第17章 贪婪算法

- 最优化问题
- 贪婪算法(greedy method)思想
- **17.3.3 拓扑排序**
- **17.3.5 单源最短路径**
- **17.3.6 最小耗费生成树**

最优化问题

- 每个最优化问题都包含一组**限制条件**(constraint)和一个**优化函数**(optimization function)；
- 符合限制条件的问题求解方案称为**可行解**(feasible solution)；
- 使优化函数取得最佳值的可行解称为**最优解**(optimal solution)。

装载问题

- 有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第i 个货箱的重量为w_i（$1 \le i \le n$），而货船的最大载重量为c。
- 目的是在货船上装入最多的货箱。

装载问题–最优化问题描述

- 设存在一组变量x_i，其可能取值为0或1。
  - $x_i=0$，则货箱i 将不被装上船；
  - $x_i=1$，则货箱i 将被装上船。
- 目的是找到一组x_i，
  - 限制条件$\Sigma w_i x_i \le c$ 且$x_i \in \{0, 1\}$, $1 \le i \le n$。
  - 优化函数是Σx_i
- 满足限制条件的每一组x_i都是一个可行解，
- 能使Σx_i取得最大值的方案是最优解。

贪婪算法思想

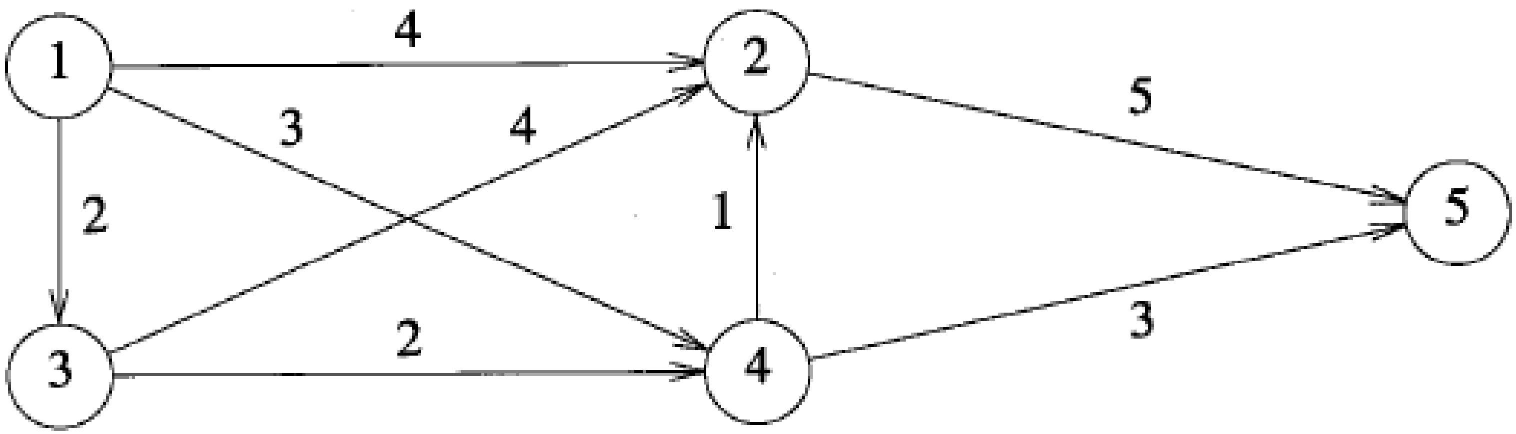
- 贪婪算法(greedy method)思想
  - 采用**逐步构造最优解**的方法。在每个阶段，都作出一个看上去最优的决策(在一定的标准下)。决策一旦作出，就不可再更改。
  - **贪婪准则:**做决策的依据.
  - **贪婪算法:**每一步,根据贪婪准则,做出一个看上去最优的决策.

货箱装船问题

- 贪婪准则：从剩下的货箱中，选择重量最小的货箱。
- 这种选择次序可以保证所选的货箱总重量最小，从而可以装载更多的货箱。
- 例
  - n=8,
  - [w1,...w8]=[100,200,50,90,150,50,20,80],c=400。
- 利用贪婪算法能产生最佳装载

