USTC

# Chapter 5
# Greedy algorithms

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn/

# 学习要点

- 理解贪心算法的概念
- 掌握贪心算法的基本要素
  - （1）最优子结构性质
  - （2）贪心选择性质
- 理解贪心算法与动态规划算法的差异
- 理解贪心算法的一般理论
- 通过应用范例学习贪心设计策略
  - （1）活动安排问题
  - （2）最优装载问题
  - （3）哈夫曼编码
  - （4）单源最短路径
  - （5）最小生成树
  - （6）多机调度问题

# 贪心算法思想

# 算法思想

顾名思义，贪心算法总是作出在当前看来最好的选择

也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择

当然，希望贪心算法得到的最终结果也是整体最优的

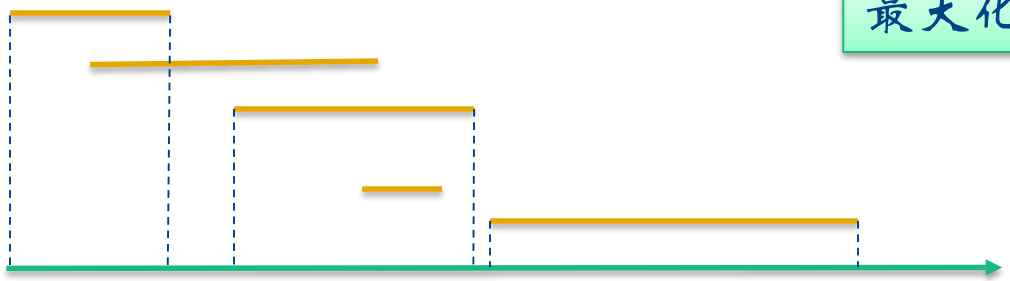
- 虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解，如：单源最短路经问题、最小生成树问题等
- 在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似

# 活动安排问题

- 活动安排问题
  - 就是要在所给的活动集合中选出最大的相容活动子集合，是可以用贪心算法有效求解的很好例子
  - 该问题要求高效地安排一系列争用某一公共资源的活动
- 贪心算法
  - 提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源

# 活动安排问题

- 设有 $n$ 个活动的集合 $E=\{1, 2, \ldots, n\}$，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源
  - 每个活动 $i$ 都有一个要求使用该资源的起始时间 $s_i$ 和一个结束时间 $f_i$，且 $s_i < f_i$，如果选择了活动 $i$，则它在半开时间区间 $[s_i, f_i)$ 内占用资源
  - 若区间 $[s_i, f_i)$ 与区间 $[s_j, f_j)$ 不相交，则称活动 $i$ 与活动 $j$ 是相容的也就是说，当 $s_i \geq f_j$ 或 $s_j \geq f_i$ 时，活动 $i$ 与活动 $j$ 相容
- 最大相容子集合
  - 0-1选择 (无序)



最大化活动数：3

# 活动安排问题

- 动态规划
  - 定义 $m[i,t]$ 为活动$[i{:}n]$在时间 $t$ 之后的最大相容活动数
  - 如果活动 $i$ 在 $t$ 之前是不允许的，要求活动$[i{+}1{:}n]$都是不允许的
    - 活动按结束时间 $f_i$ 递增排序
  - 递归关系

$$m[i,t] = \max(m[i+1,t], m[i+1,f_i]+1)$$

- 复杂度
  - $T(n) = O(\min(nT, n2^n))$
  - $O(n)$ ?
- 贪心选择策略?

# 活动安排问题

下面给出解活动安排问题的贪心算法 **GreedySelector**：

```
1.  template<class Type>
2.  void GreedySelector(int n, Type s[], Type f[], bool A[])
3.  {
4.        A[1]=true;
5.        int j=1;
6.        for (int i=2;i<=n;i++) {
7.          if (s[i]>=f[j]) { A[i]=true; j=i; }
8.          else A[i]=false;
9.          }
10. }
```

各活动的起始时间和结束时间存储于数组 s 和 f 中且按结束时间的非减序排列

# 活动安排问题

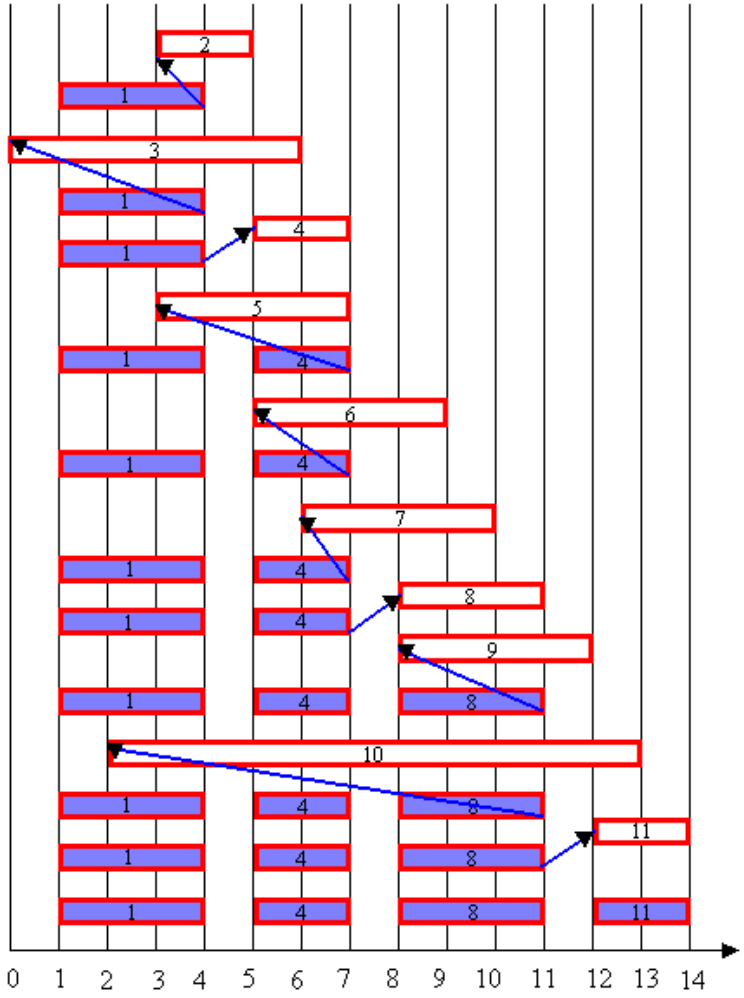
- 由于输入的活动以其完成时间的非减序排列，所以算法greedySelector每次总是选择具有最早完成时间的相容活动加入集合A中。直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间，也就是说，该算法的贪心选择的意义是使剩余的可安排时间段极大化，以便安排尽可能多的相容活动
- 算法greedySelector的效率极高
  - 当输入的活动已按结束时间的非减序排列，算法只需$O(n)$的时间安排$n$个活动，使最多的活动能相容地使用公共资源
  - 如果所给出的活动未按非减序排列，可以用$O(n\log n)$的时间重排

# 活动安排问题

例：设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| S[i] | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| f[i] | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# 活动安排问题




算法 greedySelector 的计算过程如左图所示

图中每行相应于算法的一次迭代，阴影长条表示的活动是已选入集合 $A$ 的活动，而空白长条表示的活动是当前正在检查相容性的活动

# 活动安排问题

若被检查的活动 $i$ 的开始时间 $s_i$ 小于最近选择的活动 $j$ 的结束时间 $f_i$，则不选择活动 $i$，否则选择活动 $i$ 加入集合 $A$ 中

贪心算法并不总能求得问题的**整体最优解**，但对于活动安排问题，贪心算法 greedySelector 却总能求得的整体最优解，即它最终所确定的相容活动集合$A$的规模最大

——这个结论可以用**数学归纳法**证明(贪心选择性质)

# 贪心算法的基本要素

现在着重讨论可以用贪心算法求解问题的一般特征

对于一个具体的问题，怎么知道是否可用贪心算法解此问题，以及能否得到问题的最优解呢?这个问题很难给予肯定的回答

但是，从许多可以用贪心算法求解的问题中看到这类问题一般具有2个重要的性质：贪心选择性质和最优子结构性质

# 贪心算法的基本要素

## 1、贪心选择性质

所谓**贪心选择性质**是指所求问题的**整体最优解**可以通过一系列**局部最优**的选择，即贪心选择来达到——这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别

动态规划算法通常以**自底向上**的方式解各子问题，而贪心算法则通常以**自顶向下**的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题

对于一个具体问题，要确定它是否具有贪心选择性质，必须**证明**每一步所作的贪心选择最终导致问题的整体最优解

# 4.2 贪心算法的基本要素

## 2、最优子结构性质

当一个问题的最优解包含其子问题的最优解时，称此问题具有**最优子结构性质**

——问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征

# 贪心算法的基本要素

## 3、贪心算法与动态规划算法的差异

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是两类算法的一个共同点。但是，对于具有**最优子结构**的问题应该选用贪心算法还是动态规划算法求解?是否能用动态规划算法求解的问题也能用贪心算法求解?

