Graph 2

Zibin Zheng（郑子彬）

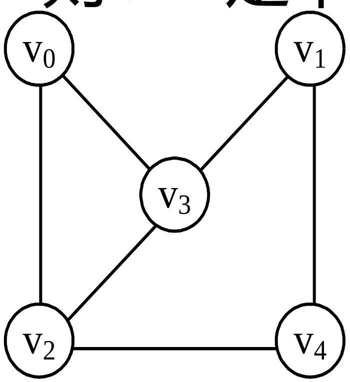
School of Data and Computer Science， SYSU

http://www.inplusslab.com

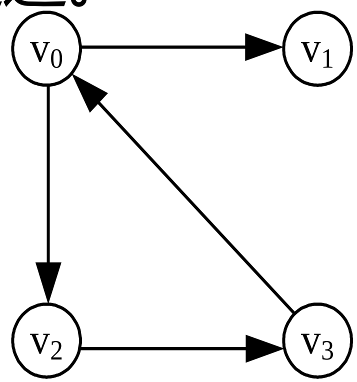
课程主页：http://inpluslab.sysu.edu.cn/dsa2016/

图的连通性问题

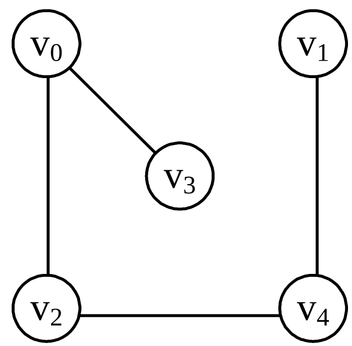
- 一个连通图的生成树是含有该连通图全部顶点的一个极小连通子图。
- 若连通图G的顶点个数为n，则G的生成树的边数为n-1。但是有n-1条边的图不一定是生成树。如果无向图G的一个生成树 G′上添加一条边，则G′中一定有环，因为依附于这条边的两个顶点有另一条路径。相反，如果G′的边数小于n-1，则G′一定不连通。



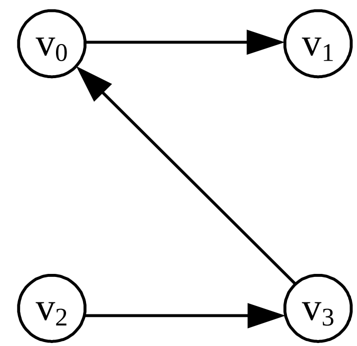
(a) 无向图G_1



(b) 有向图G_2



(a) 无向图G_1的生成树



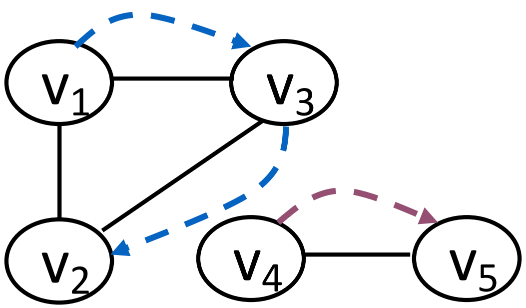
(b) 有向图G_2的生成树

图的连通性问题

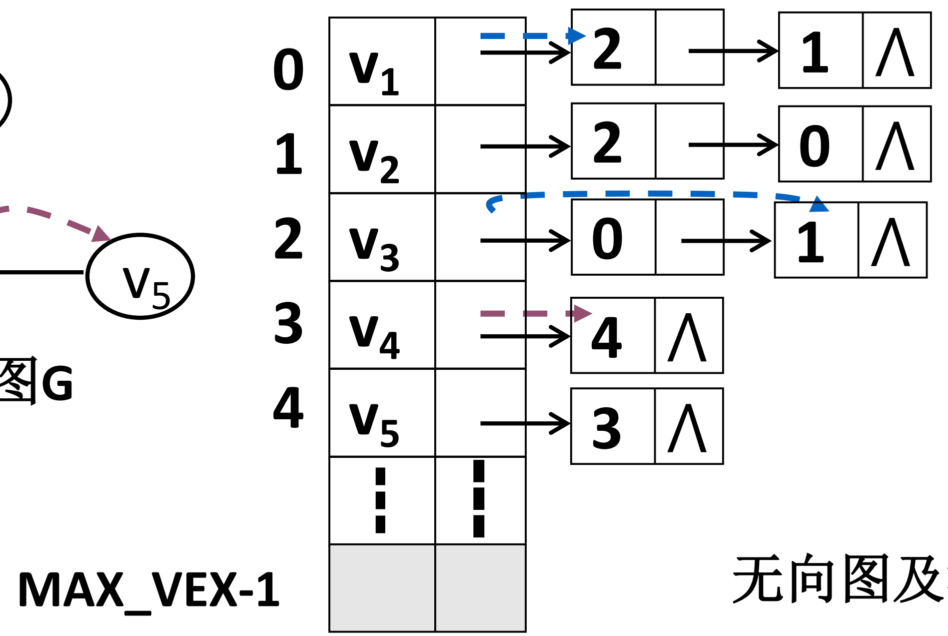
- 无向图的连通分量与生成树
- 对于无向图，对其进行遍历时：
  - ◆ 若是连通图：仅需从图中任一顶点出发，就能访问图中的所有顶点；
  - ◆ 若是非连通图：需从图中多个顶点出发。每次从一个新顶点出发所访问的顶点集序列恰好是各个连通分量的顶点集；

图的连通性问题

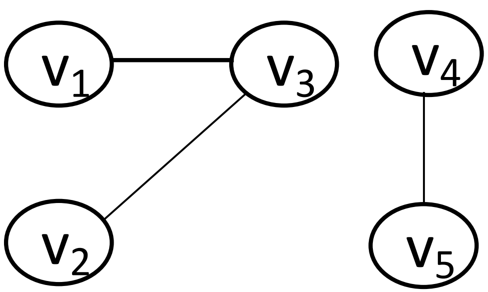
- 如下图所示的无向图是非连通图，按图中给定的邻接表进行深度优先搜索遍历，2次调用DFS所得到的顶点访问序列集是：{ v1 ,v3 ,v2}和{ v4 ,v5 }



(a) 无向图G



(b) G的邻接链表



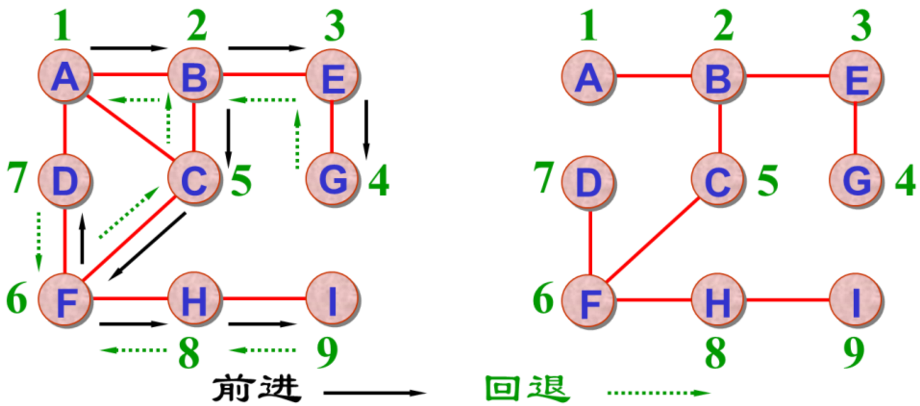
(c) 深度优先生成森林

无向图及深度优先生成森林

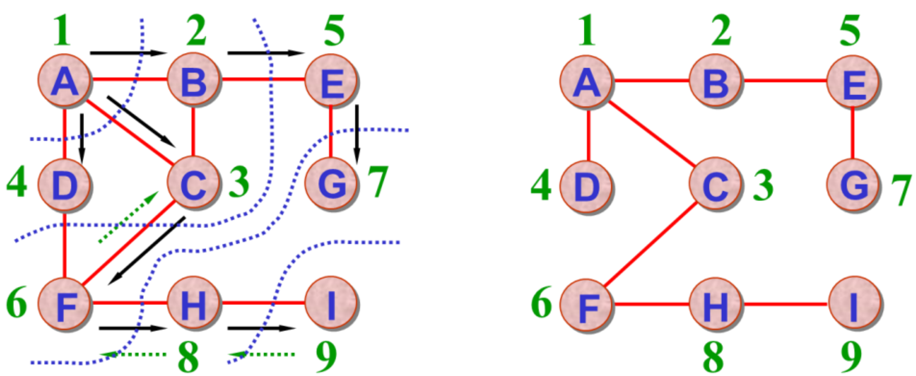
图的连通性问题

- (1) 若G=(V,E)是无向连通图，顶点集和边集分别是V(G)，E(G)。若从G中任意点出发遍历时，E(G)被分成两个互不相交的集合：
  - T(G)：遍历过程中所经过的边的集合；
  - B(G)：遍历过程中未经过的边的集合；
- 显然：$E(G)=T(G)\cup B(G)$，$T(G)\cap B(G)=\emptyset$

图的连通性问题



深度优先搜索过程　　深度优先生成树



广度优先搜索过程　　广度优先生成树

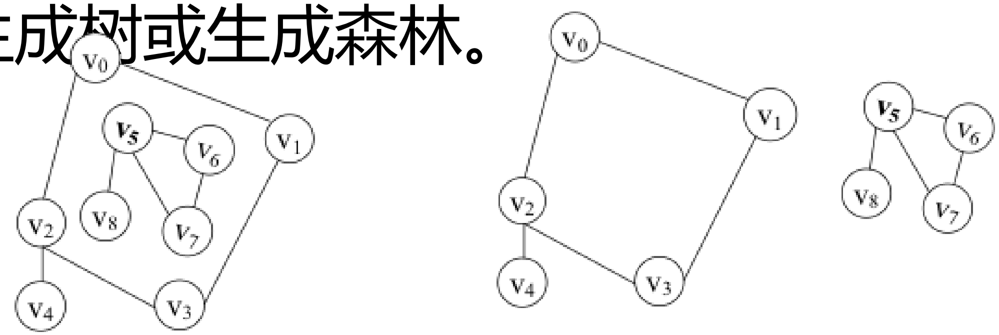
图G′=(V, T(G))是G的极小连通子图，且G′是一棵树。G′称为图G的一棵生成树。

从任意点出发按DFS算法得到生成树G′称为深度优先生成树；

按BFS算法得到的G′称为广度优先生成树。

图的连通性问题

- (2) 若G=(V,E)是无向非连通图，对图进行遍历时得到若干个连通分量的顶点集：V1(G) ,V2(G) ,…,Vn(G)和相应所经过的边集：T1(G) ,T2(G) , …,Tn(G) 。
- 则对应的顶点集和边集的二元组：Gi=(Vi(G),Ti(G)) (1≦i≦n)是对应分量的生成树，所有这些生成树构成了原来非连通图的生成森林。
- 说明：当给定无向图要求画出其对应的生成树或生成森林时，必须先给出相应的邻接表，然后才能根据邻接表画出其对应的生成树或生成森林。



(a) 非连通的无向图G3　　(b) 非连通无向图G3的连通分量

非连通无向图的连通分量示意图

最小生成树

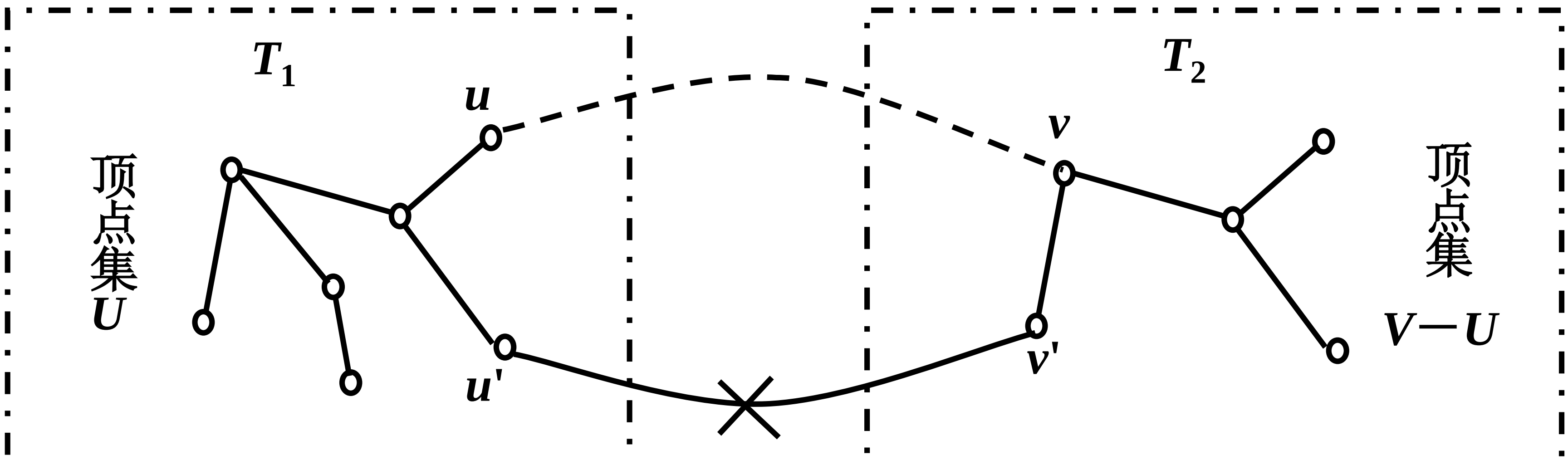
- 如果连通图是一个带权图，则其生成树中的边也带权，生成树中所有边的权值之和称为生成树的代价
- 最小生成树(Minimum Spanning Tree) ：带权连通图中代价最小的生成树称为最小生成树
- 最小生成树在实际中具有重要用途，如设计通信网。设图的顶点表示城市，边表示两个城市之间的通信线路，边的权值表示建造通信线路的费用。n个城市之间最多可以建n×(n-1)/2条线路，如何选择其中的n-1条，使总的建造费用最低?