最短路径

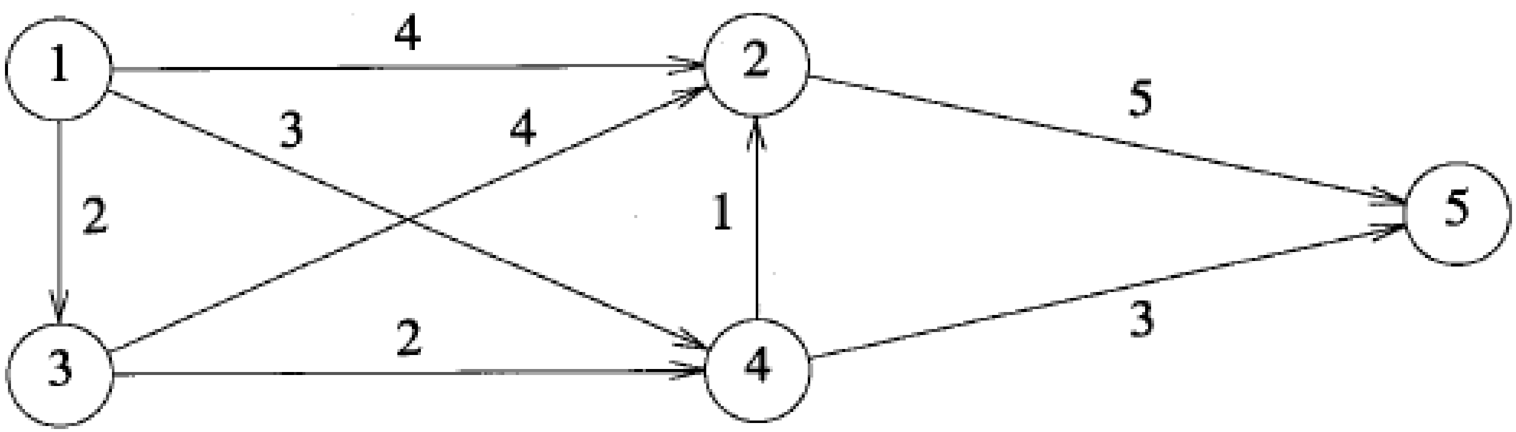
- 找一条从初始顶点s 到达目的顶点d 的最短路径



- 贪婪算法：分步构造这条路径，每一步在路径中加入一个顶点

最短路径–贪婪算法思想

- 加入下一个顶点的**贪婪准则**：
  - 假设当前路径已到达顶点q，且顶点q 并不是目的顶点d
  - 选择离q 最近且目前不在路径中的顶点



- 这种贪婪算法并不一定能获得最短路径(例1->5)

0/1背包问题

- 在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为w_i，价值为p_i。
- 可行的背包装载：背包中物品的总重量不能超过背包的容量
  - 约束条件为$\sum w_i x_i <= c$和$x_i \in [0,1] (1 \le i \le n)$。
- 最佳装载是指所装入的物品价值最高，即
- $\sum_{i=1}^{n} p_i x_i$ 取得最大值。

0/1背包问题

- 需求出x_i 的值
  - x_i =1　表示物品i 装入背包中
  - x_i =0　表示物品i 不装入背包
- 0/1背包问题是一个一般化的货箱装载问题，即每个货箱所获得的价值不同。
- 货箱装载问题转化为背包问题的形式为：船作为背包，货箱作为可装入背包的物品。

0/1背包问题贪婪策略

- 贪婪准则：从剩余的物品中，选出可以装入背包的价值最大的物品。
- 利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。
- 这种策略不能保证得到最优解。
- n=3,w=[100,10,10],p=[20,15,15],c=105。

0/1背包问题贪婪策略2

- 重量贪婪准则：从剩下的物品中选择可装入背包的重量最小的物品。
- 虽然这种规则对于前面的例子能产生最优解，但在一般情况下则不一定能得到最优解。
  - n=2,w=[10,20],p=[5,100],c=25。

0/1背包问题贪婪策略3

- 价值密度贪婪准则：从剩余物品中选择可装入包的p_i /w_i 值最大的物品。
- 这种策略也不能保证得到最优解。
  - n=3,w=[20,15,15],p=[40,25,25],c=30。