下面研究两个经典的**组合优化问题**，并以此说明贪心算法与动态规划算法的主要差别

# 贪心算法的基本要素

## ❖0-1背包问题：

给定 $n$ 种物品和一个背包，物品 $i$ 的重量是 $w_i$，其价值为 $v_i$，背包的容量为 $C$

应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

在选择装入背包的物品时，对每种物品 $i$ 只有 2 种选择，即装入背包或不装入背包，不能将物品 $i$ 装入背包多次，也不能只装入部分的物品 $i$

# 贪心算法的基本要素

## ❖ 背包问题：

与0-1背包问题类似，所不同的是在选择物品 $i$ 装入背包时，可以选择物品 $i$ 的一部分，而不一定要全部装入背包，$1 \le i \le n$

这 2 类问题都具有**最优子结构**性质，极为相似，但背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解

# 贪心算法的基本要素

用贪心算法解背包问题的基本步骤：

首先计算每种物品单位重量的价值 $v_i/w_i$，然后，依贪心选择策略，将尽可能多的**单位重量价值最高**的物品装入背包

若将这种物品全部装入背包后，背包内的物品总重量未超过 $C$，则选择单位重量价值次高的物品并尽可能多地装入背包；依此策略一直地进行下去，直到背包装满为止

# 贪心算法的基本要素

## 具体算法可描述：

```
1.  void Knapsack(int n,float M,float v[],float w[],float x[])
2.  {
3.      Sort(n,v,w);
4.      int i;
5.      for (i=1;i<=n;i++) x[i]=0;
6.      float c=M;
7.      for (i=1;i<=n;i++) {
8.        if (w[i]>c) break;
9.        x[i]=1;
10.       c-=w[i];
11.       }
12.     if (i<=n) x[i]=c/w[i];
13. }
```

算法knapsack的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为O（*n*log*n*）

为了证明算法的正确性，必须证明背包问题具有贪心选择性质

# 贪心算法的基本要素

对于**0-1背包问题**，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。事实上，在考虑0-1背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，然后再作出最好选择，由此就导出许多互相重叠的子问题——这正是该问题可用**动态规划算法**求解的另一重要特征，实际上也是如此，动态规划算法的确可以有效地解0-1背包问题

# 最优装载和哈夫曼编码

# 最优装载

有一批集装箱要装上一艘载重量为 $c$ 的轮船，其中集装箱 $i$ 的重量为 $w_i$，最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船

## 1、算法描述

最优装载问题可用贪心算法求解：采用重量最轻者先装的贪心选择策略，可产生最优装载问题的最优解

具体算法描述如下页

# 最优装载

```
1.  template<class Type>
2.  void Loading(int x[],  Type w[], Type c, int n)
3.  {
4.          int *t = new int [n+1];
5.          Sort(w, t, n);
6.          for (int i = 1; i <= n; i++) x[i] = 0;
7.          for (int i = 1; i <= n && w[t[i]] <= c; i++) {x[t[i]] = 1; c -= w[t[i]];}
8.  }
```

# 最优装载

## 2、贪心选择性质

变换目标后可以证明最优装载问题具有贪心选择性质(归纳法)

## 3、最优子结构性质

最优装载问题具有最优子结构性质(反证法)

由最优装载问题的贪心选择性质和最优子结构性质，容易证明算法**loading**的正确性

算法**loading**的主要计算量在于将集装箱依其重量从小到大排序，故算法所需的计算时间为 $O(n\log n)$

# 哈夫曼编码

**哈夫曼编码**是广泛地用于数据文件压缩的十分有效的编码方法，其压缩率通常在20%～90%之间。哈夫曼编码算法用字符在文件中出现的频率表来建立一个用 0,1 串表示各字符的最优表示方式

给出现频率高的字符较短的编码，出现频率较低的字符以较长的编码，可以大大缩短总码长

## 1、前缀码

对每一个字符规定一个0,1串作为其代码，并要求任一字符的代码都不是其它字符代码的前缀，这种编码称为**前缀码**

# 哈夫曼编码

编码的前缀性质可以使译码方法非常简单!

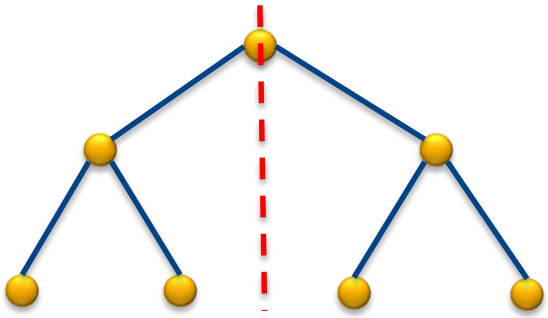
表示**最优前缀码**的二叉树总是一棵**完全二叉树**，即树中任一结点都有 2 个儿子结点

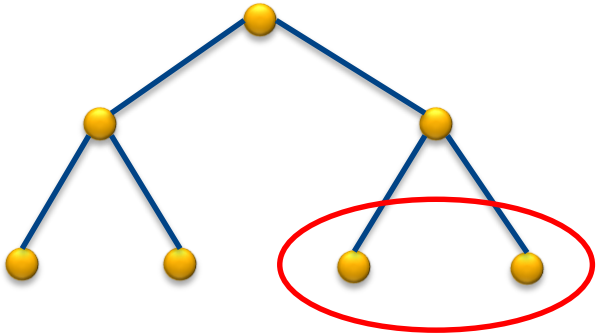
平均码长定义为：$$B(T)=\sum_{c\in C} f(c)d_T(c)$$

使平均码长达到最小的前缀码编码方案称为给定编码字符集 $C$ 的**最优前缀码**

# 哈夫曼编码

## 划分方式





**子树划分**

将字符集划分为两个集合，共有$2^n$种划分方法——划分动态性

**节点合并划分**

将字符集个数减一，选择两个字符进行操作——选择动态性

# 哈夫曼编码

## 2、构造哈夫曼编码

哈夫曼提出构造最优前缀码的贪心算法，由此产生的编码方案称为**哈夫曼编码**

哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树 $T$

算法以 $|C|$ 个叶结点开始，执行 $|C|-1$ 次的“合并”运算后产生最终所要求的树 $T$

# 哈夫曼编码

在书上给出的算法 huffmanTree 中，编码字符集中每一字符 $c$ 的频率是 $f(c)$。以$f$为键值的优先队列Q用在贪心选择时有效地确定算法当前要合并的 2 棵具有最小频率的树;一旦2棵具有最小频率的树合并后，产生一棵新的树，其频率为合并的2棵树的频率之和，并将新树插入优先队列Q;经过$n-1$次的合并后，优先队列中只剩下一棵树，即所要求的树$T$。

算法 huffmanTree 用最小堆实现优先队列Q。初始化优先队列需要$O(n)$计算时间，由于最小堆的 removeMin和put 运算均需$O(\log n)$时间，$n-1$次的合并总共需要$O(n\log n)$ 计算时间，因此，关于$n$个字符的哈夫曼算法的计算时间为$O(n\log n)$

# 哈夫曼编码

## 3、哈夫曼算法的正确性

要证明哈夫曼算法的正确性，只要证明最优前缀码问题具有贪心选择性质和最优子结构性质

(1)贪心选择性质

归纳法

(2)最优子结构性质

反证法

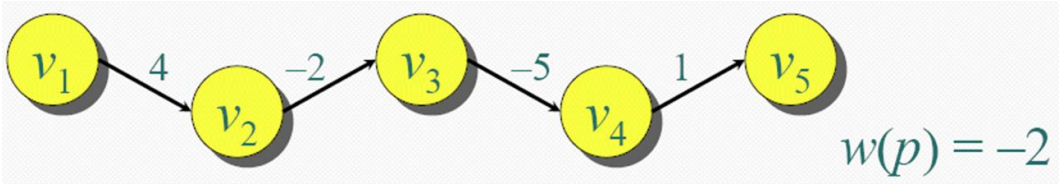
# 单源最短路径

# 最短路径

## 路径及其权重

- 给定一个图$G=(V, E)$及其上的边权函数$w: E \rightarrow R$
- 路径$p=v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$上的权重定义为

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$



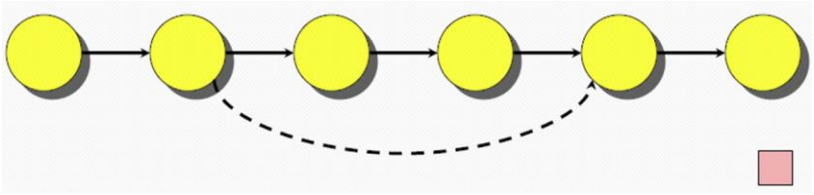

## 最短路径

- 从顶点$u$到$v$的所有路径中具有最小权重的路径
- $\delta(u, v) = \min\{w(p): P \text{ is a path from } u \text{ to } v\}$
- 没有路径的两个顶点u和v，有$\delta(u, v) = \infty$

