数据结构与算法

第 7 章　图

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

递归与非递归的拓扑排序

- 必须是有向图
- 必须是无环图
- 支持非连通图
- 不用考虑权值
- 回路
  - 非递归的算法，最后判断 (若还有顶点没有输出，肯定有回路)
  - 递归的算法要求判断有无回路

图算法需要考虑的问题

- 是否支持
  - 有向图、无向图
  - 有回路的图
  - 非连通图
  - 权值为负
- 如果不支持
  - 则修改方案?

图的运算

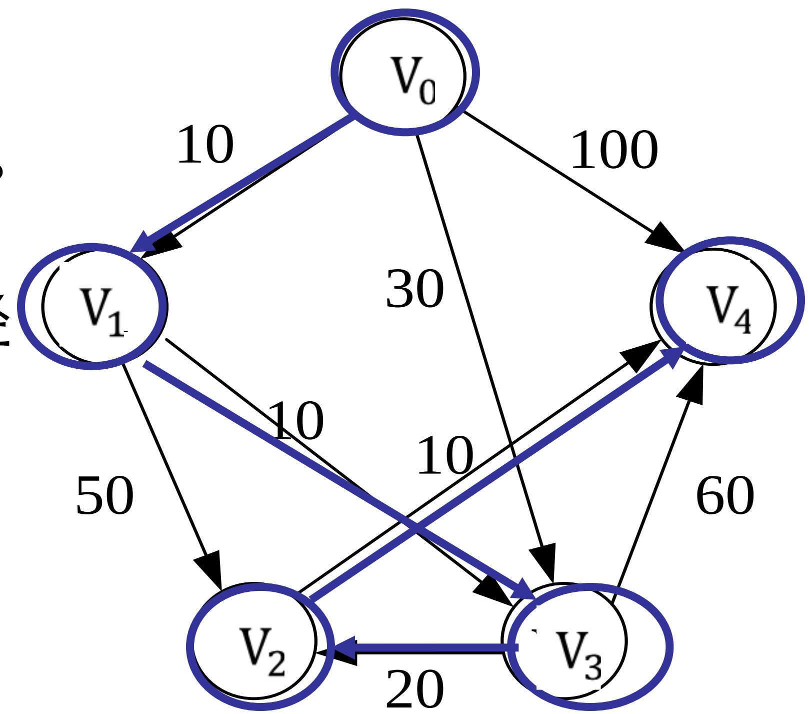
- 图的周游
- 最短路径
- 最小生成树
- 关键路径

最短路径

- 考虑两类最短路径的求解
  - 单源最短路径 (single-source shortest paths)：给定**一个带权图**和一个**指定顶点S (源点)**，求出S 到所有其他各点的最短路径
  - 每对顶点间的最短路径(all-pairs shortest paths)：给定**一个带权图**，求出图中每一对顶点间的最短路径

单源最短路径

- 单源最短路径
  - 给定带权图 G = <V，E>，其中每条边 (v_i，v_j) 上的权 W[v_i， v_j] 是一个非负实数。计算从任给的一个源点 s 到所有其他各结点的最短路径



- 单源最短路径 与 Dijkstra

Dijkstra其人

Edsger Wybe Dijkstra: 1930-2002，荷兰皇家艺术与科学学院的院士，美国科学院院士，英国计算协会的Fellow：

- 1972 Turing Award (ALGOL60)
- 1974 AFIPS Harry Goode Award
- 1982 IEEE Computer Pioneer Award
- 1989 ACM SIGCSE Award for outstanding Contributions to Computer Science Education
- 2002 ACM PODC Influential-Paper Award （2003 and later，Dijkstra Prize in Distributed Computing）
- .....

- 计算机科学与工程领域许多概念、术语的缔造者：结构化程序设计、问题分解、同步、死锁、最弱前提、控制进程同步的“信号量”、栈、向量、……

Dijkstra其人

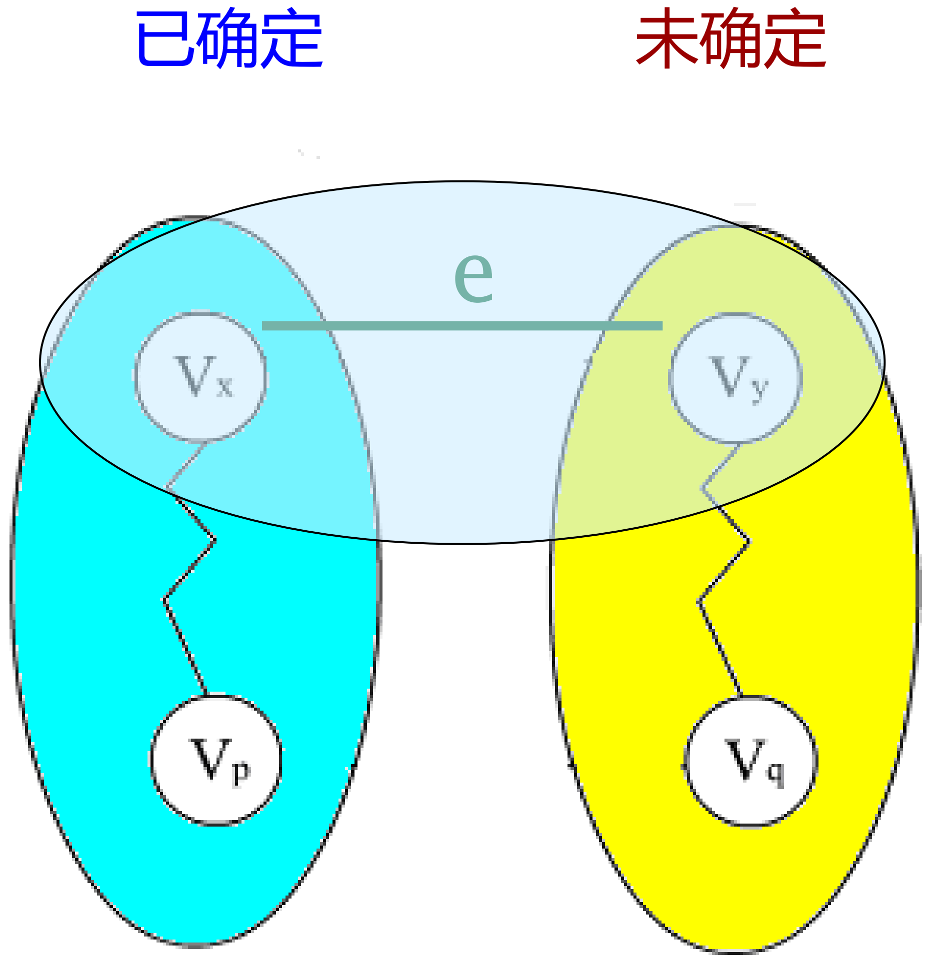
著述甚丰 (http://www.cs.utexas.edu/users/EWD)，http://www.cnblogs.com/dahang/archive/2005/03/24/124658.aspx

