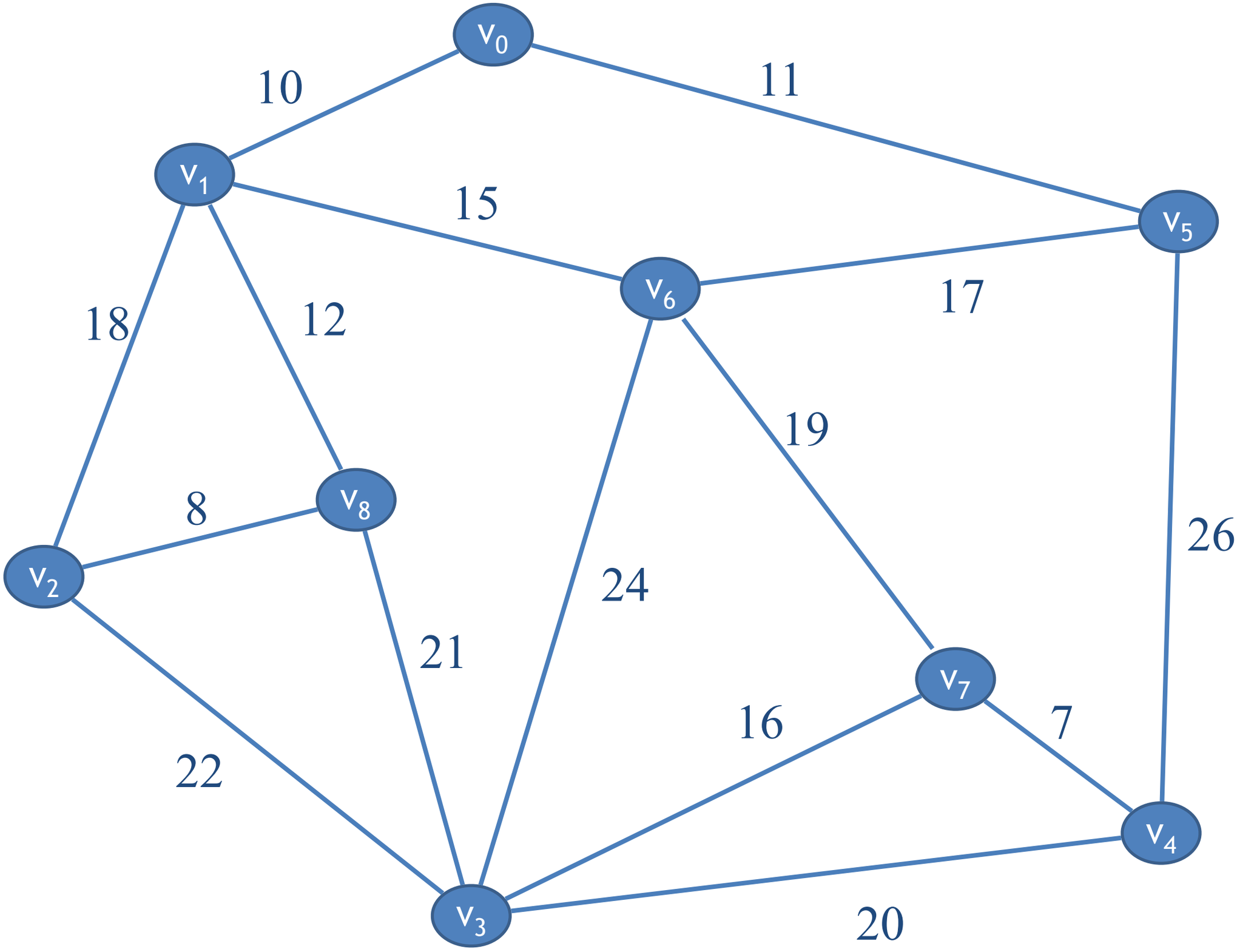
数据结构

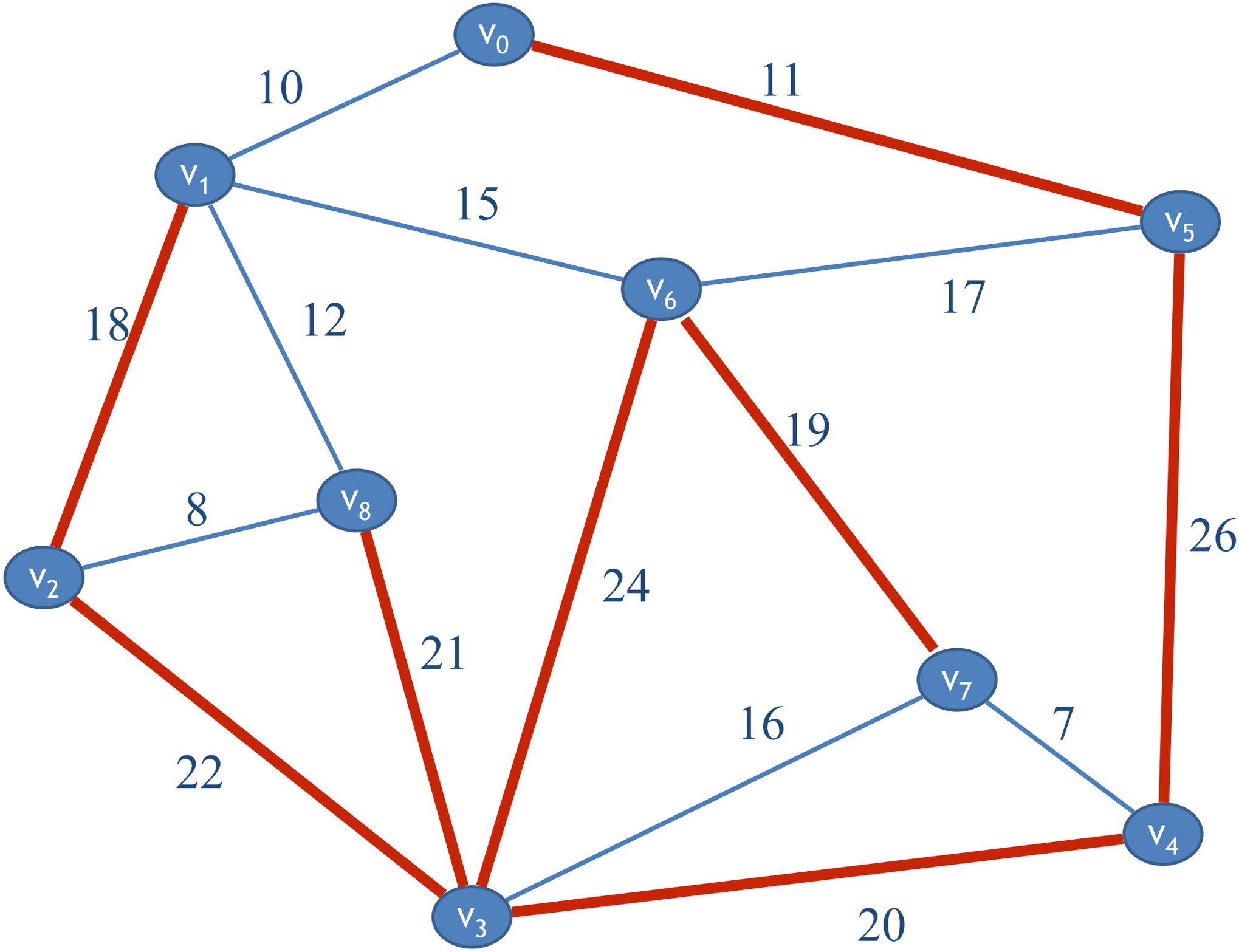
第十讲　最小生成树

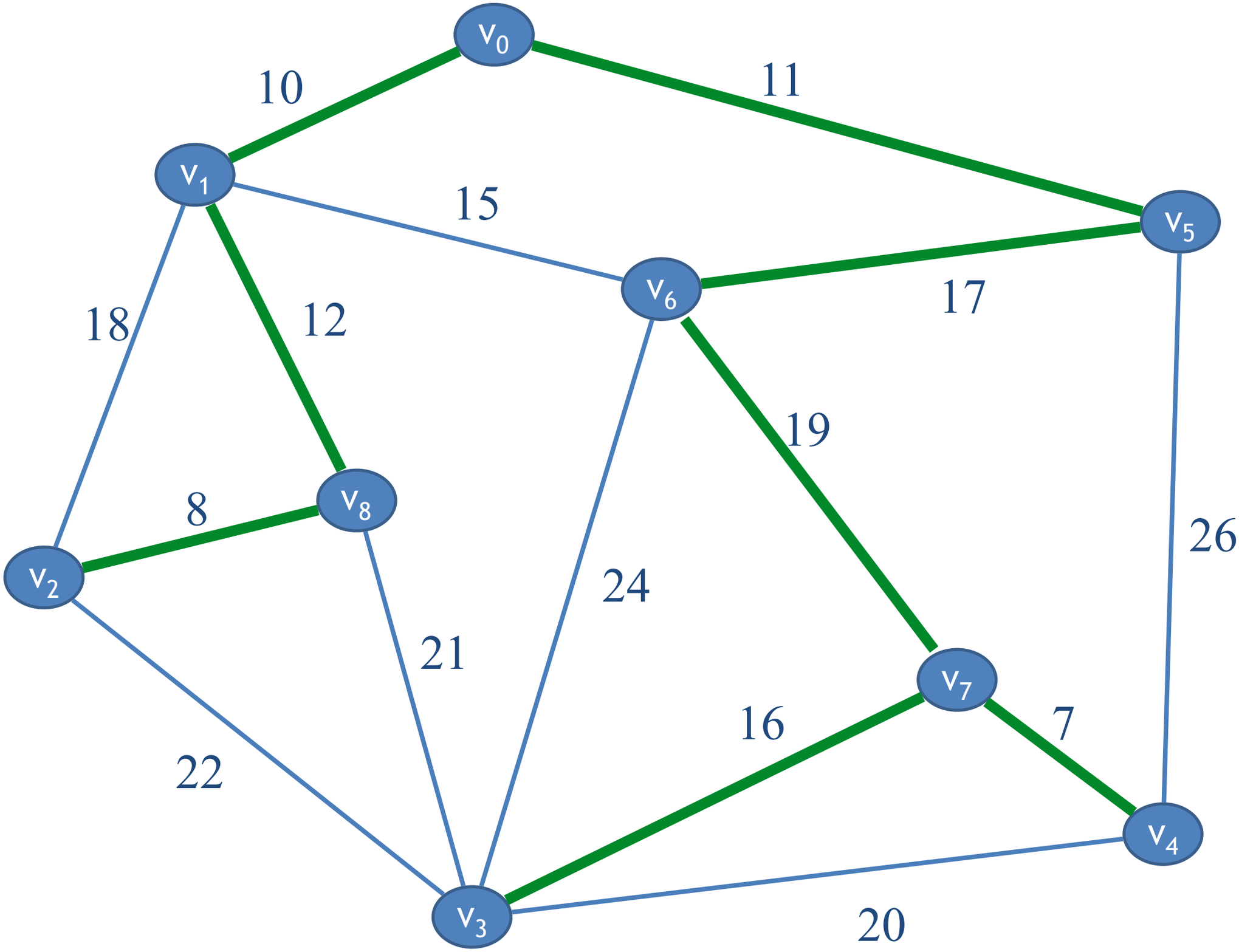
孙猛

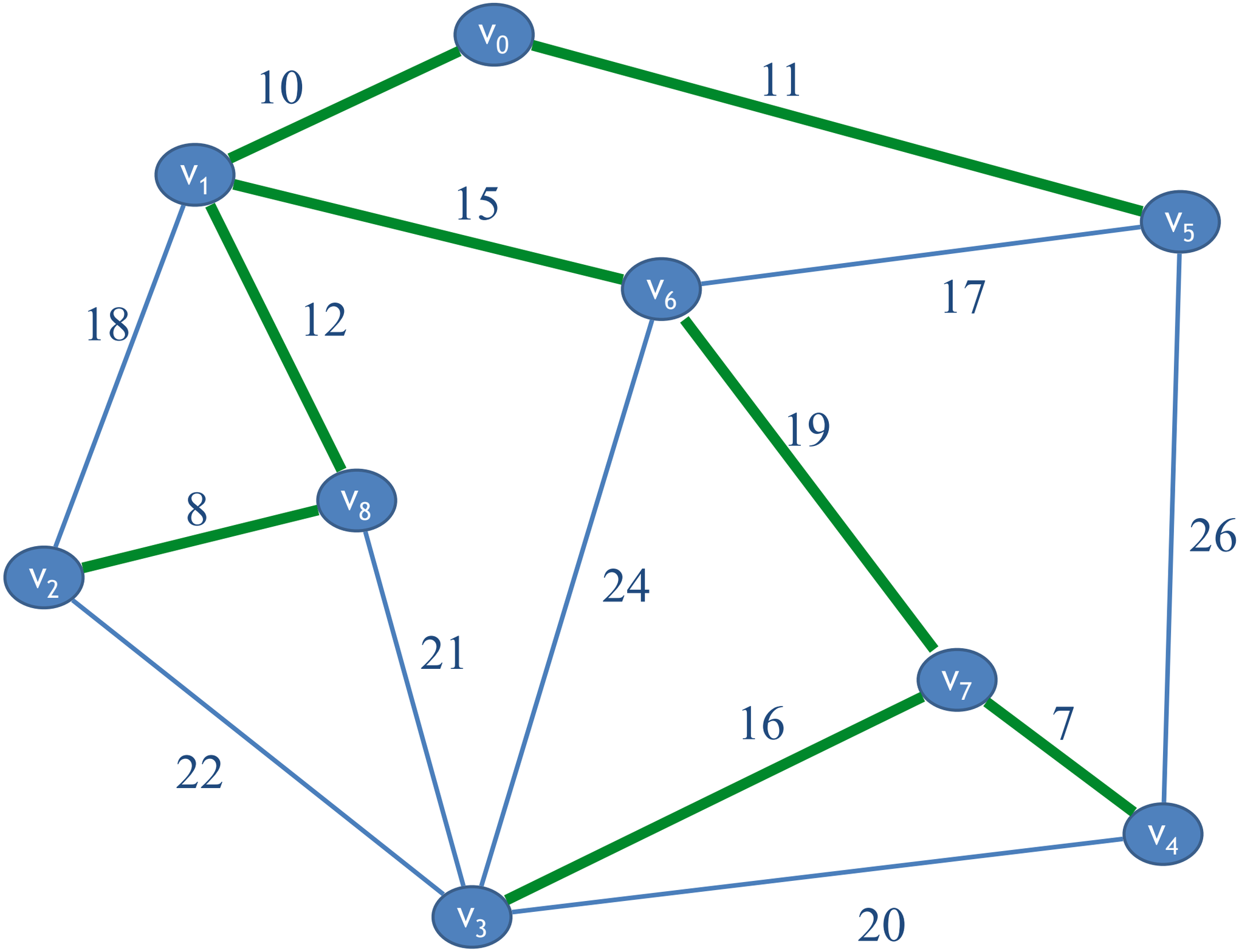
http://www.math.pku.edu.cn/teachers/sunm

2017年11月2日


v0
v1
v2
v3
v4
v5
v6
v7
v8
10
11
15
18
12
17
19
8
26
24
21
16
7
22
20


v0
v1
v2
v3
v4
v5
v6
v7
v8
10
11
15
18
12
17
19
8
26
24
21
22
16
7
20


v0
v1
v2
v3
v4
v5
v6
v7
v8