0/1背包问题

- 0/1背包问题是一个NP-复杂问题。

贪婪算法

- 在有些应用中，贪婪算法所产生的结果总是最优的解决方案。
- 但对其他一些应用，生成的算法只是一种启发式方法。
- 可能是也可能不是近似算法。

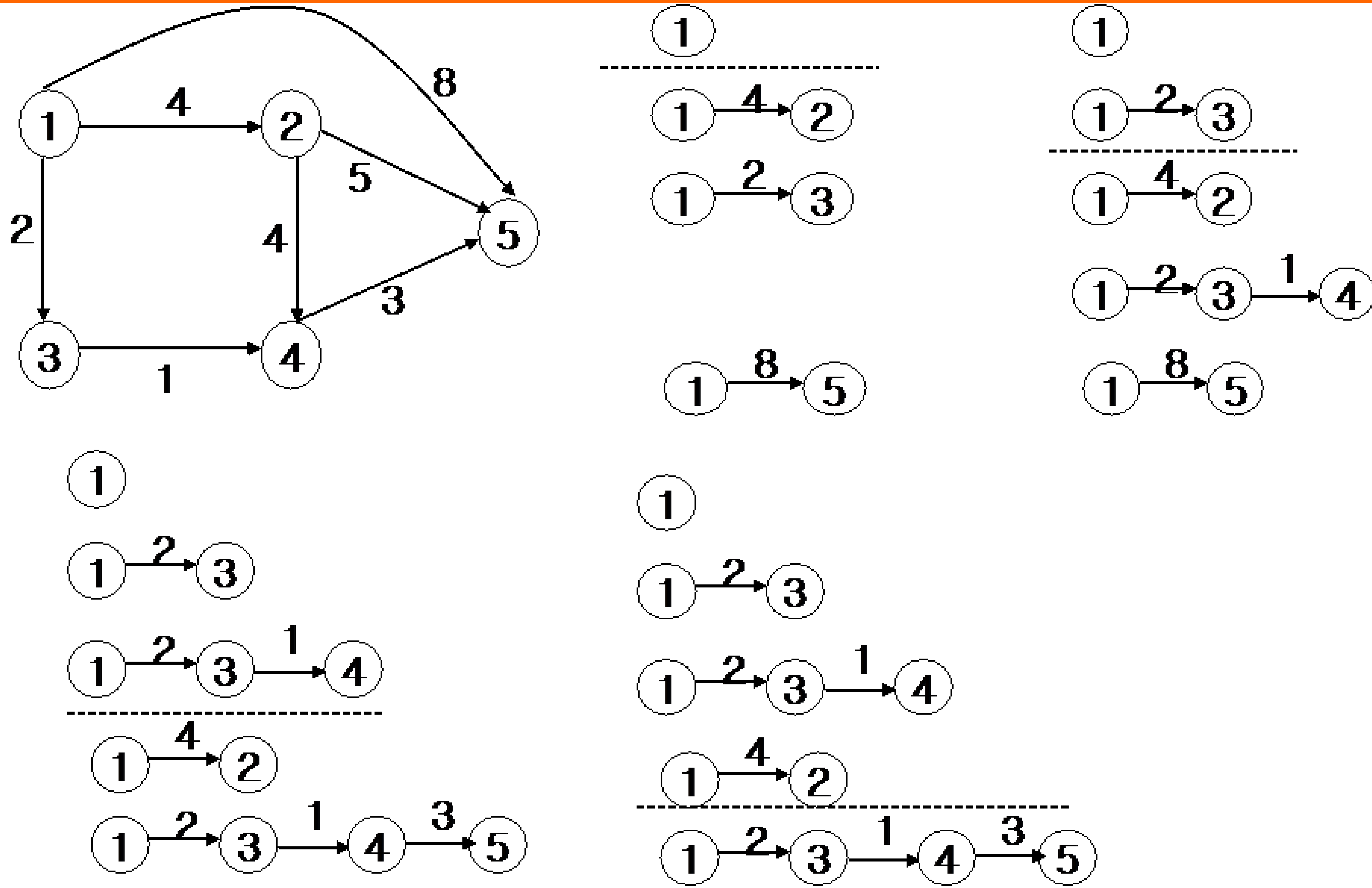
17.3.5 单源最短路径

- 带权有向图.
- 路径的长度即为此路径所经过的边的长度(耗费)之和。
- 对于给定的源顶点sourceVertex， 需找出从它到图中其他任意顶点(称为目的)的最短路径。
- 假设:
  - 边的长度(耗费) >= 0.
  - 没有路径的长度< 0.

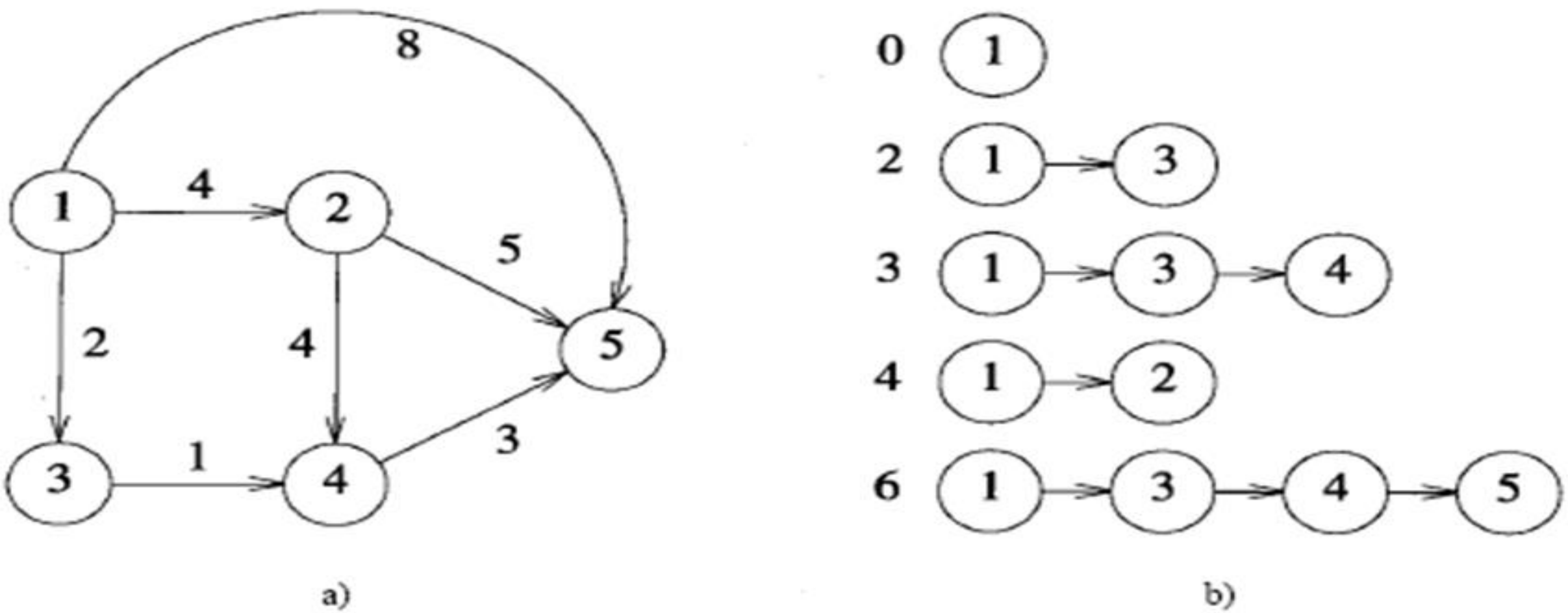
Dijkstra算法

- Dijkstra(迪克斯特拉/迪杰斯特拉)
- E. Dijkstra发明的贪婪算法：**分步**求源点**sourceVertex**到其它各顶点的最短路径。**每一步产生一个到达新的目的顶点**的最短路径。
- Dijkstra算法：
  - 按路径长度递增顺序产生最短路径。
  - 首先最初产生从sourceVertex到它自身的路径，这条路径没有边，其长度为0。
  - 在贪婪算法的**每一步**中，**产生下一个最短路径**：在目前产生的每一条最短路径中，考虑加入一条边到达未产生最短路径的顶点，再从所有这些新路径中选择最短的。