以语言机智、诙谐而著称，如

- “ the question of whether computers can think is like the question of whether submarines can swim”
- “In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the culture history of mankind” (*remarks* in his Turing Award lecture)

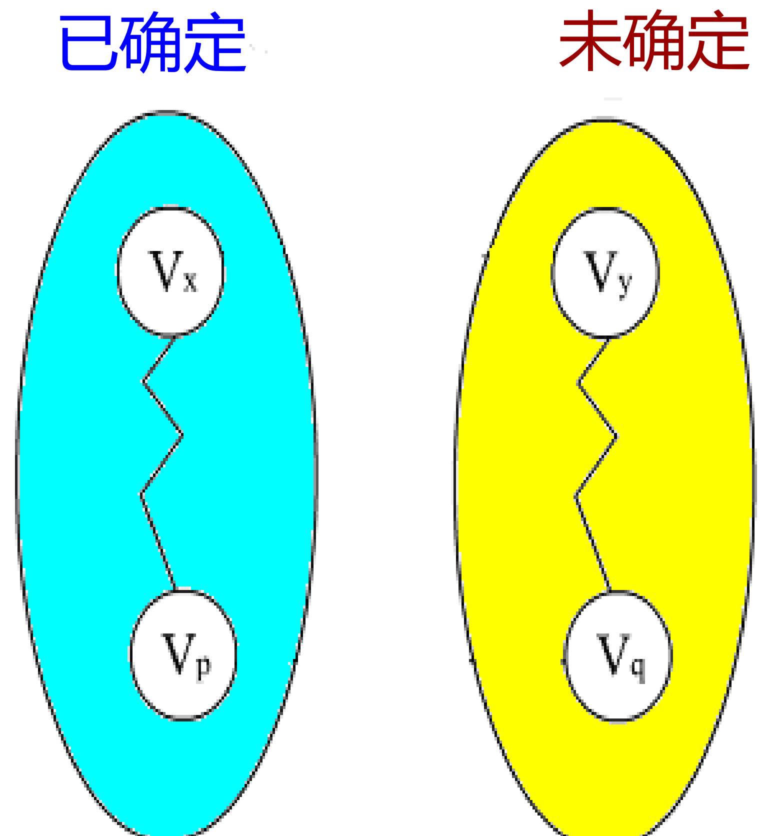
Dijkstra算法基本思想

- 把图中所有顶点分成两组：
  - 已确定最短路径的顶点为一组 S；
  - 尚未确定最短路径的顶点为另一组；
- 按最短路径长度递增的顺序将第二组的顶点逐个加入第一组，直到从指定顶点 S 出发可到达的所有顶点都已在第一组中为止



Dijkstra算法基本思想

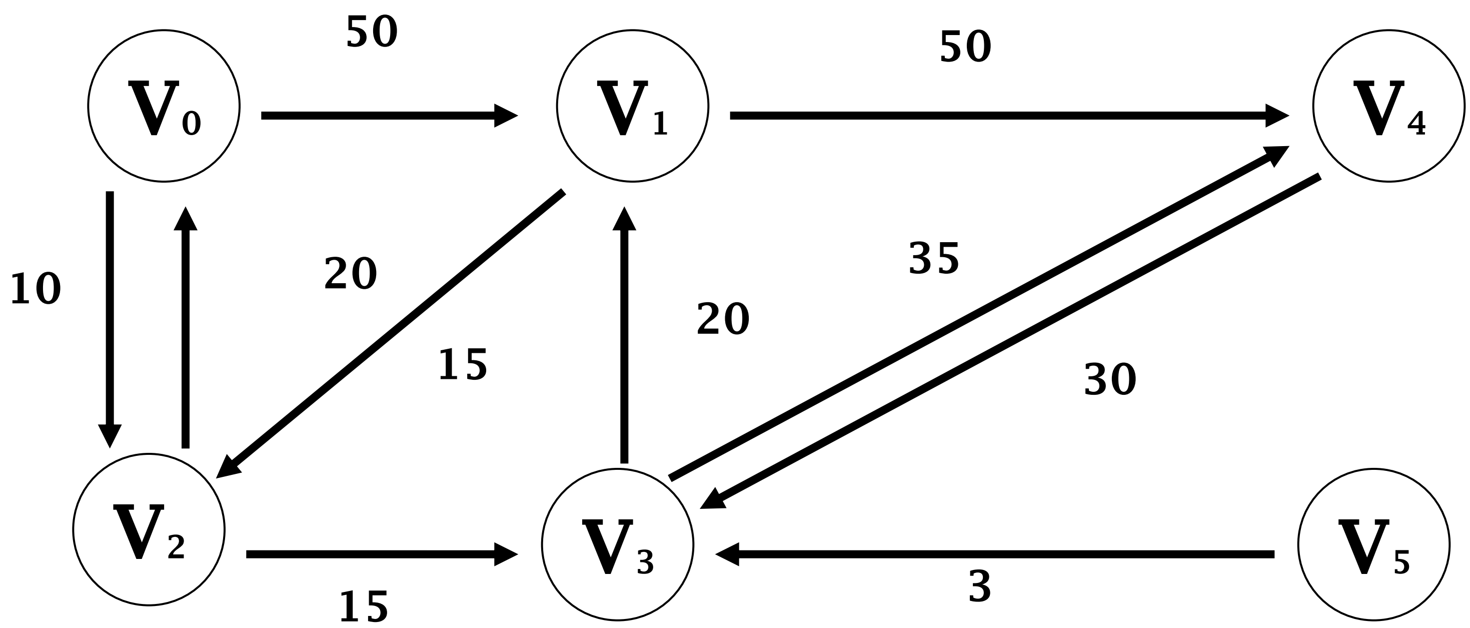
- 在此过程中，保持从 S 到第一组各顶点的最短路径长度都不大于从 S 到第二组任何顶点的最短路径长度。
- 每个顶点对应一个距离值：
  - 第1组 顶点对应的距离值就是从 S 到此顶点的最短路径长度
  - 第2组 顶点对应的距离值是从 S 到此顶点的只包括第一组顶点为中间顶点的最短路径长度



