第7章 图2

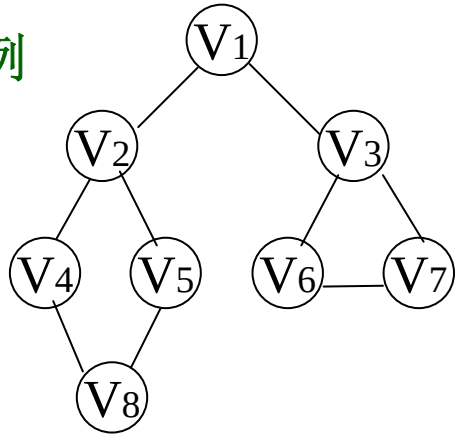
- 7.6 图的遍历
- 7.7 最短路径
- 7.9 最小支撑树
- 7.10图匹配

7.6图的遍历

深度优先遍历(DFS)

- 方法：从图的某一顶点V_0出发，访问此顶点；然后依次从V_0的未被访问的邻接点出发，深度优先遍历图，直至图中所有和V_0相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未被访问的顶点作起点，重复上述过程，直至图中所有顶点都被访问为止（问题：什么时候会出现这种情况？）

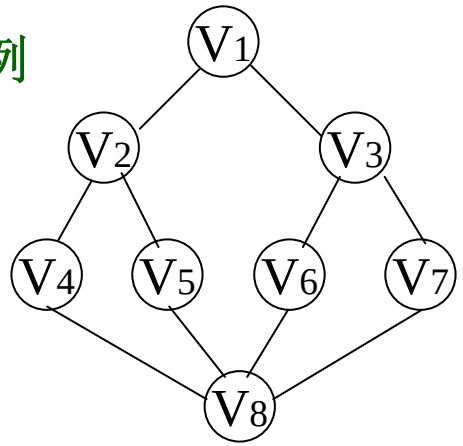
例



类似于：树的前序遍历！

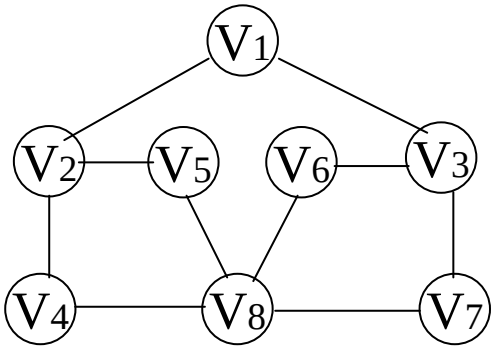
深度遍历：**V1⇒ V2 ⇒V4 ⇒ V8 ⇒V5 ⇒V3 ⇒V6 ⇒V7**

例



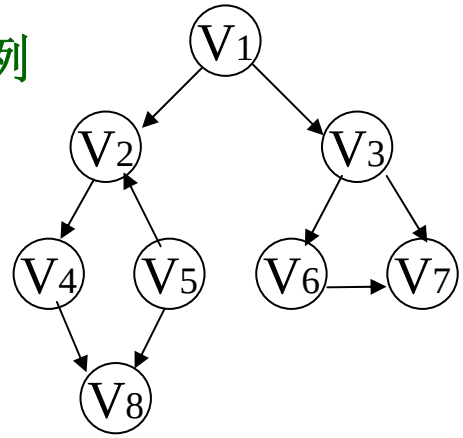
深度遍历：**V1⇒ V2 ⇒V4 ⇒ V8 ⇒V5 ⇒V6 ⇒V3 ⇒V7**

例



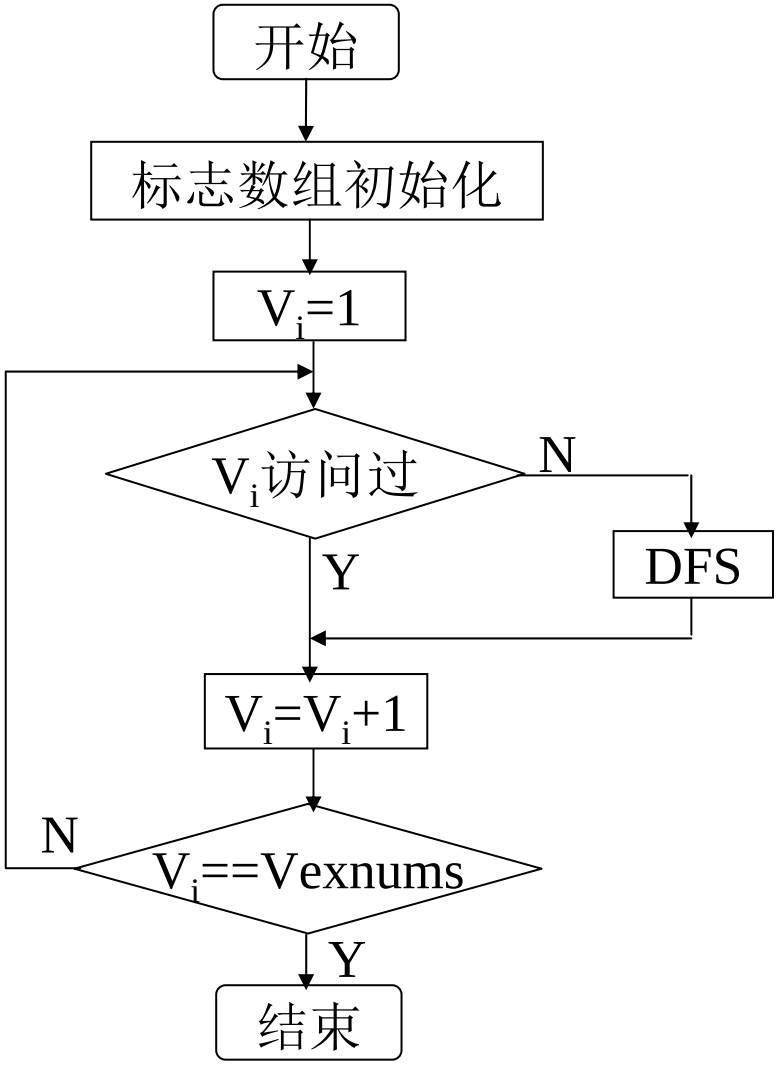
深度遍历：**V1⇒ V2 ⇒V4 ⇒ V8 ⇒V5 ⇒V6 ⇒V3 ⇒V7**