最小生成树

- 构造最小生成树的算法有许多，基本原则是：
  - ◆ 尽可能选取权值最小的边，但不能构成回路
  - ◆ 选择n-1条边构成最小生成树

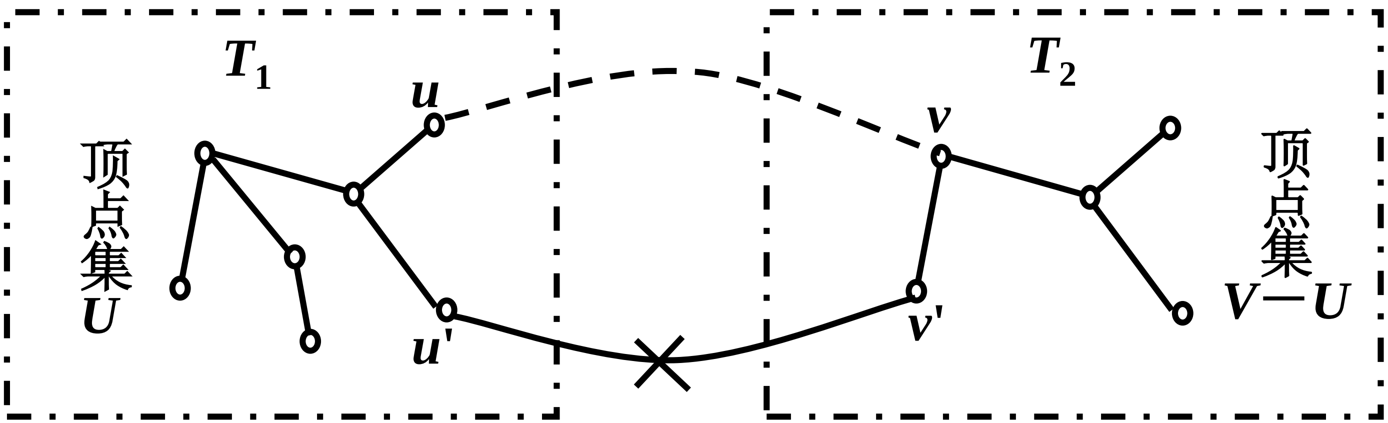
MST性质

假设$G=(V, E)$是一个无向连通网，U是顶点集V的一个非空子集。若(u, v)是一条具有最小权值的边，其中$u \in U$，$v \in V-U$，则必存在一棵包含边(u, v)的最小生成树。



最小生成树

- 反证法：设图G的任何一棵最小生成树都不包含边(u,v)。设T是G的一棵生成树，则T是连通的，从u到v必有一条路径(u,…,v)，当将边(u,v)加入到T中时就构成了回路。则路径(u, …,v)中必有一条边(u’,v’) ，满足u’∈U ，v’∈V-U 。删去边(u’,v’) 便可消除回路，同时得到另一棵生成树T’。
- 由于(u,v)是U中顶点到V-U中顶点之间权值最小的边，故(u,v)的权值不会高于(u’,v’)的权值，T’的代价也不会高于T，T’是包含(u,v) 的一棵最小生成树，与假设矛盾。



最小生成树--普里姆(Prim)算法

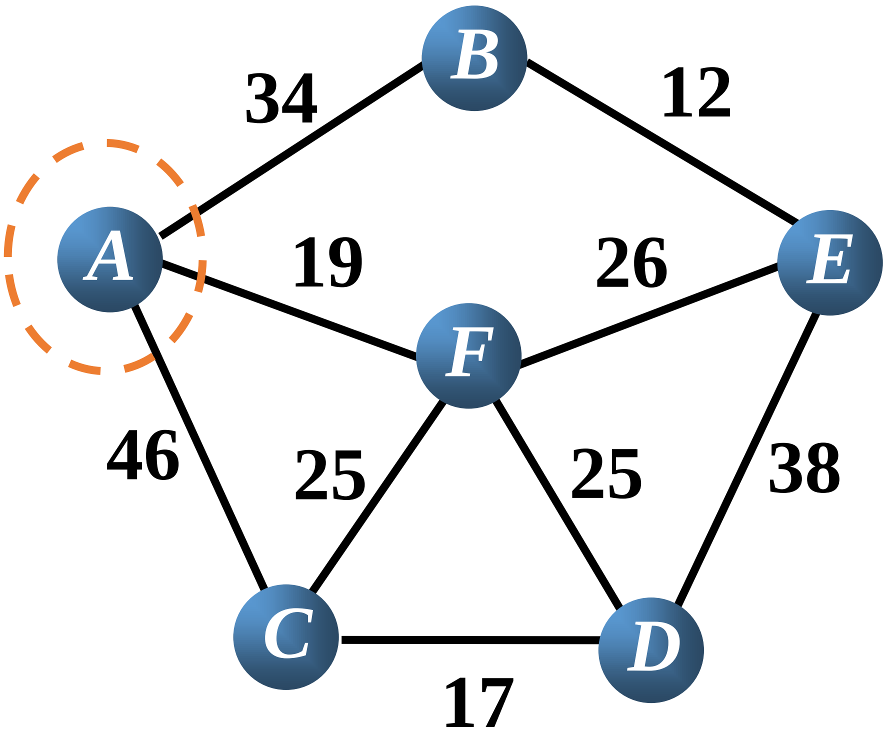
基本思想：设$G=(V, E)$是具有n个顶点的连通网，$T=(U, TE)$是G的最小生成树， T的**初始状态**为$U=\{u_0\}$（$u_0 \in V$），$TE=\{\}$，重复执行下述操作：在所有$u \in U$，$v \in V-U$的边中找一条代价最小的边(u, v)并入集合TE，同时v并入U，直至$U=V$。

Prim算法的基本思想用伪代码描述如下：

1. 初始化：U = {v0}； TE={ }；
2. 重复下述操作直到U = V：
   - 2.1 在E中寻找最短边(u，v)，且满足$u \in U$，$v \in V-U$；
   - 2.2 U = U + {v}；
   - 2.3 TE = TE + {(u，v)}；