10
11
15
18
12
17
19
8
26
24
21
16
7
22
20


v0
v1
v2
v3
v4
v5
v6
v7
v8
10
11
15
18
12
17
19
8
26
24
21
16
7
22
20

生成树

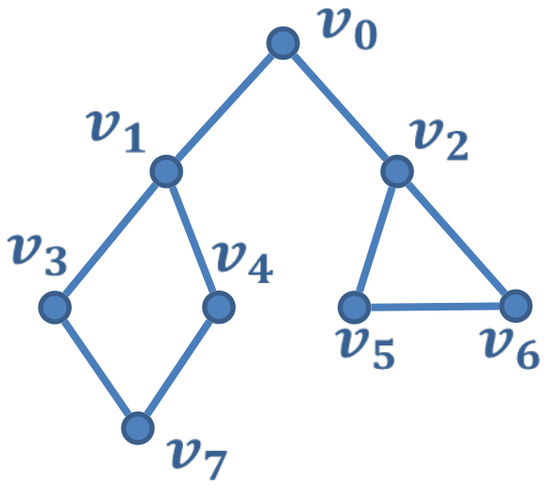
- 对于连通的无向图或强连通的有向图，从任一顶点出发周游，或是对于有根的有向图，从图的根顶点出发周游，可以访问到图中所有的顶点。
- 周游时经过的边加上所有顶点构成了图的一个连通子图，称为图的一棵**生成树**。

生成树的构造

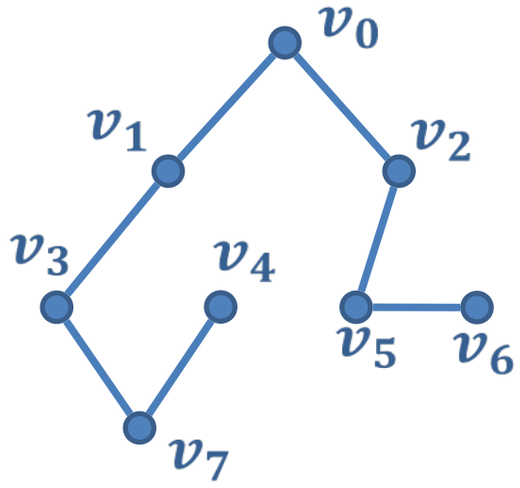
- 周游可以按深度优先搜索，也可以按广度优先搜索。
- 周游中记录访问到的所有顶点以及经过的边，得到的分别是深度优先生成树或广度优先生成树（简称为DFS生成树或BFS生成树）。