示例



示例



- 每一条路径(第1条除外)都是由一条已产生的最短路径加上一条边形成。
- 下一条最短路径总是由已产生的最短路径再扩充一条边得到的，且这条路径所到达的顶点其最短路径还未产生。

算法分析

- **distanceFromSource[i]:**
  - **在当前已产生的最短路径中加入一条边**,从而使得扩充的路径**到达顶点i的最短长度**。
- a：有向图的邻接矩阵，初始，仅有从sourceVertex到它自身的一条长度为0的路径，
  - 对于每个顶点i，distanceFromSource[i]等于a[sourceVertex][i]。
  - 当获得一条新的最短路径后，由于新的最短路径可能会产生更小的distanceFromSource值，因此有些顶点的distanceFromSource值可能会发生变化。

算法分析

- **表newReachableVertices:** 存储路径可到达顶点且未产生最短路径的顶点。
- 在路径可到达顶点且未产生最短路径的顶点中，distanceFromSource值最小的即为下一条最短路径的终点。
- 数组**predecessor[]**:存储最短路径
  - **predecessor[i]**——从sourceVertex到达i的路径中顶点i前面的那个顶点。

Dijkstra算法的伪代码-1/2

1)初始化

distanceFromSource[i]=a[sourceVertex][i]($1\le i\le n$)，

predecessor[i]=sourceVertex,//邻接于sourceVertex的顶点

predecessor[sourceVertex] = 0;

predecessor[i]=-1;//其余的顶点

创建一个表**newReachableVertices**，保存所有 **predecessor**[i]>0的顶点。

2) 当**newReachableVertices**为空时,算法停止,否则转至3);

Dijkstra算法的伪代码-2/2

3) 从**newReachableVertices**中选择并删除 **distanceFromSource**值最小的顶点i。

4) 对于所有邻接于顶点i的顶点j,

更新**distanceFromSource[j]**值为

min{distanceFromSource[j],

distanceFromSource[i] +a[i][j]}；

若distanceFromSource[j]改变，

则 置predecessor[j]=i，

而且，若j没有在表newReachableVertices中，

则将j 加入newReachableVertices。

数据结构的选择

- **distanceFromSource()** : 1维数组
- **predecessor()** : 1维数组.
- **newReachableVertices** :无序链表或最小堆 .
  - 选择采用无序链表 chain ?
  - 采用最小堆?

```
void shortestPaths(int sourceVertex, T* distanceFromSource, int* predecessor)
{// 寻找从源 sourceVertex 开始的最短路径
 // 在数组 distanceFromSource 中返回最短路径
 // 在数组 predecessor 中返回顶点在路径上的前驱的 information
   if (sourceVertex < 1 || sourceVertex > n)
      throw illegalParameterValue("Invalid source vertex");
   // 这里确认 *this 是加权图的代码

   graphChain<int> newReachableVertices;
   // 初始化
   for (int i = 1; i <= n; i++)
   {
      distanceFromSource[i] = a[sourceVertex][i];
      if (distanceFromSource[i] == noEdge)
         predecessor[i] = -1;
      else
      {
         predecessor[i] = sourceVertex;
         newReachableVertices.insert(0, i);
      }
   }
   distanceFromSource[sourceVertex] = 0;
   predecessor[sourceVertex] = 0;  // 源顶点没有前驱

   // 更新 distanceFromSource 和 predecessor
   while (!newReachableVertices.empty())
```