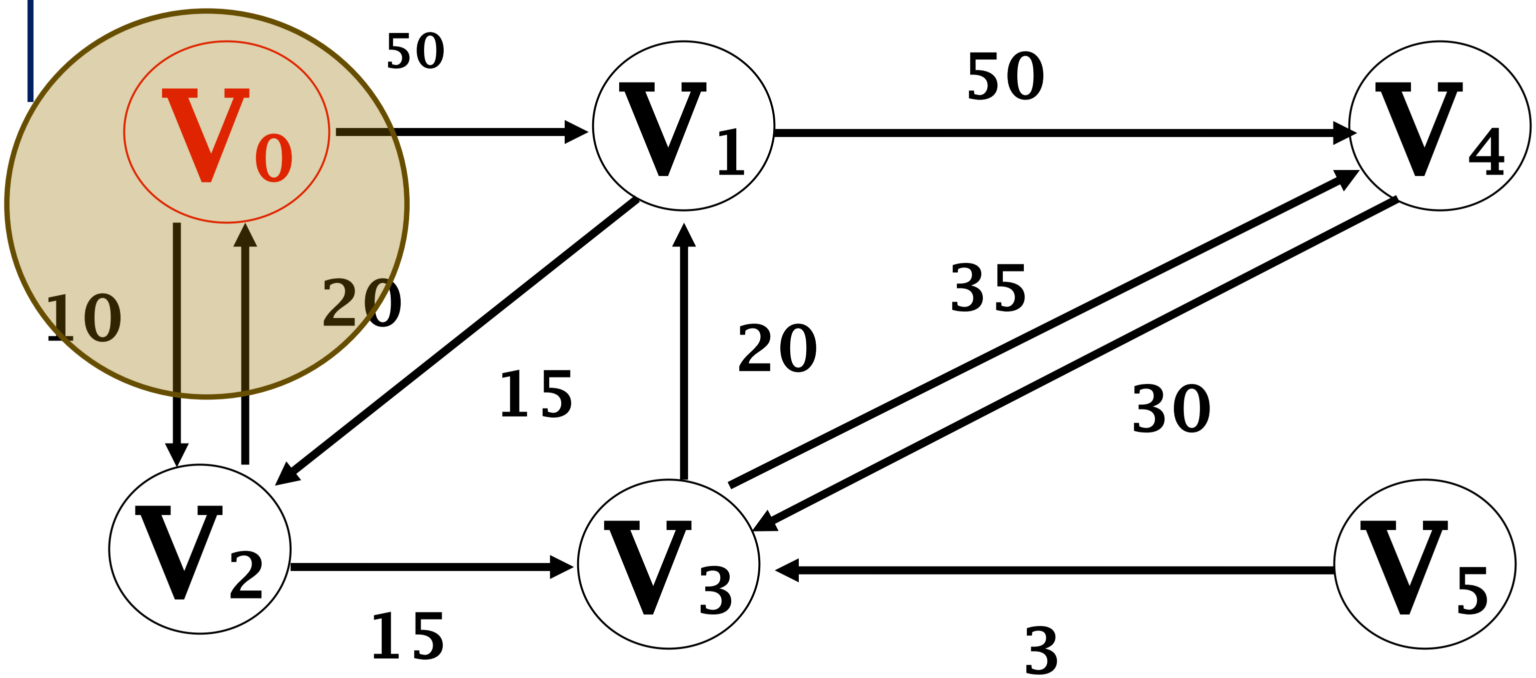
Dijkstra算法具体步骤

- 两组顶点集合的建立
  - 第1 组为其到源顶点 S 的最短路径已确定的顶点集合，则初始第1 组只包含源顶点 S 而已： S 对应的距离值为 0
  - 第2 组初始时包含除源顶点 S 之外的所有其他顶点
    - 各顶点对应的距离值如下确定：若图中有边< S, V_i >，则V_i的距离值为此边的权值，否则V_i的距离值为一个很大的数
- 第 2 组的顶点加入第 1 组的过程
  - 每次从第2 组的顶点中选择一个其距离值最小的顶点V_m加入到第1 组
  - 修改第2组中因V_m作为中间顶点而距离发生改变的各顶点的距离值
- 反复直到图的所有顶点均从 第2组 移到 第1组 为止