例子

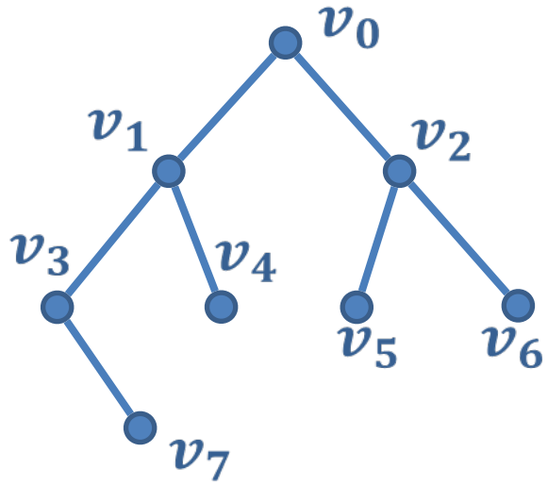
- 从无向图G的顶点v_0出发分别进行深度优先周游和广度优先周游，得到的DFS和BFS生成树如下图所示：



G



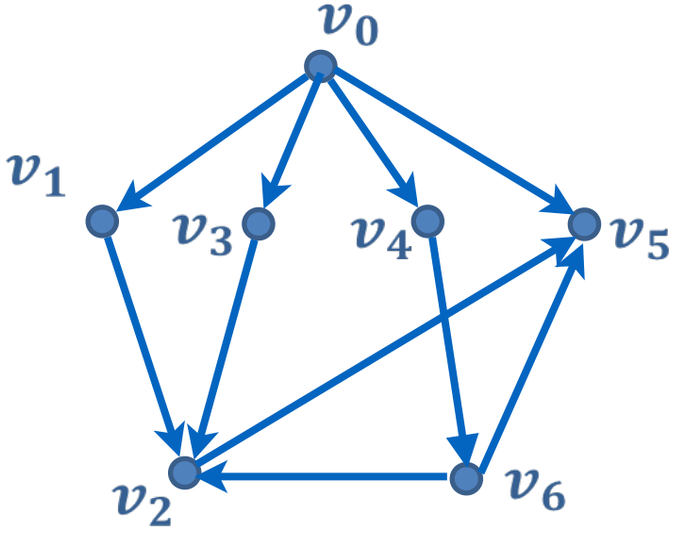
DFS生成树



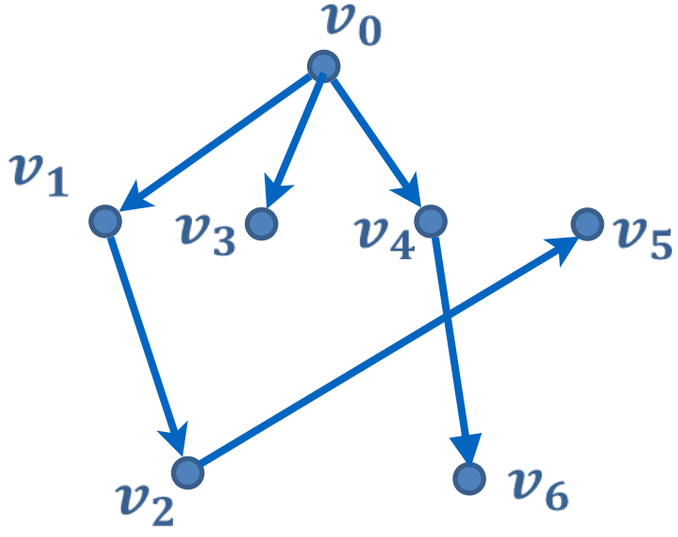
BFS生成树

例子

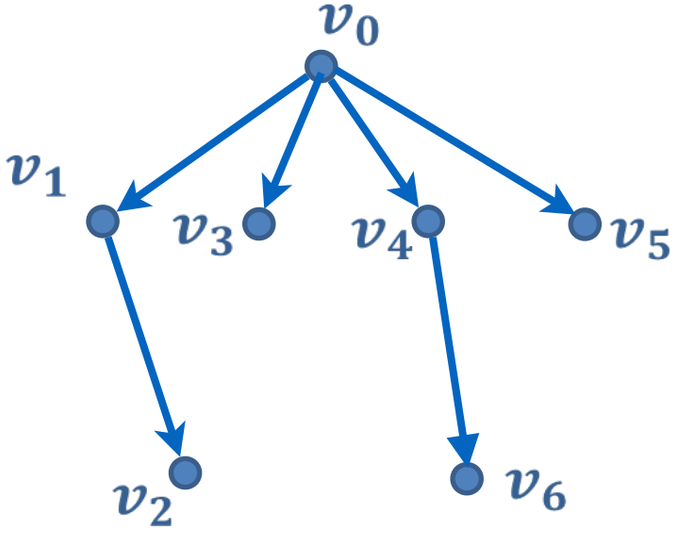
- 从有向图G的顶点v_0出发分别进行深度优先周游和广度优先周游，得到的DFS和BFS生成树如下图所示：



G



DFS生成树



BFS生成树

生成树林

- 对于非连通的无向图和非强连通的有向图，从任一顶点出发通常不能访问到图中所有的顶点，周游结果通常只能得到由各连通分量的生成树或者各个有根子图的生成树所组成的生成树林。

最小生成树

- 图的生成树不唯一，从不同顶点出发，或从同一顶点出发，周游的路径不一样，则得到的生成树都可能不同。
- 网络的生成树中的边也带权，将生成树各边的权值加起来称为**生成树的权**，把权值最小的生成树称为**最小生成树**(Minimum Spanning Tree，简称为MST)。

MST性质

G=(V,E)是网络，U是顶点集V的任一真子集。假设有边 e = (u,v)，其顶点 u∈U，v∈V-U，而且 e 是图 G 中所有一个端点在 U 另一端点在 V - U 里的边中权值最小的边，则一定存在 G 的一棵包括边 e 的最小生成树。