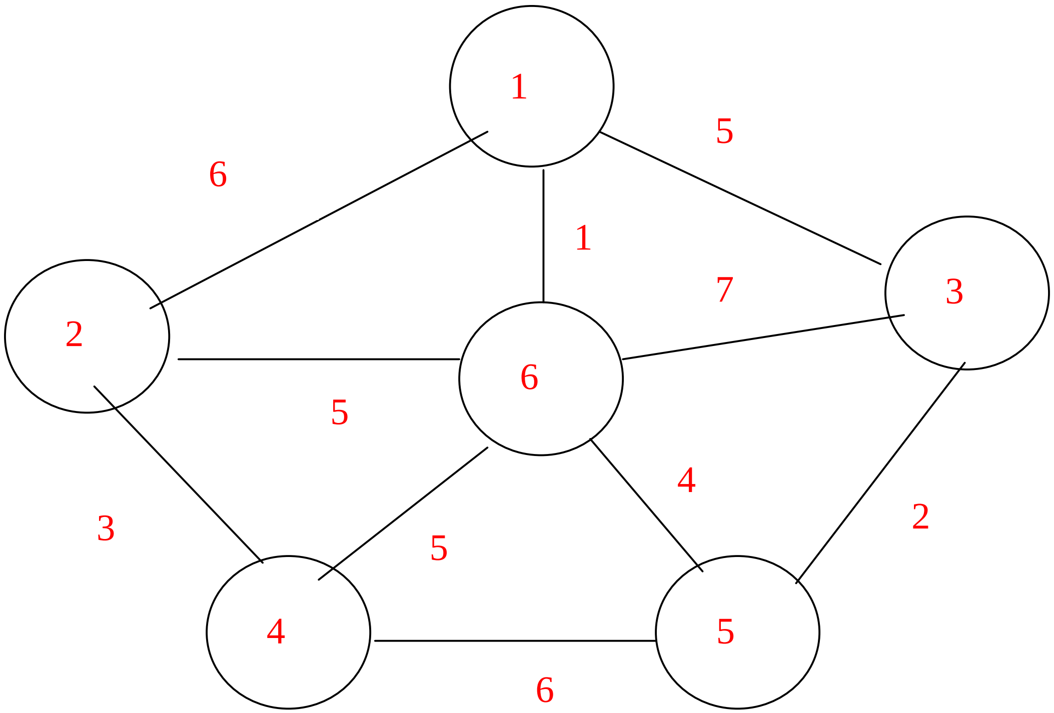
```
{// 还存在更多的路径
 // 寻找 distanceFromSource 值最小的，还未到达的顶点 v
   chain<int>::iterator iNewReachableVertices= newReachableVertices.begin();
   chain<int>::iterator theEnd = newReachableVertices.end();
   int v = *iNewReachableVertices;
   iNewReachableVertices++;
   while (iNewReachableVertices != theEnd)
   {
      int w = *iNewReachableVertices;
      iNewReachableVertices++;
      if (distanceFromSource[w] < distanceFromSource[v])
         v = w;
   }

   // 下一条最短路径是到达顶点 v
   // 从 newReachableVertices 删除顶点 v, 然后更新 distanceFromSource
   newReachableVertices.eraseElement(v);
   for (int j = 1; j <= n; j++)
   {
      if (a[v][j] != noEdge && (predecessor[j] == -1 ||
   distanceFromSource[j] > distanceFromSource[v] + a[v][j]))
      {
         // distanceFromSource[j] 减少
         distanceFromSource[j] = distanceFromSource[v] + a[v][j];
         // 把顶点 j 加到 newReachableVertices
         if (predecessor[j] == -1)
         // 以前未到达
            newReachableVertices.insert(0, j);
         predecessor[j]=v;
      }
   }
 }
}
```

复杂性分析

- 在**newReachableVertices**中选择**distanceFromSource**值最小的顶点**i**：O(n)
- 更新邻接自顶点**i** 的顶点的 **distanceFromSource**值和 **predecessor**值
  - 使用邻接表：O(顶点**i** 的出度) .
  - 使用耗费邻接矩阵：O(n) .
- 总的时间复杂性： $O(n^2)$.
- 读P435程序17-3

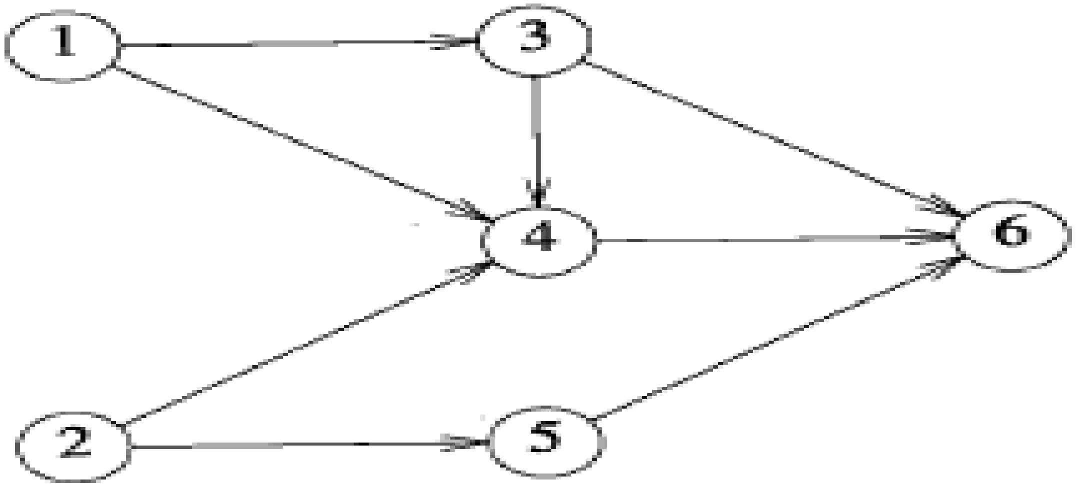
练习

- 已知图G如下所示，请给出从顶点1出发到其他顶点的最短路径



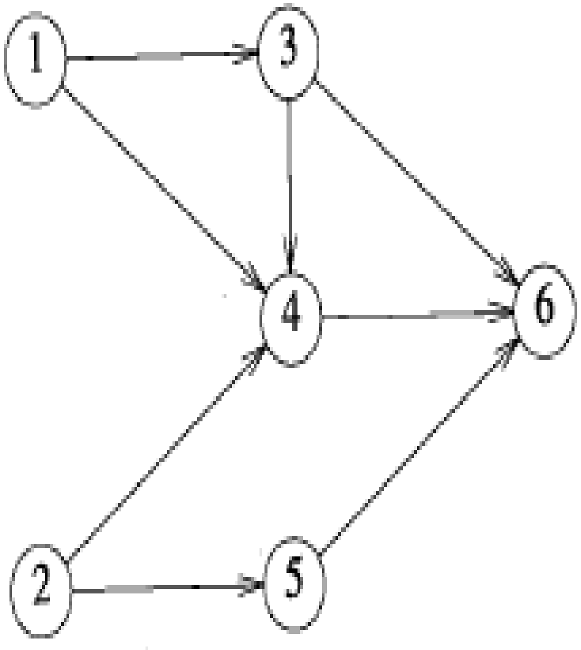
17.3.3 拓扑排序

- 一个复杂的工程通常可以分解成一组简单任务(活动)的集合，完成这些简单任务意味着整个工程的完成。
- 任务之间具有先后关系。



顶点活动网络(AOV)