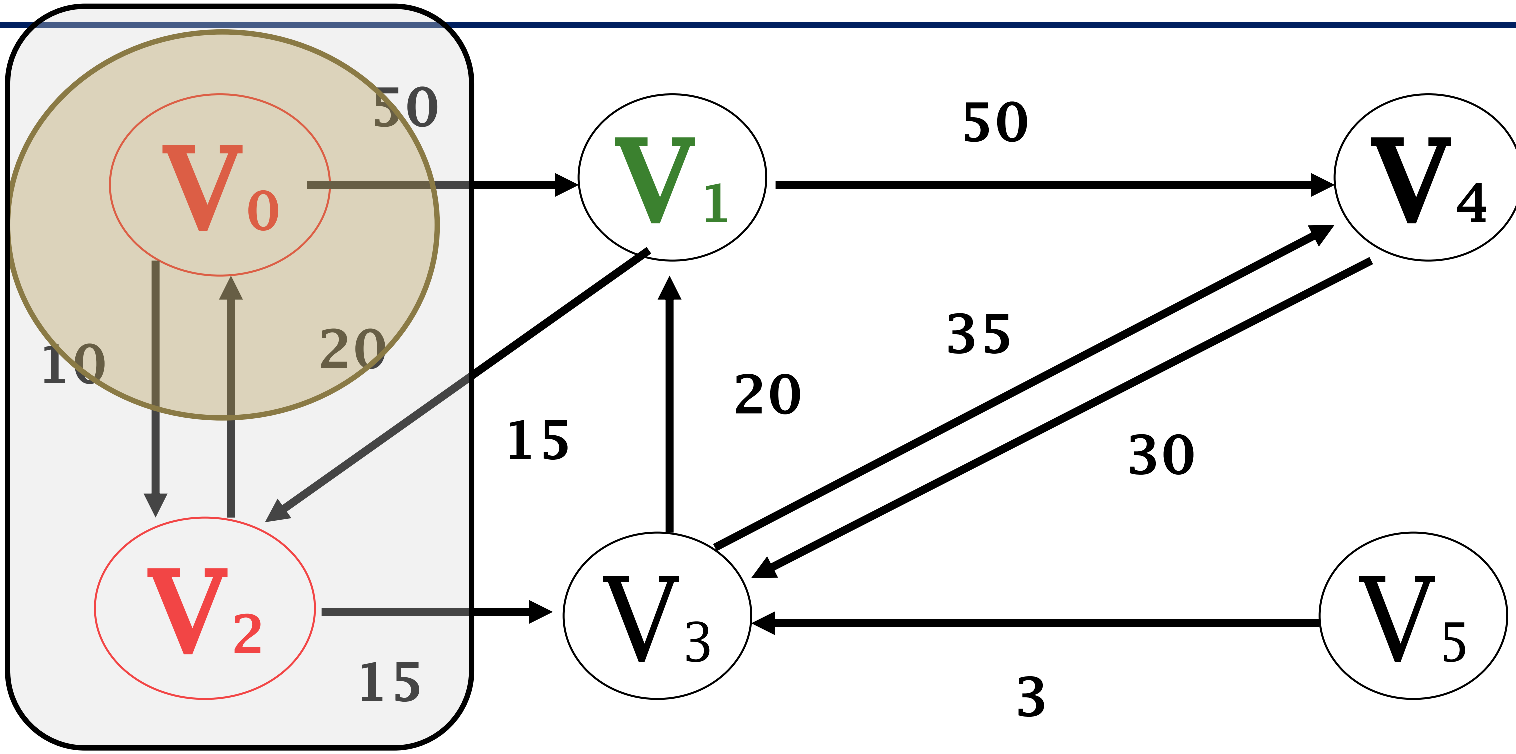
Dijkstra算法示例



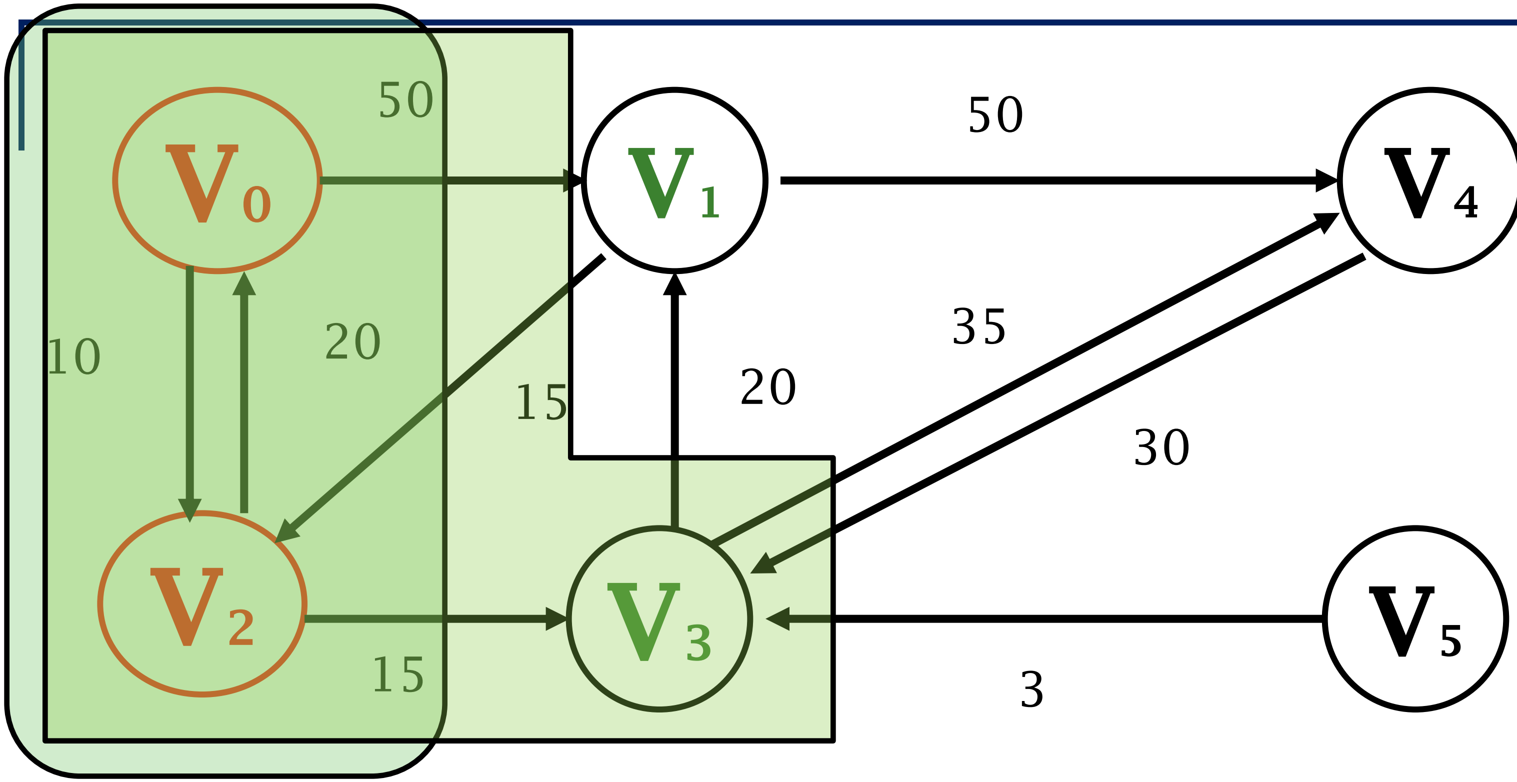
	V_0	V_1	V_2	V_3	V_4	V_5
始前状态	0 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0