关键:是如何找到连接U和$V-U$的最短边。

最小生成树--普里姆(Prim)算法

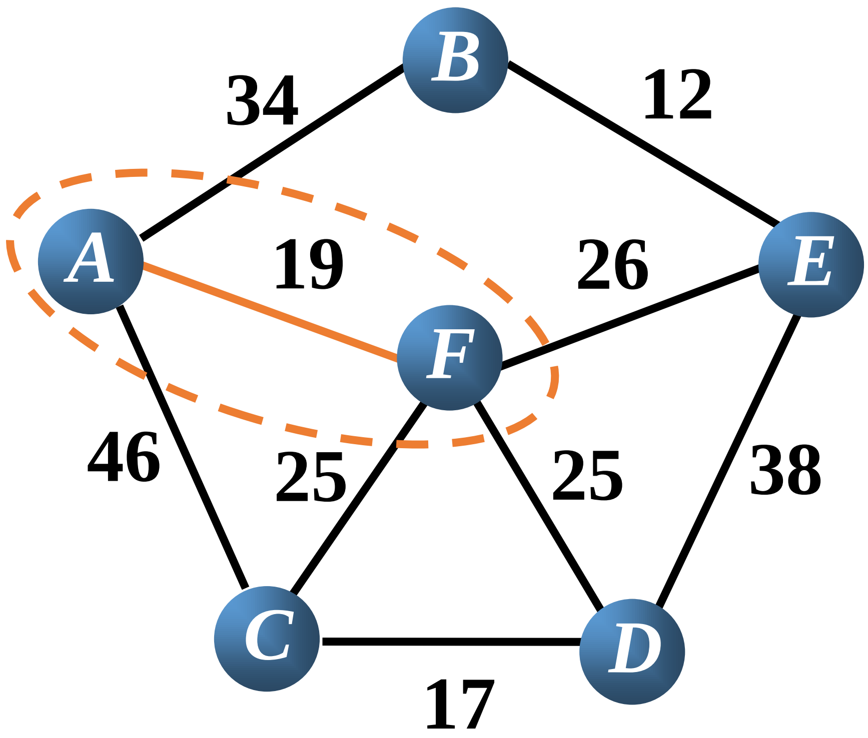


U={A}

V-U={B, C, D, E, F}

cost={(*A*, *B*)34,
(*A*, *C*)46,
(*A*, *D*)∞,
(*A*, *E*)∞,
(*A*, *F*)19}

最小生成树--普里姆(Prim)算法

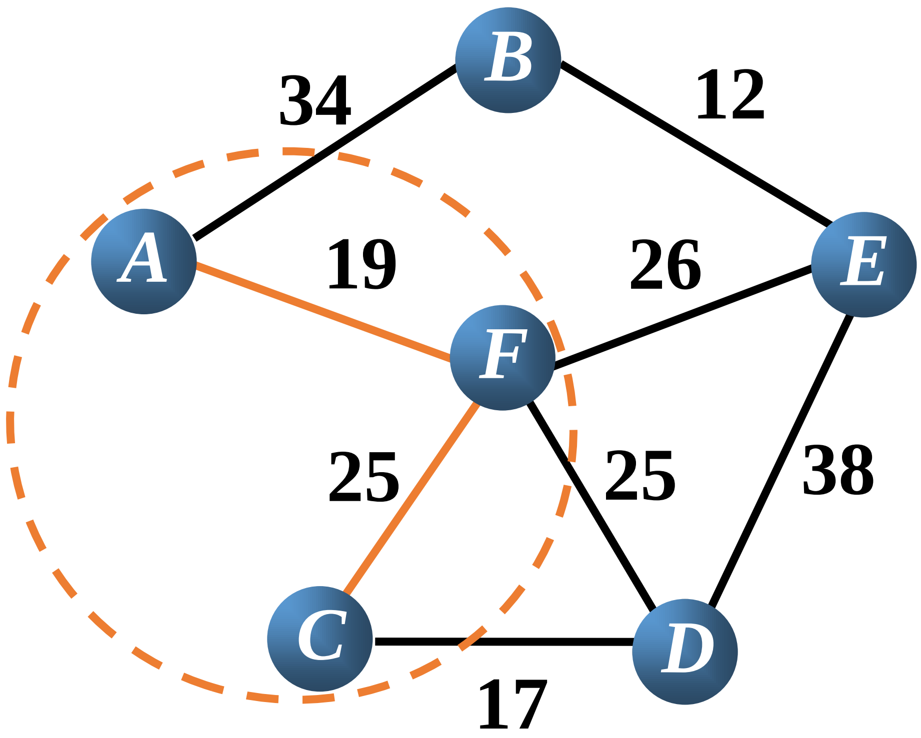


U={A, F}

V-U={B, C, D, E}

cost={(*A*, *B*)34,
(*F*, *C*)25,
(*F*, *D*)25,
(*F*, *E*)26}

最小生成树--普里姆(Prim)算法

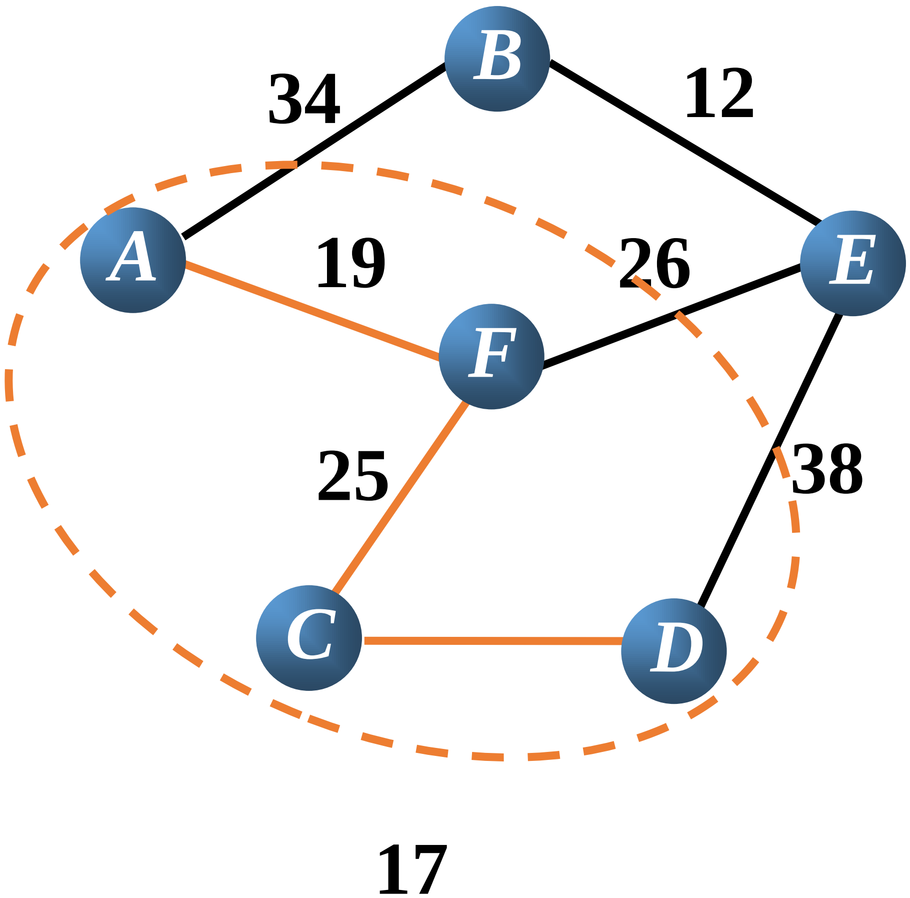


U={A, F, C}

V-U={B, D, E}

cost={(*A*, *B*)34,
(*C*, *D*)17,
(*F*, *E*)26}

最小生成树--普里姆(Prim)算法

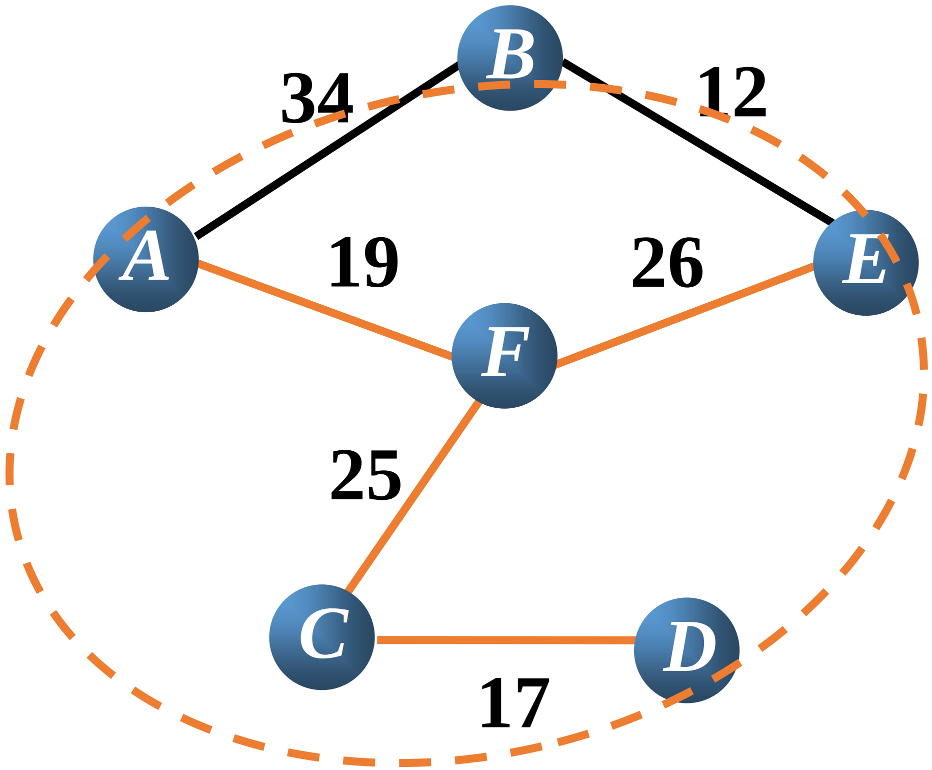


U={A, F, C, D}

V-U={B, E}

cost={(*A*, *B*)34,
(*F*, *E*)26}

最小生成树--普里姆(Prim)算法

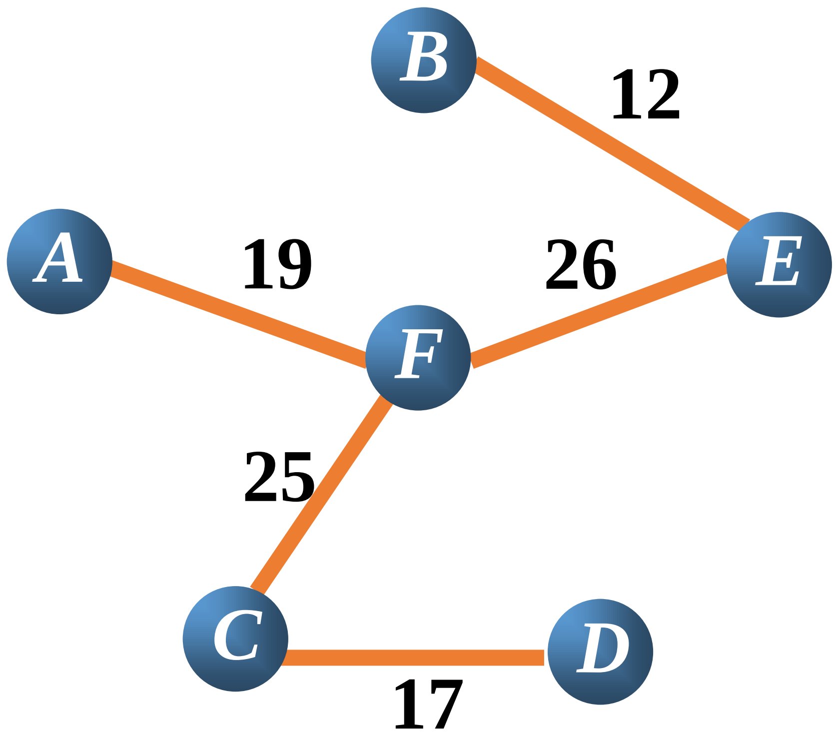


U={A, F, C, D, E}

V-U={B}

cost={(*E*, *B*)12}

最小生成树--普里姆(Prim)算法



U={A, F, C, D, E, B}

V-U={ }

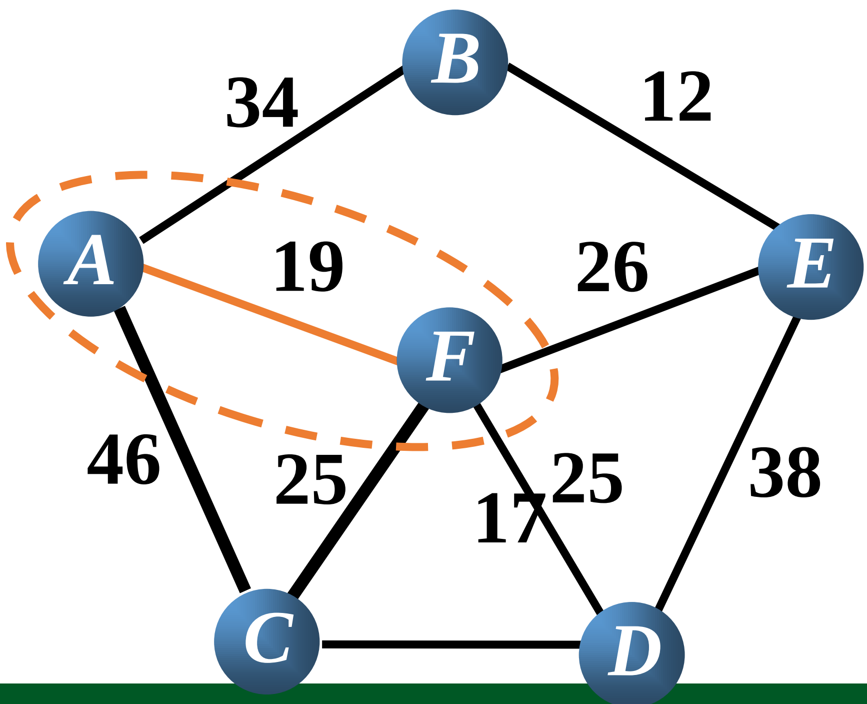
数据结构设计

1. 图的存储结构：由于在算法执行过程中，需要不断读取任意两个顶点之间边的权值，所以，图采用邻接矩阵存储。

最小生成树--普里姆(Prim)算法

数据结构设计

2. 候选最短边集：设置数组shortEdge[n]表示候选最短边集，数组元素包括adjvex和lowcost两个域，分别表示候选最短边的邻接点和权值。



对于顶点C，只需保留

min{arc[A][C], arc[F][C]}

对于V-U中的每个顶点，只保留从该顶点到U中的某顶点的最短边。

最小生成树--普里姆(Prim)算法

数据结构设计

如何用**lowcost**和**adjvex**表示候选最短边集？

- 候选最短边(v_i, v_k)的权值为w，其中v_i∈V-U，v_k∈U：

$$
\begin{cases}
\text{shortEdge[i].adjvex} = k \\
\text{shortEdge[i].lowcost} = w
\end{cases}
$$

- 顶点v_k从集合V-U进入集合U后，依据下式更新数组shortEdge：

$$
\begin{cases}
\text{shortEdge[j].lowcost} = \min\ \{\text{shortEdge[j].lowcost}\ ,\ \text{cost}(v_j, v_k)\} \\
\text{shortEdge[j].adjvex} = k\ （\text{如果边}(v_j, v_k)\text{的权值较小}）
\end{cases}
$$

最小生成树--普里姆(Prim)算法

Prim算法——伪代码

1. 初始化两个辅助数组lowcost和adjvex；

2. 输出顶点u_0，将顶点u_0加入集合U中；

3. 重复执行下列操作n−1次

   3.1 在lowcost中选取最短边，取adjvex中对应的顶点序号k；

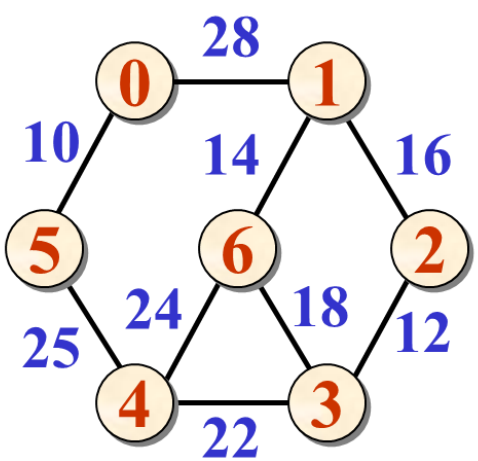
   3.2 输出顶点k和对应的权值；

   3.3 将顶点k加入集合U中；

   3.4 调整数组lowcost和adjvex；

最小生成树--普里姆(Prim)算法

- 两个辅助数组：
  - lowcost[ ] 存放生成树顶点集合内顶点到生成树外各顶点的各边上的当前最小权值
  - nearvex[ ] 记录生成树顶点集合外各顶点距离集合内哪个顶点最近(即权值最小)。
- 例子



$$
\begin{pmatrix}
0 & 28 & \infty & \infty & \infty & 10 & \infty \\
28 & 0 & 16 & \infty & \infty & \infty & 14 \\
\infty & 16 & 0 & 12 & \infty & \infty & \infty \\
\infty & \infty & 12 & 0 & 22 & \infty & 18 \\
\infty & \infty & \infty & 22 & 0 & 25 & 24 \\
10 & \infty & \infty & \infty & 25 & 0 & \infty \\
\infty & 14 & \infty & 18 & 24 & \infty & 0
\end{pmatrix}
$$

最小生成树--普里姆(Prim)算法

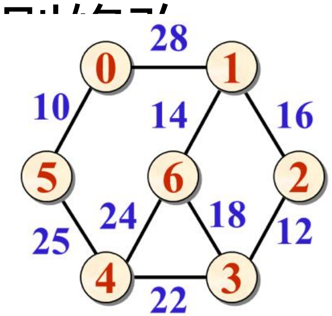
- 若选择从顶点0出发，即u0 = 0，则两个辅助数组的初始状态为：

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞
nearvex	−1	0	0	0	0	0	0

- 然后反复做以下工作：
  - 在 lowcost [ ]中选择 nearvex[i] ≠ -1 && lowcost[i]最小的边, 用 v 标记它。则选中的权值最小的边为(nearvex[v], v), 相应的权值为 lowcost[v]。

最小生成树--普里姆(Prim)算法

- 将 nearvex[v] 改为-1，表示它已加入生成树顶点集合。
- 将边 (nearvex[v], v, lowcost[v]) 加入生成树的边集合。
- 取 lowcost[i] = min{ lowcost[i], Edge[v][i] }，即用生成树顶点集合外各顶点 i 到刚加入该集合的新顶点 v 的距离 Edge[v][i] 与原来它们到生成树顶点集合中顶点的最短距离 lowcost[i] 做比较，取距离近的作为这些集合外顶点到生成树顶点集合内顶点的最短距离。
- 如果生成树顶点集合外顶点 i 到刚加入该集合的新顶点 v 的距离比原来它到生成树顶点集合中顶点的最短距离还要近，
x[i] : nearvex[i] = v，表示生成树外顶点i到生成
前距离最近



	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞
nearvex	−1	0	0	0	0	0	0

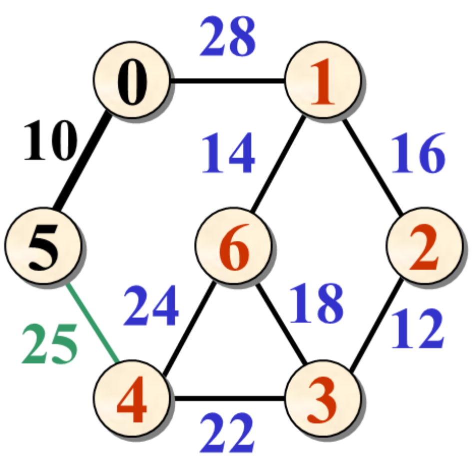
选 v=5　　选边 (0,5)

最小生成树--普里姆(Prim)算法

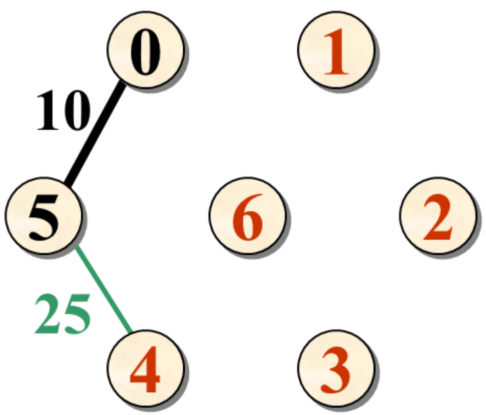
顶点v=5加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	25	10	∞
nearvex	−1	0	0	0	5	−1	0

选 v=4　选边 (5,4)



原图



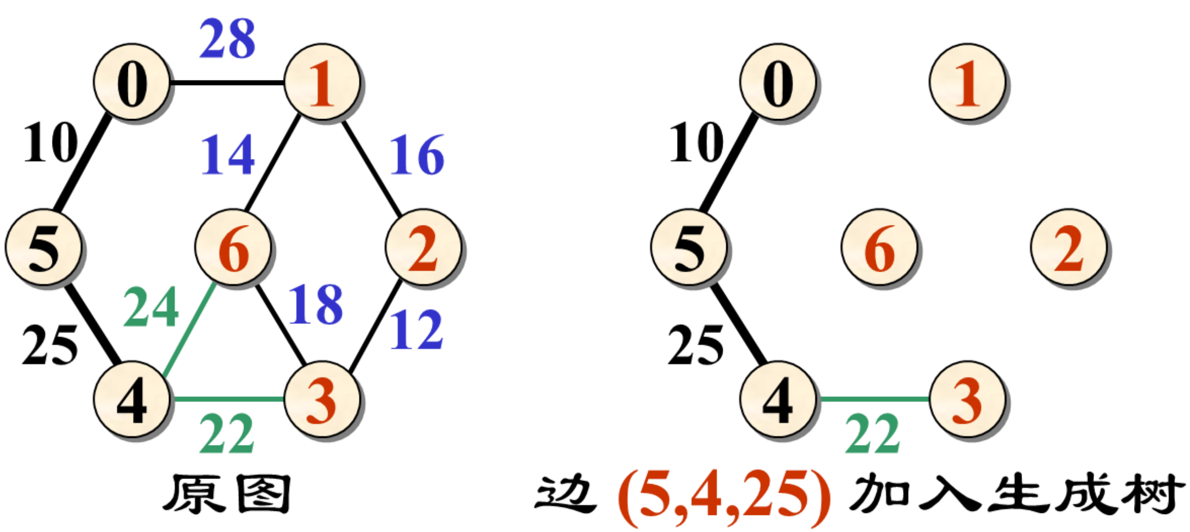
边 (0,5,10) 加入生成树

最小生成树--普里姆(Prim)算法

顶点v=4加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	∞	22	25	10	24
nearvex	−1	0	0	4	−1	−1	4

选 v=3　　选边 (4,3)

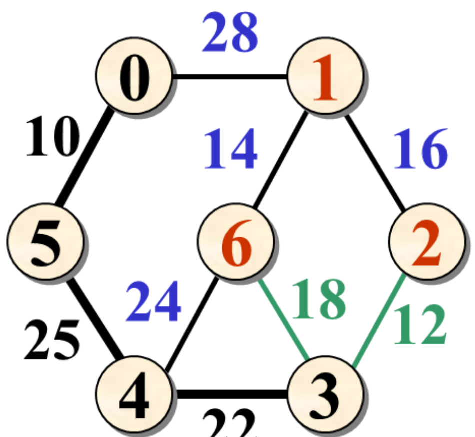


最小生成树--普里姆(Prim)算法

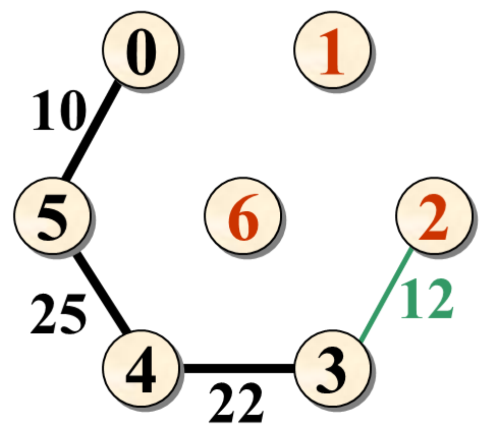
顶点v=3加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	28	12	22	25	10	18
nearvex	−1	0	3	−1	−1	−1	3

选 v=2　选边 (3,2)