# 最短路径

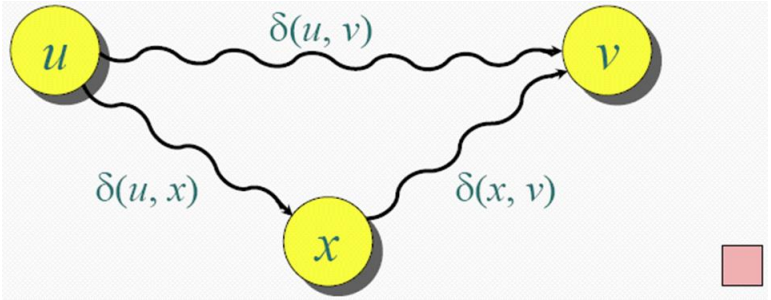
## 最短路径性质

- 最优子结构性质
  - 一个最短路径的子路径也是最短路径



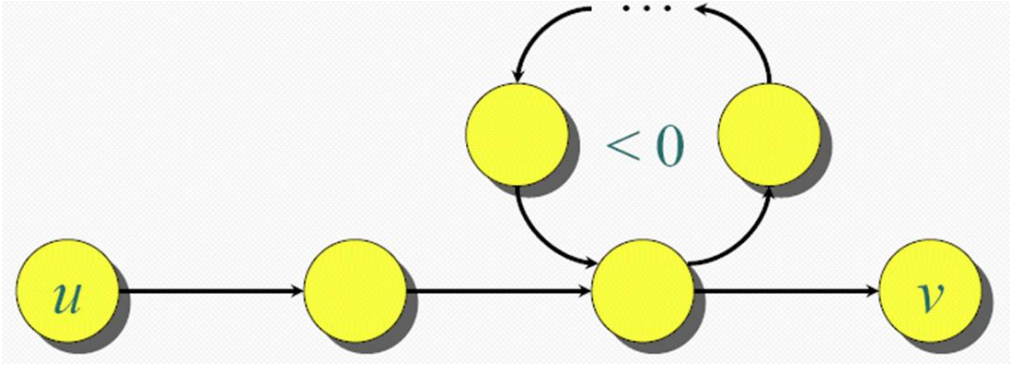
- 三角不等式
  - 对任意三个顶点$u, v, x$，有：

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v)$$



# 最短路径

- 最短路径的定义说明
  - 如果图$G=(V, E)$上存在一个负权重的环，则一些节点间的最短路径可能不存在
  - 示例



# 单源最短路径

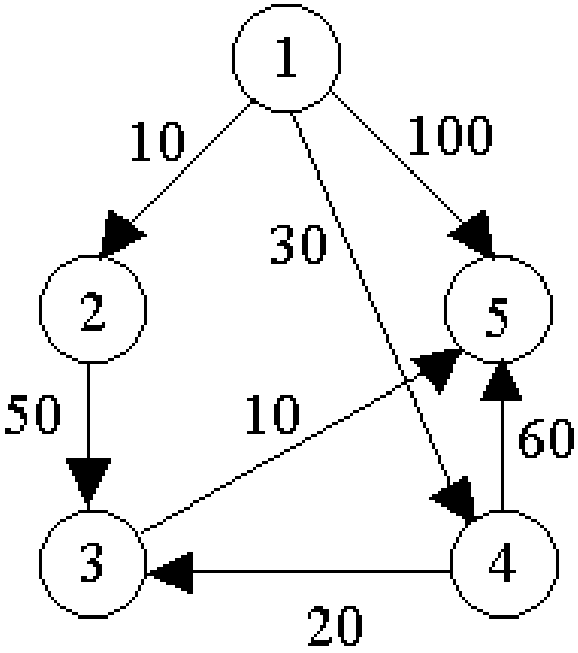
- 单源最短路径问题
  - 给定有向带权图 $G=(V,E)$，其中每条边的权是非负实数
  - 给定 $V$ 中的一个顶点 s（称为源），现在要计算从源到所有其它各顶点 $v$ 的最短路径长度 $\delta(u,v)$
- 算法基本思想——贪心
  - 维护一个顶点集合 $S$，它们从原点 s 的最短路径已知
  - 每一步添加 $v \in V-S$ 中具有最小距离的顶点到 $S$ 中
  - 更新剩余顶点 $v$ 的距离估计

# 单源最短路径

- Dijkstra算法
  - 是解单源最短路径问题的贪心算法
- 过程
  - 初始时，$S$中仅含有源
  - 设$v$是$G$的某一个顶点，把从源到$v$且中间只经过$S$中顶点的路称为从源到$v$的特殊路径，并用数组dist记录当前每个顶点所对应的最短特殊路径长度
  - 每次从$V$-$S$中取出具有最短特殊路长度的顶点$v$，将$v$添加到$S$中，同时对数组dist作必要的修改
  - 一旦$S$包含了所有$V$中顶点，dist就记录了从源到所有其它顶点之间的最短路径长度

# 单源最短路径

例如，对右图中的有向图，应用 Dijkstra 算法计算从源顶点1到其它顶点间最短路径的过程列在下页的表中



# 单源最短路径

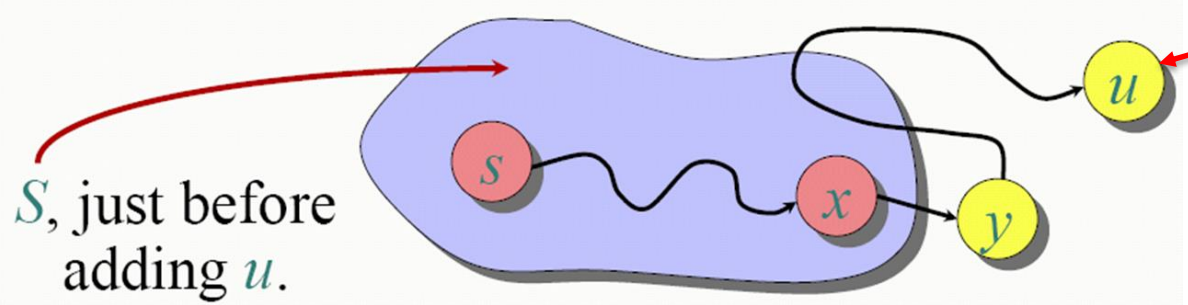
Dijkstra 算法的迭代过程：

| 迭代 | S | u | dist[2] | dist[3] | dist[4] | dist[5] |
|---|---|---|---|---|---|---|
| 初始 | {1} | - | 10 | maxint | 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 |

# 单源最短路径

## ❖ 算法的正确性

- 初始化时 $d[s]=0, d[v]=\infty$ for $v \in V-\{s\}$
- 循环不变式：$d[v] \geq \delta(s,v)$ ——反证法
- 对所有的节点，Dijkstra算法在$d[v]=\delta(s,v)$ 时收敛
  - 反正：假设$u$是第一个违反点
  - $d[u] \leq d[y]$ && $d[u] \geq d[y] \longrightarrow d[y]=d[u]$



# Dijkstra 复杂度分析

$|V|$ times

**while** $Q \neq \varnothing$

**do** $u \leftarrow$ EXTRACT-MIN($Q$)

$S \leftarrow S \cup \{u\}$

$degree(u)$ times

**for** each $v \in Adj[u]$

**do if** $d[v] > d[u] + w(u, v)$

**then** $d[v] \leftarrow d[u] + w(u, v)$

握手定理 $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

$$T(G) = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

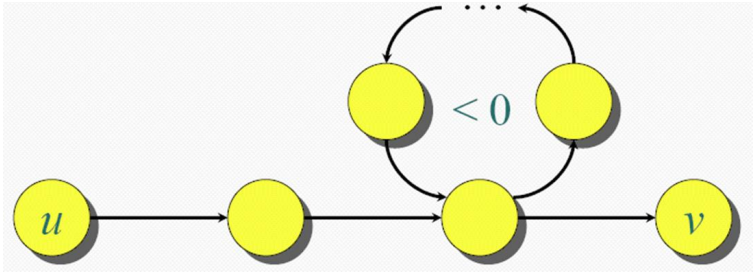
# Dijkstra 复杂度分析

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

| $Q$ | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total |
|---|---|---|---|
| array | $O(V)$ | $O(1)$ | $O(V^2)$ |
| binary heap | $O(\lg V)$ | $O(\lg V)$ | $O(E \lg V)$ |
| Fibonacci heap | $O(\lg V)$ amortized | $O(1)$ amortized | $O(E + V \lg V)$ worst case |

# 最短路径

- 最短路径的定义说明
  - 如果图$G=(V, E)$上存在一个负权重的环，则一些节点间的最短路径可能不存在




- Bellman-Ford 算法
  - 针对给定的 $G=(V, E)$，找到从源节点 s到所有其他节点的最短路径或判别出存在一个负权重环

# Bellman-Ford 算法

$d[s] \leftarrow 0$
**for** each $v \in V - \{s\}$
  **do** $d[v] \leftarrow \infty$

} initialization

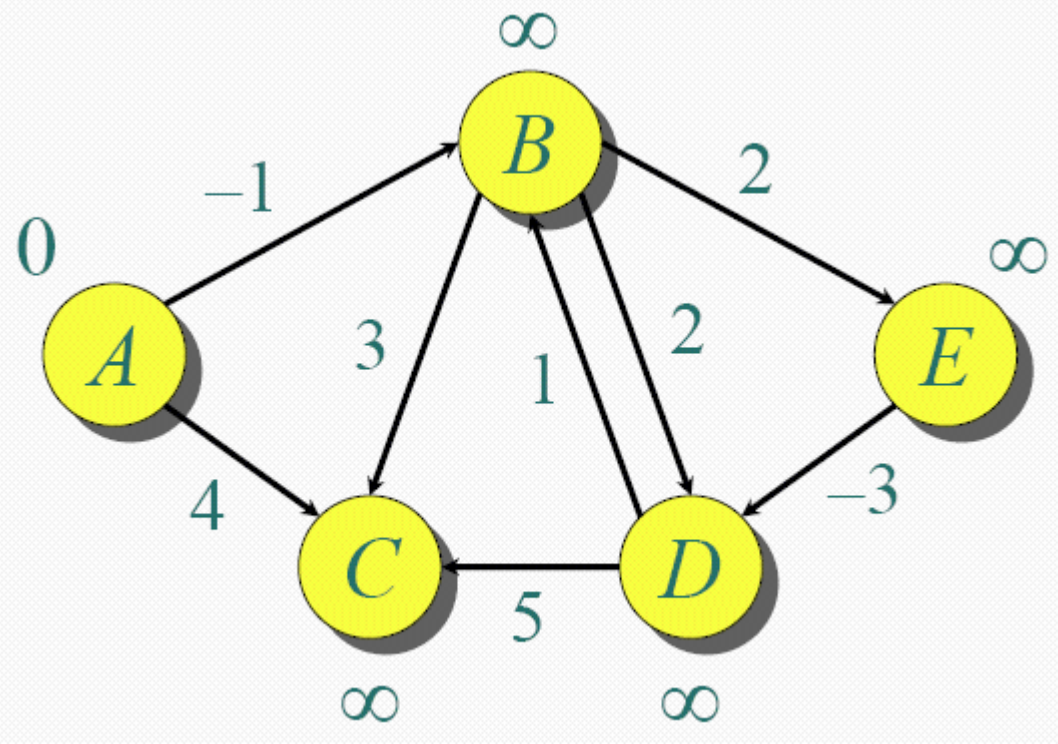
**for** $i \leftarrow 1$ **to** $|V| - 1$
  **do for** each edge $(u, v) \in E$
    **do if** $d[v] > d[u] + w(u, v)$
      **then** $d[v] \leftarrow d[u] + w(u, v)$

*relaxation step*

**for** each edge $(u, v) \in E$
  **do if** $d[v] > d[u] + w(u, v)$
    **then** report that a negative-weight cycle exists