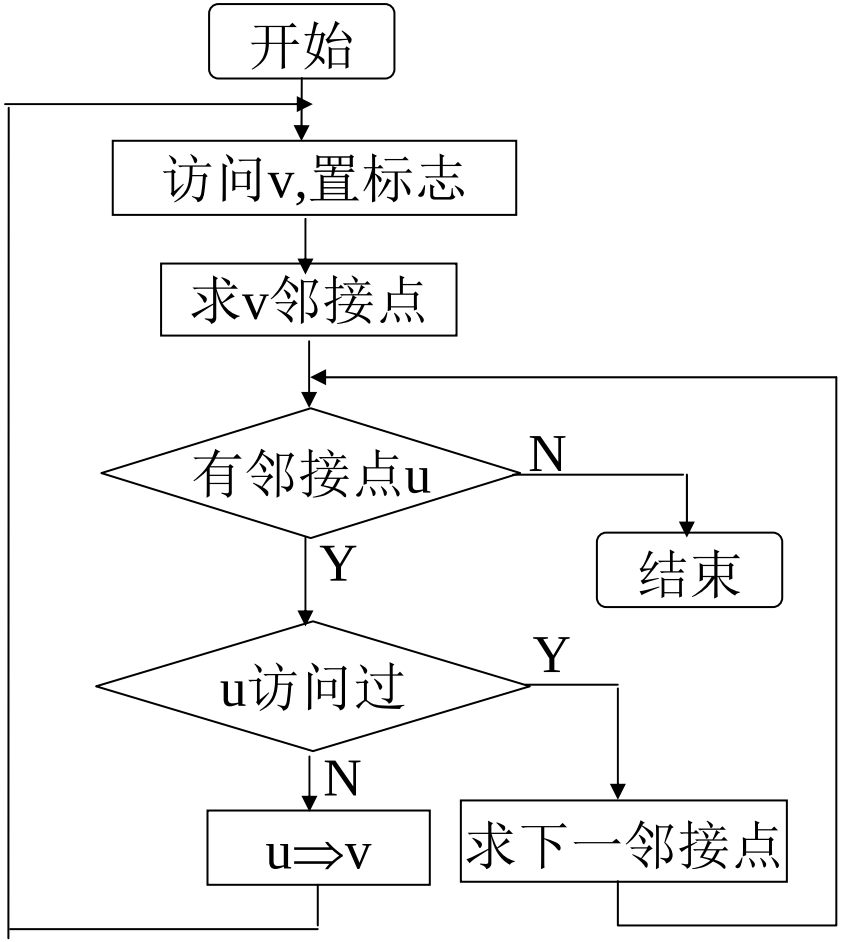
例



深度遍历：**V1⇒ V2 ⇒V4 ⇒ V8 ⇒V3 ⇒V6 ⇒V7 ⇒V5**

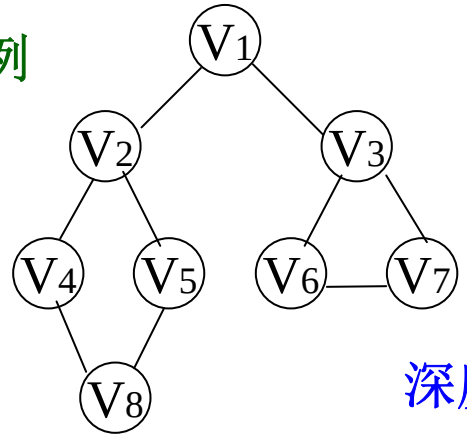
- 深度优先遍历算法
  - 递归算法



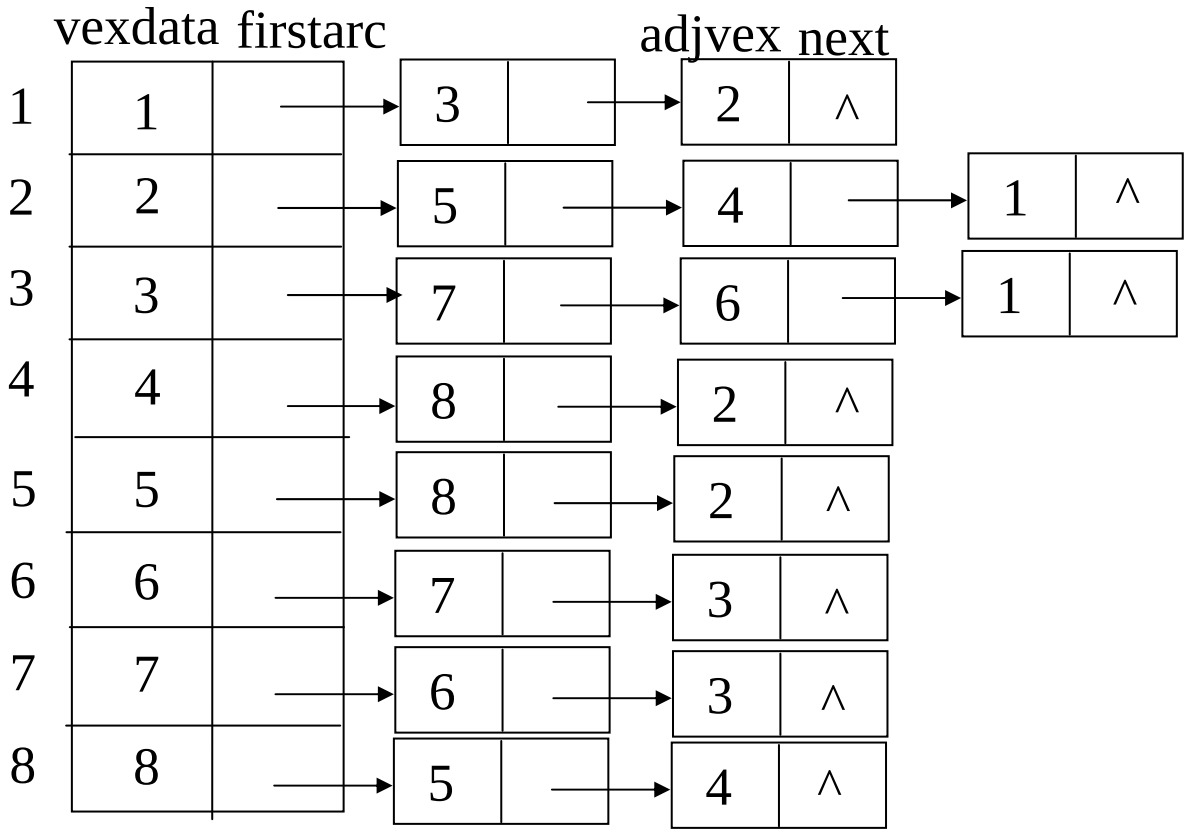


DFS

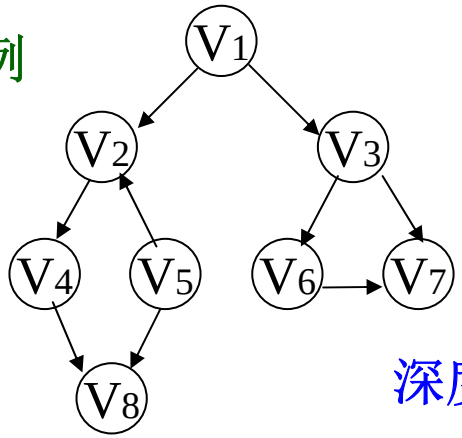
例



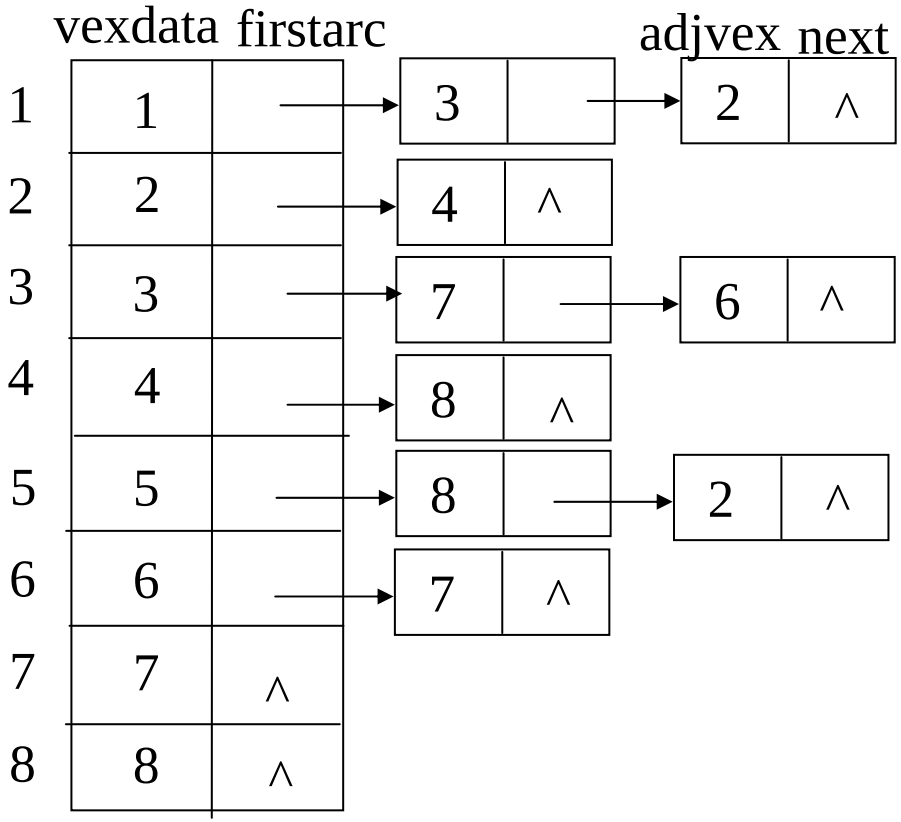
深度遍历：**V1⇒V3 ⇒V7 ⇒V6 ⇒V2 ⇒V5 ⇒V8 ⇒V4**



例



深度遍历：**V1⇒V3 ⇒V7 ⇒V6 ⇒V2 ⇒V4 ⇒V8 ⇒V5**

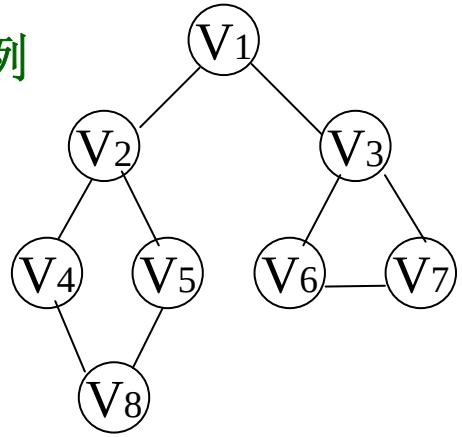


7.6 图的遍历

广度优先搜索(BFS)

- 方法：从图的某一顶点V_0出发，访问此顶点后，依次访问V_0的各个未曾访问过的邻接顶点；然后分别从这些邻接顶点出发，广度优先遍历图，直至图中所有已被访问的顶点的邻接点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未被访问的顶点作起点，重复上述过程，直至图中所有顶点都被访问为止。

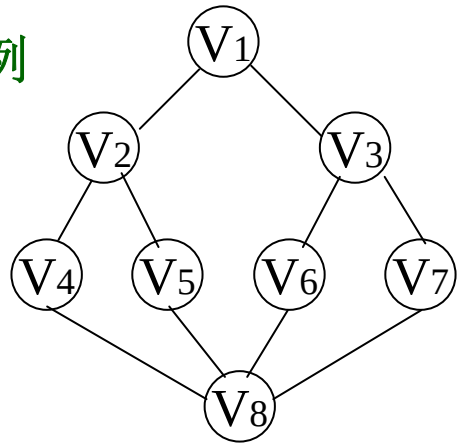
例



类似于：树的层次遍历！

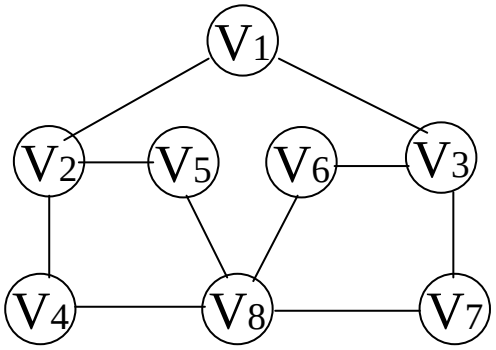
广度遍历：**V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8**