原图



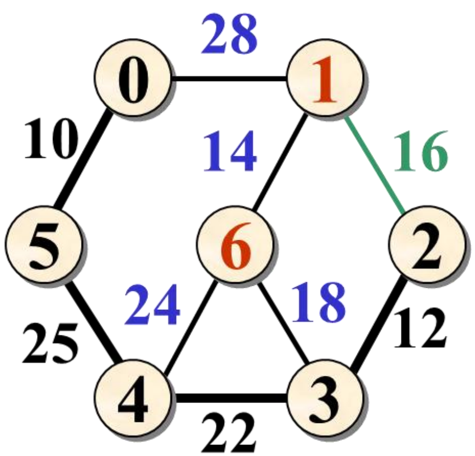
边 (4,3,22) 加入生成树

最小生成树--普里姆(Prim)算法

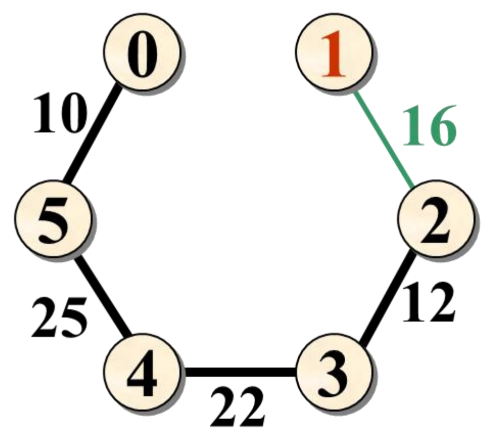
顶点v=2加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	18
nearvex	−1	2	−1	−1	−1	−1	3

选 v=1　　选边 (2,1)



原图



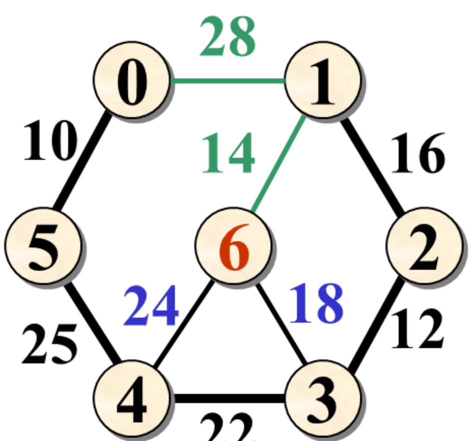
边 (3,2,12) 加入生成树

最小生成树--普里姆(Prim)算法

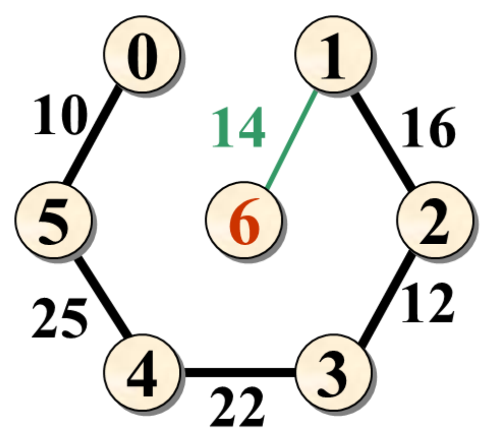
顶点v=1加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
nearvex	−1	−1	−1	−1	−1	−1	1

选 v=6 选边 (1,6)



原图

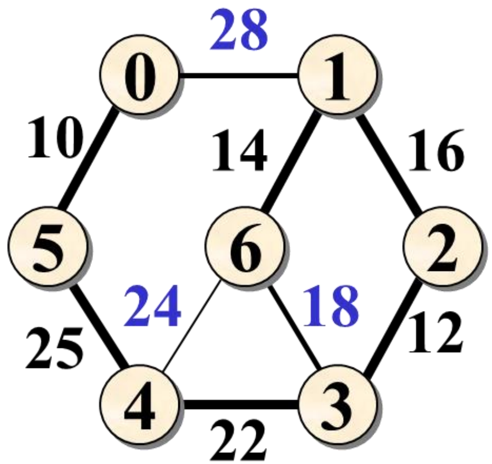


边 (2,1,16) 加入生成树

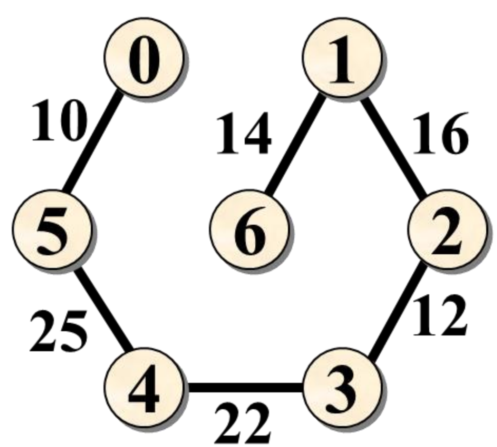
最小生成树--普里姆(Prim)算法

顶点v=6加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
nearvex	−1	−1	−1	−1	−1	−1	−1



原图



边 (1,6,14) 加入生成树

最小生成树--克鲁斯卡尔 (Kruskal) 算法

- 基本思想：设无向连通网为$G=(V, E)$，令G的最小生成树为$T=(U, TE)$，其初态为$U=V$，$TE=\{\ \}$
- 按照边的权值由小到大的顺序，考察G的边集E中的各条边。若被考察的边的两个顶点属于T的两个不同的连通分量，则将此边作为最小生成树的边加入到T中，同时把两个连通分量连接为一个连通分量
- 若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路
- 如此下去，当T中的连通分量个数为1时，此连通分量便为G的一棵最小生成树

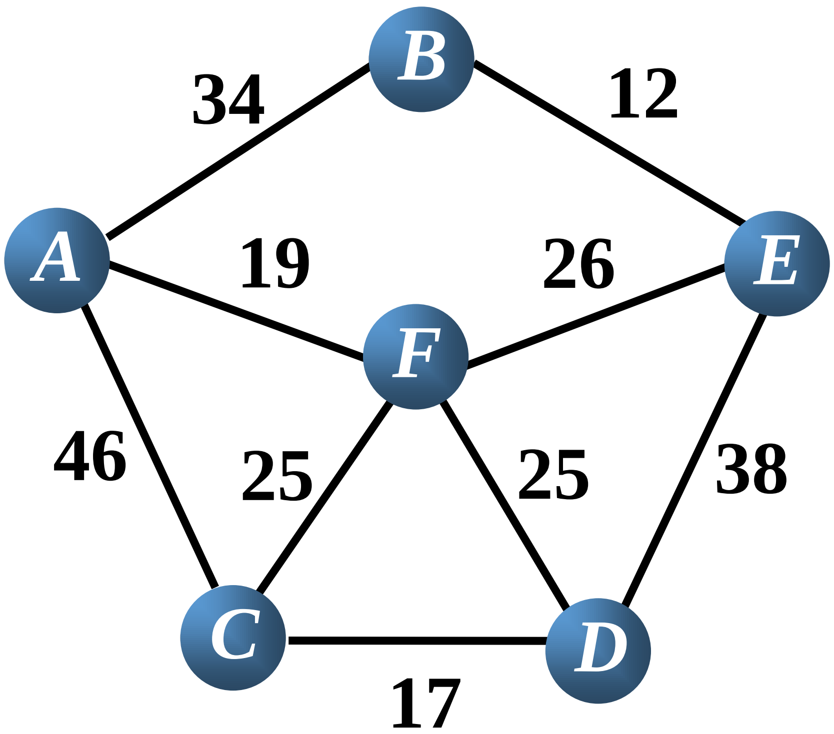
最小生成树--克鲁斯卡尔 (Kruskal) 算法

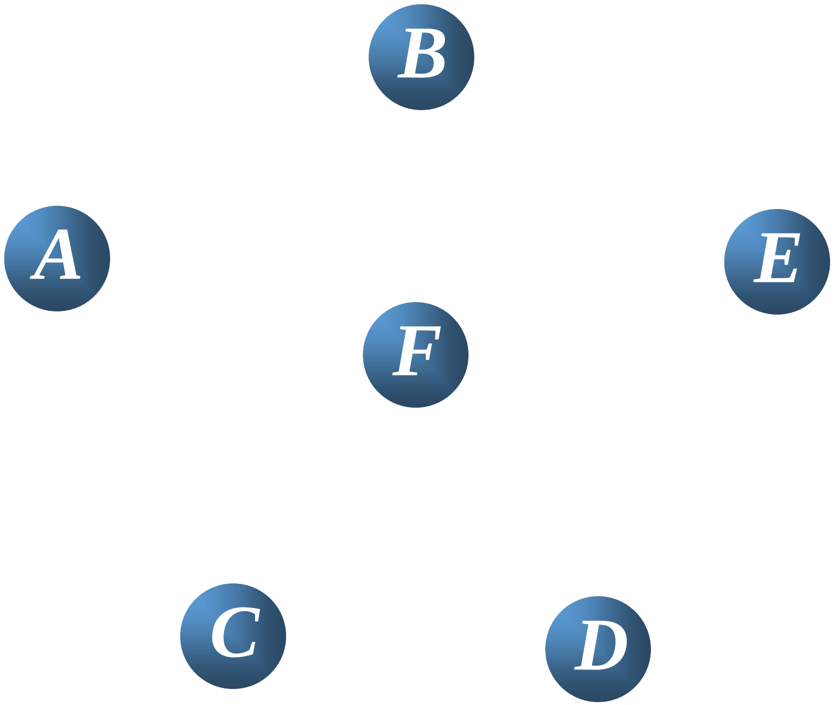
Kruskal算法的基本思想用伪代码描述如下：

1. 初始化：U=V；TE={ }；
2. 重复下述操作直到T中的连通分量个数为1：
   2.1 在E中寻找最短边(u，v)；
   2.2 如果顶点u、v位于T的两个不同连通分量，则
      2.2.1 将边(u，v)并入TE；
      2.2.2 将这两个连通分量合为一个；
   2.3 标记边(u，v)，使得(u，v)不参加后续最短边的选取；