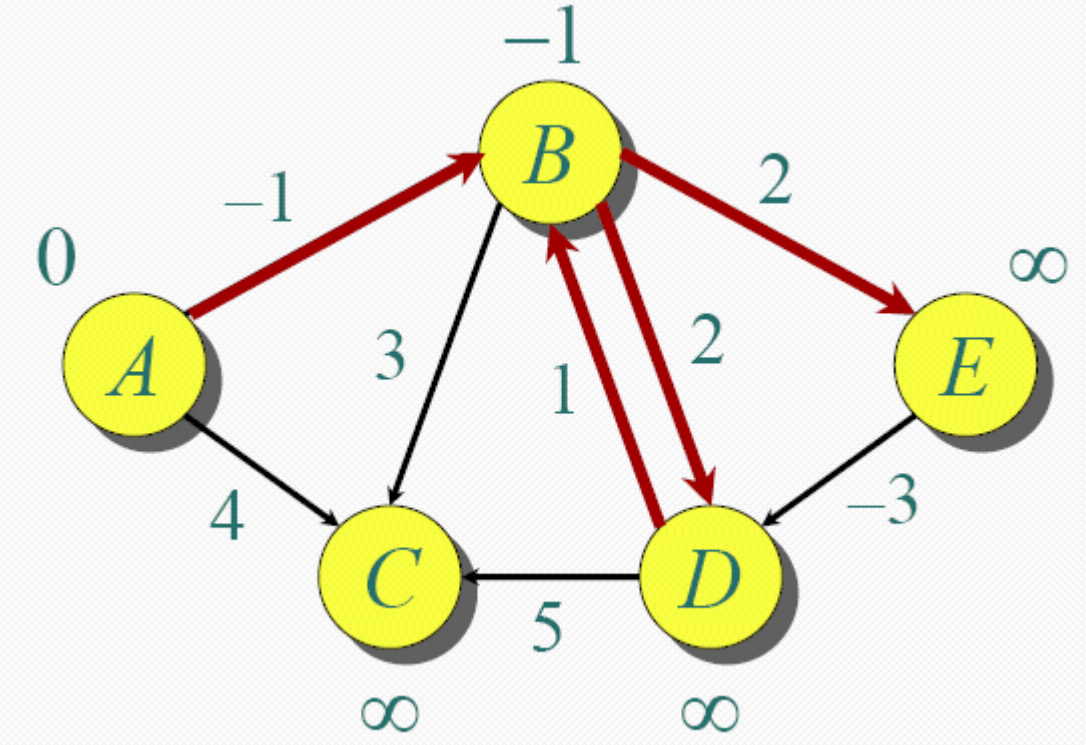
At the end, $d[v] = \delta(s, v)$. Time = $O(VE)$

# Bellman-Ford 示例



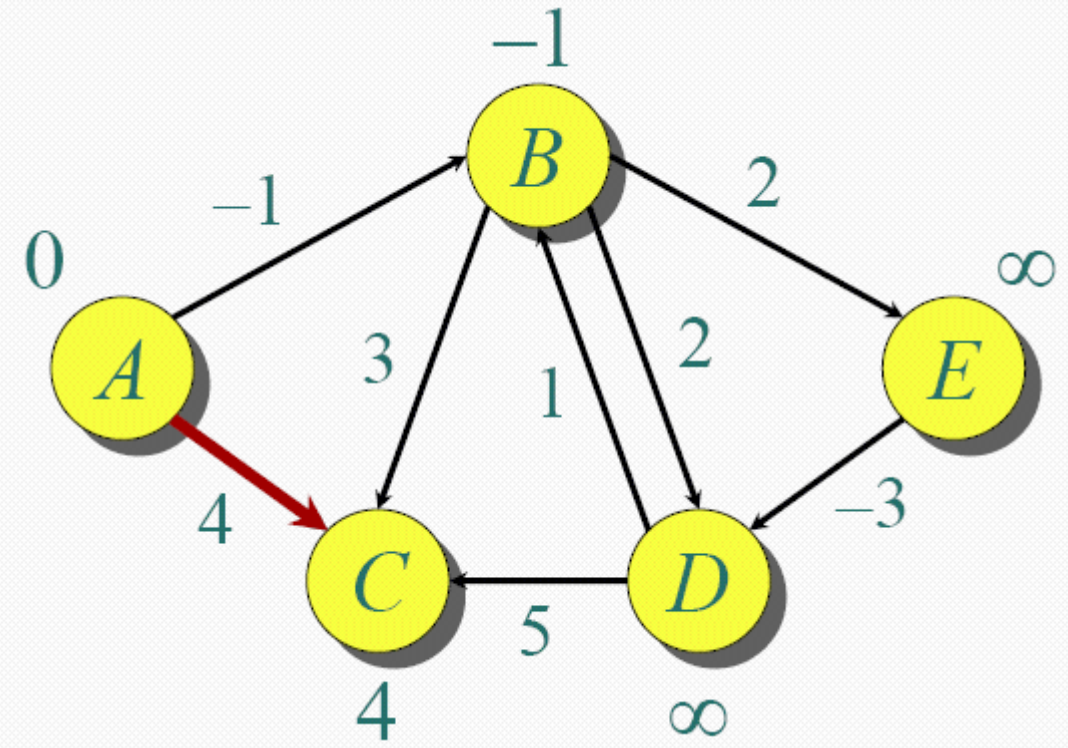

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |

# Bellman-Ford 示例



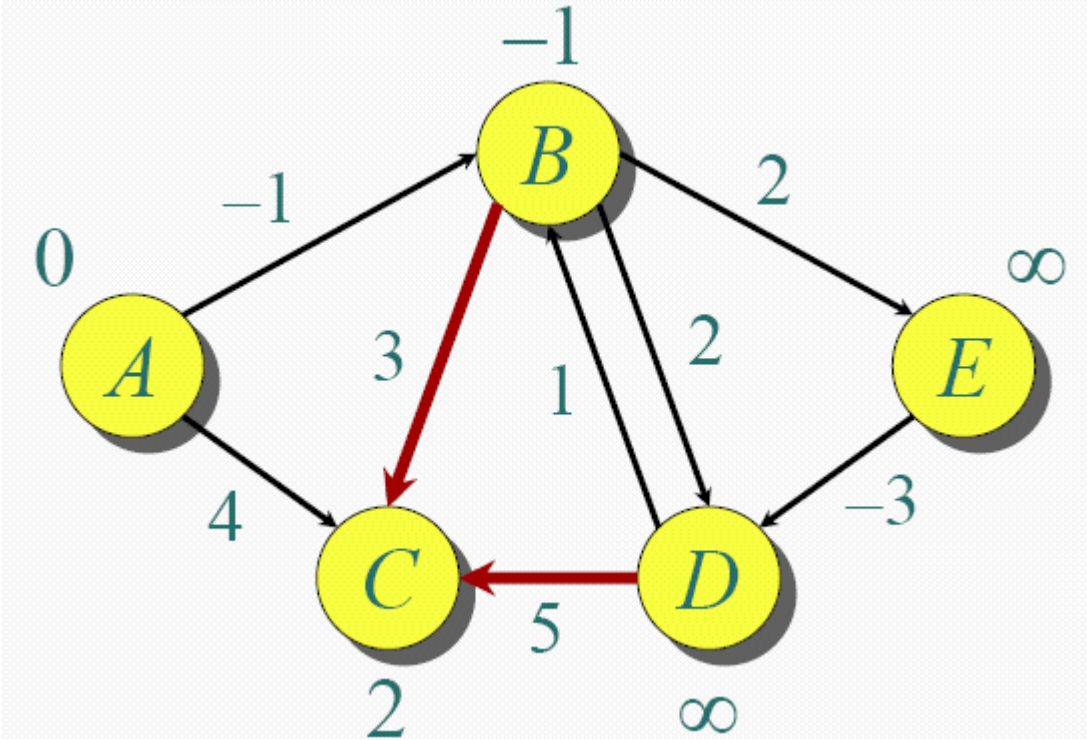

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | –1 | ∞ | ∞ | ∞ |

# Bellman-Ford 示例



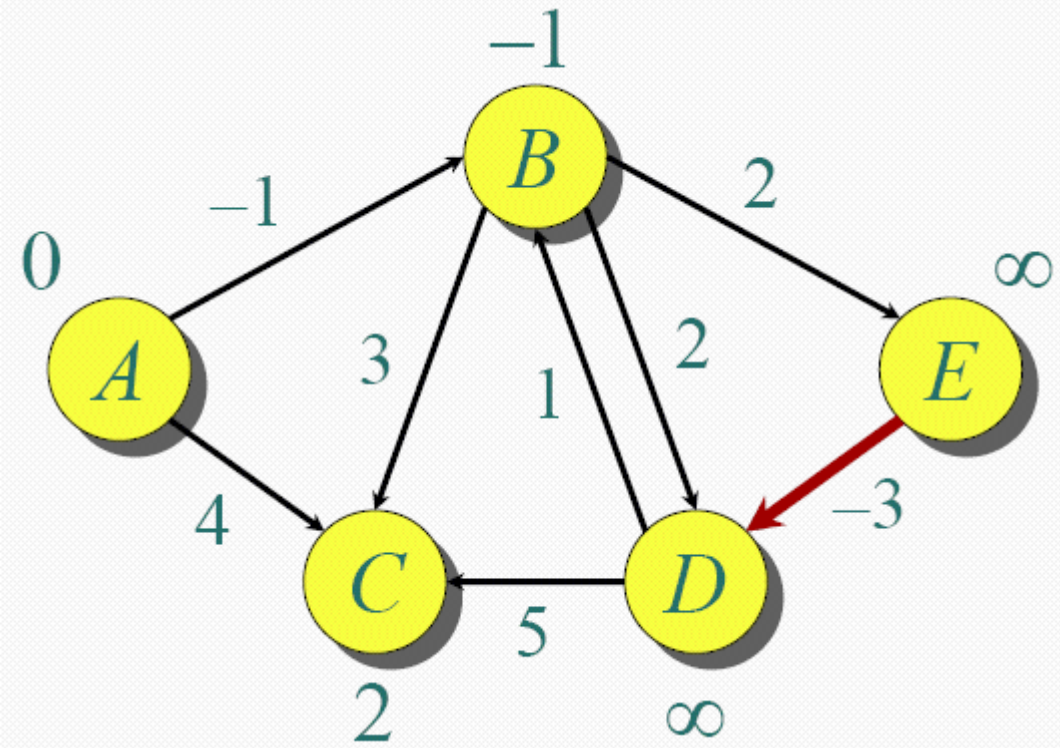

| $A$ | $B$ | $C$ | $D$ | $E$ |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | −1 | ∞ | ∞ | ∞ |
| 0 | −1 | 4 | ∞ | ∞ |