例



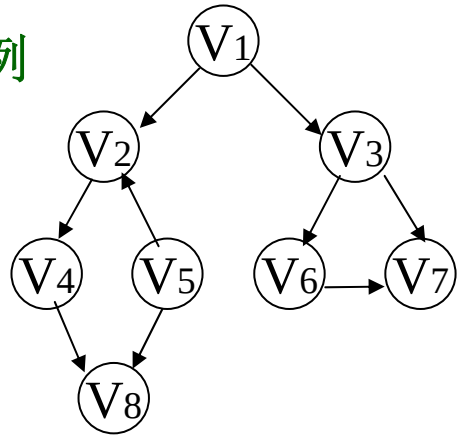
广度遍历：**V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8**

例



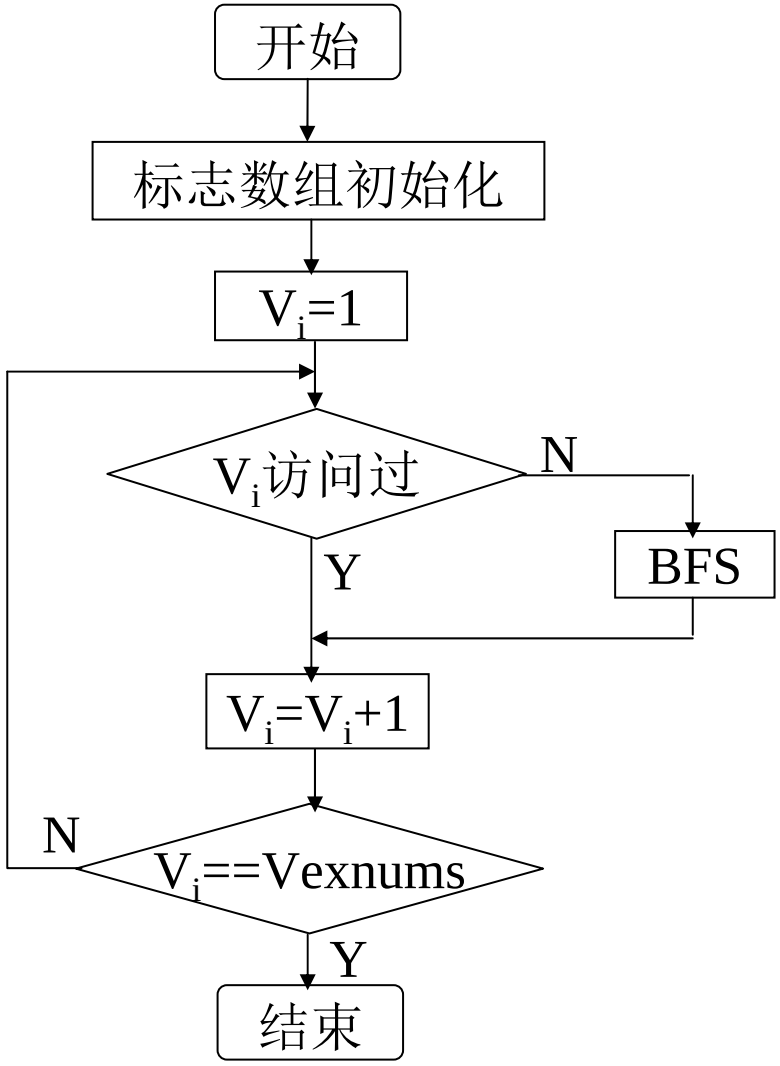
广度遍历：**V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8**

例



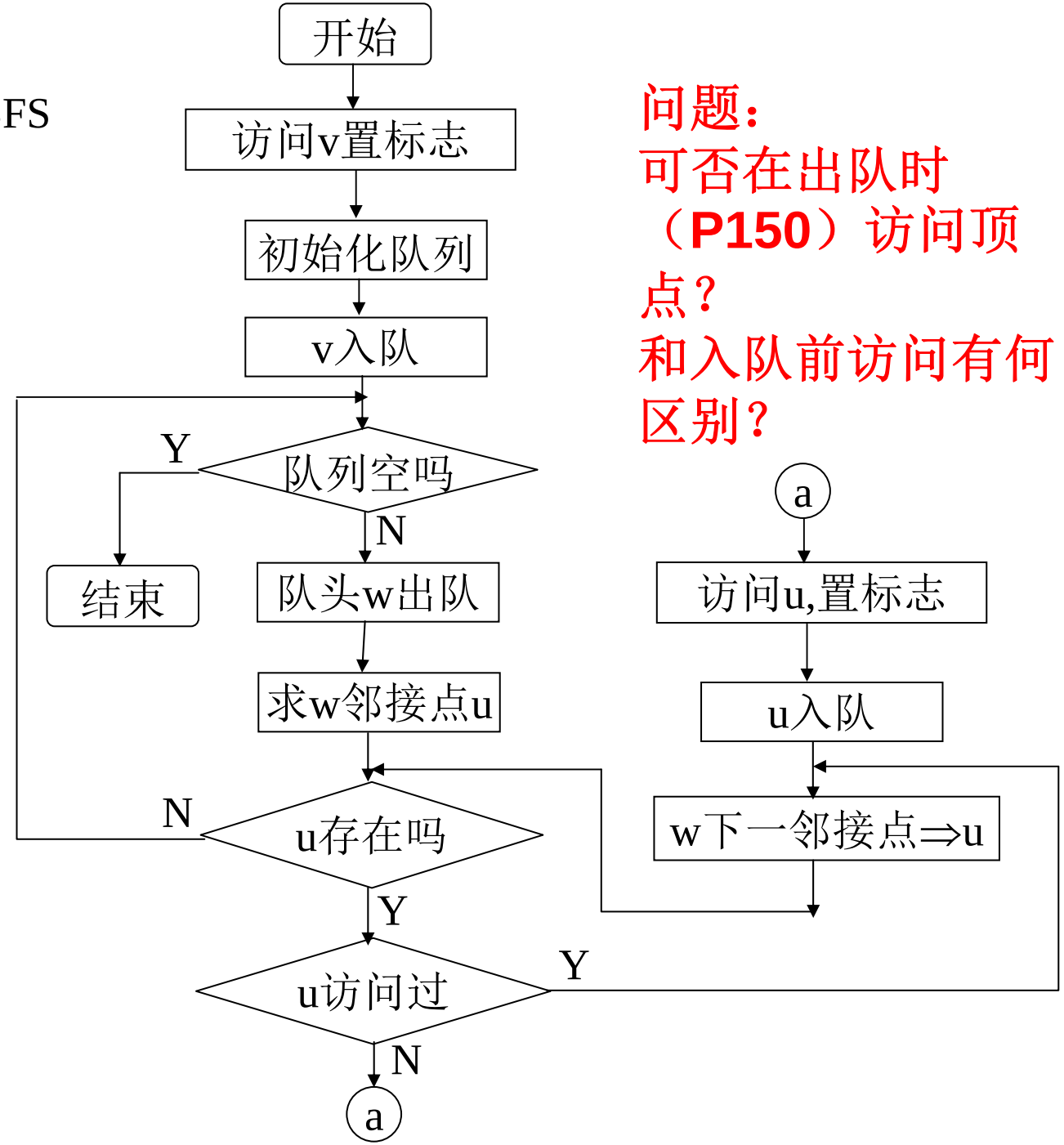
广度遍历：**V1⇒ V2 ⇒V3 ⇒ V4 ⇒V6 ⇒V7 ⇒V8 ⇒V5**