关键：如何判别被考察边的两个顶点是否位于两个连通分量

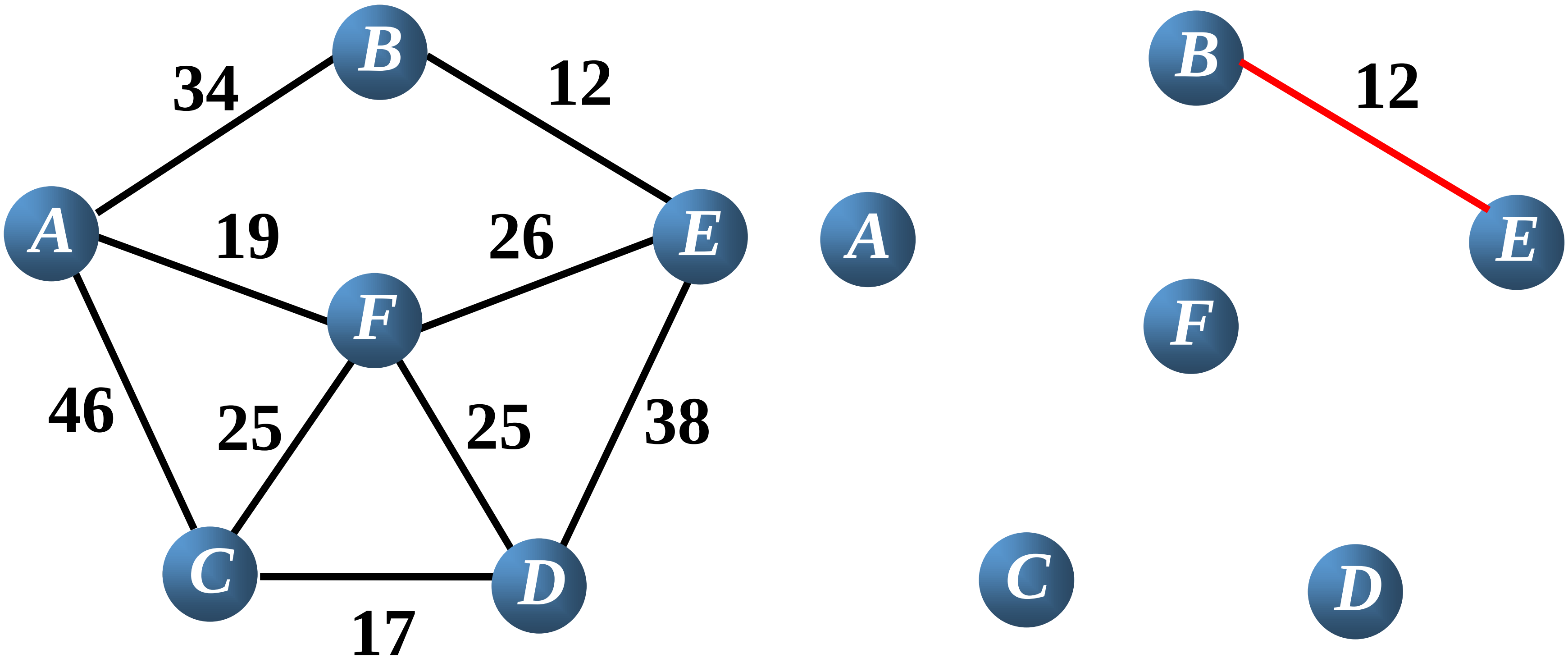
最小生成树--克鲁斯卡尔 (Kruskal) 算法





连通分量＝{A}, {B}, {C}, {D}, {E}, {F}

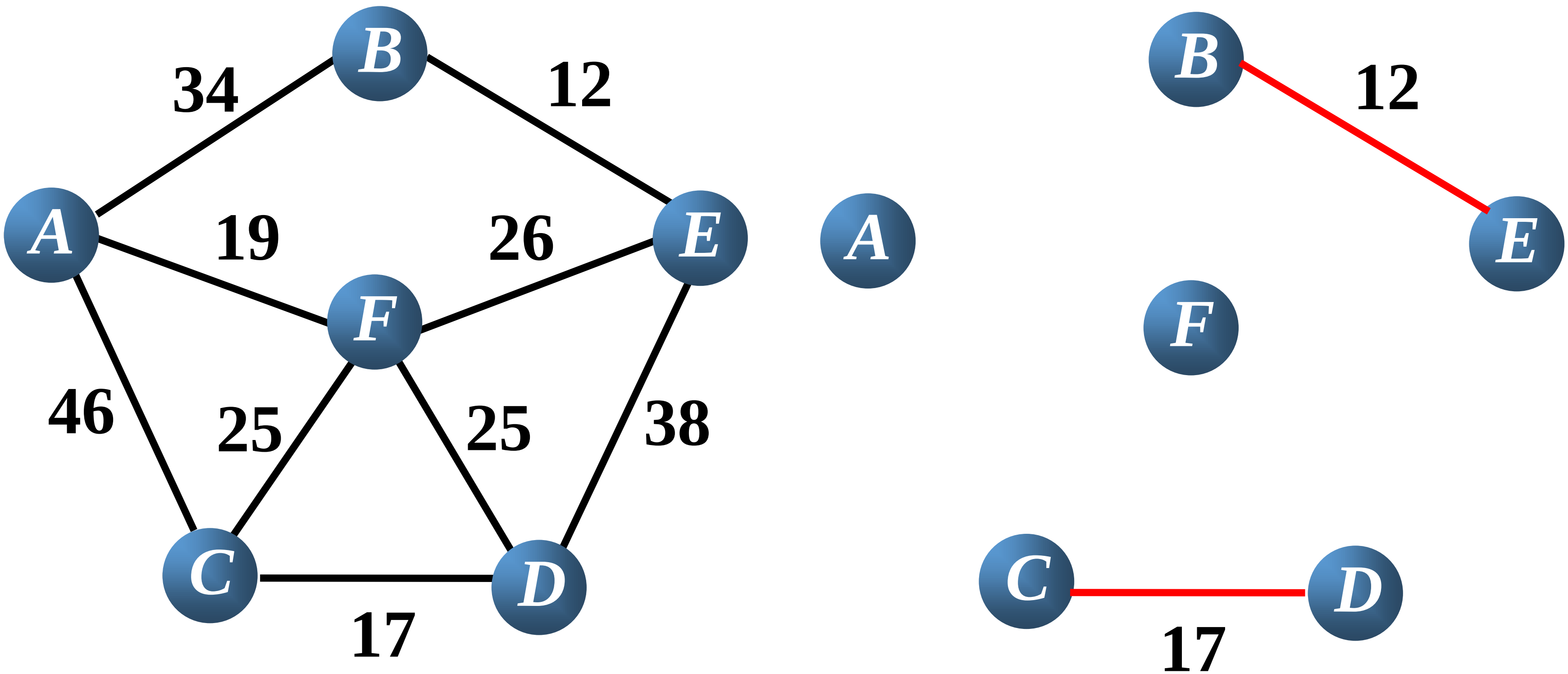
最小生成树--克鲁斯卡尔 (Kruskal) 算法



连通分量＝{A}, {B}, {C}, {D}, {E}, {F}

连通分量＝{A}, {B, E}, {C}, {D}, {F}

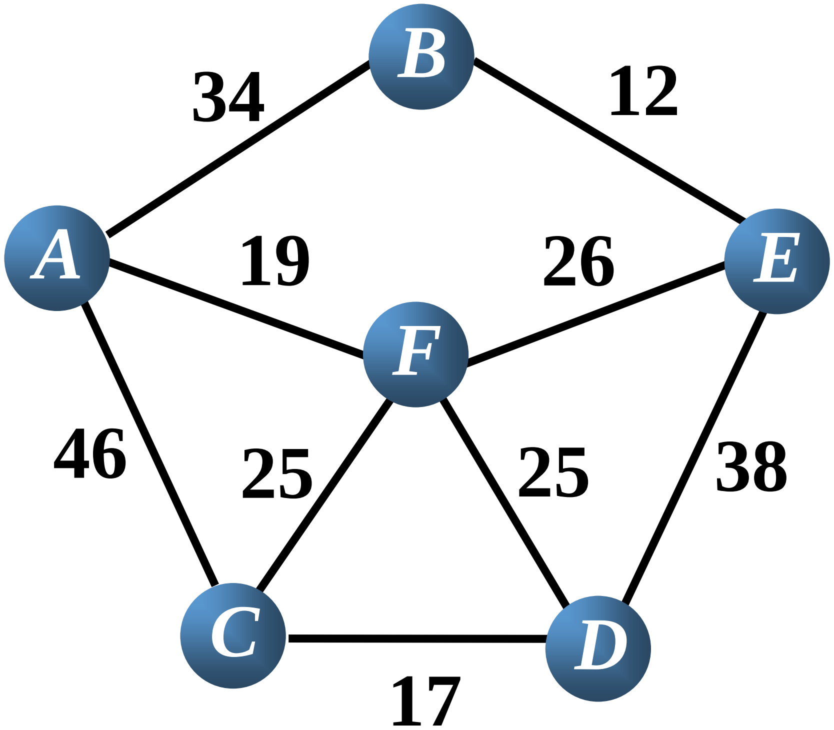
最小生成树--克鲁斯卡尔 (Kruskal) 算法

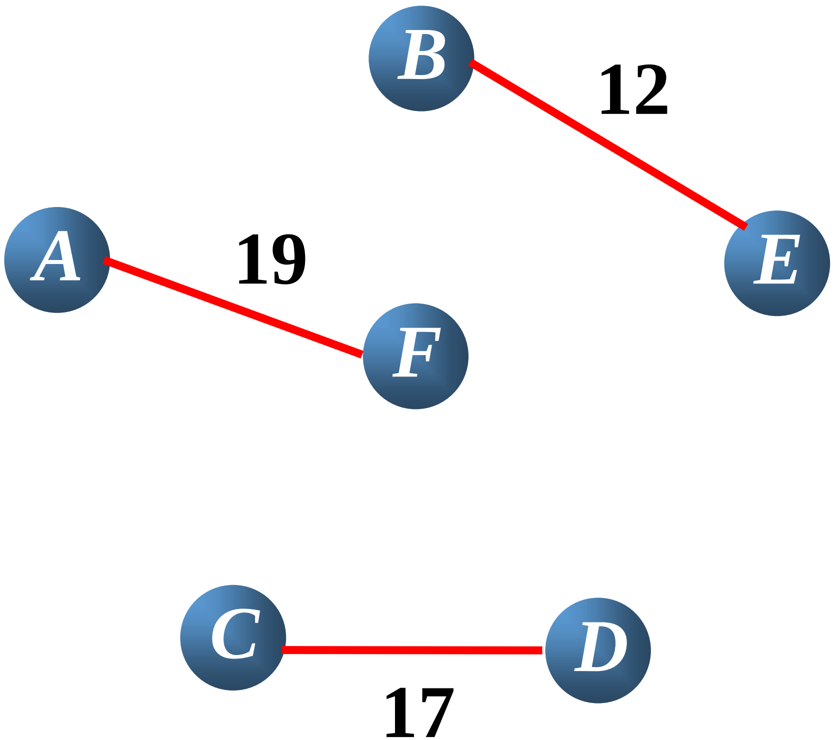


连通分量＝{A}, {F}, {B, E}, {C}, {D}

连通分量＝{A}, {F}, {B, E}, {C, D}

最小生成树--克鲁斯卡尔 (Kruskal) 算法

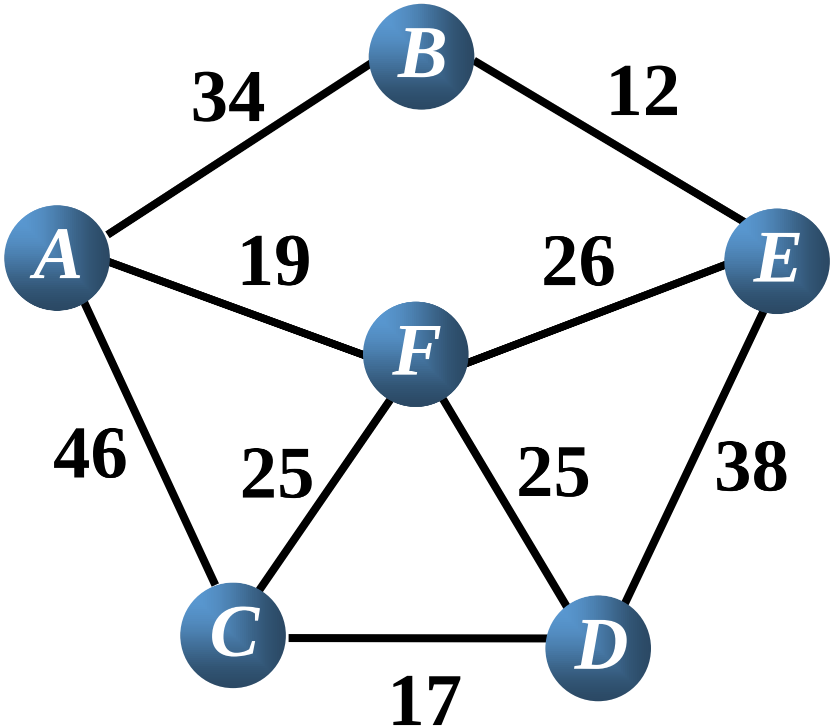


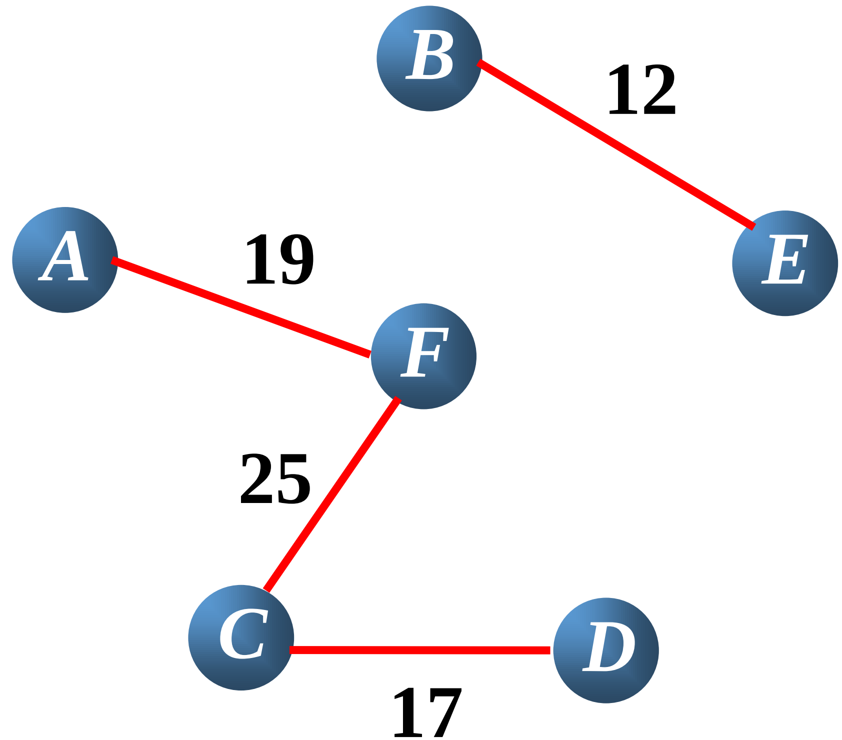


连通分量＝{A}, {B, E}, {C, D}, {F}

连通分量＝{A, F}, {B, E}, {C, D}

最小生成树--克鲁斯卡尔 (Kruskal) 算法

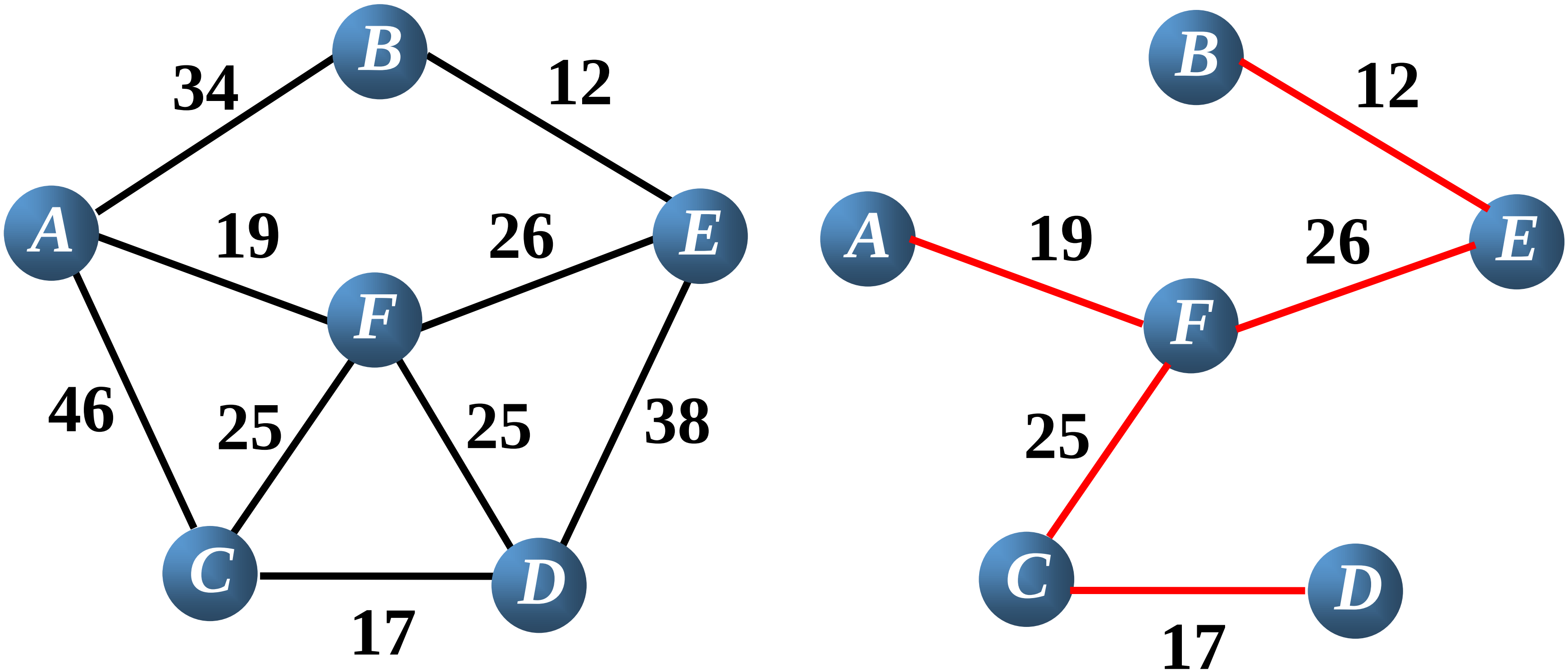




连通分量＝{A, F}, {B, E}, {C, D}

连通分量＝{A, F, C, D}, {B, E}

最小生成树--克鲁斯卡尔 (Kruskal) 算法



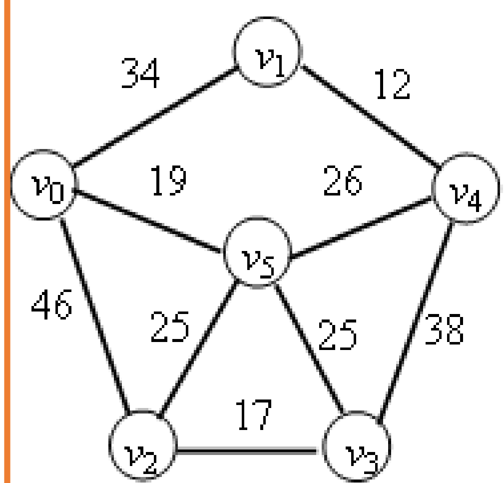
连通分量＝{A, F, C, D}, {B, E}

连通分量＝{A, F, C, D, B, E}

最小生成树--克鲁斯卡尔 (Kruskal) 算法

数据结构设计

1. 图的存储结构：因为Kruskal算法是依次对图中的边进行操作，因此考虑用边集数组存储图中的边，为了提高查找速度，将边集数组按边上的权排序。



(a) 无向网图

vertex[6] =

v_0	v_1	v_2	v_3	v_4	v_5

下标	0	1	2	3	4	5	6	7	8
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

(b) 边集数组存储

最小生成树--克鲁斯卡尔 (Kruskal) 算法

数据结构设计

2. 连通分量。Kruskal算法实质上是使生成树以一种随意的方式生长，初始时每个顶点构成一棵生成树，然后每生长一次就将两棵树合并，到最后合并成一棵树。

因此，可以设置一个数组parent[n]，元素parent[i]表示顶点i的双亲结点，初始时，parent[i]= -1，表示顶点i没有双亲，即该结点是所在生成树的根结点；对于边(u, v)，设vex1和vex2分别表示两个顶点所在树的根结点，如果vex1≠vex2，则顶点u和v必位于不同的连通分量，令parent[vex2] = vex1，实现将两棵树合并。