- 顶点活动网络(AOV—Activity on vertex network)：任务的集合以及任务的先后顺序
  - 顶点：表示任务(活动)
  - 有向边(i, j)：表示任务间先后关系——任务j 开始前任务i 必须完成。

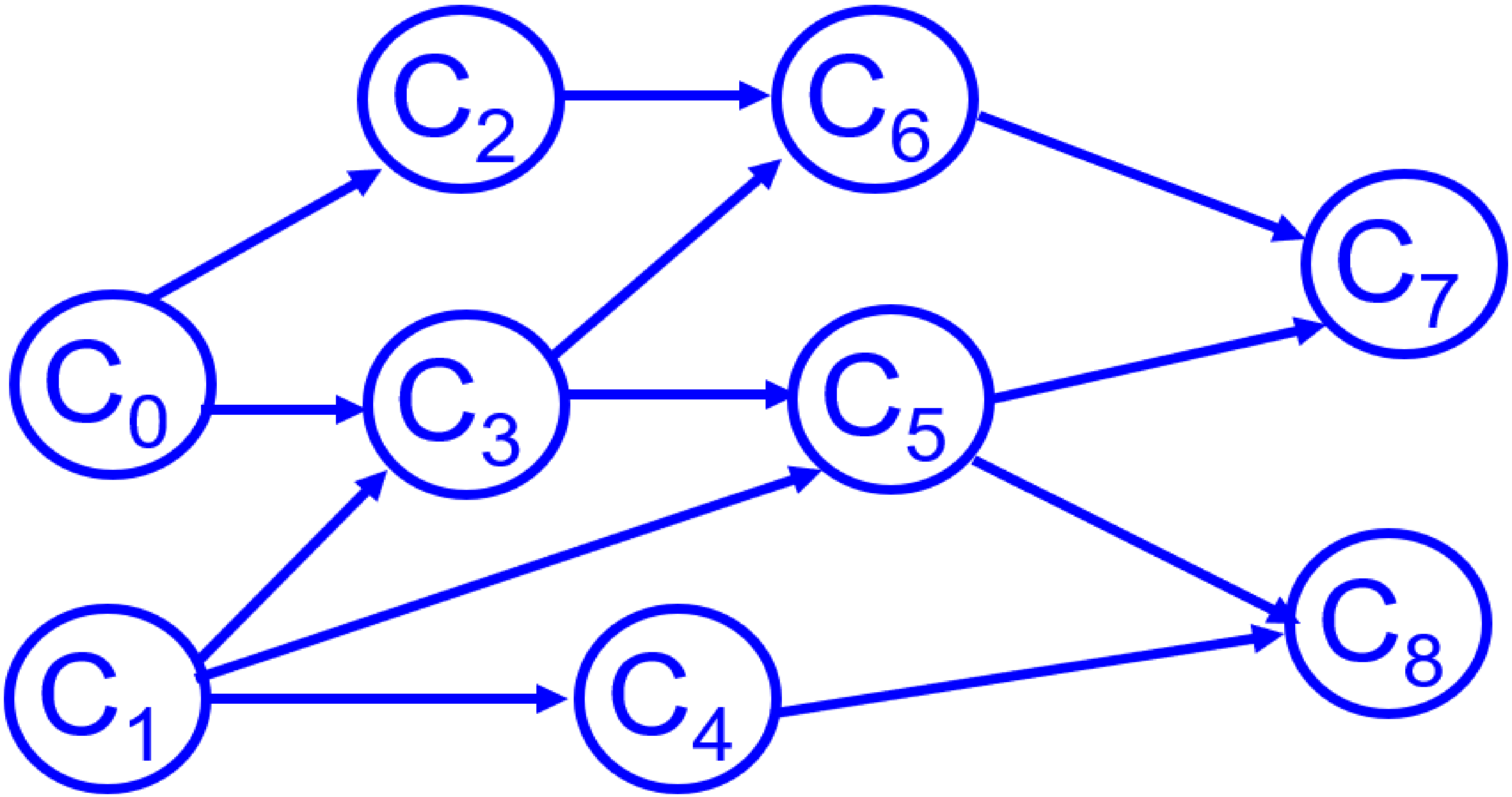


拓扑排序应用示例

- 计算机专业的学习就是一个工程，每一门课程的学习就是整个工程的一些活动(任务)。其中有些课程要求先修课程。

课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1，C_2
C_4	数据结构	C_3，C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5，C_4
C_7	操作系统	C_4，C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8