- 广度优先遍历算法



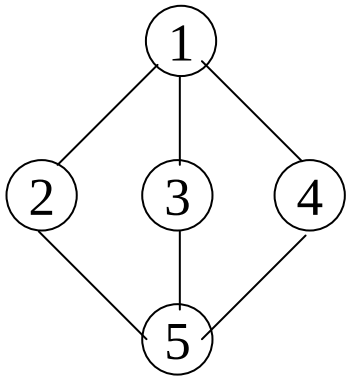
- 广度优先遍历算法

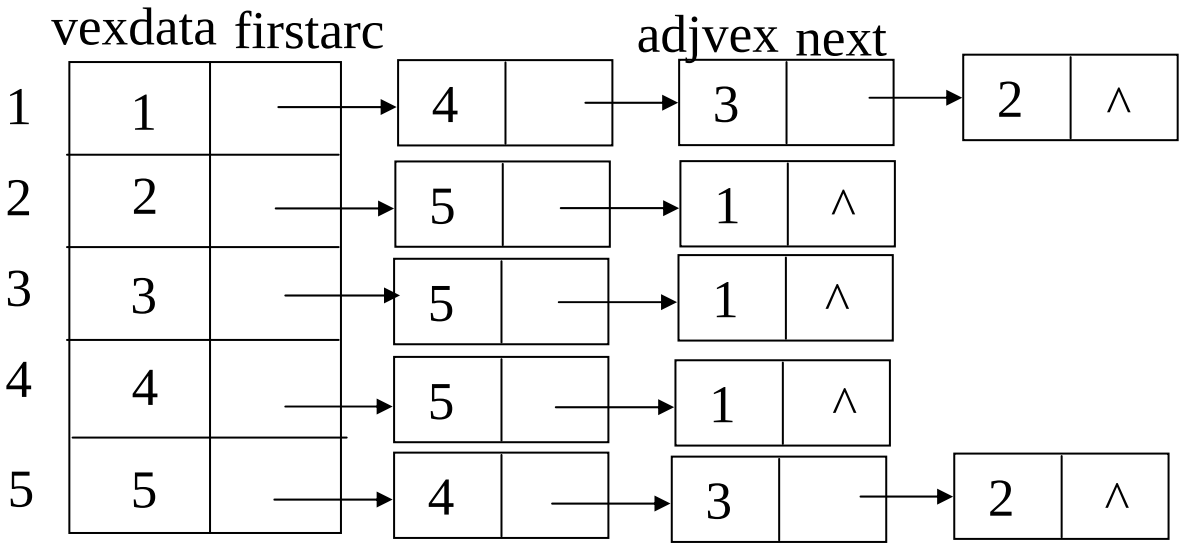
BFS

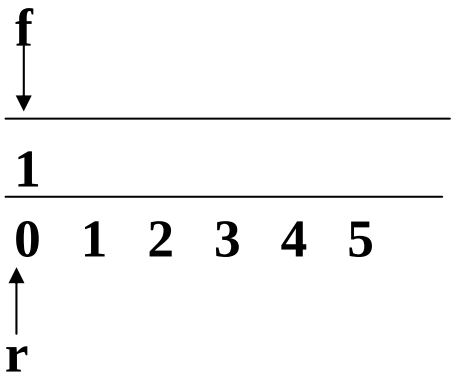


问题：
可否在出队时（P150）访问顶点？
和入队前访问有何区别？

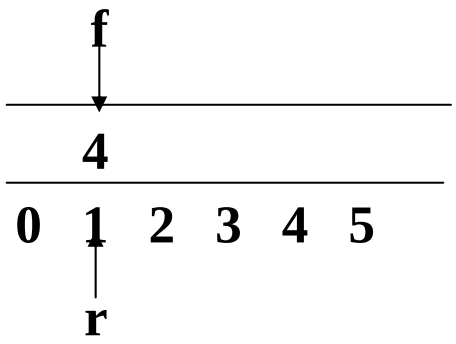
例



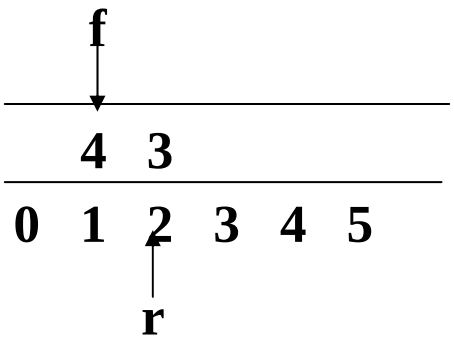




遍历序列：1

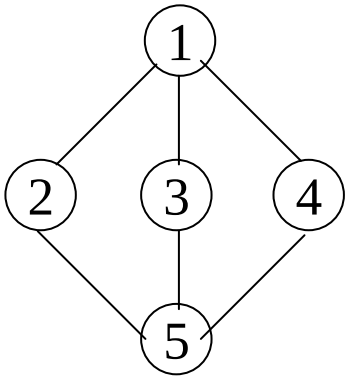


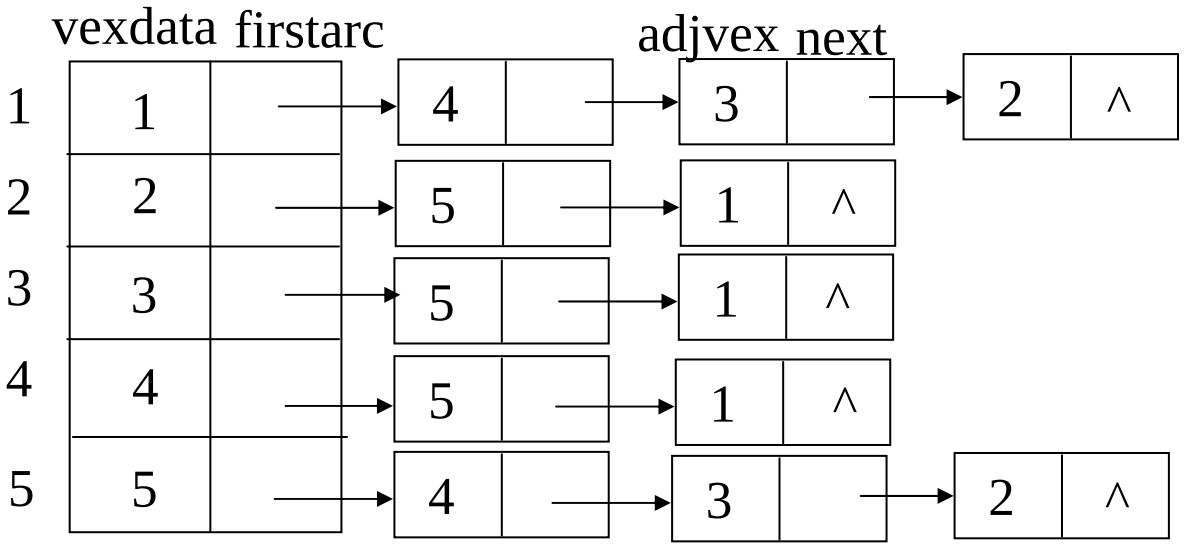
遍历序列：1 4

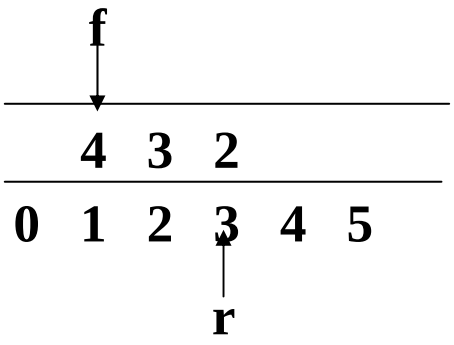


遍历序列：1 4 3

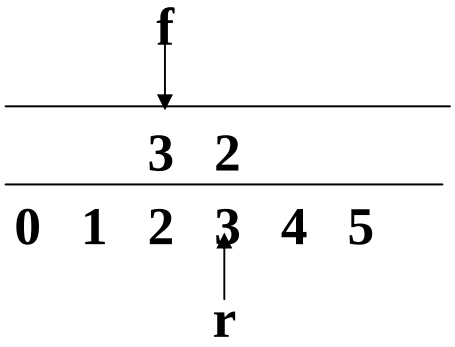
例



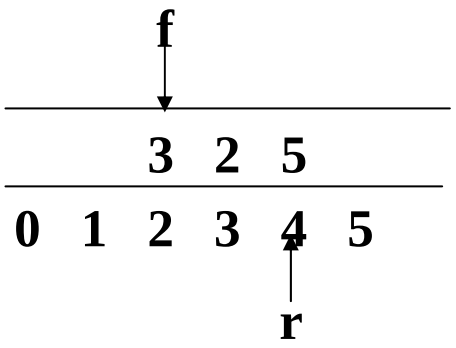




遍历序列：1 4 3 2

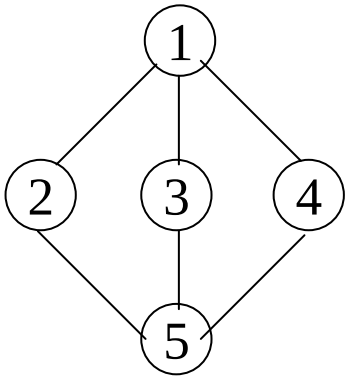


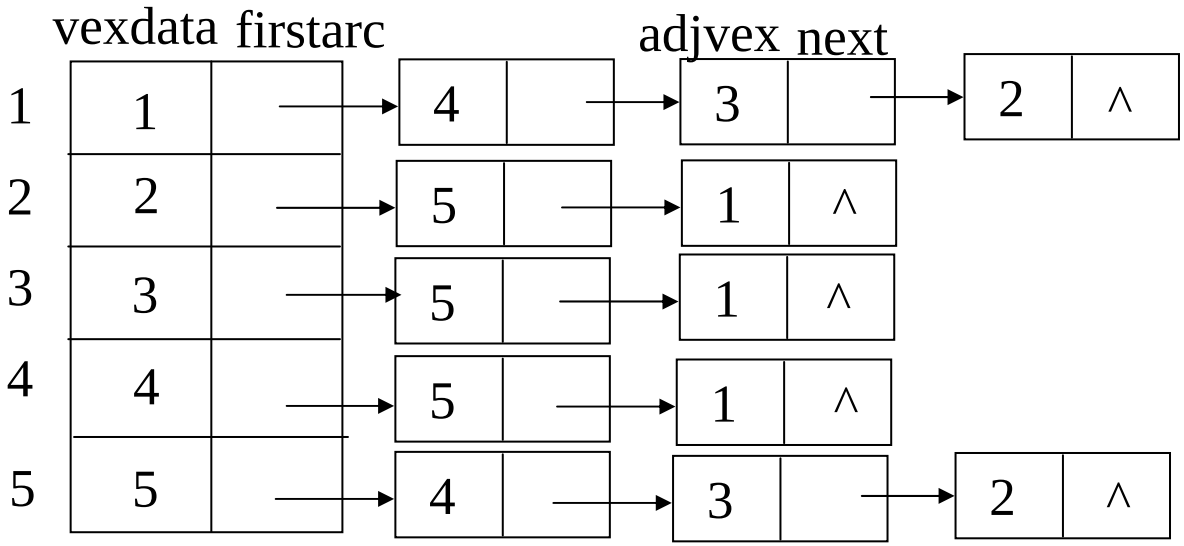
遍历序列：1 4 3 2

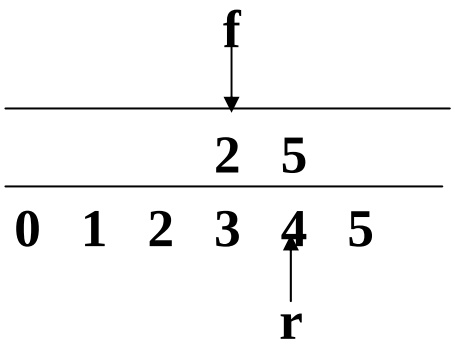


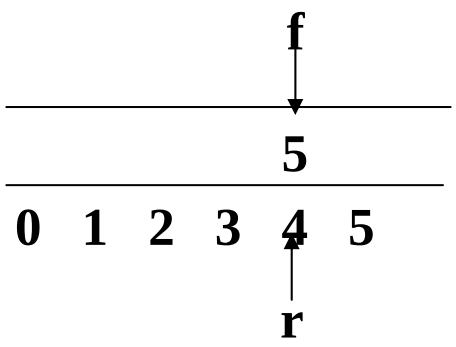
遍历序列：1 4 3 2 5

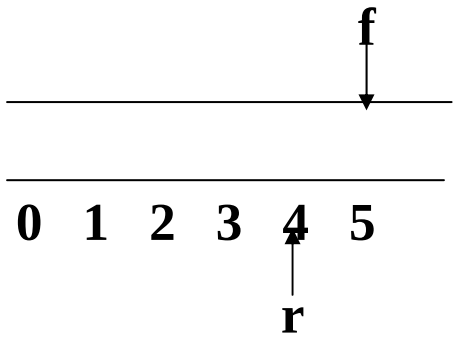
例











遍历序列：1 4 3 2 5

遍历序列：1 4 3 2 5

遍历序列：1 4 3 2 5

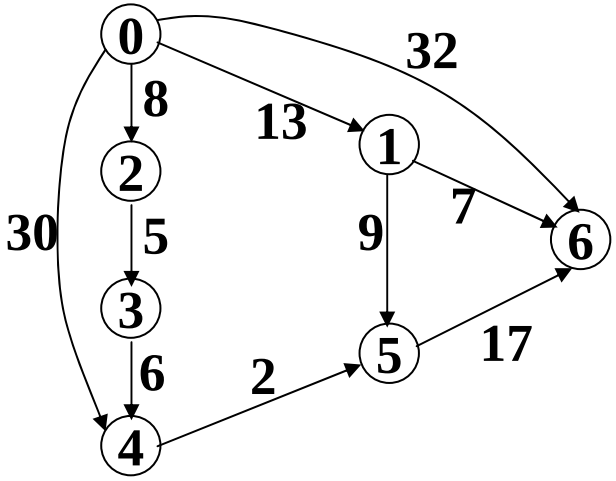
7.7 最短路径

- 问题提出

  用带权的有向图表示一个交通运输网，图中：

  - 顶点——表示城市
  - 边——表示城市间的交通联系
  - 权——表示此线路的长度或沿此线路运输所花的时间或费用等
  - 问题：从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径——最短路径。

- 单源最短路径



最短路径	长度
<V0,V1>	13
<V0,V2>	8
<V0,V2,V3>	13
<V0,V2,V3,V4>	19
<V0,V2,V3,V4,V5>	21
<V0,V1,V6>	20

➔ Dijkstra算法思想

按路径长度递增次序产生最短路径算法：

- 把V分成两组：
  - （1）S：已求出最短路径的顶点的集合
  - （2）V-S=T：尚未确定最短路径的顶点集合
- 将T中顶点按最短路径递增的次序加入到S中，保证：
  - （1）从源点V_0到S中各顶点的最短路径长度都不大于从V_0到T中任何顶点的最短路径长度
  - （2）每个顶点对应一个距离值
    - S中顶点：从V_0到此顶点的最短路径长度
    - T中顶点：从V_0到此顶点的只包括S中顶点作中间顶点的最短路径长度

依据：可以证明V_0到T中顶点V_k的最短路径，或是从V_0到V_k的直接路径的权值；或是从V_0经S中顶点到V_k的路径权值之和。