求某顶点v所在生成树的根结点只需沿数组v = parent[v]不断查找v的双亲，直到parent[v]等于-1。

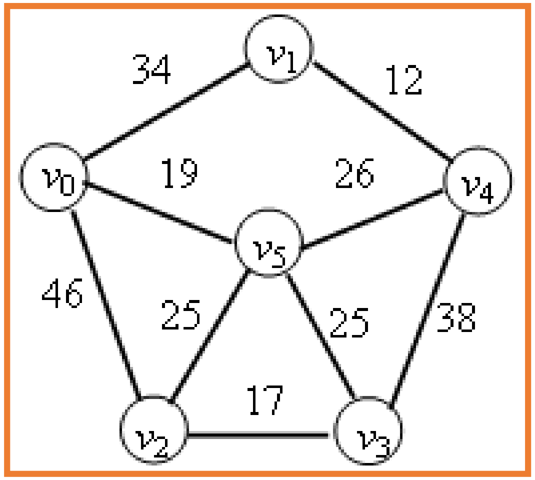
最小生成树--克鲁斯卡尔 (Kruskal) 算法

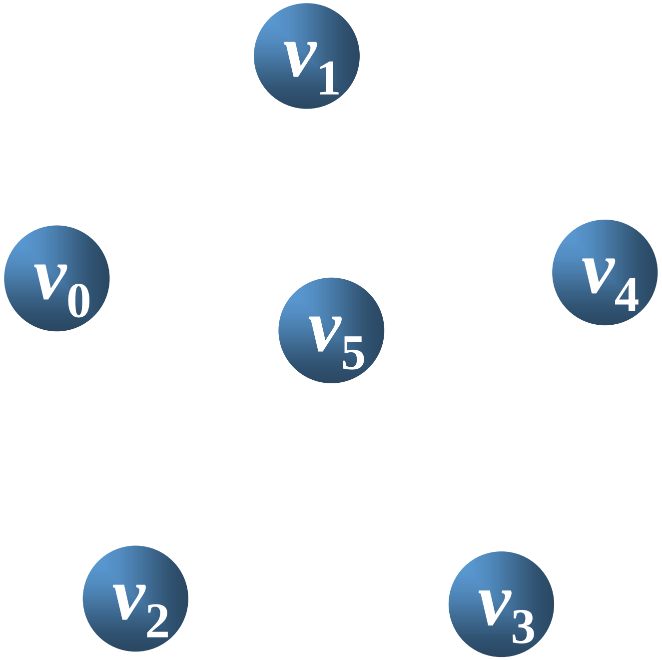
数据结构设计

Kruskal算法用伪代码进一步描述为：

```
1. 初始化辅助数组parent[n]; num = 0;
2. 依次考查每一条边for (i = 0; i < arcNum; i++)
    2.1 vex1 = edge[i].from所在生成树的根结点;
    2.2 vex2 = edge[i].to所在生成树的根结点;
    2.3 如果vex1 != vex2，执行下述操作:
         2.3.1 parent[vex2] = vex1;
         2.3.2 num++;
         2.3.3 if (num == n-1) 算法结束;
```

