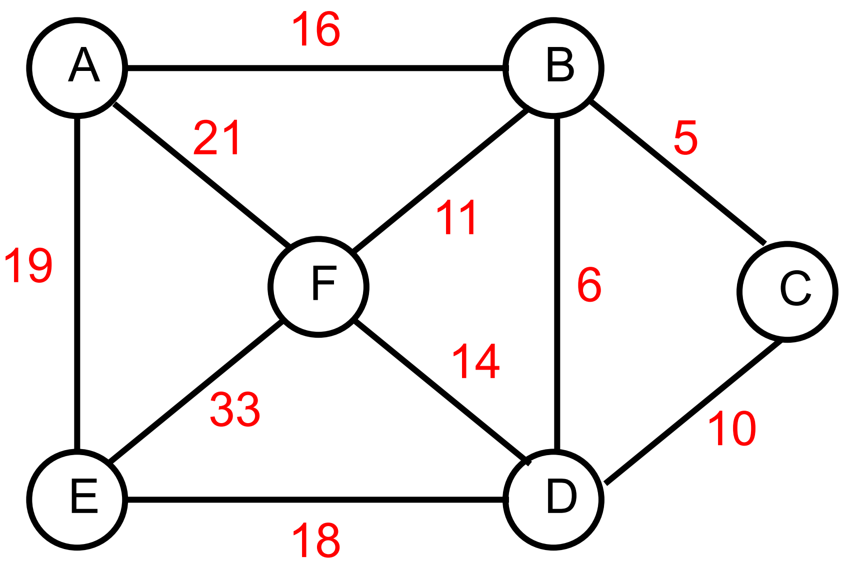
时间复杂度? $\Rightarrow$ $O(e\log_2 e)$

小结

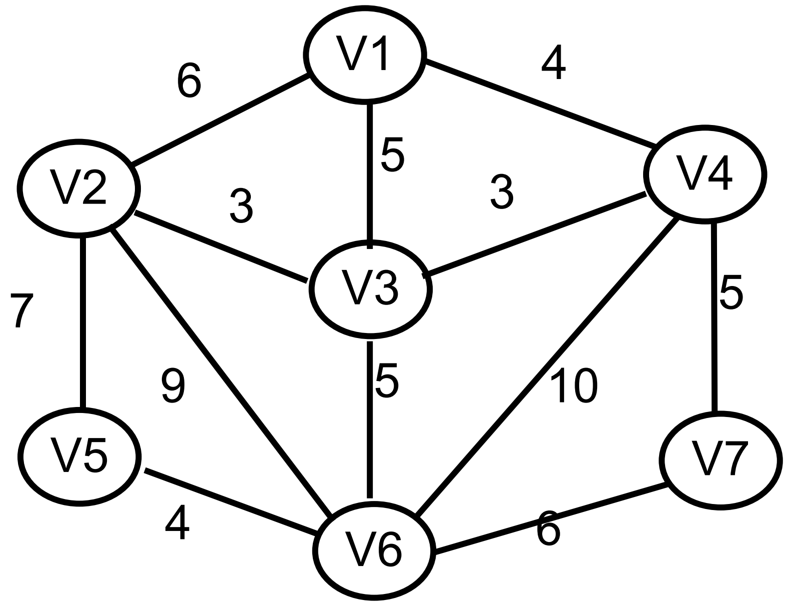
1. 掌握Prim算法及实现方法
2. 理解Kruskal算法及实现方法
3. 理解Prim算法与Kruskal算法的区别

作业

带权无向图如下图所示，请分别画出对应的最小生成树。



G1



G2

实验安排

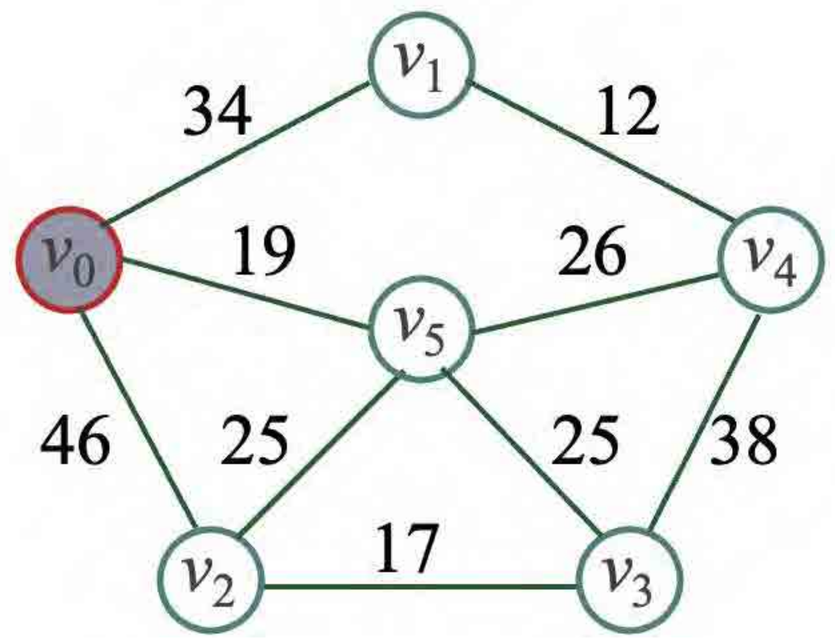
实验七、最小生成树和最短路径的实现与应用

一、实验目的

1. 掌握图的邻接矩阵存储及实现方法
2. 掌握Prim和Kruskal最小生成树算法原理
3. 掌握Dijkstra和Floyd最短路径算法原理
3. 用C++语言实现相关算法，并上机调试。

二、实验内容

1. 实现Prim算法，完成最小生成树的生成和输出。
2. 实现Dijkstra算法，完成最短路径的生成和输出。
3. 实现Kruskal算法和Floyd算法（扩展）
4. 给出测试过程和测试结果。



测试用例

实验时间：　第14周周四晚
22网安：18:30-20:10　22物联网：20:10-21:50
实验地点：软件基础实验室301（老干部处）

Thank You !

Q & A