# Bellman-Ford 示例



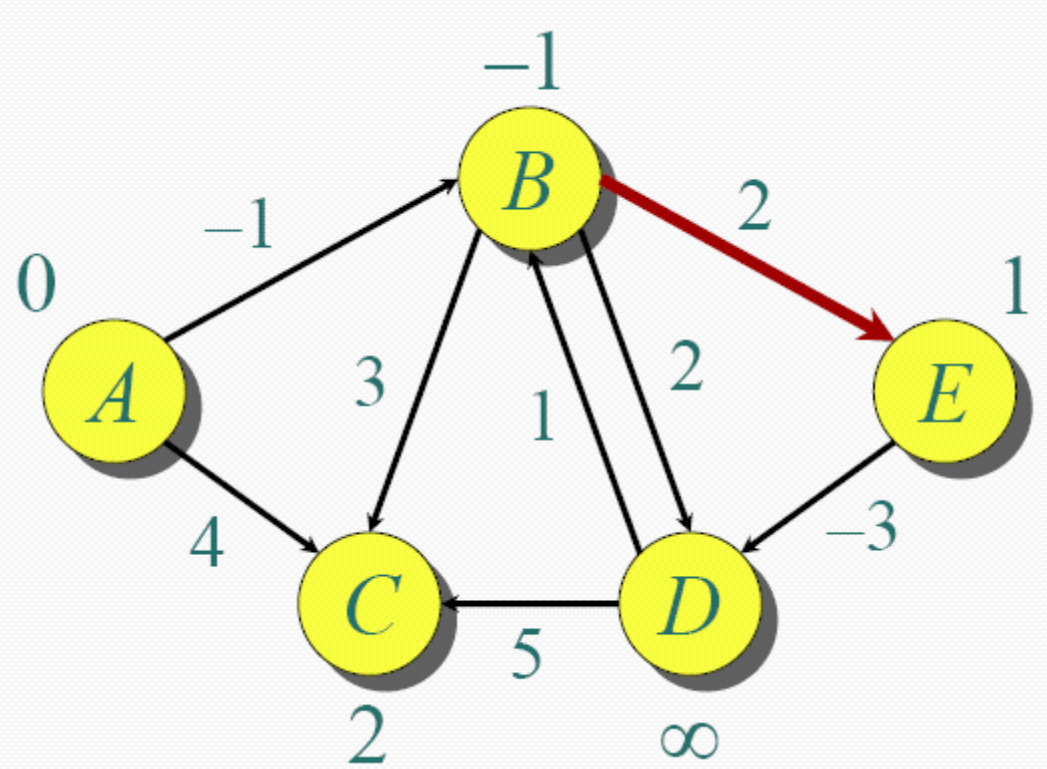

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | –1 | ∞ | ∞ | ∞ |
| 0 | –1 | 4 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | ∞ |

# Bellman-Ford 示例



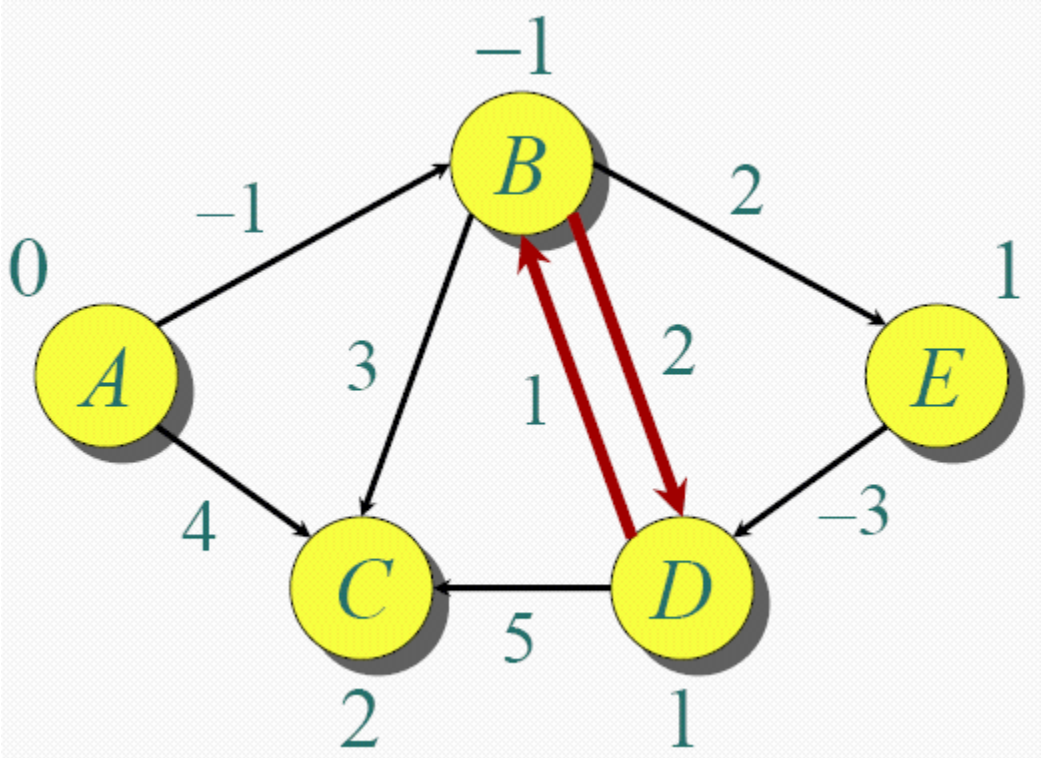

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | –1 | ∞ | ∞ | ∞ |
| 0 | –1 | 4 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | ∞ |

# Bellman-Ford 示例



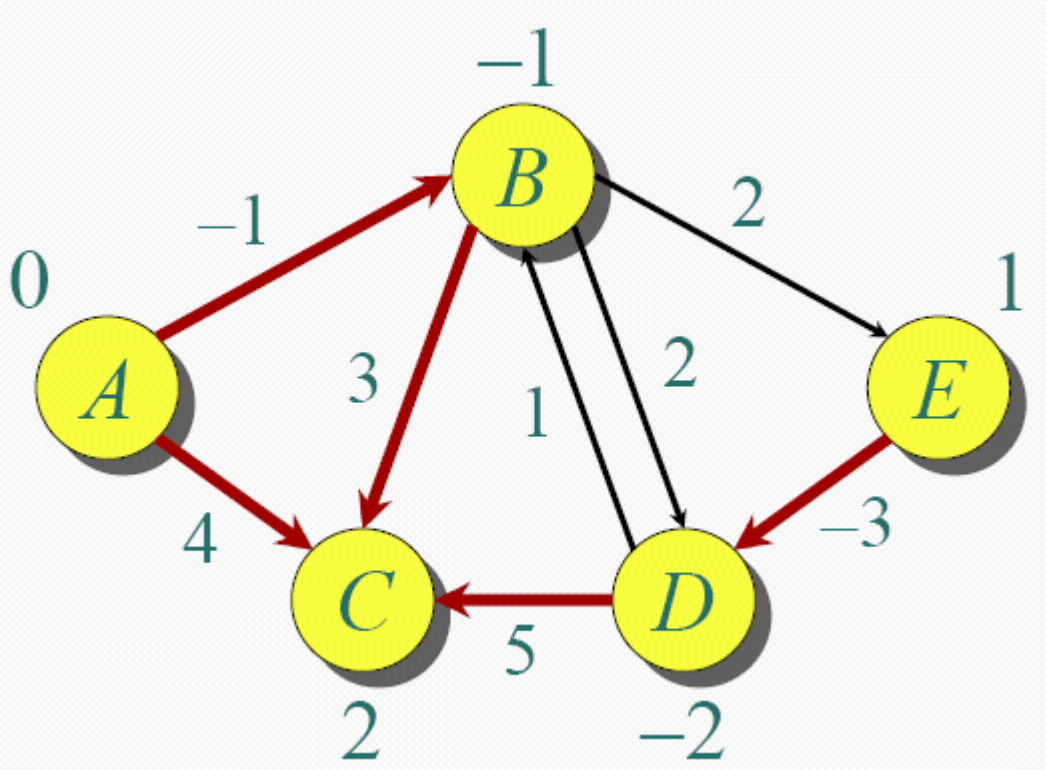

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | –1 | ∞ | ∞ | ∞ |
| 0 | –1 | 4 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | 1 |