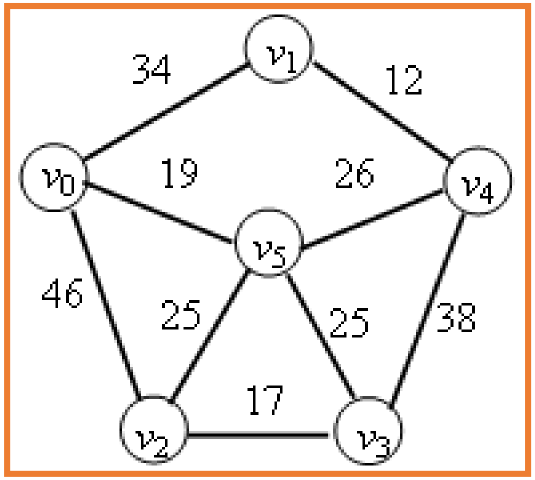
最小生成树--克鲁斯卡尔 (Kruskal) 算法

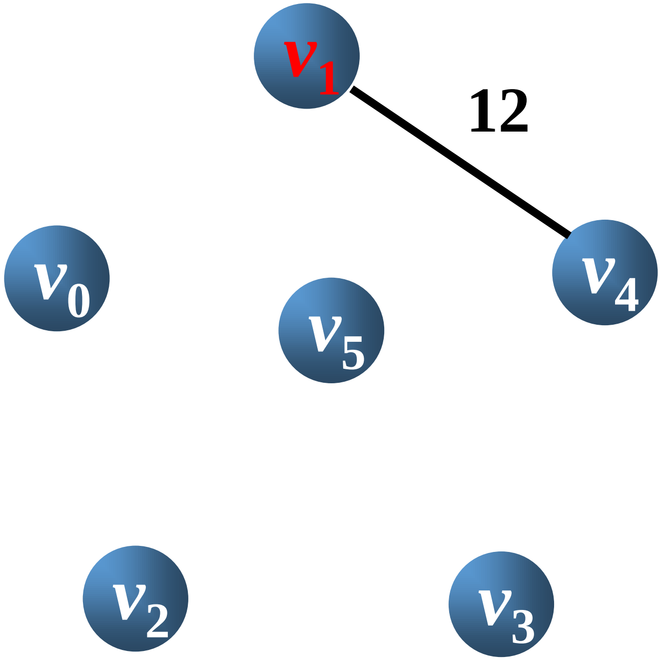




parent[0]=-1
parent[1]=-1
parent[2]=-1
parent[3]=-1
parent[4]=-1
parent[5]=-1

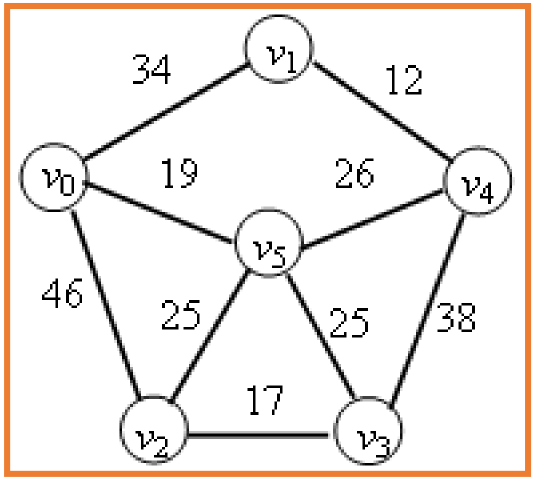
最小生成树--克鲁斯卡尔 (Kruskal) 算法

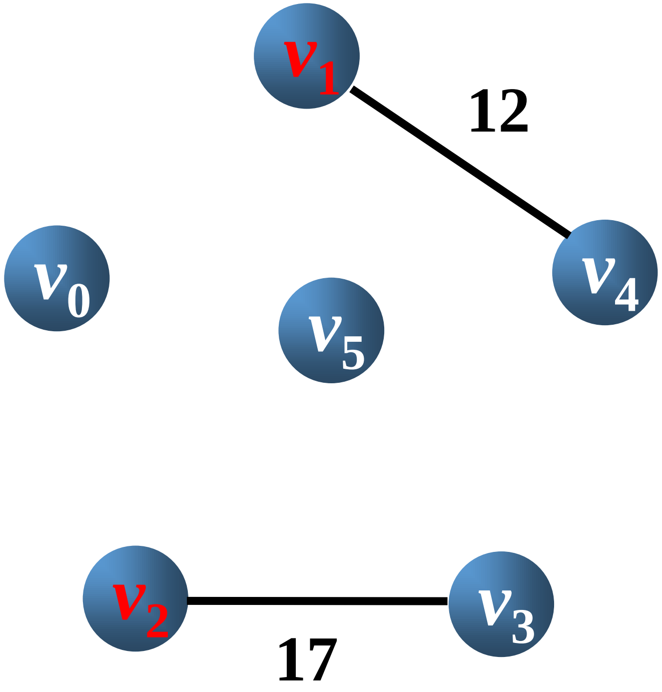




parent[0]=-1
parent[1]=-1
parent[2]=-1
parent[3]=-1
parent[4]=1
parent[5]=-1

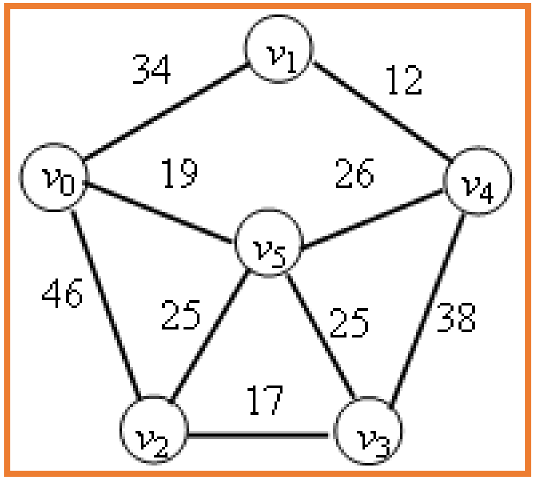
最小生成树--克鲁斯卡尔 (Kruskal) 算法

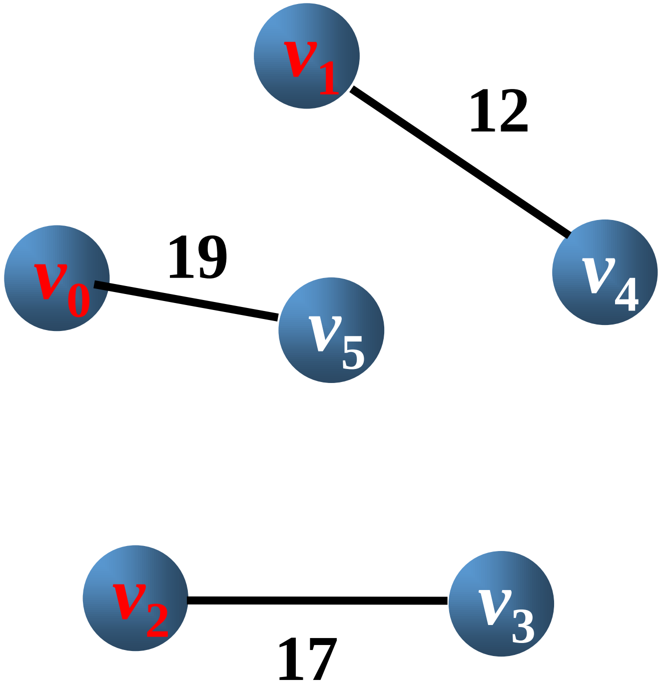




parent[0]=-1
parent[1]=-1
parent[2]=-1
parent[3]=2
parent[4]=1
parent[5]=-1

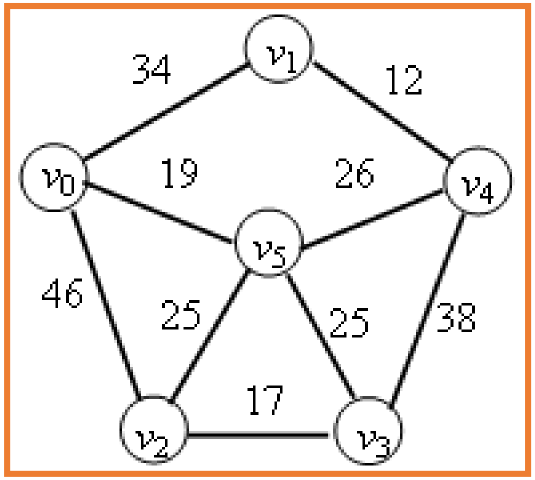
最小生成树--克鲁斯卡尔 (Kruskal) 算法



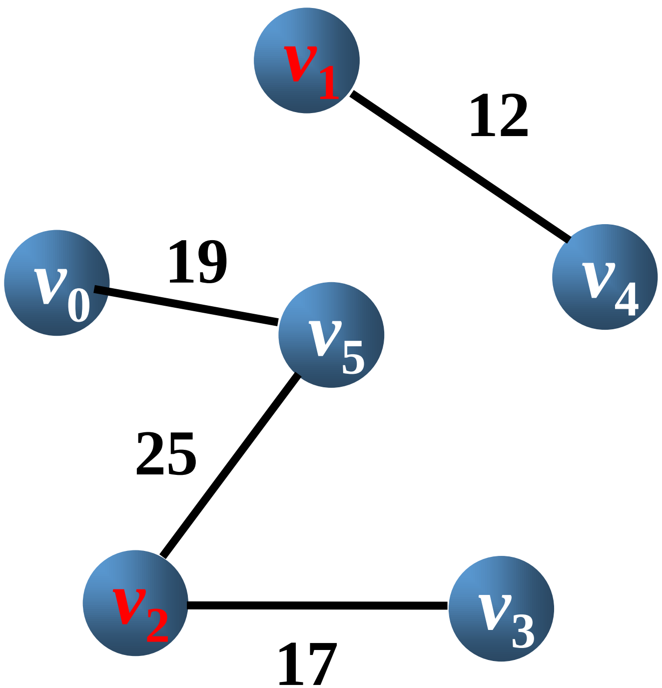


parent[0]=-1
parent[1]=-1
parent[2]=-1
parent[3]=2
parent[4]=1
parent[5]=0

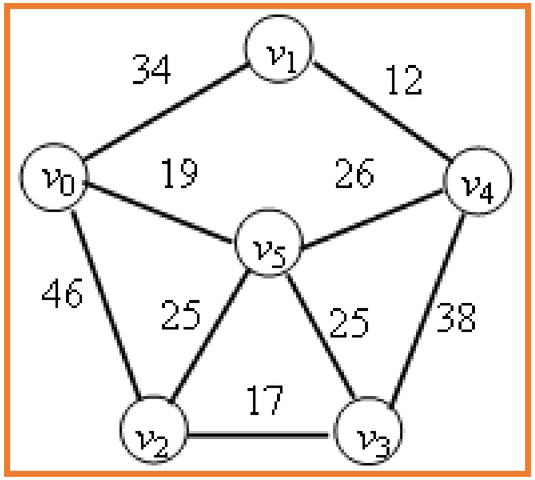
最小生成树--克鲁斯卡尔 (Kruskal) 算法

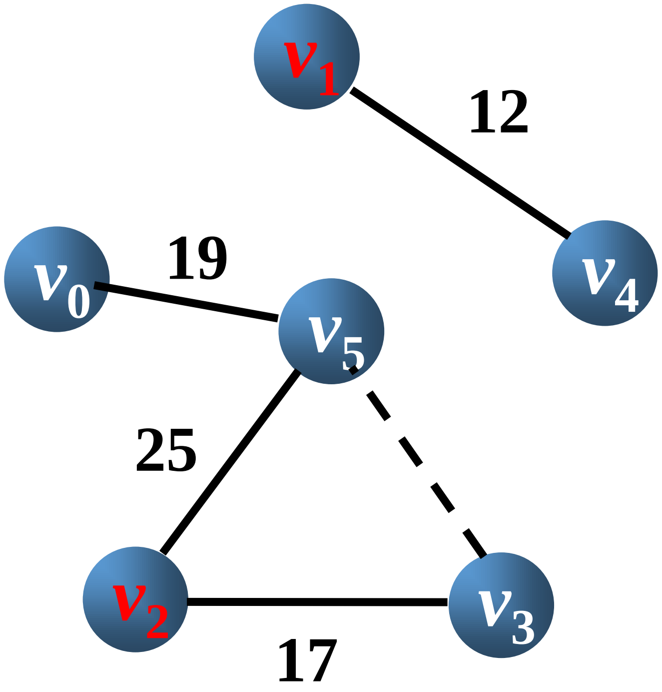


parent[0]=2
parent[1]=-1
parent[2]=-1
parent[3]=2
parent[4]=1
parent[5]=0



最小生成树--克鲁斯卡尔 (Kruskal) 算法

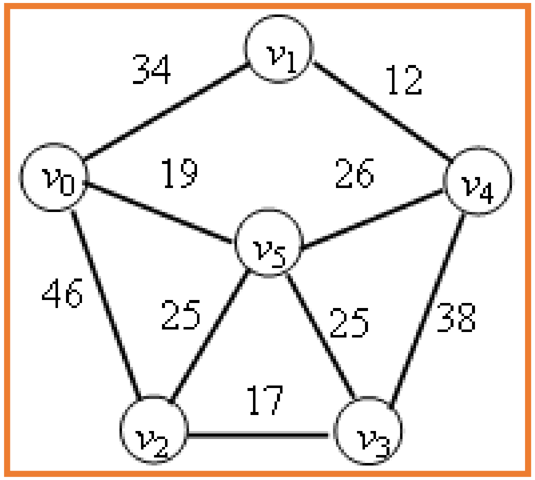




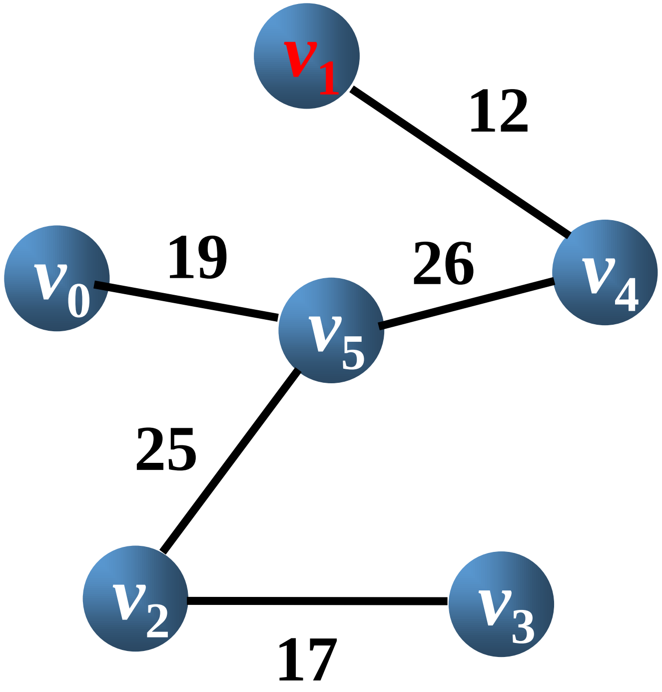
parent[0]=2
parent[1]=-1
parent[2]=-1
parent[3]=2
parent[4]=1
parent[5]=0

考查边(v_3, v_5)，由于parent[3]=2，parent[2]=-1，则v_3所在生成树的根结点是v_2，而parent[5]=0，parent[0]=2，则v_5所在生成树的根结点也是v_2，故舍去此边。

最小生成树--克鲁斯卡尔 (Kruskal) 算法



parent[0]=2
parent[1]=-1
parent[2]=1
parent[3]=2
parent[4]=1
parent[5]=0

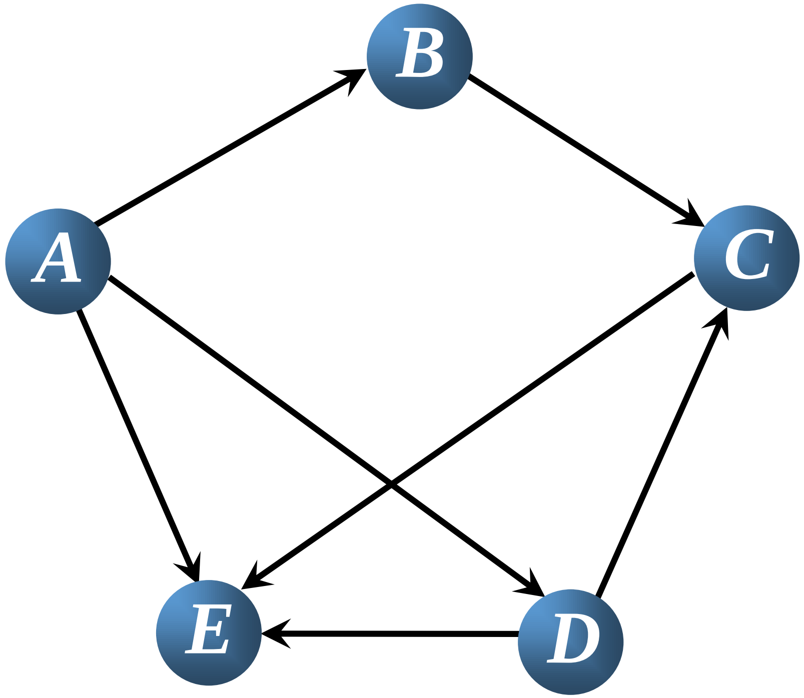


普里姆(Prim)算法 VS.克鲁斯卡尔 (Kruskal) 算法

- 分析普里姆算法, 设连通网络有 n 个顶点，则该算法的时间复杂度为 O(n2)，它适用于边稠密的网络，与边的数目无关。
- 克鲁斯卡尔算法主要针对边展开，边数少时效率会很高，所以对于边稀疏的图有优势。
- 注意：当各边有相同权值时，由于选择的随意性，产生的生成树可能不惟一。

最短路径

在非网图中，最短路径是指两顶点之间经历的边数最少的路径。



AE：1

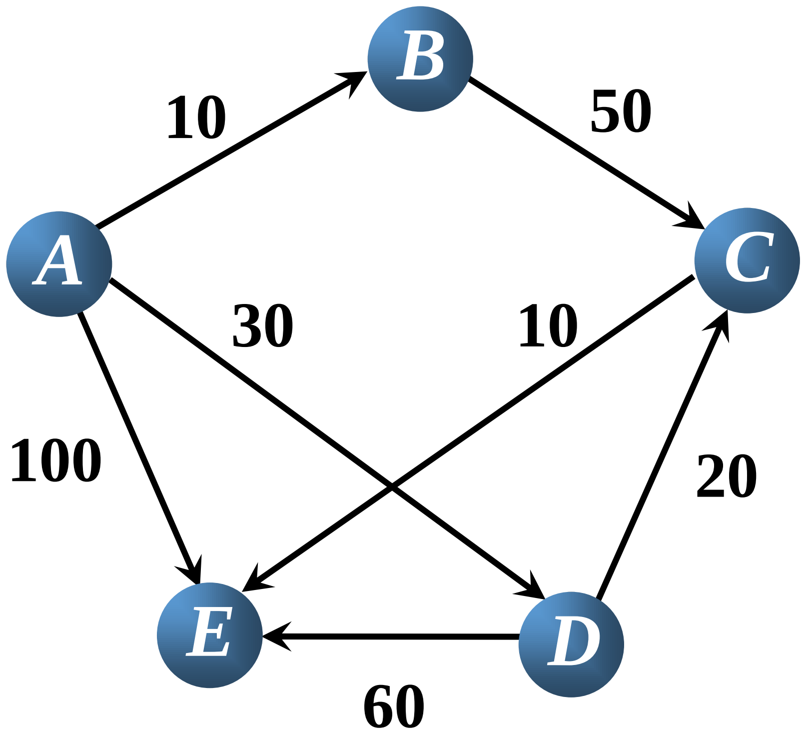
ADE：2

ADCE：3

ABCE：3

最短路径

在网图中，最短路径是指两顶点之间经历的边上权值之和最短的路径。



AE：100

ADE：90

ADCE：60

ABCE：70

单源点最短路径问题

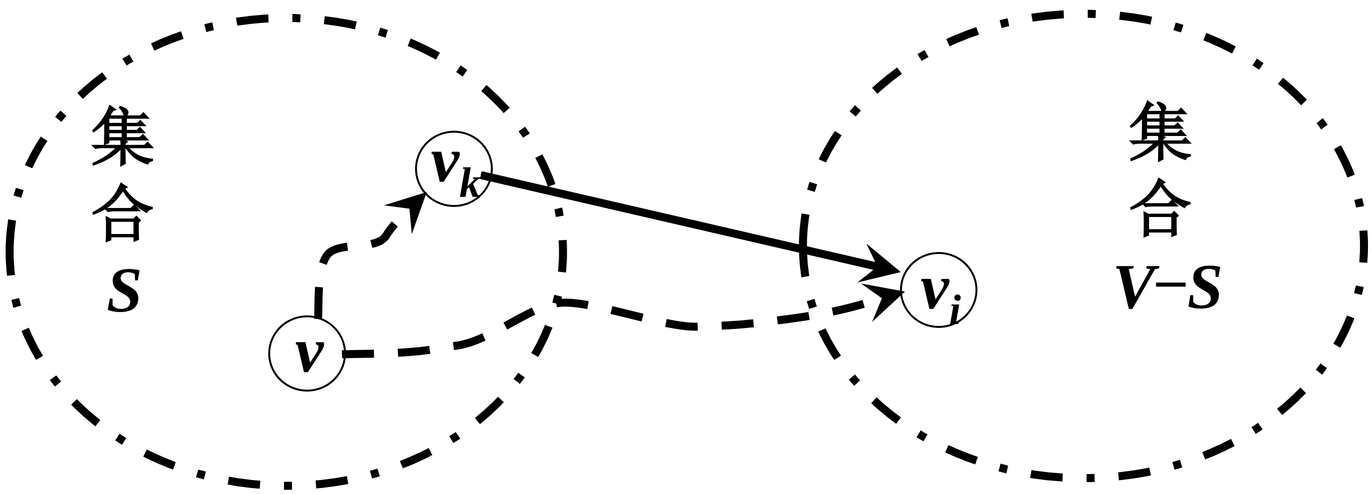
问题描述： 给定带权有向图$G=(V, E)$和源点$v \in V$，求从v到G中其余各顶点的最短路径。

应用实例——计算机网络传输的问题：怎样找到一种最经济的方式，从一台计算机向网上所有其它计算机发送一条消息。

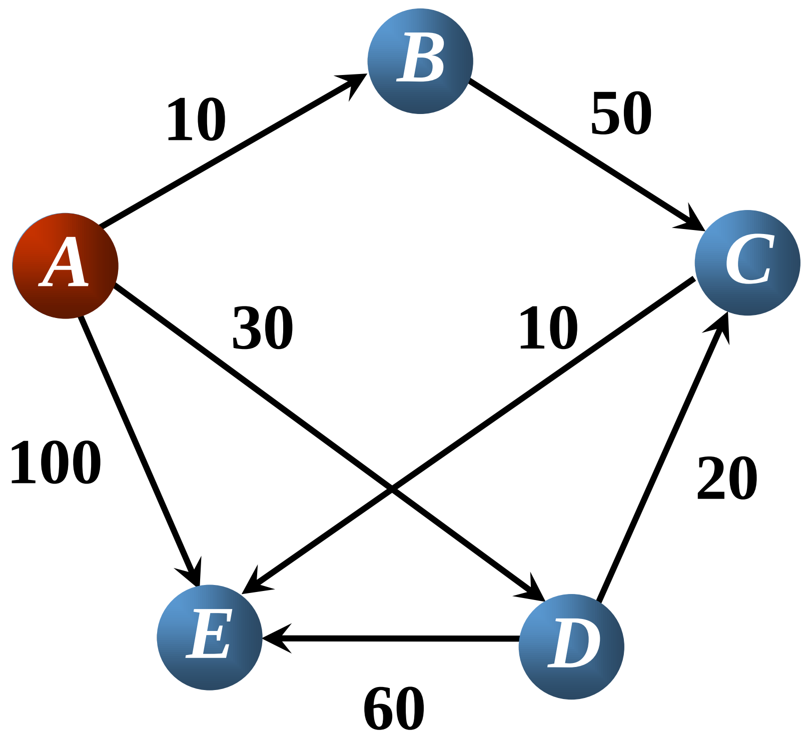
迪杰斯特拉（Dijkstra）提出了一个按路径长度递增的次序产生最短路径的算法——Dijkstra算法。

最短路径--Dijkstra算法

基本思想：设置一个集合S存放已经找到最短路径的顶点，S的初始状态只包含源点v，对$v_i \in V-S$，假设从源点v到v_i的有向边为最短路径。以后每求得一条最短路径$v, \ldots, v_k$，就将v_k加入集合S中，并将路径$v, \ldots, v_k, v_i$与原来的假设相比较，取路径长度较小者为最短路径。重复上述过程，直到集合V中全部顶点加入到集合S中。