拓扑排序应用示例



课程学习工程图

拓扑排序问题

- 在很多条件下，任务的执行是连续进行的，需要按照一个顺序来执行。
- 任务序列？

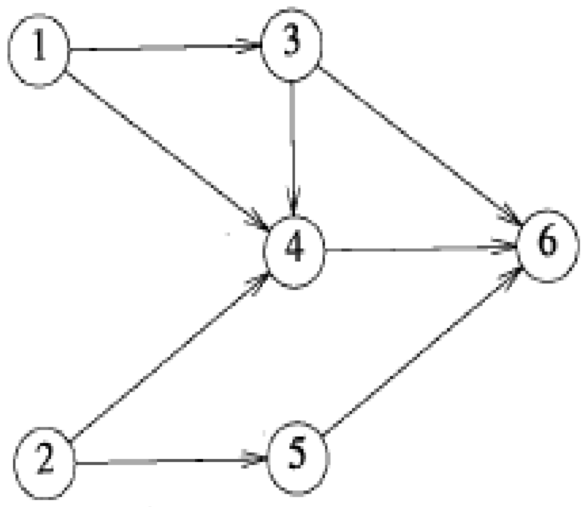
拓扑序列和拓扑排序

- **拓扑序列**(Topological orders/topological sequences):

⇒满足：对于在任务的有向图中的任一边 (i,j)，在序列中任务*i* 在任务*j* 的前面

- **拓扑排序**(Topological Sorting):

⇒根据任务的有向图建立拓扑序列的过程



- 拓扑序列:
  - 123456
  - 132456
  - 215346

拓扑排序方法

- 从一个空序列V开始，逐步构造拓扑序列;
- 每一步在排好的序列中加入一个顶点。
  - 利用如下贪婪准则来选择顶点：
  - 从剩下的顶点中，选择顶点*w*，使得*w* 不存在这样的入边(*v*,*w*)，其中顶点*v* 不在已排好的序列结构中出现。

拓扑排序算法

设*n*是有向图中的顶点数；

设*theOrder*是一个空序列；

While (*true*)

{设*w*是任意一个不存在入边(*v*,*w*)，其中顶点*v*不在*theOrder*中

如果没有这样的*w*，*break*。

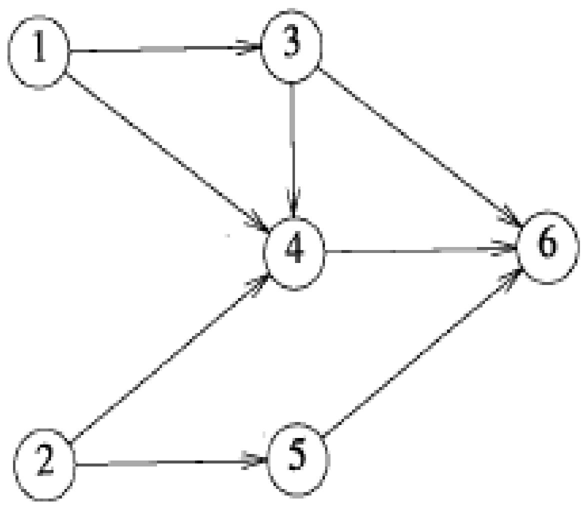
把*w*添加到*theOrder*的尾部

}

If (*theOrder*中的顶点数少于*n*) 算法失败

else *theOrder*是一个拓扑序列

拓扑排序示例



- 拓扑序列:
  – 123456
  – 251346
  – ……

贪婪算法的正确性

- 为保证贪婪算法算的正确性，需要证明：
  - 1)当算法失败时，有向图没有拓扑序列；
  - 2)若算法没有失败，V即是拓扑序列。
    - 2)即是用贪婪准则来选取下一个顶点的直接结果
    - 1)的证明:
      - 如果算法失败，则有向图含有环路。见定理17-2.
      - 若有向图中包含环$q_j q_{j+1} \cdots q_k q_j$,则它没有拓扑序列，因为该序列暗示了$q_j$一定要在$q_j$开始前完成。

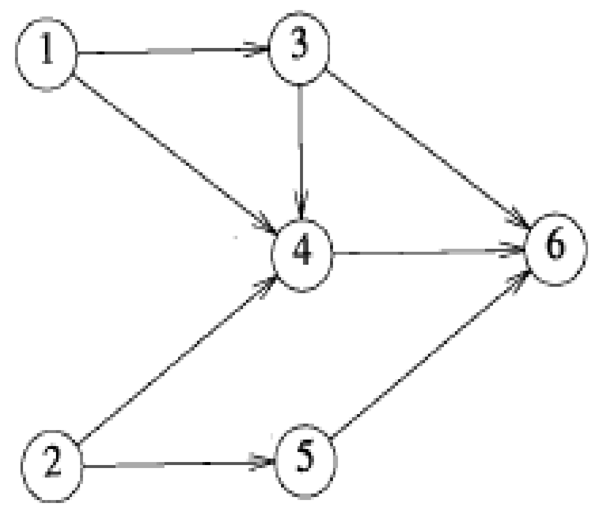
贪婪算法的正确性

- 定理17-2:如果图17-5算法失败，则有向图含有环路
- 证明:
  - 注意到当失败时|V|<n,且没有候选顶点能加入V中，因此至少有一个顶点q_1不在V中，
  - 有向图中必包含边(q_2,q_1)且q_2不在V中，否则，q_1是可加入V的候选顶点。
  - 同样，必有边(q_3, q_2)使得q_3不在V中，若q_3= q_1,则q_1 q_2 q_3是有向图中的一个环路;
  - 若$q_3 \neq q_1$，则必存在q_4使(q_4, q_3)是有向图的边且q_4不在V中，否则，q_3便是V的一个候选顶点。
  - 若q_4为q_1, q_2, q_3中的任何一个，则又可知有向图含有环，否则，因为有向图具有有限个顶点数n，继续利用上述方法，最后总能找到一个环路。