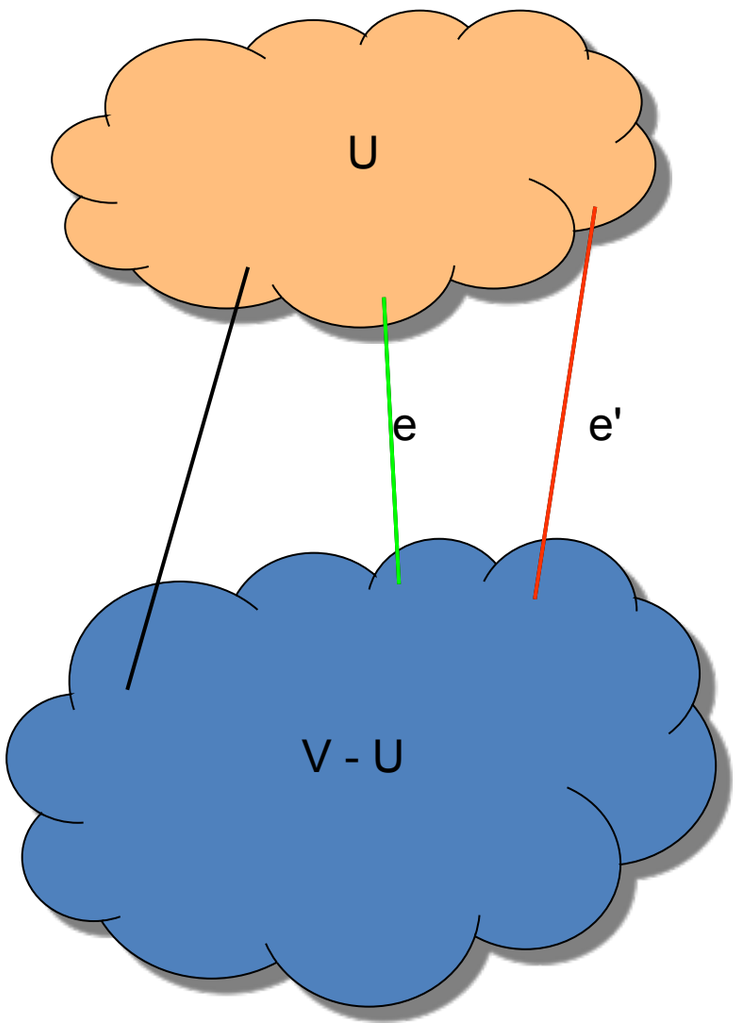


图 G

MST性质的证明：

对 G 的任一最小生成树 T，其中必有一条一端在U另一端在V-U的边e'。加上e得到一个包含环路的图，去掉 e' 得到另一生成树，其权不大于T 的权。

构造最小生成树的Prim算法

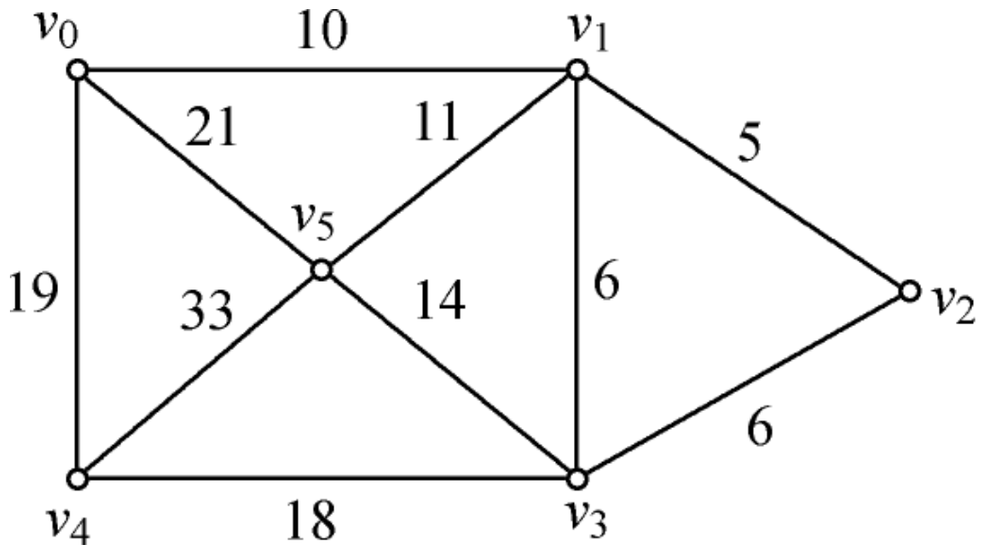
- 设$G=(V,E)$是具有n个顶点的网络，$T=(U,TE)$为（构造中）G的最小生成树，其中U是T的顶点集合，TE是T的边集合，T初始状态为空树。

1. 从集合V中任取一顶点（例如取顶点v_0）放入集合U中，这时$U=\{v_0\}$，TE为空集；
2. 在所有一个顶点在集合U里，另一个顶点在集合$V-U$里的边中，找出权最小的边(u,v)，其中$u\in U$，$v\in V-U$。将该边放入TE，并将顶点v加入集合U。