- 求最短路径步骤：
  - 初始时令 S={V_0},T={其余顶点}，T中顶点对应的距离值
    - 若存在<V_0,V_i>，距离值为<V_0,V_i>弧上的权值
    - 若不存在<V_0,V_i>，距离值为∞
  - 从T中选取一个其距离值为最小的顶点W，加入S
  - 对T中顶点的距离值进行修改：若加进W作中间顶点，从V_0到V_i的距离值比不加W的路径要短，则修改此距离值
  - 重复上述步骤，直到S中包含所有顶点，即S=V为止

算法实现

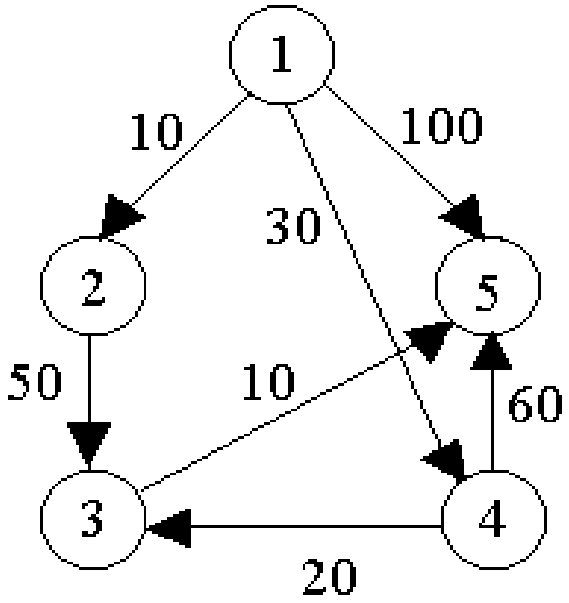
- 图用带权邻接矩阵存储**a[ ][ ]**
- 数组**dist[ ]**存放当前找到的从源点V_0到每个终点的最短路径长度，其初态为图中直接路径权值
- 数组**pre[ ]**表示从V_0到各终点的最短路径上，此顶点的前一顶点的序号；若从V_0到某终点无路径，则用**0**作为其前一顶点的序号

算法描述

步骤1：初始化dist[v]=a[s][v],0<=v<n;
对于所有与s邻接的顶点v置prev[v]=s;
其余顶点u置prev[u]=0;
建立表L包含所有prev[v]!=0的顶点v.
步骤2：若表L空则算法结束,否则转步骤3.
步骤3：从表L中取出dist值最小的顶点v.
步骤4：对于顶点v的所有邻接顶点u置
dist[u]=min{dist[u],dist[v]+a[v][u]};
若dist[u]改变则置prev[u]=v,且若u不在表L中
将u加入L;转步骤2.

dist[v]表示当前从源s到顶点v的最短特殊路径长度。

prev[v]表示从源s到顶点v的最短特殊路径上顶点v的前驱顶点。



迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

- **Dijkstra算法的计算复杂性**
- 对于一个具有***n***个顶点和***e***条边的赋权有向图，如果用赋权邻接矩阵表示这个图，那么**Dijkstra**算法的主循环体需要$O(n)$时间。这个循环需要执行$n-1$次，所以完成循环需要$O(n^2)$时间。算法的其余部分所需要时间不超过$O(n^2)$ 。

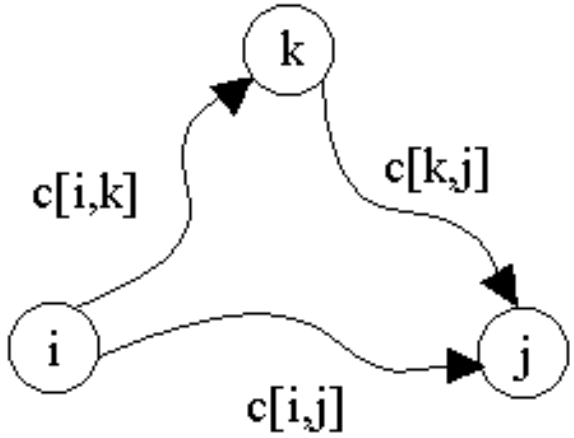
Bellman-Ford算法

- 如果允许有向赋权图中某些边的权为负实数，则**Dijkstra**算法不能正确地求出从源到所有其他顶点的最短路径长度。
- 此时可对**Dijkstra**算法做适当修改，得到如下的算法。
  - 该算法可返回一个布尔值，表明图**G**中是否有一个从源可达的负权圈。若有这样的圈，算法判定该问题无解。
  - **Bellman-Ford**算法的基本思想是，对图中除了源顶点**s**外的任一顶点**u**，依次构造从**s**到**u**的最短路长序列$dist^1[u]$， $dist^2[u]$ … $dist^{n-1}[u]$。其中$dist^{n-1}[u]$就是从**s**到**u**的最短路长。
  - 对于$k>1$， $dist^k[u]=\min\{dist^{k-1}[v]+e(v,u) \mid (v,u) \in E\}$。**Bellman-Ford**最短路算法就是依次递归式计算最短路。