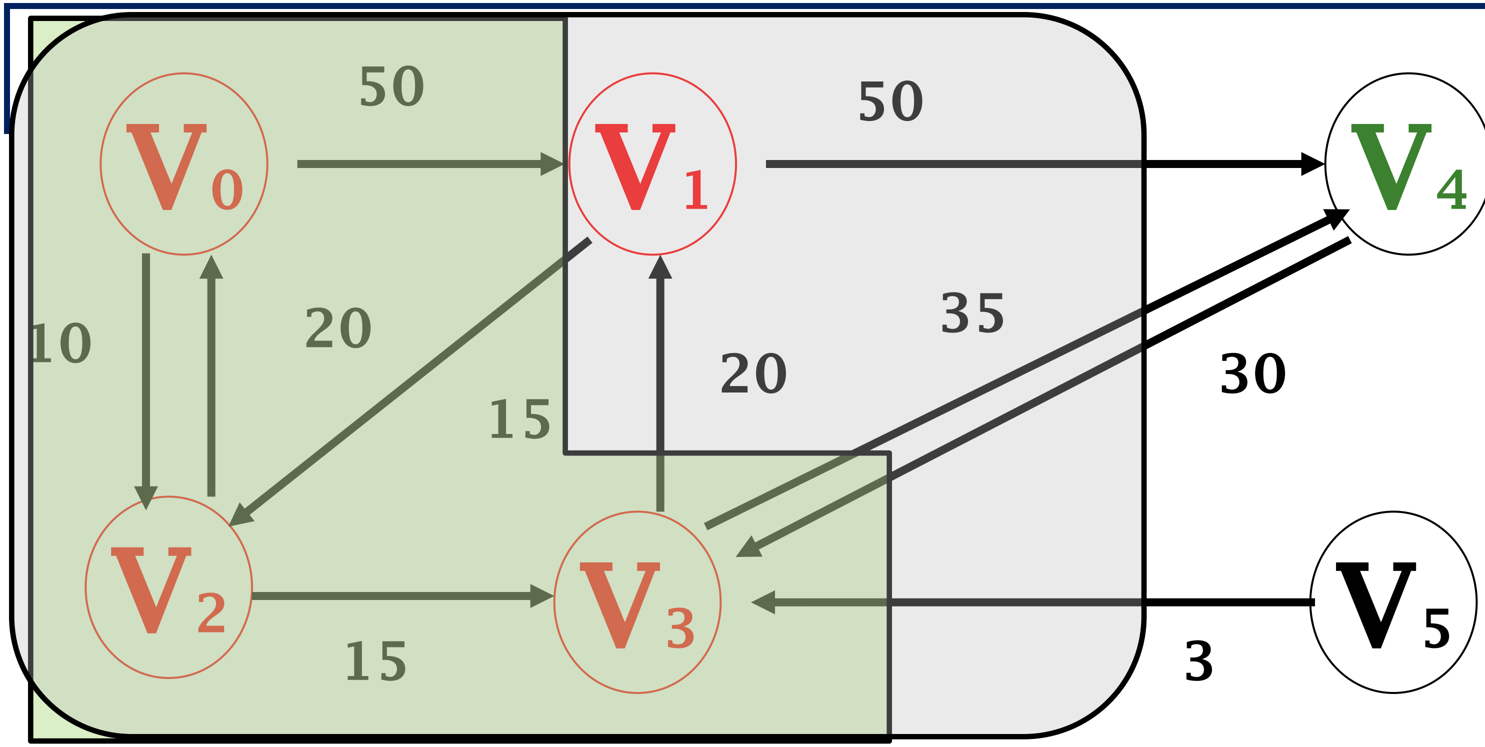
	V_0	V_1	V_2	V_3	V_4	V_5
初始	0 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0
V_0进入第一组	0 Pre:0	50 Pre:0	10 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0