拓扑排序的实现

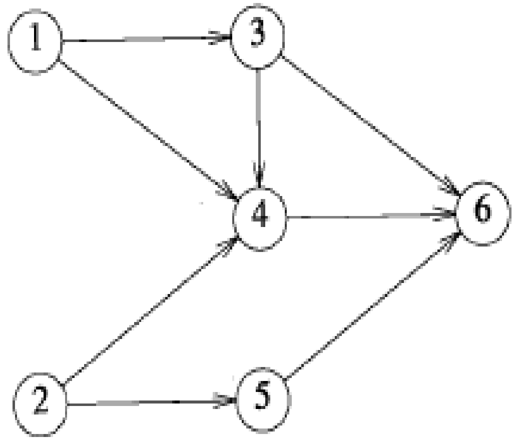
- 用一维数组*v*来描述序列V
- 用一个栈来保存可加入V的候选顶点。
- 用一维数组InDegree
  - InDegree[j]表示与顶点j相连的节点*i*的数目，其中顶点*i*不是V中的成员，它们之间的有向图的边表示为(*i*,*j*)。
- 序列V初始时为空。InDegree[*j*]为顶点*j*的入度。
- 每次向V中加入一个顶点w时，所有新加入顶点w邻接到的顶点*j*，其InDegree[*j*]减1。
- 当InDegree[*j*]变为0时表示*j*成为一个候选节点。

```
bool topologicalOrder(int *theOrder)
{ //求有向图中顶点的拓扑序列;如果找到了一个拓扑序列
   ，则返回true，此时，在theOrder[0:n-1]中记录拓扑序列;
   如果不存在拓扑序列，则返回false
   ……//确定图是有向图
   int n=numberofVertices();
   //计算入度
   int *inDegree = new int [n+1];
   fill(indegree+1, indegree+n+1, 0); //初始化
   for (i=1; i<=n; i++) {// i的出边
        vertexIterator<T> *ii=iterator(i);
        int u ;
        while ((u=ii->next())!=0) {
             inDegree[u]++;}
        }

   //把入度为 0 的顶点压入栈
   arrayStack<int> stack;
   for (i = 1; i <= n; i++)
        if (!inDegree[i]) stack.push(i);
```



// 生成拓扑序列

```
j = 0; // 数组theOrder 的索引
while (!stack.empty()) // 从堆栈中选择
  {int nextVertex= stack.top(); // 从栈中提取下一个顶点
    stack.pop();
    theOrder[j++] = nextVertex;
    //更新nextVertex邻接到的顶点的入度
    vertexIterator<T> *inextVertex =iterator(nextVertex);
    int u;
    while (u= inextVertex->next())!=0)
        { inDegree[u]--;
          If   (inDegree[u]==0)    stack.push(u);
        }
    }

return (j == n);
}
```

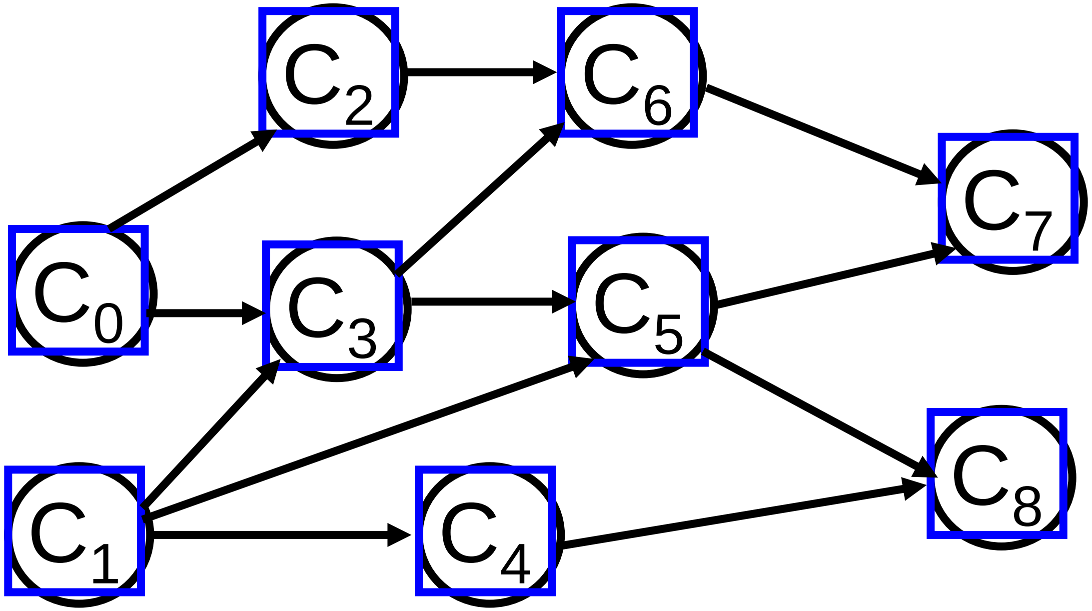


topologicalOrder的复杂性

- 使用(耗费)邻接矩阵描述：
  $\Theta(n^2)$
- 使用邻接链表描述:
  $\Theta(n+e)$

拓扑序列?

假设某专业的学生在学习期间总共要学9门课程(分别用C_0,C_1,…, C_8表示)，其中有些课程是独立于其它课程的，而另一些课程必须在学完其先修课程后才能开始。对各课程间的先后关系可以用一个AOV网表示，如下图所示。



练习

- 已知图G的邻接矩阵如下所示：$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

- 由邻接矩阵画出相应的图G；图中所有顶点是否都在它的拓扑有序序列中？