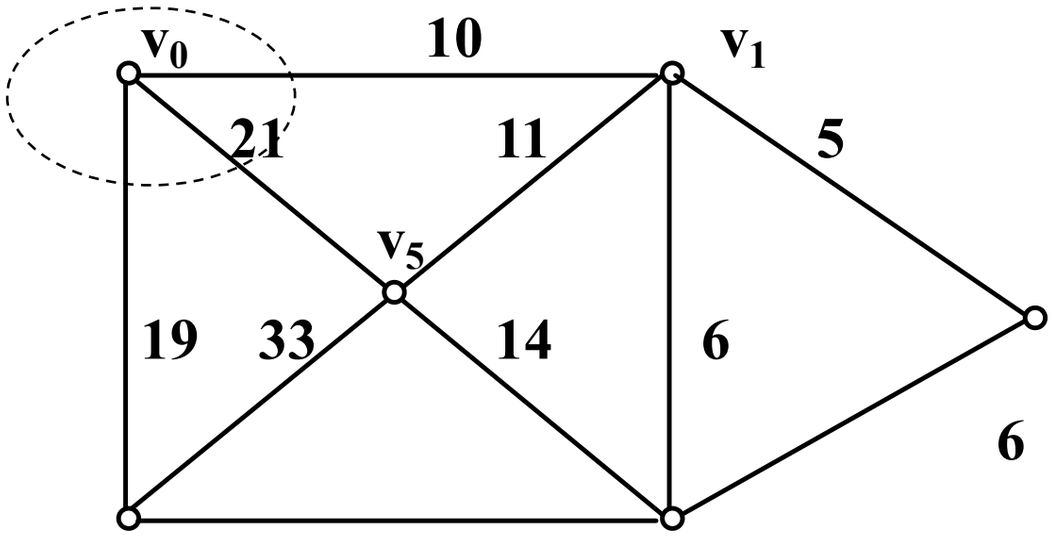
重复上述操作2直到$U=V$为止。

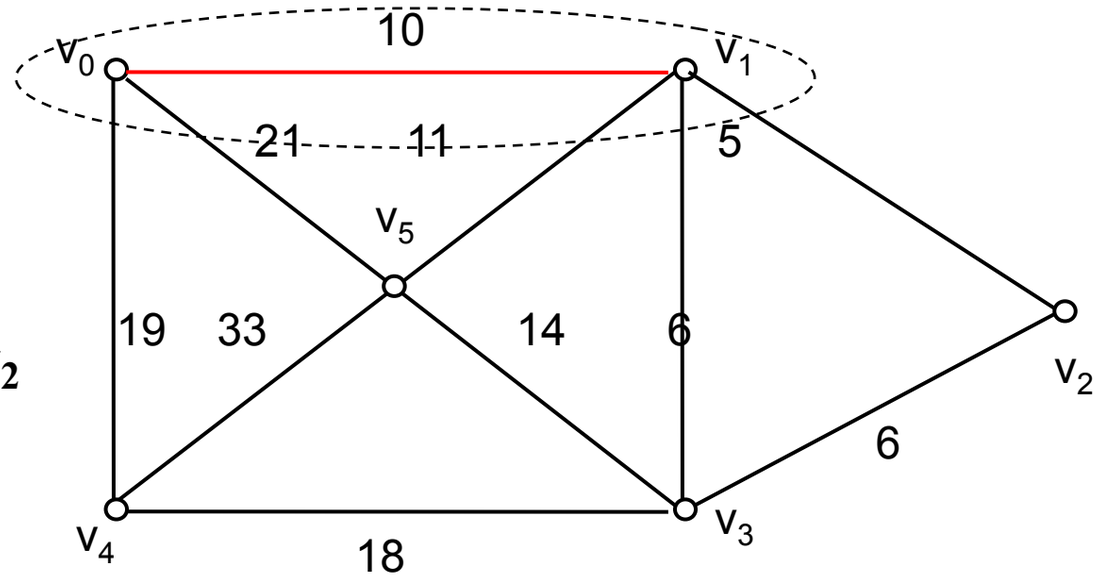
结果TE有$n-1$条边，$T=(U,TE)$就是G的一棵最小生成树。

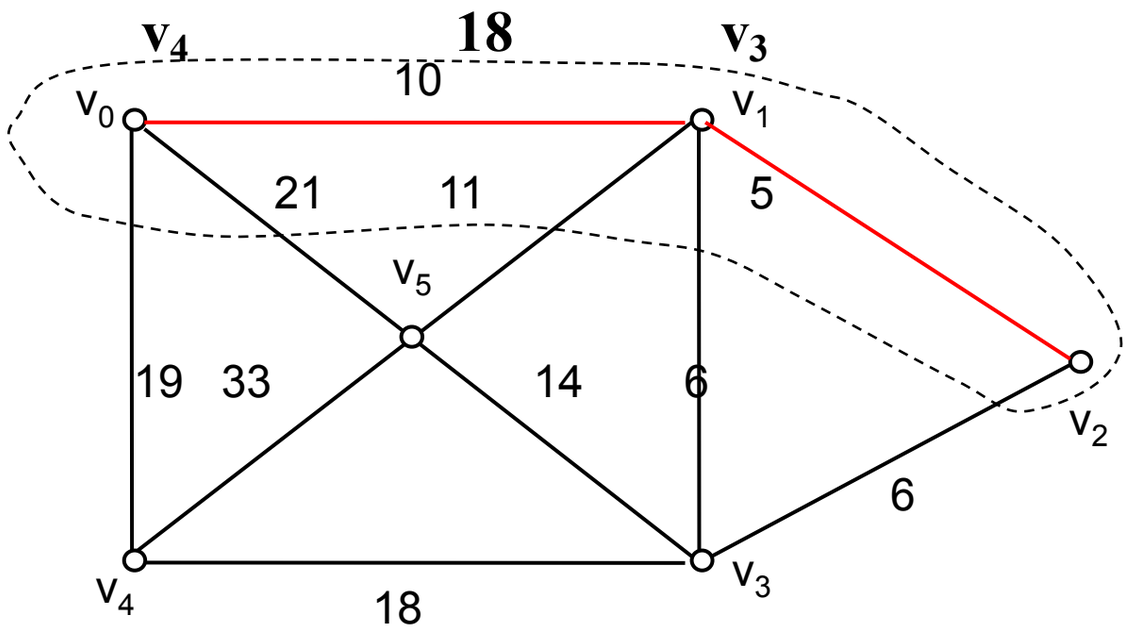
图及其关系矩阵

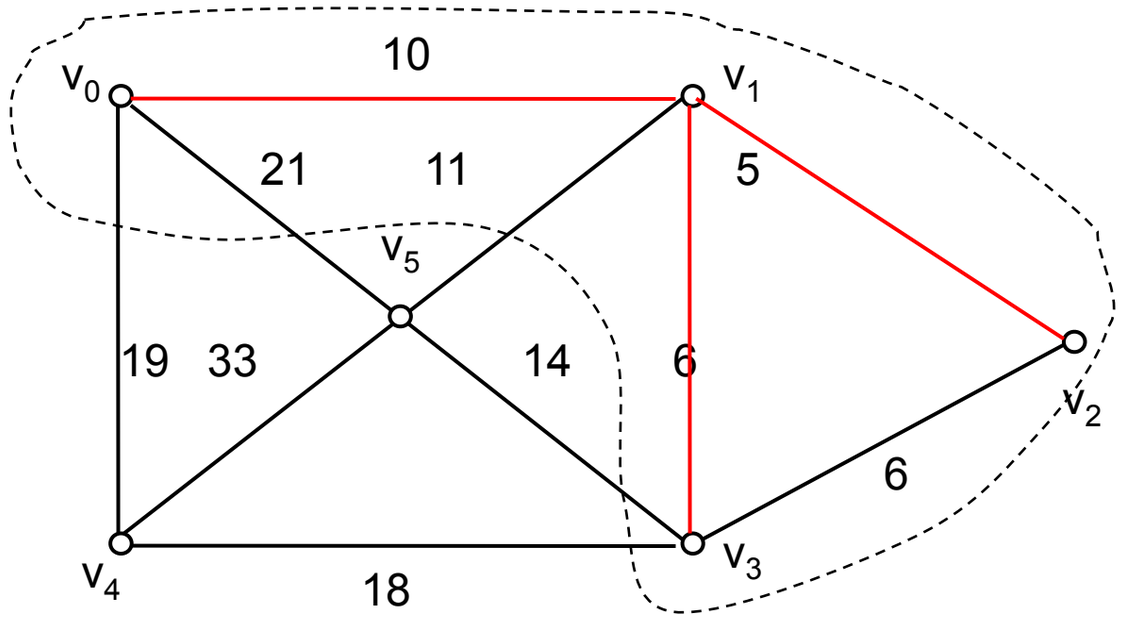


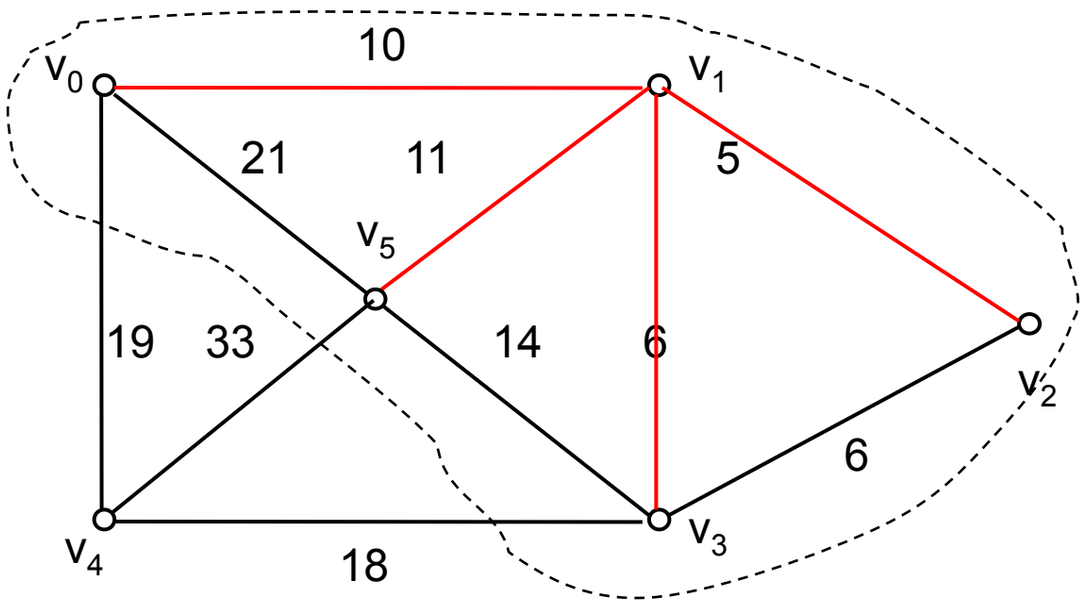
$$\begin{bmatrix} 0 & 10 & \infty & \infty & 19 & 21 \\ 10 & 0 & 5 & 6 & \infty & 11 \\ \infty & 5 & 0 & 6 & \infty & \infty \\ \infty & 6 & 6 & 0 & 18 & 14 \\ 19 & \infty & \infty & 18 & 0 & 33 \\ 21 & 11 & \infty & 14 & 33 & 0 \end{bmatrix}$$

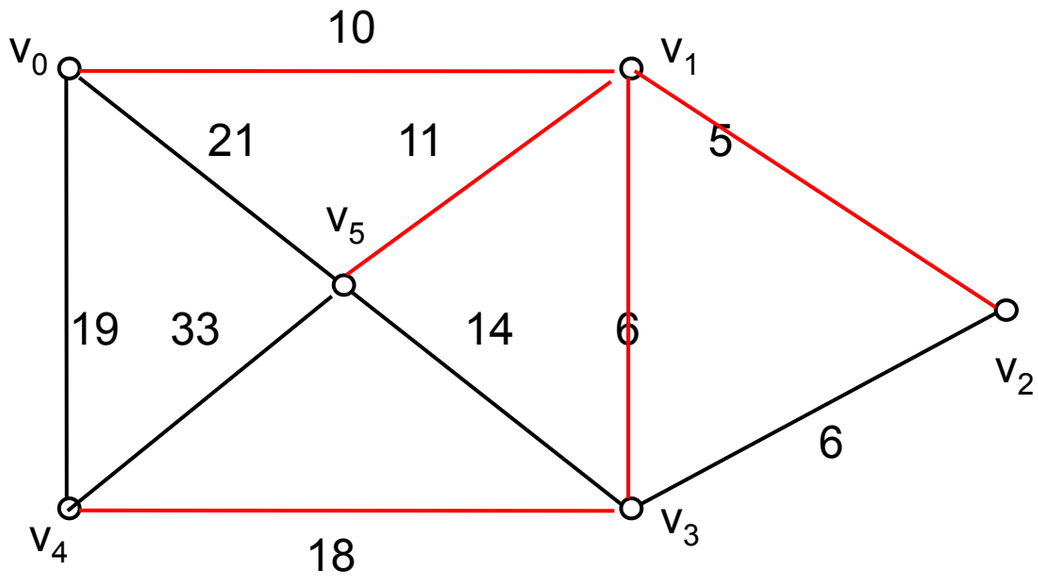

v0
10
v1
21
11
5
v5
19
33
14
6
v2
6
v4
18
v3


10
v0
v1
21
11
5
v5
19
33
14
6
v2
6
v4
18
v3


v4
18
v3
10
v0
v1
21
11
5
v5
19
33
14
6
v2
6
v4
18
v3


10
v0
v1
21
11
5
v5
19
33
14
6
v2
6
v4
18
v3


10
v0
v1
21
11
5
v5
19
33
14
6
v2
6
v4
18
v3


10
v0
v1
21
11
5
v5
19
33
14
6
v2
6
v4
18
v3

存储表示

图采用邻接矩阵表示。

用一对顶点在顶点表中的下标表示一条边，定义如下：

```
typedef struct{
    int  start_vex, stop_vex;      /* 边的起点和终点 */
    AdjType  weight;               /* 边的权 */
}Edge;
```

定义一个类型为Edge的数组：Edge mst[n-1];

算法结束时，mst中存放求出的最小生成树的n-1条边。

数组mst的计算

①. n=6，只有顶点v_0在最小生成树中。

mst=[(0,1,10), (0,2,∞), (0,3,∞), (0,4,19), (0,5,21)]

②. 在mst[0]到mst[4]中取出权值最小的边(0,1,10)，即(v_0, v_1)，将顶点v_1及边(v_0, v_1)加入最小生成树。

数组mst的调整

③. 调整mst[1]到mst[4]。

(v_1, v_2)=5　小于(v_0, v_2)　　调整

(v_1, v_3)=6　小于(v_0, v_3)　　调整

(v_1, v_4)=∞　大于(v_0, v_4)　　不需要调整

(v_1, v_5)=11 小于(v_0, v_5)　　调整

④. 在mst[1]到mst[4]中找出权值最小的边mst[1]，即(v_1, v_2)，将顶点v_2及边(v_1, v_2)加入最小生成树。

mst = [(0,1,10), (1,2,5), (1,3,6), (0,4,19), (1,5,11)]