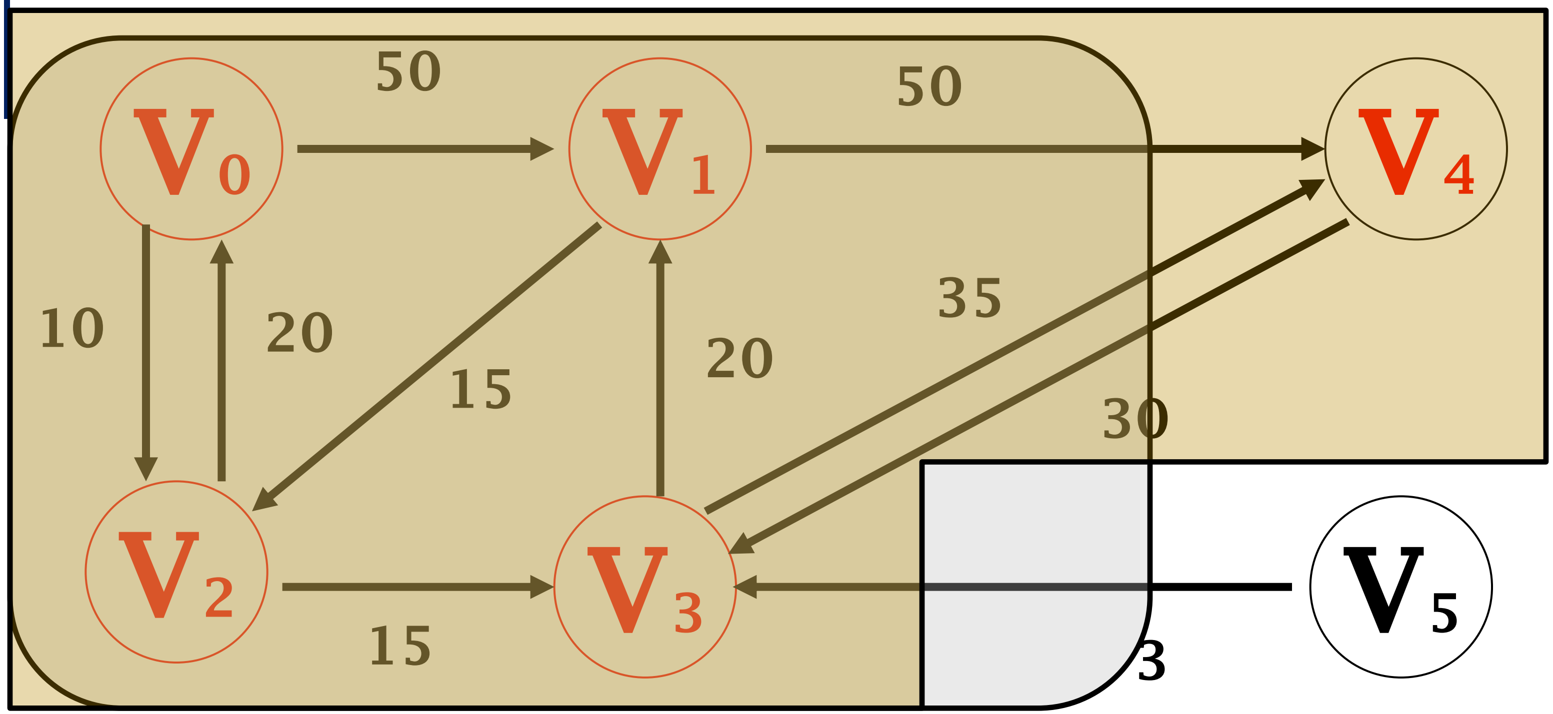
	V_0	V_1	V_2	V_3	V_4	V_5
V_2进入之前	0 Pre:0	50 Pre:0	10 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0
V_2进入第一组	0 Pre:0	50 Pre:0	10 Pre:0	25 Pre:2	∞ Pre:0	∞ Pre:0



	V_0	V_1	V_2	V_3	V_4	V_5
V_3进入之前	0 Pre:0	50 Pre:0	10 Pre:0	25 Pre:2	∞ Pre:0	∞ Pre:0
V_3进入第一组	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0



	V_0	V_1	V_2	V_3	V_4	V_5
V_1进入之前	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0
V_1进入第一组	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0



	V_0	V_1	V_2	V_3	V_4	V_5
V_4进入之前	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0
V_4进入第一组	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0

Dijkstra单源最短路径迭代过程

步数	S	v_0	v_1	v_2	v_3	v_4
初始	$\{v_0\}$	Length:0 pre:0	length:<u>50</u> pre:0	length:<u>10</u> pre:0	length:∞ pre:0	length:∞ pre:0
1	$\{v_0, v_2\}$	Length:0 pre:0	length:50 pre:0	length:10 pre:0	length:<u>25</u> pre:2	length:∞ pre:0
2	$\{v_0, v_2, v_3\}$	Length:0 pre:0	length: <u>45</u> pre:3	length:10 pre:0	length:25 pre:2	length:<u>60</u> pre:3
3	$\{v_0, v_2, v_3, v_1\}$	Length:0 pre:0	length: 45 pre:3	length:10 pre:0	length:25 pre:2	length:60 pre:3
4	$\{v_0, v_2, v_3, v_1, v_4\}$	Length:0 pre:0	length: 45 pre:3	length:10 pre:0	length:25 pre:2	length:60 pre:3

Dijkstra算法的正确性

- 分以下两点来证明:
  - 初始对两个组的 划分 以及 各顶点的距离值的 确定 的正确性
  - 每次往第1 组加入顶点后，两个组的划分以及顶点距离值 再确定 的正确性（反证）