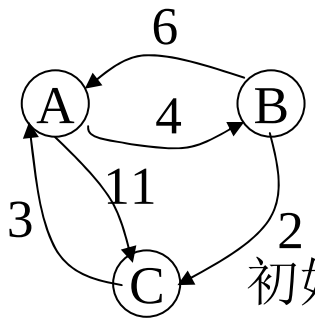
- 所有顶点对之间的最短路径
  - 方法一：每次以一个顶点为源点，重复执行**Dijkstra**算法**n**次—— **$T(n)=O(n^3)$**
  - 方法二： **Floyd**算法
    - 算法思想：逐个顶点试探法
    - 求最短路径步骤
      - 初始时设置一个**n**阶方阵，令其对角线元素为**0**，若存在弧**$<V_i,V_j>$**，则对应元素为权值；否则为
      - 逐步试着在原直接路径中增加中间顶点，若加入中间点后路径变短，则修改之；否则，维持原值
      - 所有顶点试探完毕，算法结束

Floyd算法实现

- 设V={ 0,1,…， *n* -1}。设置一个*n*×*n*矩阵c，初始时c[i,j]= a[i,j]。
- 在矩阵c上做*n*次迭代。经第k次迭代之后，c[i,j]的值是从顶点i到顶点j，且中间不经过编号大于k的顶点的最短路径长度。
- 在c上做第k次迭代时，用下面的公式来计算：
- c[i,j] = min { c[i,j], c[i,k]+c[k,j] }



例



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

	AB	AC
BA		BC
CA		

path= $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

加入A：$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	AC
BA		BC
CA	CAB	

path= $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

加入B：$\begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	ABC
BA		BC
CA	CAB	

path= $\begin{pmatrix} 0 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

加入C：$\begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	ABC
BCA		BC
CA	CAB	

path= $\begin{pmatrix} 0 & 0 & 2 \\ 3 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

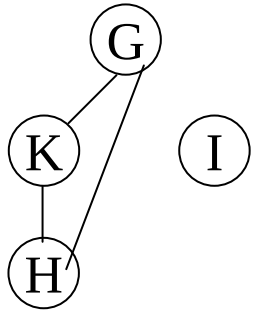
- 算法实现
  - 图用邻接矩阵存储
  - 二维数组c[ ][ ]存放最短路径长度
  - path[i][j]是从V_i到V_j的最短路径上V_j前一顶点序号
- 算法描述
- 算法分析 $T(n)=O(n^3)$

这样看来， Floyd算法似乎没带来更多的好处？？！

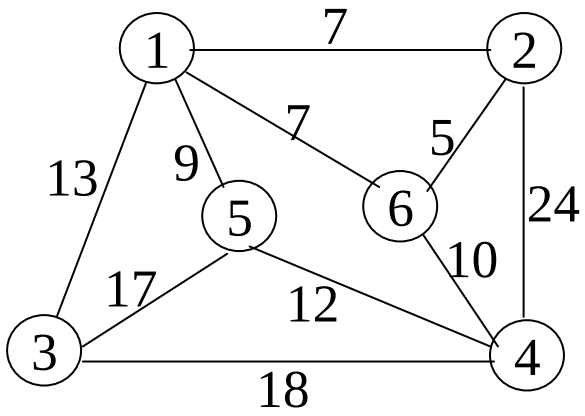
实际上，从实现代码来看：
Floyd算法的代码比用Dijkstra算法要简明得多！！！

7.9 支撑树（生成树）

- 定义：所有顶点均由边连接在一起，但不存在回路的图叫~
- 说明：
  - 一个图可以有许多棵不同的生成树
  - 所有生成树具有以下共同特点：
    - 生成树的顶点个数与图的顶点个数相同
    - 生成树是图的极小连通子图
    - 一个有**n**个顶点的连通图的生成树有**n-1**条边
    - 生成树中任意两个顶点间的路径是唯一的
    - 在生成树中再加一条边必然形成回路
  - 含**n**个顶点**n-1**条边的图不一定是生成树



最小支撑树（最小生成树）



- 问题提出

  要在**n**个城市间建立通信联络网，

  顶点——表示城市

  权——城市间建立通信线路所需花费代价

  希望找到一棵生成树，它的每条边上的权值之和（即建立该通信网所需花费的总代价）最小———最小代价生成树

- 问题分析

  n个城市间，最多可设置**n(n-1)/2**条线路

  n个城市间建立通信网，只需**n-1**条线路

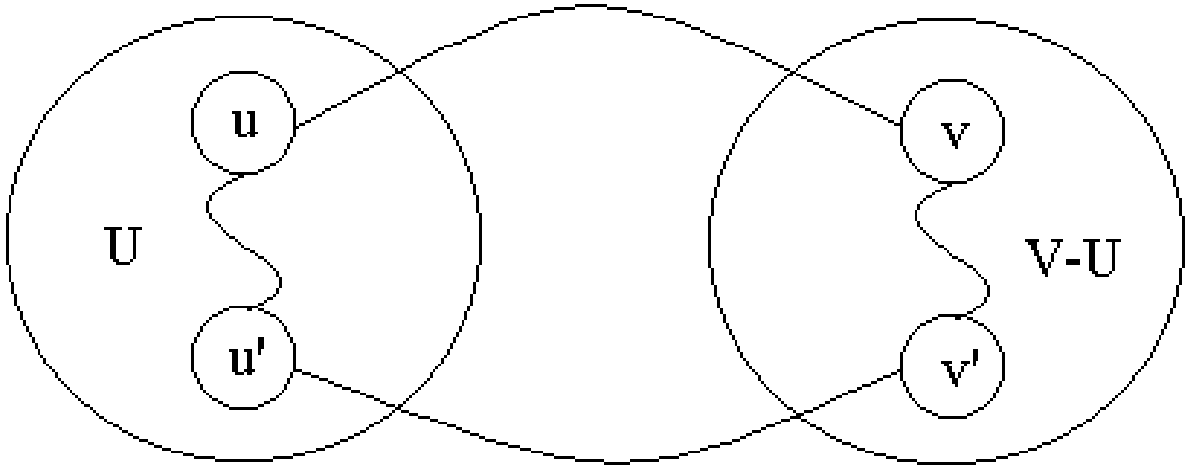
  问题转化为：如何在可能的线路中选择**n-1**条，能把所有城市（顶点）均连起来，且总耗费（各边权值之和）最小

最小生成树性质

- 用贪心算法设计策略可以设计出构造最小生成树的有效算法。本节介绍的构造最小生成树的**Prim**算法和**Kruskal**算法都可以看作是应用贪心算法设计策略的例子。尽管这**2**个算法做贪心选择的方式不同，它们都利用了下面的最小生成树性质：
  - 设**G=(V,E)**是连通带权图，**U**是**V**的真子集。如果**(u,v)∈E**，且**u∈U**，**v∈V-U**，且在所有这样的边中，**(u,v)**的权**c[u][v]**最小，那么一定存在**G**的一棵最小生成树，它以**(u,v)**为其中一条边。这个性质有时也称为**MST**性质。

✪ MST性质证明

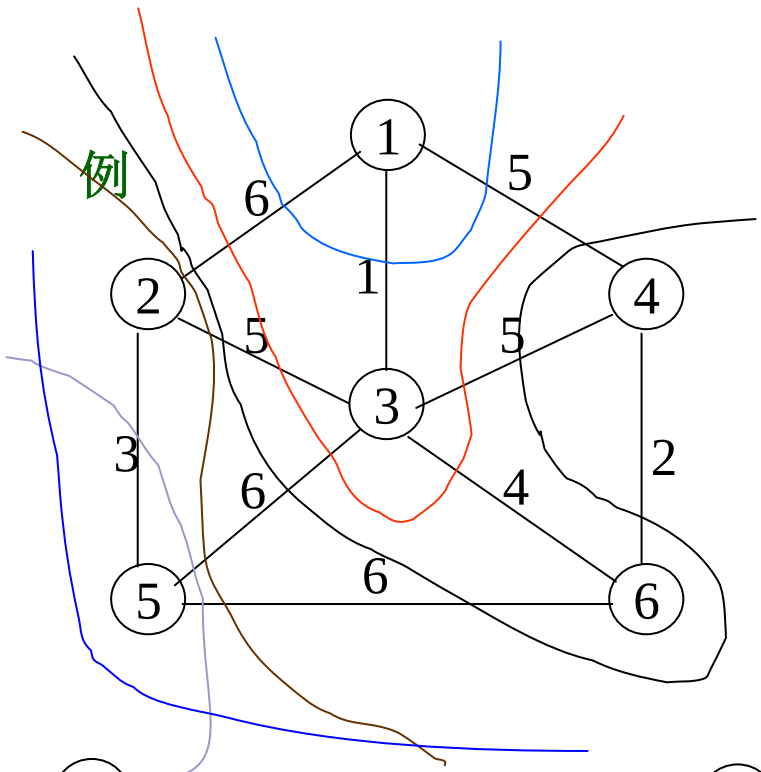
假设G的任何一棵最小生成树都不含边(u,v)。将边(u,v)添加到G的一棵最小生成树T上，将产生含有边(u,v)的圈，并且在这个圈上有一条不同于(u,v)的边(u',v')，使得u'∈U,v'∈V-U，如下图所示。