# Bellman-Ford 示例



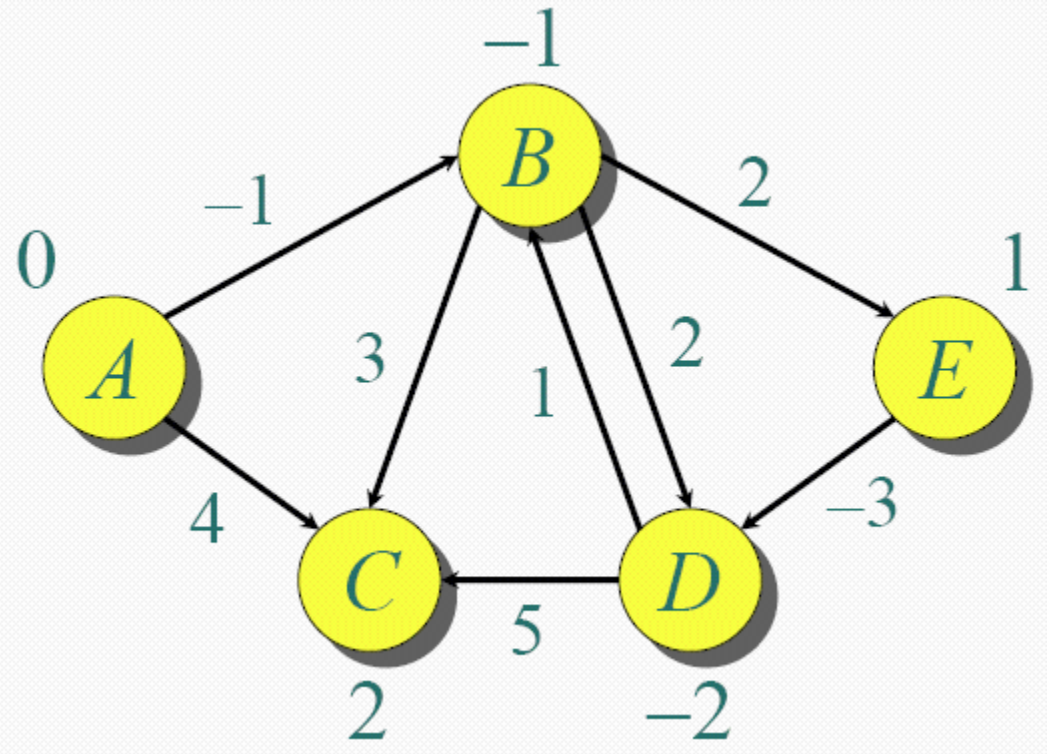

| $A$ | $B$ | $C$ | $D$ | $E$ |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | –1 | ∞ | ∞ | ∞ |
| 0 | –1 | 4 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | 1 |
| 0 | –1 | 2 | 1 | 1 |

# Bellman-Ford 示例




| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | −1 | ∞ | ∞ | ∞ |
| 0 | −1 | 4 | ∞ | ∞ |
| 0 | −1 | 2 | ∞ | ∞ |
| 0 | −1 | 2 | ∞ | 1 |
| 0 | −1 | 2 | 1 | 1 |
| 0 | −1 | 2 | −2 | 1 |

# Bellman-Ford 示例




| $A$ | $B$ | $C$ | $D$ | $E$ |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | –1 | ∞ | ∞ | ∞ |
| 0 | –1 | 4 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | ∞ |
| 0 | –1 | 2 | ∞ | 1 |
| 0 | –1 | 2 | 1 | 1 |
| 0 | –1 | 2 | –2 | 1 |

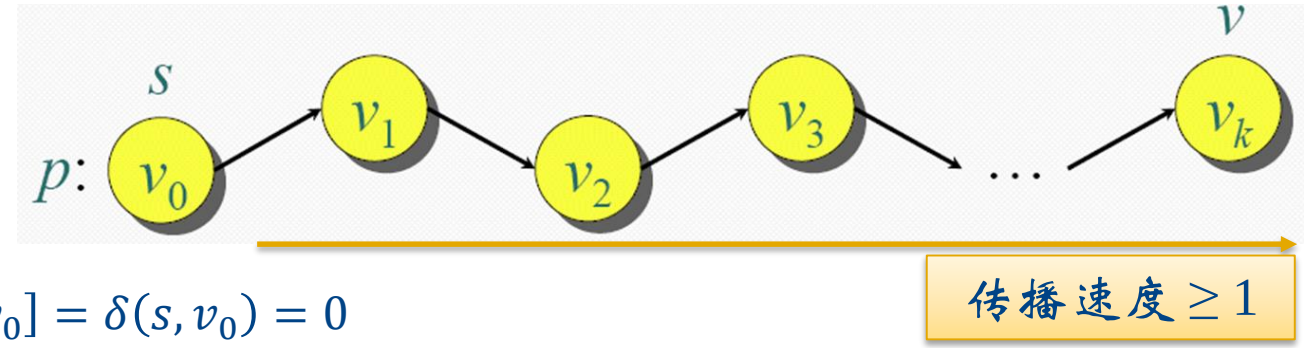
**Note:** Values decrease monotonically !

# Bellman-Ford 算法

- 正确性
  - 如果图$G=(V,E)$没有负权重环，则BF算法执行后对所有的节点$v$具有$d[v]=\delta(s,v)$
  - 证明：
    - $v$是任一节点，$p$是其最短路径

$$\delta(s,v_i)=\delta(s,v_{i-1})+w(v_{i-1},v_i)$$

    - 最长简单路径最多具有$|V|-1$个结点



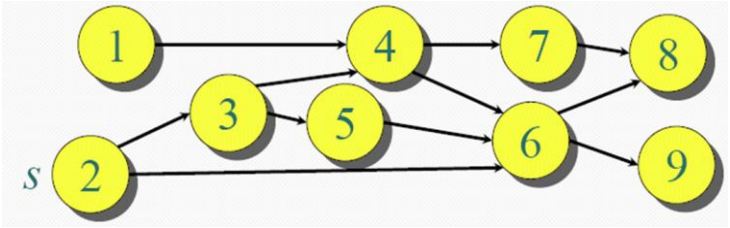

$d[v_0]=\delta(s,v_0)=0$

# Bellman-Ford 算法

- 正确性
  - 如果图$G=(V,E)$没有负权重环，则BF算法执行后对所有的节点$v$具有$d[v]=\delta(s,v)$
- 推论
  - 如果一个节点的$d[v]$在经过$|V|-1$次传递后仍不能收敛，则$G$中一定存在一个负权重环

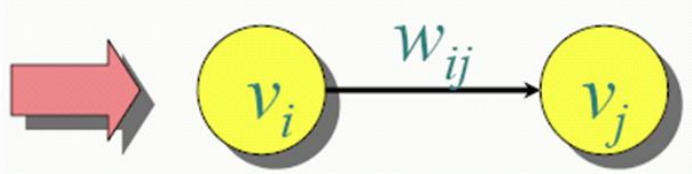
# Bellman-Ford 算法

- 有向无环图 DAG (directed acyclic graph)
  - 深度遍历下 $O(V+E)$
  - One-pass BF算法



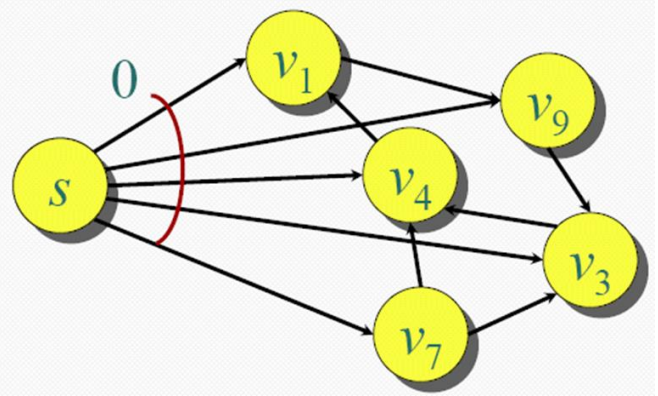

- 线性规划
  - 差分约束
    - 系数中一个 1 和一个 -1，其他都为 0
  - 约束图

$$x_j - x_i \le w_{ij}$$



# Bellman-Ford 算法

- BF算法能够求解一个 $n$ 变量 $m$ 差分约束的线性规划问题，复杂度为 $O(mn)$
  - 无负权重环，否则无解
  - $x_i = \delta(s, v_i)$
  - 三角不等式→差分约束




- 形式变换
  - BF算法实际上是最大化 $\sum_{i=1}^{n} x_i$，满足 $x_j - x_i \leq w_{ij}, x_i \leq 0$
  - BF算法也最小化了 $\max_i\{x_i\} - \min_i\{x_i\}$

# 最短路径算法总结

- 单源最短路径 Single-source shortest paths
  - 边权重非负
    - Dijkstra's algorithm: $O(E + V \lg V)$
  - 通用算法
    - Bellman-Ford: $O(VE)$
  - DAG
    - One pass of Bellman-Ford: $O(V + E)$
- 所有节点对的最短路径 All-pairs shortest paths
  - 边权重非负
    - Dijkstra's algorithm $|V|$ times: $O(VE + V\,2 \lg V)$
  - 通用方法
    - 参考CLRS Chapter 25 (三种方法)

# 最小生成树

**MST**

# 最小生成树

## 最小生成树 Minimum spanning tree

设 $G=(V, E)$ 是无向连通带权图，即一个网络，$E$ 中每条边 $(v,w)$ 的权为 $c[v][w]$。如果 $G$ 的子图 $G'$ 是一棵包含 $G$ 的所有顶点的树，则称 $G'$ 为 $G$ 的生成树，生成树上各边权的总和称为该生成树的耗费。在 $G$ 的所有生成树中，耗费最小的生成树称为 $G$ 的最小生成树

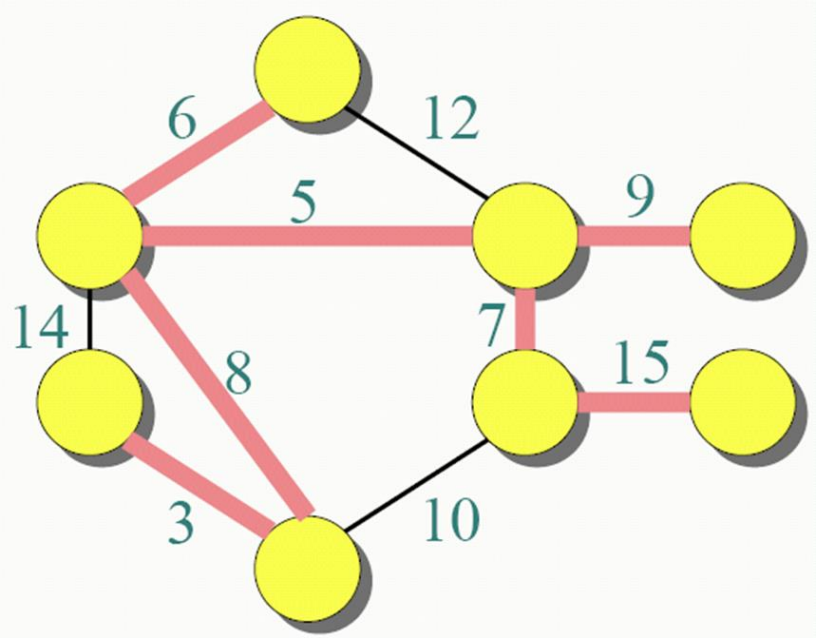
网络的最小生成树在实际中有广泛应用。例如，在设计通信网络时，用图的顶点表示城市，用边 $(v,w)$ 的权 $c[v][w]$ 表示建立城市 $v$ 和城市 $w$ 之间的通信线路所需的费用，则最小生成树就给出了建立通信网络的最经济的方案