Dijkstra 算法中关键

- 最小值的计算
  - 扫描D数组，两两比较
  - 最小值堆

Dijkstra 算法的时间复杂度

- 对于 n 个顶点e条边的图，图中的任一条边都可能在最短路径中出现，因此对每条边至少都要检查一次
- 采用最小堆来选择权值最小的边，那么每次改变最短路径长度时需要对堆重排一次，其时间代价为 $O((n+e)\log e)$，适合于稀疏图
  - 直接比较D数组元素，确定代价最小的边就需要总时间$O(n^2)$；取出最短路径长度最小的顶点后，修改最短路径长度共需要时间O(e)，因此共需花费$O(n^2)$的时间，这种方法适合于稠密图
- **局限**
  - Dijkstra算法要求边的权值为非负数才可

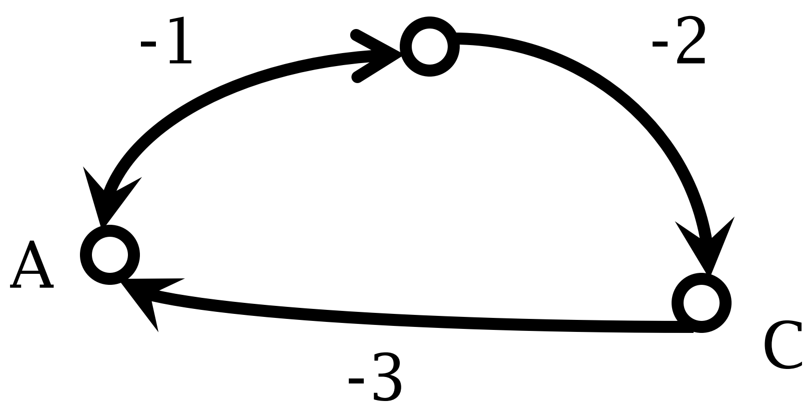
Dijkstra算法讨论

- 是否支持
  - 有向图、无向图
  - 非连通
  - 有回路的图
  - 权值为负
- 如果不支持
  - 修改方案?

- 针对有向图 (且“有源”)
  - 若输入无向图?
  - 照样能够处理 (边都双向)
- 对非连通图，有不可达
  - 没有必要修改
- 支持回路
- 支持负权值?

Dijkstra算法讨论

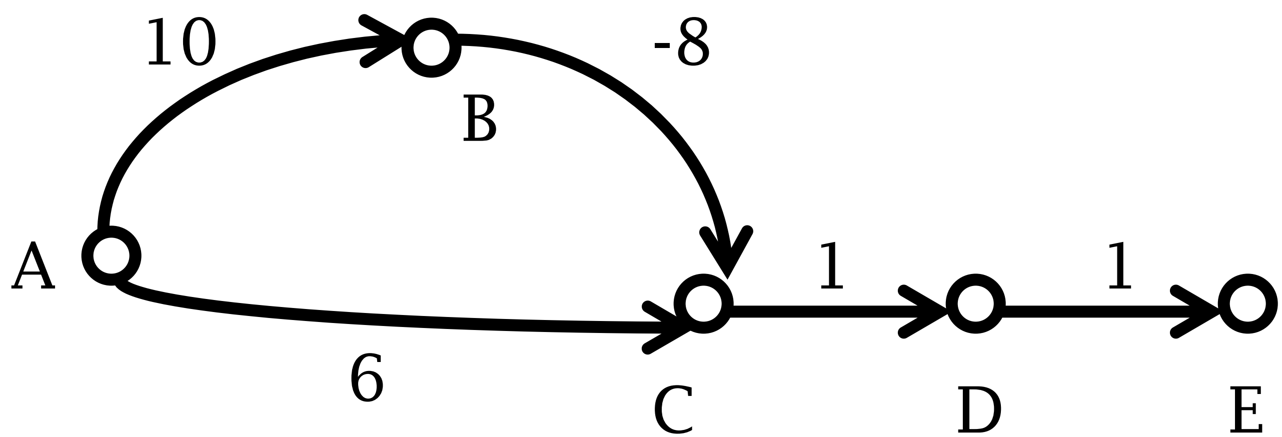
- 若存在总权值为负的回路，则将出现权值为 -∞ 的情况



- Dijkstra算法不支持负权值

Dijkstra算法讨论

- 即使不存在负的回路，若后面出现的负权值，也会导致整体计算错误
- 主要原因是 Dijkstra 算法是贪心法，最为最小进入第1组后，不会返回去重新计算



- Dijkstra算法**不支持负权值**

Dijkstra算法讨论

- 支持负权值的最短路径算法
  - Bellman – Ford 算法
    - 参考书 MIT “Introduction to Algorithms”
  - SPFA 算法

贪心算法（Greedy algorithm）

- **基本思路：从问题的某一个初始解出发逐步逼近给定目标，以尽可能快求得更好的解。当达到算法中的某一步不能再继续前进时，算法停止。**
  - 亦即，总是选择当前来说最佳的方案，寄希望于由局部的最优解构建最终的全局最优解
- **该类算法的实现过程：**

  从问题的某一初始解出发；
  while 能朝给定总目标前进一步 do
  　　求出可行解的一个解元素；
  由所有解元素组合成问题的一个可行解；

贪心算法

- 存在问题:
  1. 不能保证求得的最后解是最佳的;
  2. 不能用来求最大或最小解问题;
  3. 只能求满足某些约束条件的可行解的范围
- 贪心算法虽然并不总能产生最优解，但对很多问题确实有效且可获得最优解:
  - 最小生成树 (minimum-spanning-tree)
  - Dijkstra的单源最短路径 (shortest paths)
  - 集合涵盖 (set-covering heuristic)