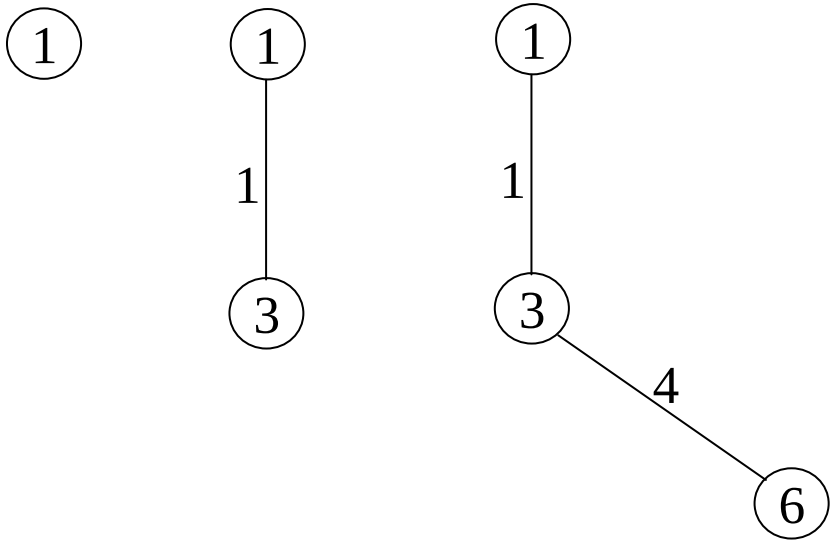


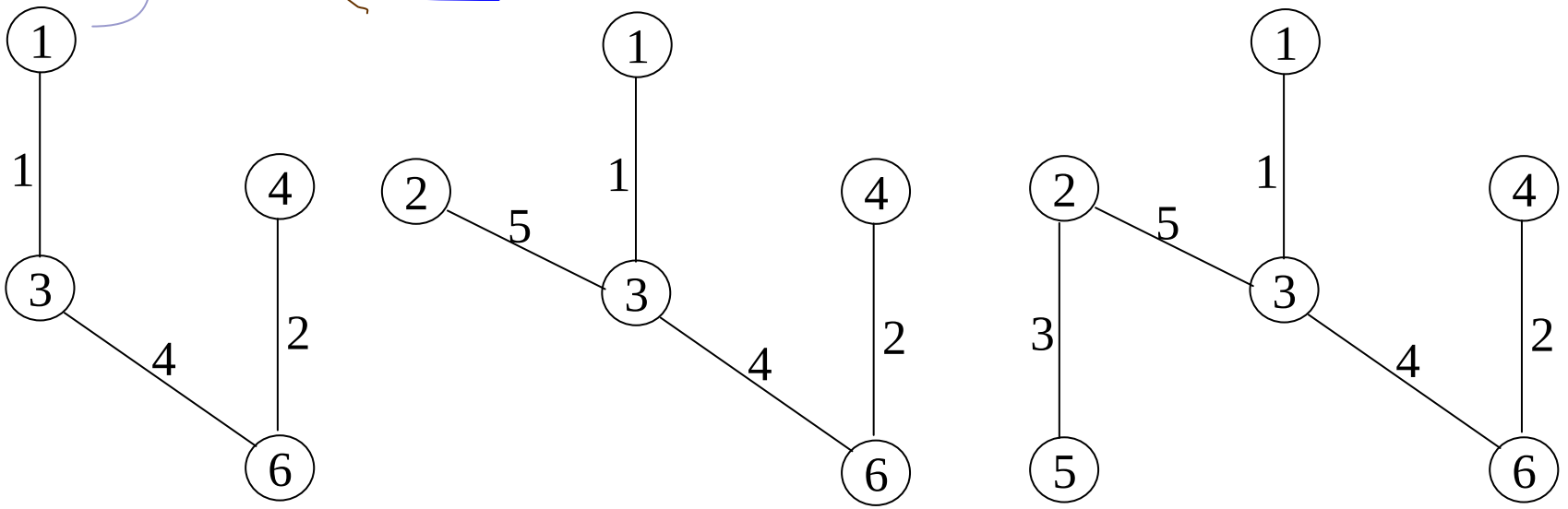
图示 含边(u,v)的圈

将边(u',v')删去，得到G的另一棵生成树T'。由于c[u][v]≤c[u'][v']，所以T'的耗费≤T的耗费。于是T'是一棵含有边(u,v)的最小生成树，这与假设矛盾。

- 构造最小生成树方法
  - 方法一：**Prim**算法
    - 算法思想：设**G=(V,E)**是连通网，**T**是**N**上最小生成树中边的集合
      - 初始令**U={u_0},(u_0∈V), T=Φ**
      - 在所有**u∈U,v∈V-U**的边**(u,v)∈E**中，找一条代价最小的边**(u_0,v_0)**
      - 将**(u_0,v_0)**并入集合**T**，同时**v_0**并入**U**
      - 重复上述操作直至**U=V**为止，则**T=(V, T)**为**N**的最小生成树
    - 算法实现：图用邻接矩阵表示
    - 算法描述
    - 算法评价：**$T(n)=O(n^2)$**

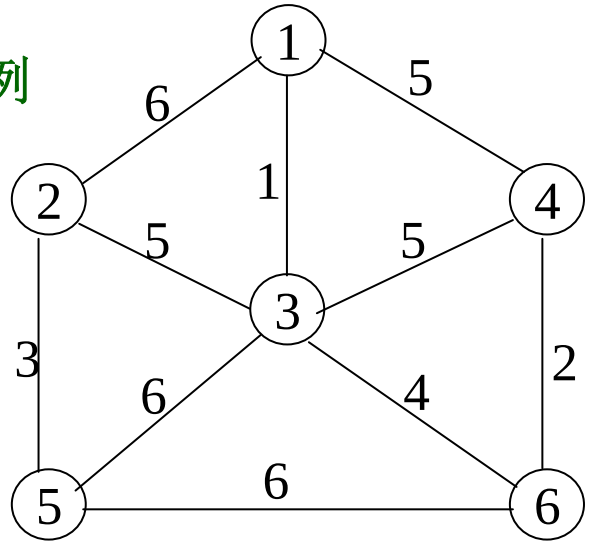

例
1
2
3
4
5
6
6
5
1
5
5
3
6
4
2
6

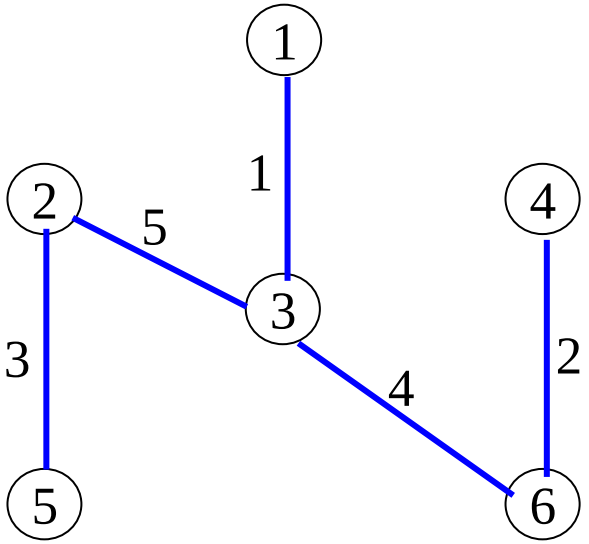

1
1
1
3
1
1
3
4
6


1
1
4
3
4
2
6
2
5
1
4
3
4
2
6
1
2
5
1
4
3
3
4
2
5
6

- 构造最小生成树方法
  - 方法二：**Kruskal**算法
    - 算法思想：设**G=(V,E)**，令最小生成树
      - 初始状态为只有**n**个顶点而无边的非连通图**T=(V,{Φ})**，每个顶点自成一个连通分量
      - 在**E**中选取代价最小的边，若该边依附的顶点落在**T**中不同的连通分量上，则将此边加入到**T**中；否则，舍去此边，选取下一条代价最小的边
      - 依此类推，直至**T**中所有顶点都在同一连通分量上为止

例





- 方法二：**Kruskal**算法
  - 算法描述：
  - 算法分析：

    设输入的连通赋权图有**e**条边，则将这些边依其权值组成优先队列需要**O(e)**时间；**while**循环中，**DeleteMin**运算需要**O(loge)**时间，因此关于优先队列所作运算的时间为**O(eloge)**。

    实现**UnionFind**所需的时间为**O(eloge)**。

    ∴ Kruskal算法所需的计算时间是**O(eloge)**。

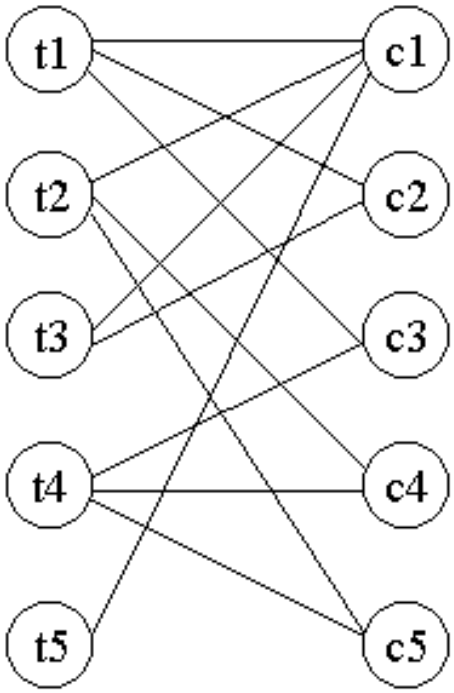
- **Prim**算法与**Kruskal**算法的比较

  从算法的时间复杂性看:

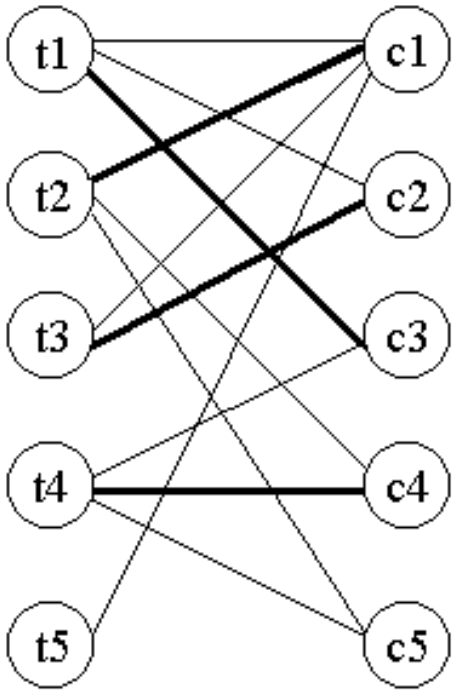
  - 当**e= $\Omega(n^2)$**时，**Kruskal**算法比**Prim**算法差，
  - 但当**e=O(n^2)**时，**Kruskal**算法却比**Prim**算法好得多。

7.10 图匹配算法

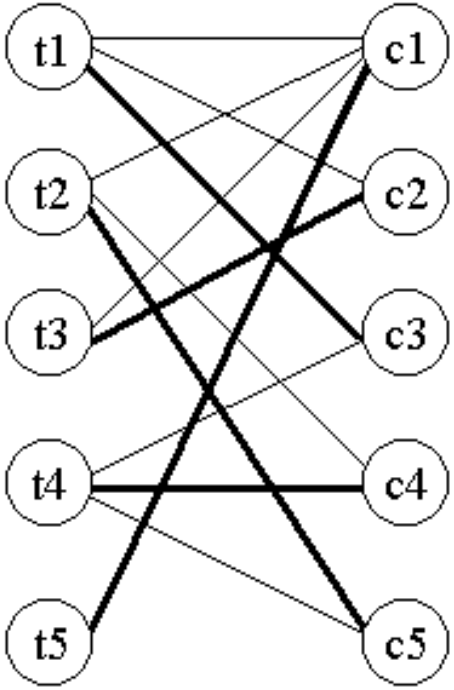
- 设G=(V,E)是一个无向图。如果顶点集合V可分割为2个互不相交的子集，并且图中每条边(i,j)所关联的2顶点i和j分属于这2个不同的顶点集，则称图G为一个二分图。
- 在学校的教务管理中，排课表是一项例行工作。一般情况下每位教师可胜任多门课程的教学，而每个学期只讲授一门所胜任的课程。反之每学期的一门课程只需一位教师讲授。这就需要对课程和教师作合理安排。可以用一个二分图来表示教师与课程的这种关系。教师和课程都是图的顶点，边(t,c)表示教师t胜任课程c。


t1
t2
t3
t4
t5
c1
c2
c3
c4
c5

(a)


t1
t2
t3
t4
t5
c1
c2
c3
c4
c5

(b)


t1
t2
t3
t4
t5
c1
c2
c3
c4
c5

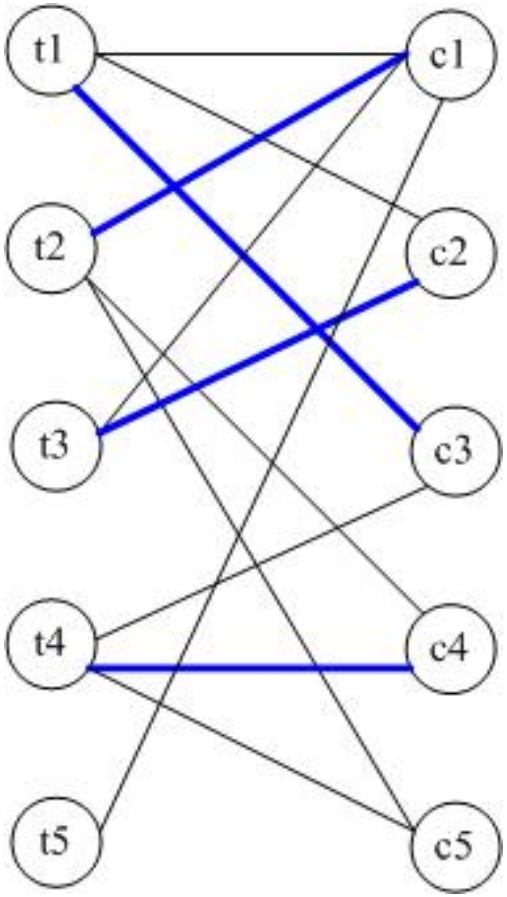
(c)

- 图匹配问题可描述如下：设**G =(V,E)**是一个图。如果**M ⊆ E**，且**M**中任何**2**条边都不与同一个顶点相关联，则称**M**是**G**的一个匹配。**G**的边数最多的匹配称为**G**的最大匹配。
- 如果图的一个匹配使得图中每个顶点都是该匹配中某条边的端点，那么就称这个匹配为图的一个完全匹配。一个图的完全匹配一定是这个图的一个最大匹配。
- 为了求一个图的最大匹配，可以系统地列举出该图的所有匹配，然后从中选出边数最多者。这种方法所需要的时间是图中边数的一个指数函数。因此，需要一种更有效的算法。

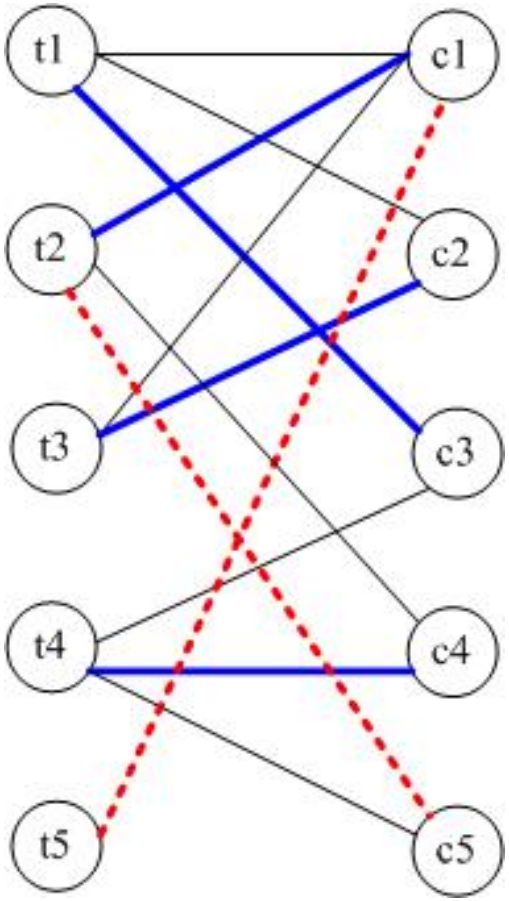
- 设M是图G的一个匹配，将M中每边所关联的顶点称为已匹配顶点，其余顶点称为未匹配顶点。若p是图G中一条连通2个未匹配顶点的路径，并且在路径p上属于M的边和不属于M的边交替出现，则称p为一条关于M的增广路径。由此定义可知增广路径具有以下性质：
- (1) 一条关于M的增广路径的长度必为奇数，且路上的第一条边和最后一条边都不属于M。
- (2) 对于一条关于M的增广路径P，将M中属于P的边删去，将P中不属于M的边添加到M中，所得到的边集合记为M⊕P，则M⊕P是一个比M更大的匹配。
- (3) M为G的一个最大匹配当且仅当不存在关于M的增广路径。

最大匹配的增广路径算法

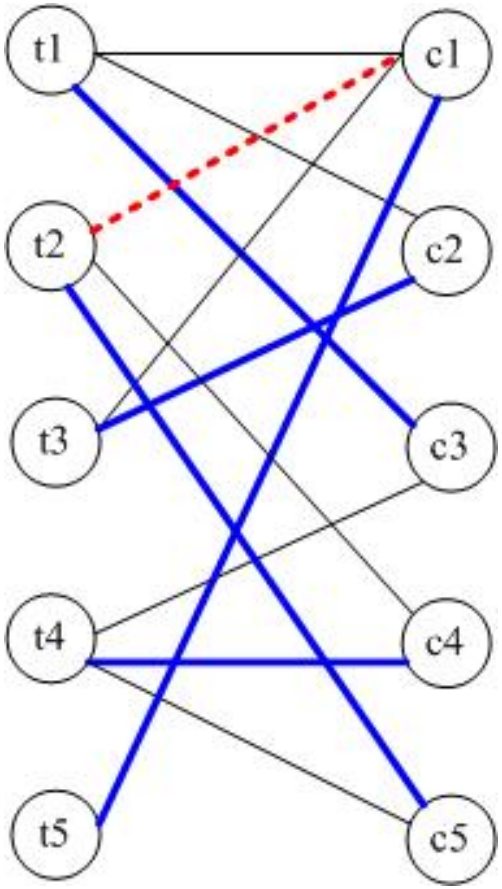
- **(1)** 置**M**为空集；
- **(2)** 找出一条关于**M**的增广路径**P**，并用**M⊕P**代替**M**；
- **(3)** 重复步骤**(2)**直至不存在关于**M**的增广路径，最后得到的匹配就是**G**的一个最大匹配。



图G的一个匹配M
(如图加粗的蓝边)



关于M的一条增广
路经P：t5,c1,t2,c5



M⊕P代替M，得到
G的一个更大匹配

THE END