数组mst的调整

⑤. 调整mst[2]到mst[4]

(v_2, v_3)=6 不小于(v_1, v_3)　　不需要调整

(v_2, v_4)=∞ 大于(v_0, v_4)　　不需要调整

(v_2, v_5)=∞ 大于(v_1, v_5)　　不需要调整

⑥. 在mst[2]到mst[4]中找出权值最小的边mst[2]，即(v_1, v_3)，将顶点v_3及边(v_1, v_3)加入最小生成树。

mst = [(0,1,10), (1,2,5), (1,3,6), (0,4,19), (1,5,11)]

数组mst的调整

⑦. 调整mst[3]到mst[4]

(v_3, v_4)=18 小于(v_0, v_4)　　调整

(v_3, v_5)=14 大于(v_1, v_5)　　不需要调整

⑧. 在mst[3]到mst[4]中找出权值最小的边mst[4]，即(v_1, v_5)，将顶点v_5及边(v_1, v_5)加入最小生成树。互换mst[3]和mst[4]。

mst = [(0,1,10), (1,2,5), (1,3,6), (1,5,11), (3,4,18)]

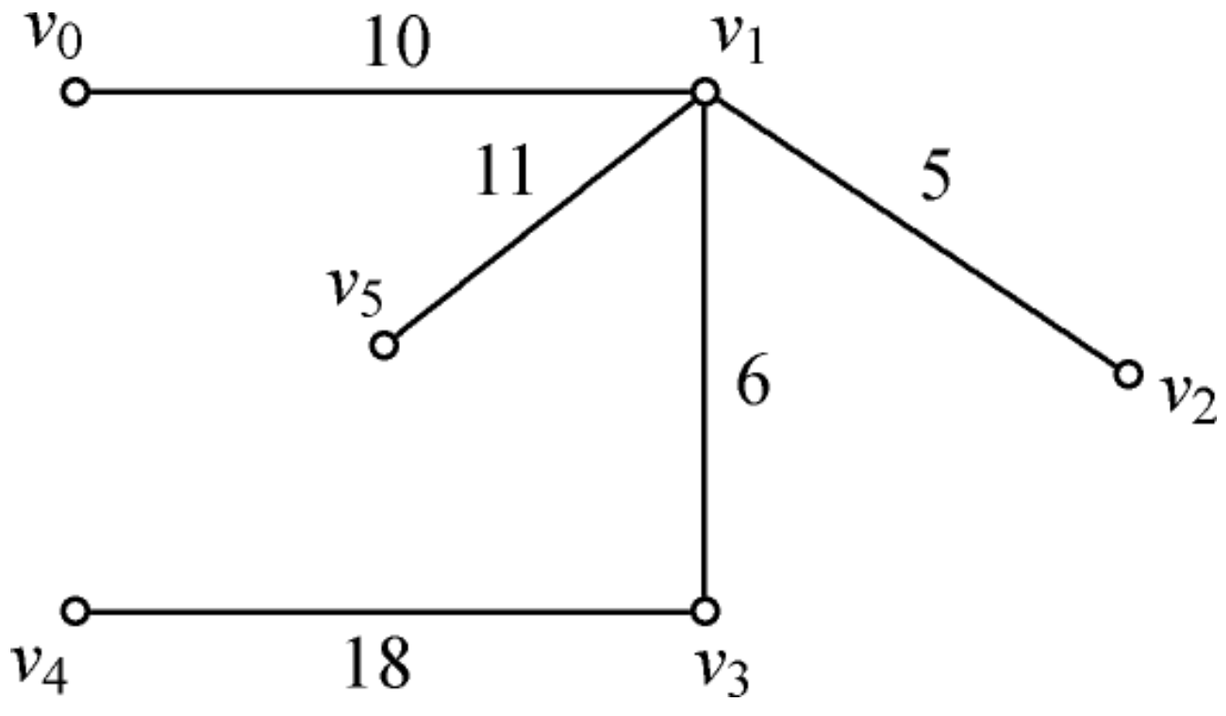
⑨. 调整mst[4]

(v_5, v_4)=33 大于(v_3, v_4)　　不需要调整

数组mst的调整

⑩．将mst[4]加入最小生成树。得到如下图的最小生成树。

mst = [(0,1,10), (1,2,5), (1,3,6), (1,5,11), (3,4,18)]



Prim算法

```
void prim(GraphMatrix * pgraph, Edge mst[ ] ) {
    int i, j, min, vx, vy, n = pgraph->n;
    double weight;     Edge edge;
    for (i = 0; i < n-1; ++ i) { /* mst[ ]初始化为(v0, vi) */
        mst[i].start_vex = 0;
        mst[i].stop_vex = i+1; /* mst[i] 保存v0到vi+1的边信息 */
        mst[i].weight = pgraph->arcs[0][i+1];
    }
    /* mst[i]..mst[n-2]记录从 U 到 V-U 顶点的最短边 */
    for (i = 0; i < n-1; ++ i) {  /* 循环加入各边，共n-1条 */
        weight = MAX;  min = i;
        for (j = i; j < n-1; ++ j)
            /* 从边mst[i] .. mst[n-2]中选最短的 (vx,vy) */
            if(mst[j].weight < weight) {
                weight = mst[j].weight;
                min = j;
            }
```

Prim算法

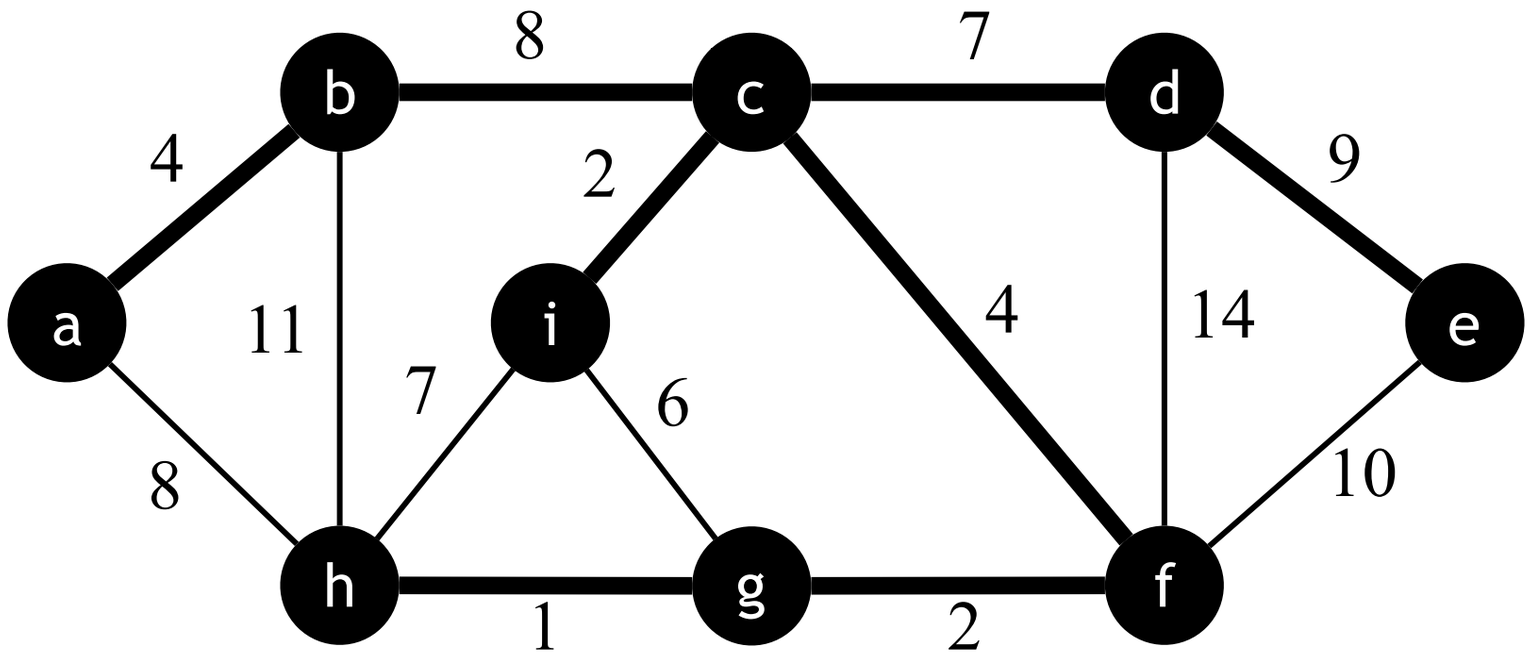
```
/* mst[min]是 U 到 V-U 的边 (vx,vy) 中最短的,
    将mst[min]加入MST（换到 mst[i]） */
edge = mst[min];  mst[min] = mst[i];   mst[i] = edge;
vx = mst[i].stop_vex; /* vx为刚加入mst的顶点的下标 */
/* 考察mst[i+1] .. mst[n-2]，是否该用来自vx的边取代 */
        for(j = i+1; j < n-1; ++ j) {
            vy = mst[j].stop_vex;
            weight = pgraph->arcs[vx][vy];
            if (weight < mst[j].weight) {
                mst[j].weight = weight;
                mst[j].start_vex = vx;
            }
        }
    }
}
```