最短路径--Dijkstra算法



S={A}

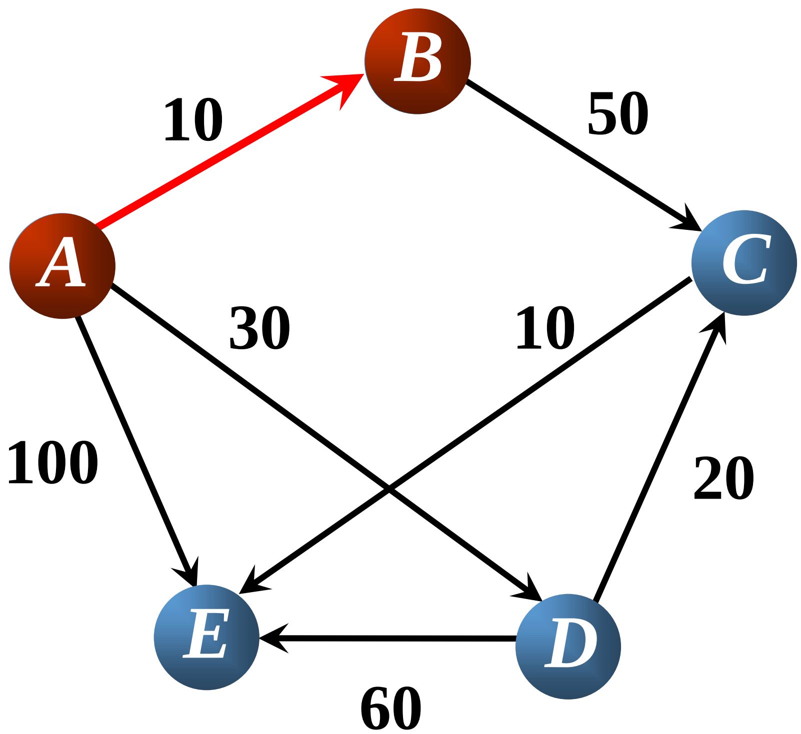
A→B:(A, B)10

A→C:(A, C)∞

A→D: (A, D)30

A→E: (A, E)100

最短路径--Dijkstra算法



S={A, B}

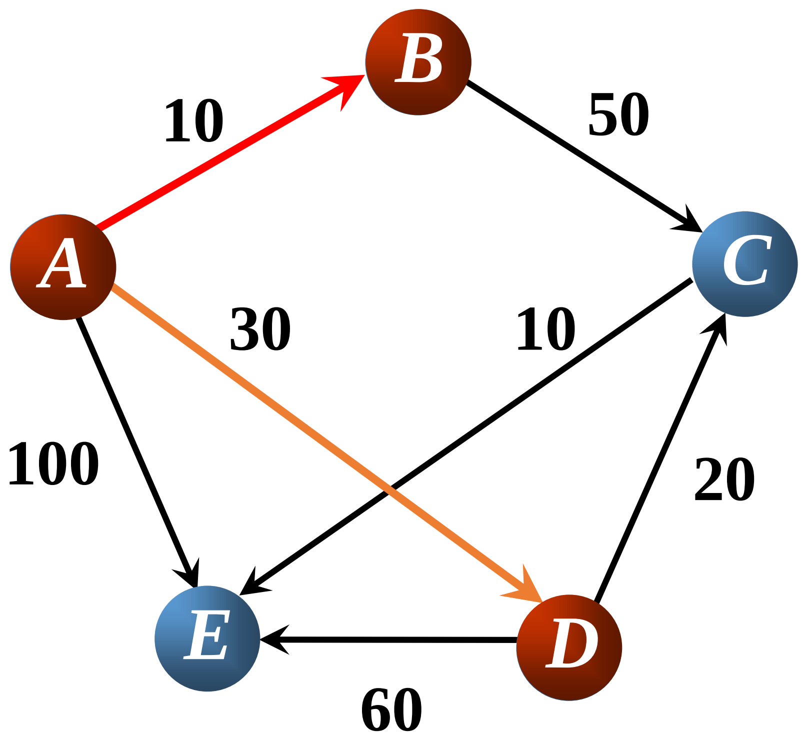
A→B:(A, B)10

A→C:(A, B, C)60

A→D: (A, D)30

A→E: (A, E)100

最短路径--Dijkstra算法



S={A, B, D}

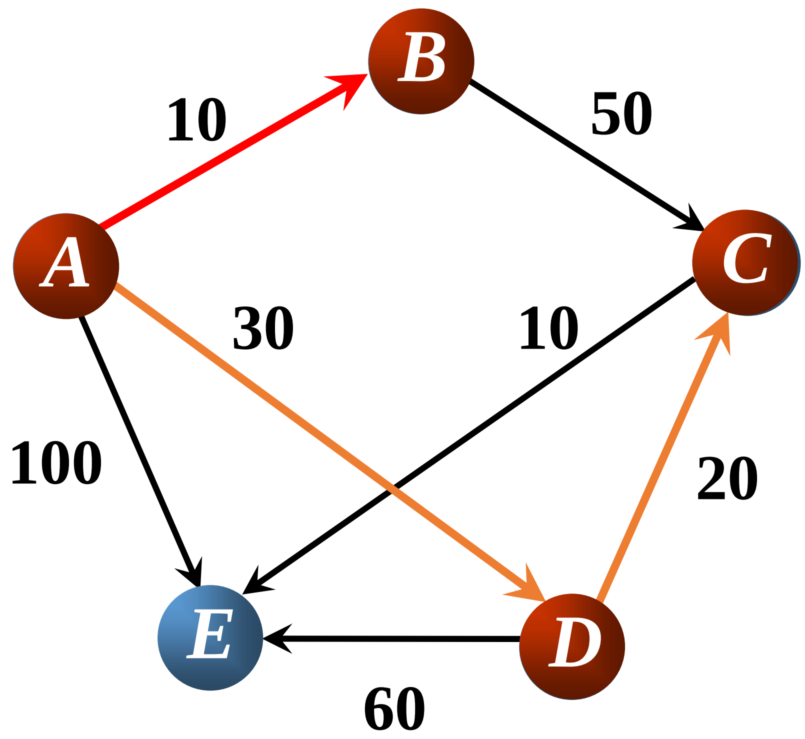
A→B:(A, B)10

A→C:(A, D, C)50

A→D: (A, D)30

A→E: (A, D, E)90

最短路径--Dijkstra算法



S={A, B, D, C}

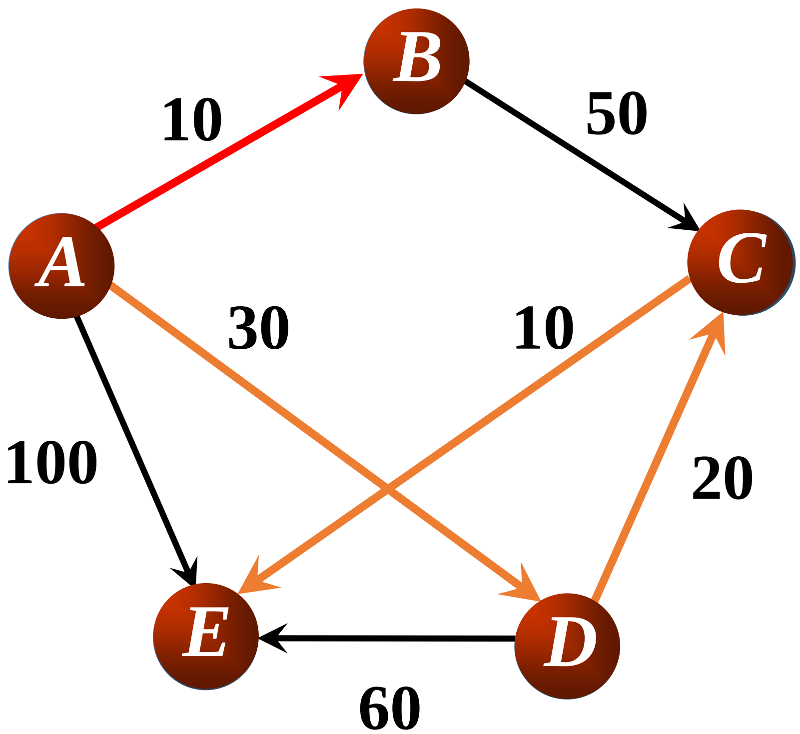
A→B:(A, B)10

A→C:(A, D, C)50

A→D: (A, D)30

A→E: (A, D, C, E)60

最短路径--Dijkstra算法



S={A, B, D, C, E}

A→B:(A, B)10

A→C:(A, D, C)50

A→D: (A, D)30

A→E: (A, D, C, E)60

最短路径--Dijkstra算法

- 图的存储结构：带权的邻接矩阵存储结构
- 数组dist[n]：每个分量dist[i]表示当前所找到的从始点v到终点v_i的最短路径的长度。初态为：若从v到v_i有弧，则dist[i]为弧上权值；否则置dist[i]为∞。
- 数组path[n]：path[i]是一个字符串，表示当前所找到的从始点v到终点v_i的最短路径。初态为：若从v到v_i有弧，则path[i]为vv_i；否则置path[i]空串。
- 数组s[n]：存放源点和已经生成的终点，其初态为只有一个源点v。

最短路径--Dijkstra算法

伪代码

1. 初始化数组dist、path和s；

2. while（s中的元素个数<n）

2.1 在dist[n]中求最小值，其下标为k；

2.2 输出dist[j]和path[j]；

2.3 修改数组dist和path；

2.4 将顶点v_k添加到数组s中；

每一对顶点之间的最短路径

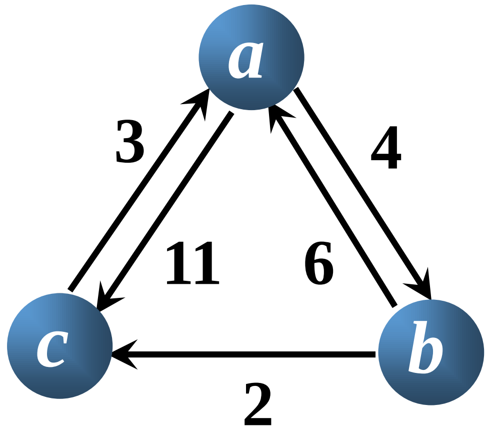
问题描述：给定带权有向图$G=(V, E)$，对任意顶点$v_i, v_j \in V$（$i \neq j$），求顶点v_i到顶点v_j的最短路径。

- 解决办法1：每次以一个顶点为源点调用Dijkstra算法。显然，时间复杂度为$O(n^3)$。
- 解决办法2：弗洛伊德提出的求每一对顶点之间的最短路径算法——Floyd算法，其时间复杂度也是$O(n^3)$，但形式上要简单些。

最短路径--Floyd算法

基本思想：对于从v_i到v_j的弧，进行n次试探：首先考虑路径v_i, v_0, v_j是否存在，如果存在，则比较v_i, v_j和v_i, v_0, v_j的路径长度，取较短者为从v_i到v_j的中间顶点的序号不大于0的最短路径。在路径上再增加一个顶点v_1，依此类推，在经过n次比较后，最后求得的必是从顶点v_i到顶点v_j的最短路径。

最短路径--Floyd算法

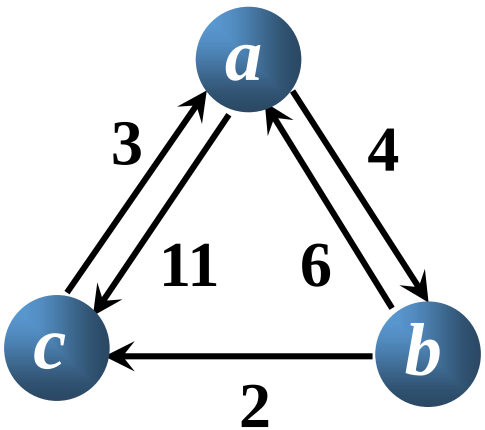


有向网图

$$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

邻接矩阵

最短路径--Floyd算法

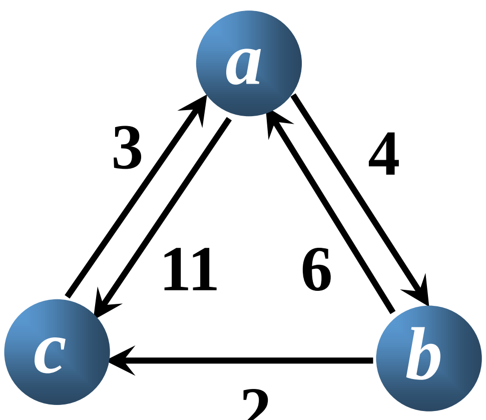


初始化

$$dist_{-1} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

$$path_{-1} = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & & \end{pmatrix}$$

最短路径--Floyd算法

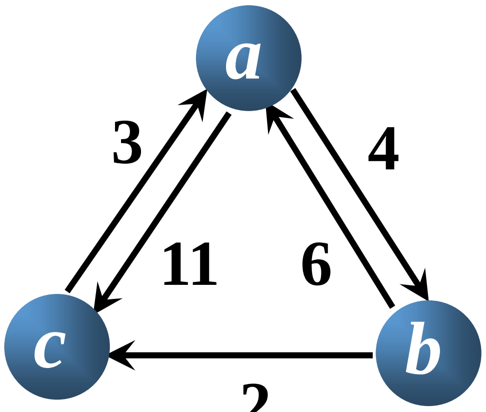


第1次迭代

$$dist_{-1} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix} \quad dist_0 = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \color{red}{7} & 0 \end{pmatrix}$$

$$path_{-1} = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & & \end{pmatrix} \quad path_0 = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & \color{red}{cab} & \end{pmatrix}$$

最短路径--Floyd算法

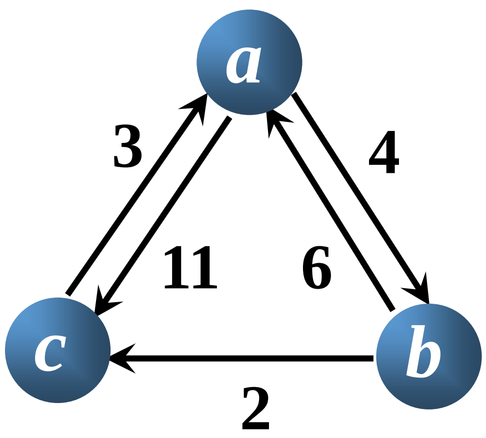


第2次迭代

$$\mathrm{dist}_0 = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad \mathrm{dist}_1 = \begin{pmatrix} 0 & 4 & \color{red}{6} \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$$\mathrm{path}_0 = \begin{pmatrix} & \mathrm{ab} & \mathrm{ac} \\ \mathrm{ba} & & \mathrm{bc} \\ \mathrm{ca} & \mathrm{cab} & \end{pmatrix} \quad \mathrm{path}_1 = \begin{pmatrix} & \mathrm{ab} & \color{red}{\mathrm{abc}} \\ \mathrm{ba} & & \mathrm{bc} \\ \mathrm{ca} & \mathrm{cab} & \end{pmatrix}$$

最短路径--Floyd算法



第3次迭代

$$dist_1 = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \qquad dist_2 = \begin{pmatrix} 0 & 4 & 6 \\ \color{red}{5} & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$$path_1 = \begin{pmatrix} & ab & abc \\ ba & & bc \\ ca & cab & \end{pmatrix} \qquad path_2 = \begin{pmatrix} & ab & abc \\ \color{red}{bca} & & bc \\ ca & cab & \end{pmatrix}$$

数据结构

图的存储结构：带权的邻接矩阵存储结构

数组dist[n][n]：存放在迭代过程中求得的最短路径长度。迭代公式：

$$\begin{cases} dist_{-1}[i][j]=arc[i][j] \\ dist_{k}[i][j]=\min\{dist_{k-1}[i][j],\ dist_{k-1}[i][k]+dist_{k-1}[k][j]\} \\ \qquad 0\le k \le n-1 \end{cases}$$

数组path[n][n]：存放从v_i到v_j的最短路径，初始为path[i][j]="v_iv_j"。

最短路径--Floyd算法

C++描述

```
void Floyd(MGraph G)
{
    for (i=0; i<G.vertexNum; i++)
      for (j=0; j<G.vertexNum; j++)
      {
         dist[i][j]=G.arc[i][j];
         if (dist[i][j]!=∞)
             path[i][j]=G.vertex[i]+G.vertex[j];
         else path[i][j]="";
      }
    for (k=0; k<G.vertexNum; k++)
       for (i=0; i<G.vertexNum; i++)
         for (j=0; j<G.vertexNum; j++)
             if (dist[i][k]+dist[k][j]<dist[i][j]) {
                 dist[i][j]=dist[i][k]+dist[k][j];
                 path[i][j]=path[i][k]+path[k][j];
           }
}
```