# 最小生成树

- 优化目标

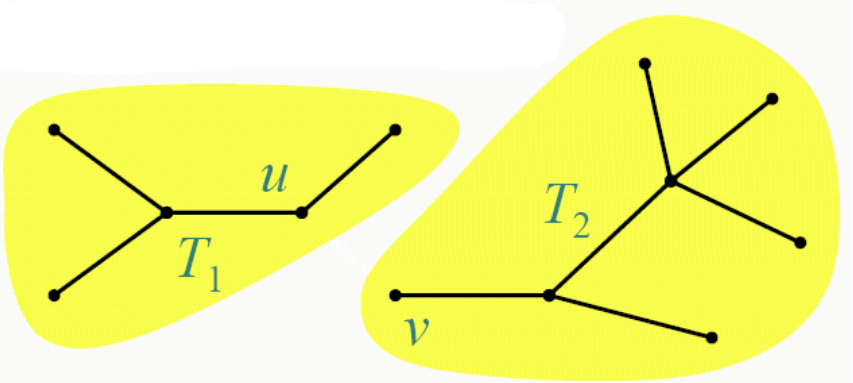
$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

- 划分方法
  - $n$ 个点必定是 $n$-1 个边
  - 从无到有方向的选择策略
    - 点选择
    - 边选择
  - 规模每次降 1



# MST性质

- 最优子结构
  - 假设MST为 $T$,删除 $(u,v)$ 后 $T$ 被分割成两个子树 $T_1$ 和 $T_2$
  - 子树 $T_1$是$G$子图$G_1=(V_1, E_1)$的MST，其中 $V_1$ 是 $T_1$ 的顶点，$E_1 = \{(x,y) \in E: x,y \in V_1\}$
  - $T_2$ 有相同的结论



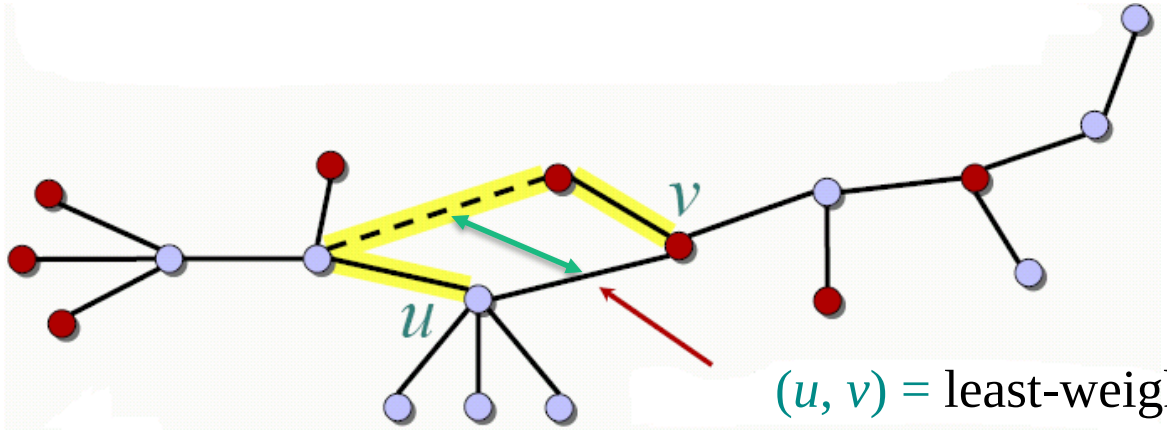

- 证明
  - 目标是

$$w(T) = w(u,v) + w(T_1) + w(T_2)$$

  - 加法具有子结构性质
- 重叠子问题 → 动态规划？

# MST性质

- 贪心选择性质
  - 设 $G=(V, E)$ 是连通带权图，$U$ 是 $V$ 的真子集。如果$(u, v)\in E$，且 $u\in U$，$v\in V-U$，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
  - 证明




- 考虑 $u$ 到 $v$ 的简单路径
- 交换 $(u, v)$和该路径上的 $U, V-U$ 连接线

# 最小生成树

- 用贪心算法设计策略可以设计出有效算法
  - 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
  - 这 2 个算法做贪心选择的方式不同，它们都利用了最小生成树性质
    - Prim：点选择
    - Kruskal：边选择

# Prim最小生成树

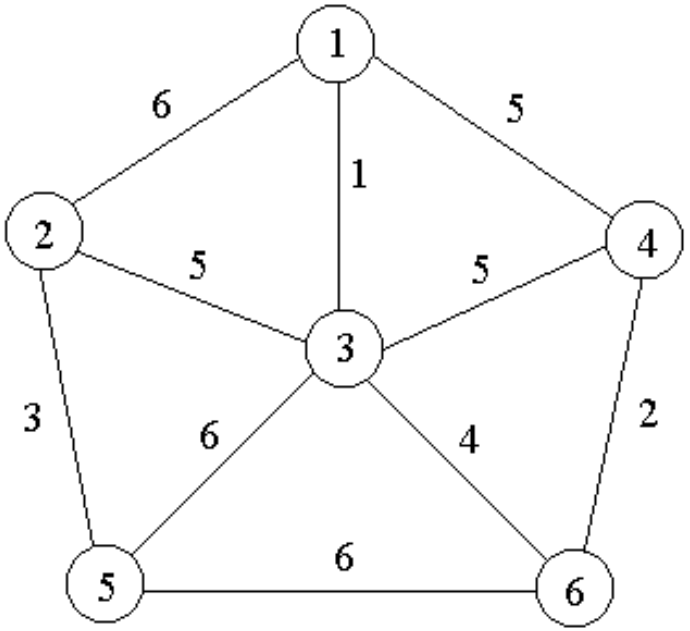
## Prim算法

- 设 $G=(V, E)$ 是连通带权图，$V=\{1,2,\ldots,n\}$
- 构造 $G$ 的最小生成树的 Prim 算法的基本思想
  - 首先置 $S=\{1\}$
  - 只要 $S$ 是 V的真子集，就作如下的贪心选择：选取满足条件 $i\in S$，$j\in V-S$，且 $c[i][j]$ 最小的边，将顶点 $j$ 添加到 $S$ 中
  - 一直进行到 $S=V$ 时为止
- 在这个过程中选取到的所有边恰好构成 G 的一棵最小生成树

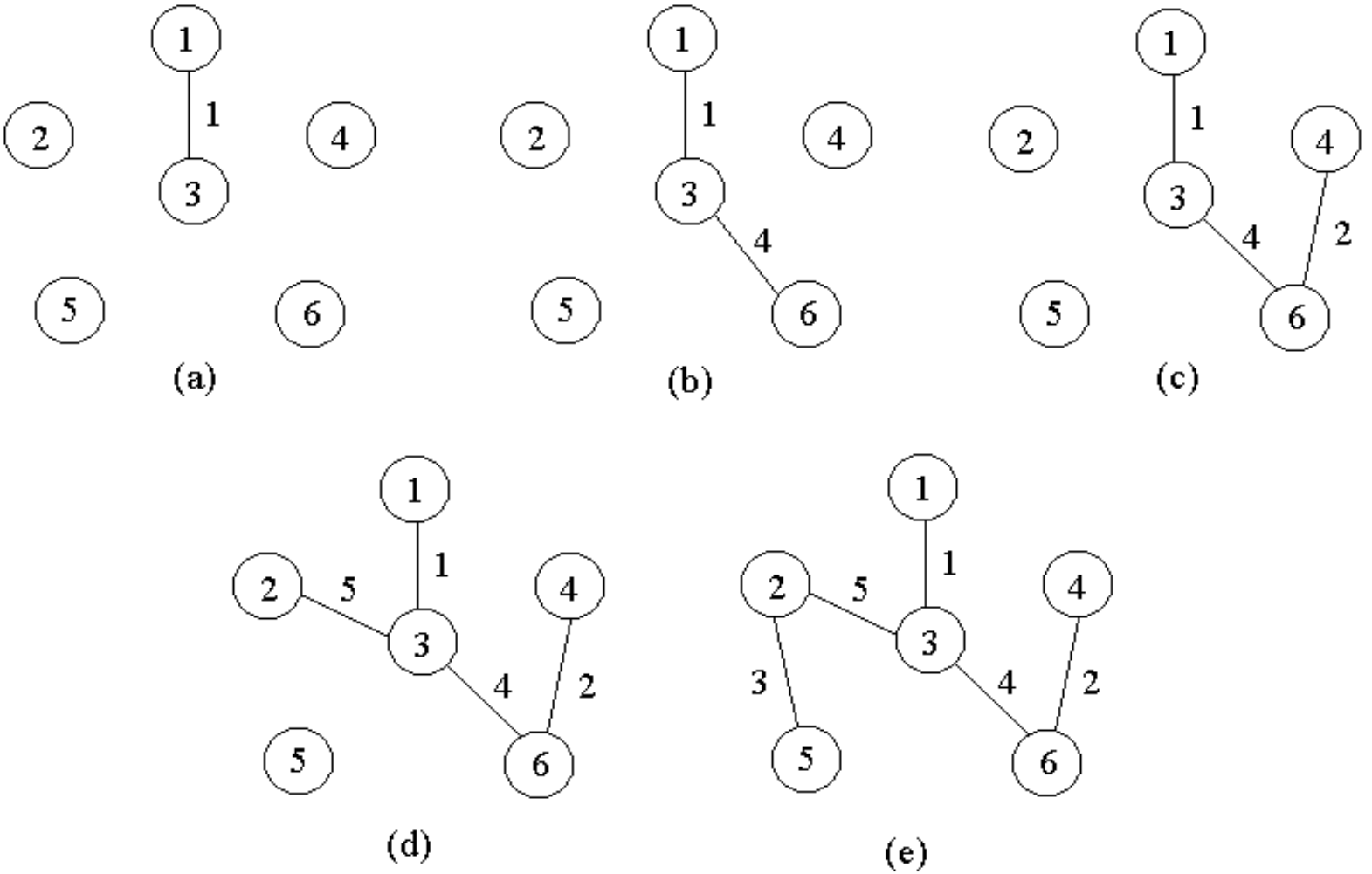
# Prim最小生成树

利用最小生成树性质和数学归纳法容易证明，上述算法中的**边集合 $T$ 始终包含 $G$ 的某棵最小生成树中的边**(**循环不变式**)。因此，在算法结束时，$T$ 中的所有边构成 $G$ 的一棵最小生成树