Prim算法时间代价

- 主要花费在选择最小生成树的n-1条边上。外循环执行n-1次，内循环两个，时间耗费为：

$$\sum_{i=0}^{n-2}(\sum_{j=i}^{n-2}O(1)+\sum_{j=i+1}^{n-2}O(1))\approx 2\sum_{i=0}^{n-2}\sum_{j=i}^{n-2}O(1)$$

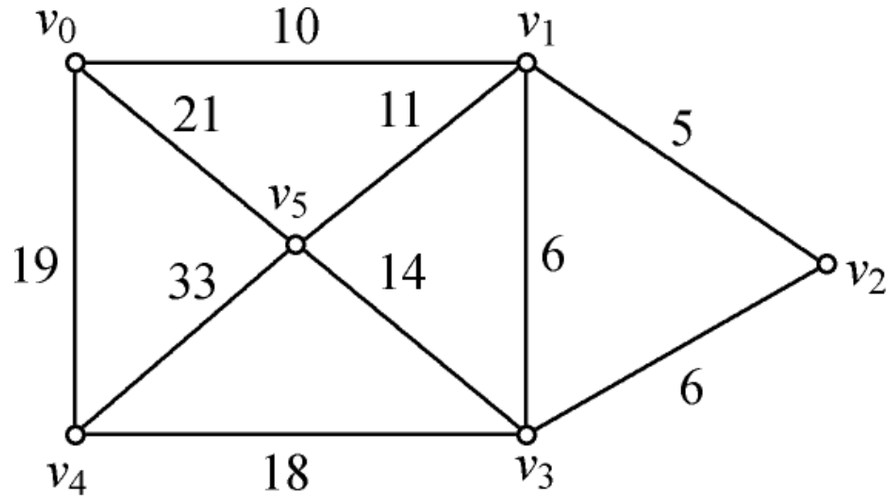
O(1)为一常数，因此，整个算法的时间复杂度为$O(n^2)$。


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2

构造最小生成树的Kruskαl算法

- 设$G=(V,E)$是网络，最小生成树的初始状态为只有n个顶点而无边的非连通图$T=(V,\varphi)$，T中每个顶点自成为一个连通分量。
- 将集合E中的边按权递增顺序排列，从小到大依次选择顶点分别在两个连通分量中的边加入图T，则原来的两个连通分量由于该边的连接而成为一个连通分量。
- 依次类推，直到T中所有顶点都在同一个连通分量上为止，该连通分量就是G的一棵最小生成树。
- 算法正确性可用归纳法证明。

例子



***E*中的边按权递增顺序排列为：**

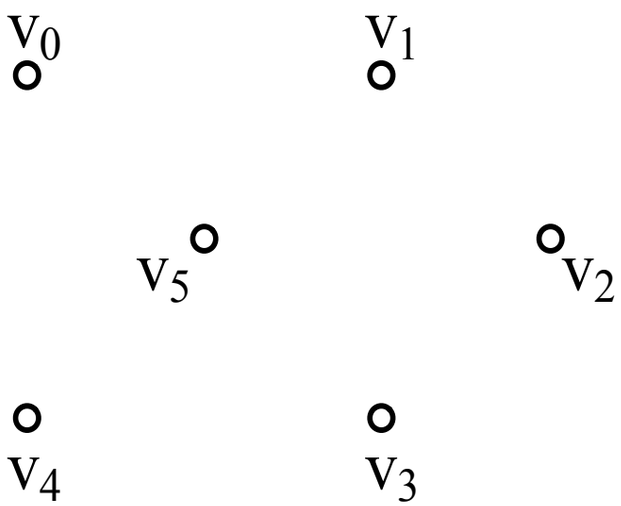
(v_1, v_2):5, (v_1, v_3):6, (v_2, v_3):6,

(v_4, v_5):7, (v_0, v_1):10, (v_1, v_5):11,

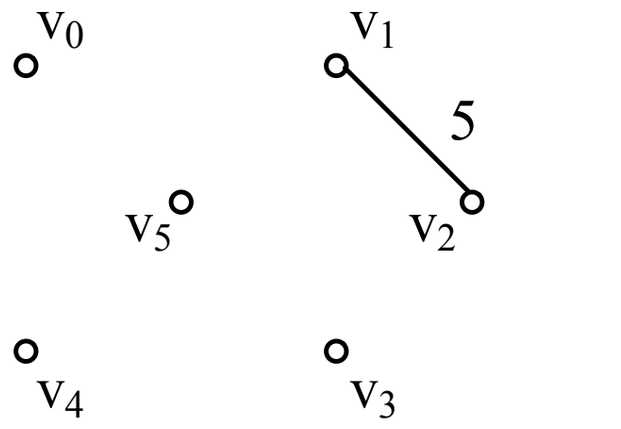
(v_3, v_5):14, (v_3, v_4):18, (v_0, v_4):19,

(v_0, v_5):21,

①初始, *T*为只包含6个顶点的非连通图:

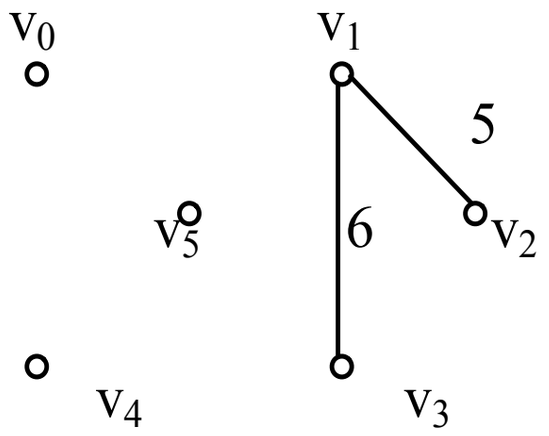


② 边(v_1, v_2)的两个顶点v_1,v_2分别属于两个连通分量，将边(v_1, v_2)加入T。



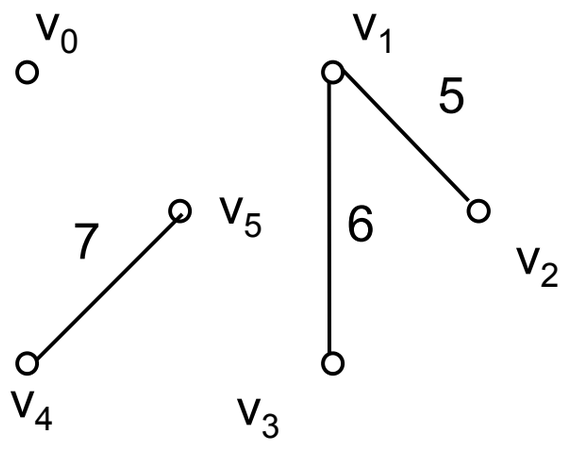
---

③ 同理，将边 (v_1, v_3) 加入T。

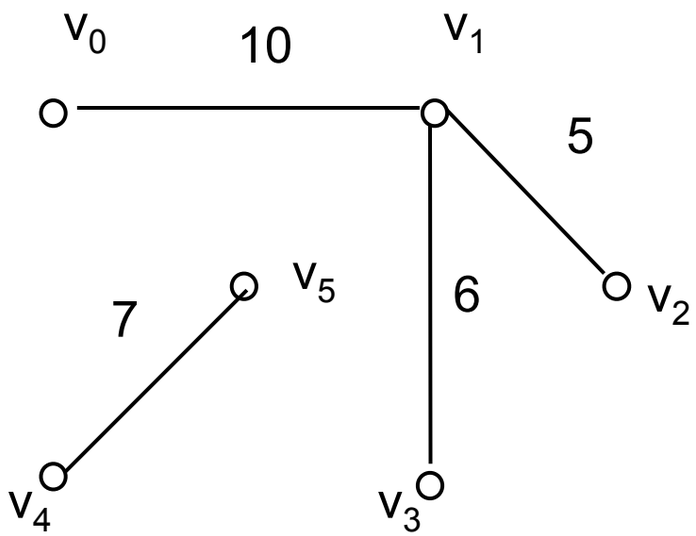


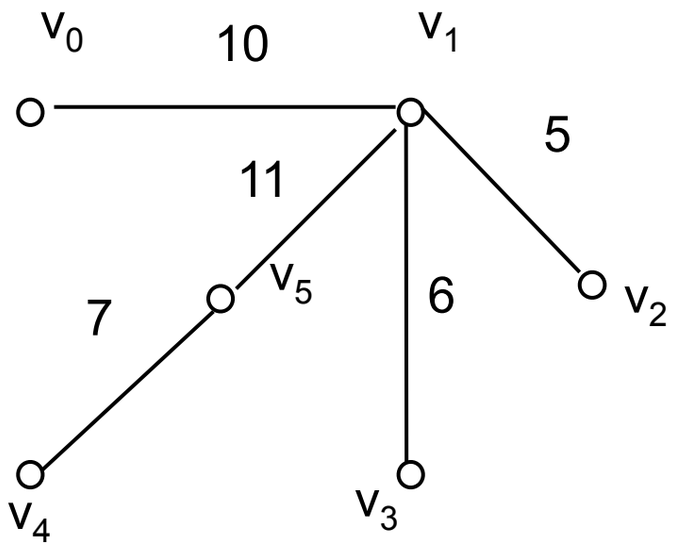
---

④由于边(v_2, v_3)的两个顶点v_2、v_3属于同一个连通分量，因此舍去这条边。下一最短边是(v_4, v_5)，将它加入T。

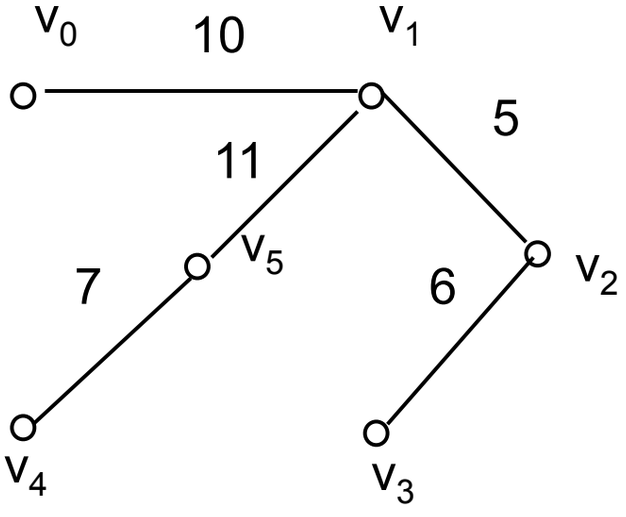


⑤将边(v_0, v_1)、(v_1, v_5)加入T。这时T中含的边数为5条，成为一个连通分量，T就是G的一棵最小生成树。





说明: 图的最小生成树不一定唯一。上例中边(v_1, v_3)和(v_2, v_3)长度相同，该图另一棵最小生成树如右图所示:



算法框架

$T=(V,\varphi)$

while (T **中所含边数** $<n-1$**)**

{

从E**中选取当前最短边**(u,v)**;**

从E**中删去边**(u,v)**;**

if ((u,v)**加入**T**中后不产生回路)**

将边(u,v)**加入**T**中;**

}

数据结构

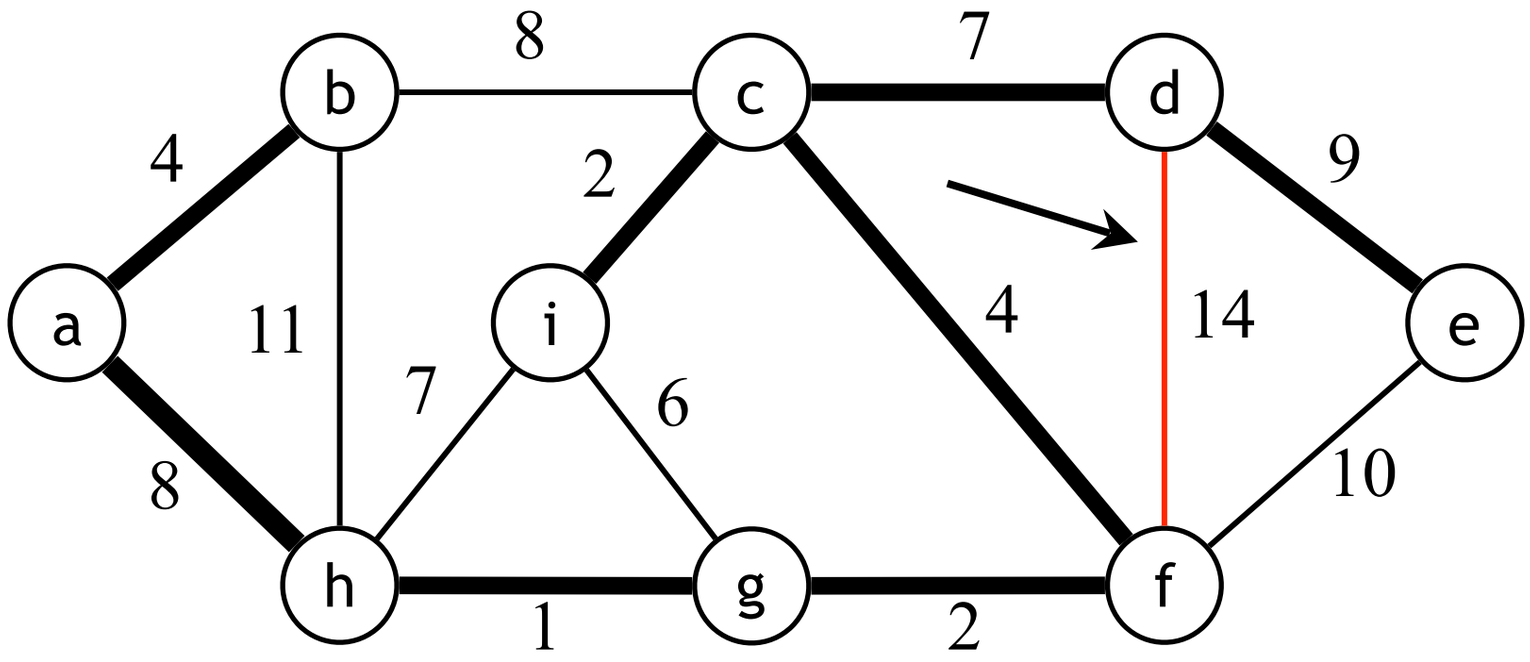
- 边数组**mst**记录构造中的最小生成树，**num**是已定边数；
- 整数数组 **status[0 … n-1]** 记录各个结点所属的连通分支，每个连通分支用一个顶点(代表顶点)的下标代表；
- 初始时各顶点属于自己的连通分支，构造过程中修改为新确定的代表顶点的下标。

Kruskal算法

```
int kruskal(GraphMatrix  *graph, Edge mst[ ] ) {
    int i, j, num = 0, start, stop, n = graph->n;
    double minweight;
    int* status = (int *)malloc( sizeof(int)*n );
    for (i = 0; i < n; ++ i)
        status[i] = i; /* 每个顶点属于自己代表的连通分支 */
    while (num < n - 1){
    minweight = MAX;
    for (i = 0; i < n-1; ++ i)
        for (j = i+1; j < n; ++ j)
            if ( graph->arcs[i][j] < minweight ){
                start = i; stop = j;
                minweight = graph->arcs[i][j];
            }
    /* start和stop是找到的最短边的起点和终点 */
    if (minweight == MAX) return FALSE; /* 无 MST */
```

Kruskal算法

```
/* 加入start和stop组成的边不产生回路*/
if (status[start] != status[stop]){ /* 不属于同一连通分支 */
    mst[num].start_vex = start;
    mst[num].stop_vex = stop;
    mst[num].weight = graph->arcs[start][stop];
    ++ num;
        /* 原 status[stop] 代表的连通分支 的结点，现都属于
           status[start] 代表的连通分支，该代表顶点 */
        for ( j = status[stop], i = 0; i < n; ++ i)
            if (status[i] == j) status[i] = status[start];
    }
    /* 删除start和stop组成的边*/
    graph->arcs[start][stop] = MAX;
  }
  return TRUE; /* 得到了最小生成树*/
}
```


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2

Kruskal算法分析

- 算法的时间复杂性由三重循环确定。
- 外层循环做n-1次，两个内层循环做 n*(n+1)/2次。
- 因此复杂性为 $O(n^3)$。

问题、方法、算法和程序

- 以求网络的最小生成树为例，这是一个问题。Prim 和 Kruskal 提出了解决方法，分别基于网络的 MST 性质和简单连通分支扩充。两者都是抽象的算法。基于它们都可能设计出多种不同的具体算法
  - 其中可能选用不同的数据结构，具体过程也可能有异
  - 不同的实现（实际算法）又可能具有不同的复杂性
- 对最小生成树问题，已知其算法时间复杂度的下界 O(|E|)，但
  - 尚未证明这是下确界（是否可能达到？），也没找到 O(|E|) 算法
  - 因此最小生成树问题的最快算法仍是一个 open problem
  - 可见，这个问题既有实际价值，也有理论追求
- 基于某种算法，可以（用某个编程语言）写出具体的程序
  - 虽然，编程不当也可能达不到算法可能达到的最高效率（复杂度）
  - 需要理解所用的语言机制和数据结构

本讲重点

- 构造最小生成树的prim和kruskal算法。

- prim 算法和kruskal算法的设计都是贪心法。

- 为什么贪心法在这里都能够构造出（最小生成树）最优解?