**例如**，对于右图中的带权图，按 **Prim算法** 选取边的过程如下页图所示



# Prim最小生成树



# Prim分析

## 复杂度分析

在上述Prim算法中，还应当考虑如何有效地找出满足条件 $i \in S, j \in V\text{-}S$，且权 $c[i][j]$ 最小的边 $(i, j)$，实现这个目的的较简单的办法是设置 2 个数组 closest 和 lowcost

在Prim算法执行过程中，先找出 $V\text{-}S$ 中使 lowcost 值最小的顶点 $j$，然后根据数组 closest 选取边 $(j, \text{closest}[j])$，最后将 $j$ 添加到 $S$ 中，并对 closest 和 lowcost 作必要的修改

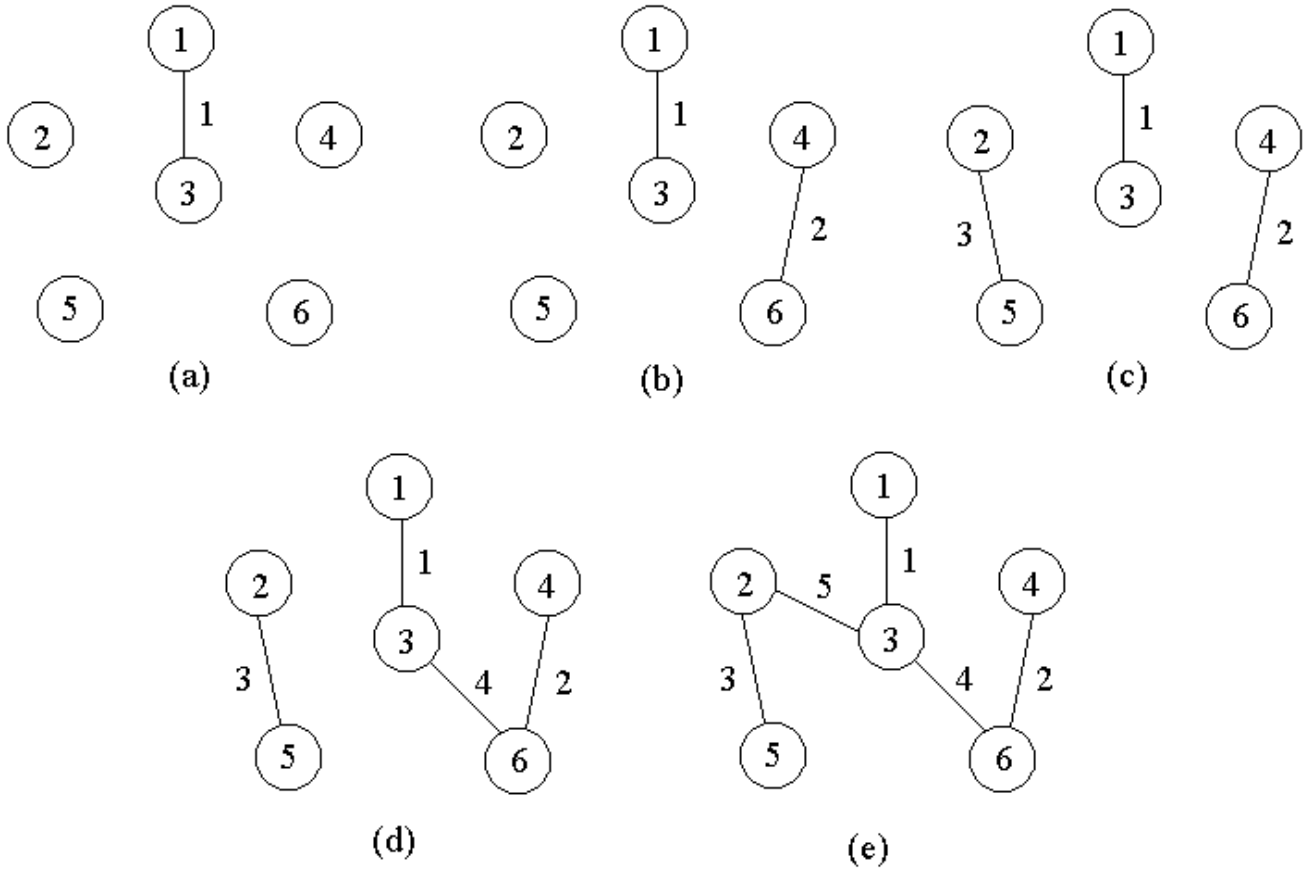
用这个办法实现的 Prim 算法所需的计算时间为 $O(n^2)$

# Kruskal最小生成树

- Kruskal算法构造 G 的最小生成树的基本思想
  - 首先将 $G$ 的 $n$ 个顶点看成 $n$ 个孤立的连通分支，将所有的边按权从小到大排序
  - 从第一条边开始，依边权递增的顺序查看每一条边，并按下述方法连接 2 个不同的连通分支：
    - 当查看到第 $k$ 条边 $(v,w)$ 时，如果端点 $v$ 和 $w$ 分别是当前 2 个不同的连通分支 $T_1$ 和 $T_2$ 中的顶点时，就用边 $(v,w)$ 将 $T_1$ 和 $T_2$ 连接成一个连通分支，然后继续查看第 $k$+1 条边
    - 如果端点 $v$ 和 $w$ 在当前的同一个连通分支中，就直接再查看第 $k$+1 条边
  - 这个过程一直进行到只剩下一个连通分支时为止

# Kruskal 最小生成树

例如，对前面的连通带权图，按 Kruskal 算法顺序得到的最小生成树上的边如下图所示。



# 最小生成树

- 关于集合的一些基本运算可用于实现 Kruskal 算法
  - 按权的递增顺序查看等价于对优先队列执行 removeMin 运算，可以用堆实现这个优先队列
  - 对一个由连通分支组成的集合不断进行修改，需要用到抽象数据类型并查集 UnionFind 所支持的基本运算 (参考CLRS)
- 当图的顶点数为 $n$，边数为 e 时
  - Kruskal算法所需的计算时间是 $O(e\log e)$
  - Prim算法的计算时间为 $O(e\log n)$
  - 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

# 多机调度问题

# 多机调度问题

**多机调度问题**要求给出一种作业调度方案，使所给的 $n$ 个作业在尽可能短的时间内由 $m$ 台机器加工处理完成

约定，每个作业均可在任何一台机器上加工处理，但未完工前**不允许中断**处理，作业**不能拆分**成更小的子作业

优化目标：

$$f = \min \max\{\sum_{i \in m_1} t_i, \sum_{i \in m_2} t_i, \cdots, \sum_{i \in m_m} t_m\}$$

这个问题是**NP完全问题**，到目前为止还没有有效的解法，对于这一类问题，用**贪心选择策略**有时可以设计出较好的近似算法

# 多机调度问题

采用**最长处理时间作业优先**的贪心选择策略可以设计出解多机调度问题的较好的近似算法

按此策略，当 $n \leq m$ 时，只要将机器 $i$ 的 $[0, t_i]$ 时间区间分配给作业 $i$ 即可，算法只需要 $O(1)$ 时间

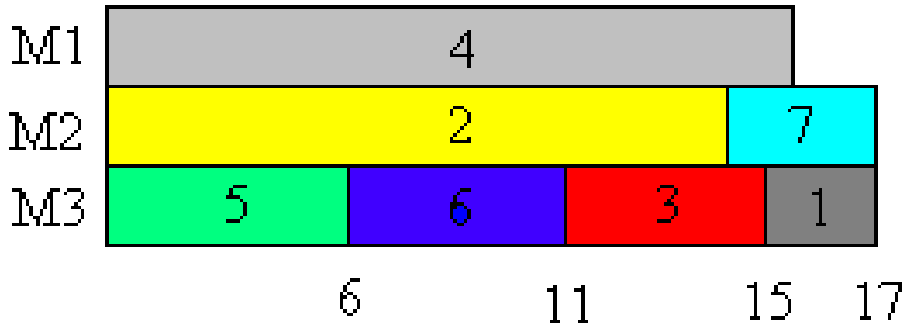
当 $n > m$ 时，首先将 $n$ 个作业依其所需的处理时间从大到小排序；然后依此顺序将作业分配给空闲的处理机

算法所需的计算时间为 $O(n\log n)$

# 多机调度问题

倒如，设 7 个独立作业 {1,2,3,4,5,6,7} 由 3 台机器 $M_1$，$M_2$ 和 $M_3$ 加工处理，各作业所需的处理时间分别为 {2,14,4,16,6,5,3}

按算法greedy产生的作业调度如下图所示，所需的加工时间为 17 ：




反例：

$m$=2, $n$=5,

{7, 6, 3, 2, 2}

# Next

- 回溯法
  - Backtracking