最短路径

- 考虑两类最短路径的求解
  - 单源最短路径 (single-source shortest paths)：给定一个带权图和一个指定顶点S (源点)，求出S 到所有其他各点的最短路径
  - 每对顶点间的最短路径(all-pairs shortest paths)：给定一个带权图，求出图中每一对顶点间的最短路径

每对顶点间的最短路径

- G=(V, E)是一个带权图，对任意的u，v∈ V ，计算u 到 v 的最短路径的值，即d(u, v)
- 常用的解决方案
  - Dijkstra算法
    - Greedy algorithm
  - Floyd-Warshall 算法
    - Dynamic programming
  - Matrix multiplication

利用Dijkstra算法

- 使用|V|次Dijkstra算法
  - 每次从不同顶点出发计算最短路径
  - 若图G为稀疏图，这不失为一种好方法。对于采用优先队列的Dijkstra算法，总的时间代价为$O(|V|+|E|)\cdot\log|E|)$
  - 对密集图来说，基于优先队列的Dijkstra 算法时间代价为$O(|V|^3 \log |E|)$；而时间代价为$O(|V|^2)$的Dijkstra算法在此处的代价则为$O(|V|^3)$
- Dijkstra算法要求边的权值为非负数才可

Floyd-Warshall 算法

- 允许图中有权值为负的边存在，但设定图中没有权值总合为负的回路
- 时间复杂度为$\Theta(|V|^3)$

- 采用动态规划（dynamic programming）方法
- 简称为Floyd算法

动态规划方法

- 适用于优化问题(optimization problems)：即将一个问题的解决方案视为一系列决策的结果
  - 每一个决策往往导致相同类型的子问题的出现，且子问题各自并不独立，可能会共享某些子-子问题
  - 与每采用一次贪婪准则便做出一个不可撤回决策的贪心算法不同，动态规划考察每个最优决策序列中是否包含一个最优子序列

动态规划方法

- 采用最优原则（principle of optimality）来建立用于计算最优解的递归式或迭代式
  - 所谓最优原则即不管前面的策略如何，此后的决策必须是基于当前状态（由上一次决策产生）的最优决策
    - 对于有些问题的某些递归式来说并不一定能保证最优原则，因此在求解问题时有必要对其进行验证。若不能保持最优原则，则不可应用动态规划方法
  - 得到最优解的递归式后，需要执行回溯（traceback）以构造最优解

动态规划方法

- 求解动态规划问题一种简单方式为编写递归程序
  - 若避免不了重复计算，递归程序的复杂性有可能将非常可观
  - 动态规划递归方程通常可用迭代方式求解，以避免重复计算
- 关键是如何存储子问题的解决方案以备复用
  - 自底向上构造
  - 在原问题的小子集中计算，每一步列出局部最优解，并**保留这些局部最优解**，以避免重复计算。逐步增大处理的子集，**最终在问题的全集**上计算，所求解的即为整体最优解

Floyd算法基本思路

- 具有n个顶点的图G=(V, E) 采用相邻矩阵adj作为其存储结构
  - 若存在边$(v_i,v_j)\in E$，则从顶点v_i到顶点v_j存在一条长度为adj[i][j] 的路径；但该路径并不一定是从顶点v_i到顶点v_j的最短路径，因可能存在从v_i到v_j且包含其它顶点作为中间顶点的路径
  - 故，应在所有从 v_i 到 v_j 允许其它顶点为中间顶点的路径中，找出长度最短的路径

Floyd算法基本思路

- k-path 定义 为任意一条从顶点v到u的、途径顶点序号小于k的路径
  - 0-path 即为从v到u，经过V_0的路径
- 若已知从v到u的最短 k-path，则最短的(k+1)-path分为以下两种情况：
  1. 经过顶点k，则最短 (k+1)-path 的一部分是从v到k的最短k-path，另一部分是从k到u的最短k-path
  2. 不经过顶点k，最短 (k+1)-path保持其值为最短k-path不变

k-path

- 初始从图的相邻矩阵开始，此时顶点v 到 u 的路径为 从v到u的边
- 依次加入顶点v_0 , v_1 , v_2 , ..., v_K等作为中间顶点，分形成 0-path, 1-path, ..., k-path：
  - (v, ..., v_k) 是从v到v_k 允许 k个顶点v_0, v_1, ..., v_{k-1}为中间顶点的最短路径
  - (v_k, ..., u) 从v_k到u允许 k个顶点v_0, v_1, ..., v_{k-1}为中间顶点的最短路径
  - 计算 (v, ..., v_k , ..., u) 和 已经得到的从 v 到u 允许 k个顶点v_0, v_1, ..., v_{k-1}为中间顶点的最短路径中的较短者
  - 较短者则是 从v到u **允许k+1个顶点**v_0, v_1, ..., v_k为中间顶点的最短路径

(n-1)-path

- 依此类推，直到加入顶点v_{n-1}为止，则得到的是从 v 到 u 允许 *n* 个顶点v_0, v_1, …, v_{n-1}为中间顶点的最短路径
  - 考虑了所有顶点作为中间顶点的可能性，故得到的即为从v到u的最短路径

Floyd算法

- 用相邻矩阵adj来表示带权有向图
  - 初始化$adj^{(0)}$为相邻矩阵adj
  - 在矩阵$adj^{(0)}$上做 n次 迭代，递归地产生一个 矩阵序列 $adj^{(1)}$，…，$adj^{(k)}$，…，$adj^{(n)}$
  - 其中, 第 k 次迭代的结果，$adj^{(k)}[i，j]$ 的值等于从顶点v_i到顶点v_j路径上所经过的顶点序号不大于k的最短路径长度

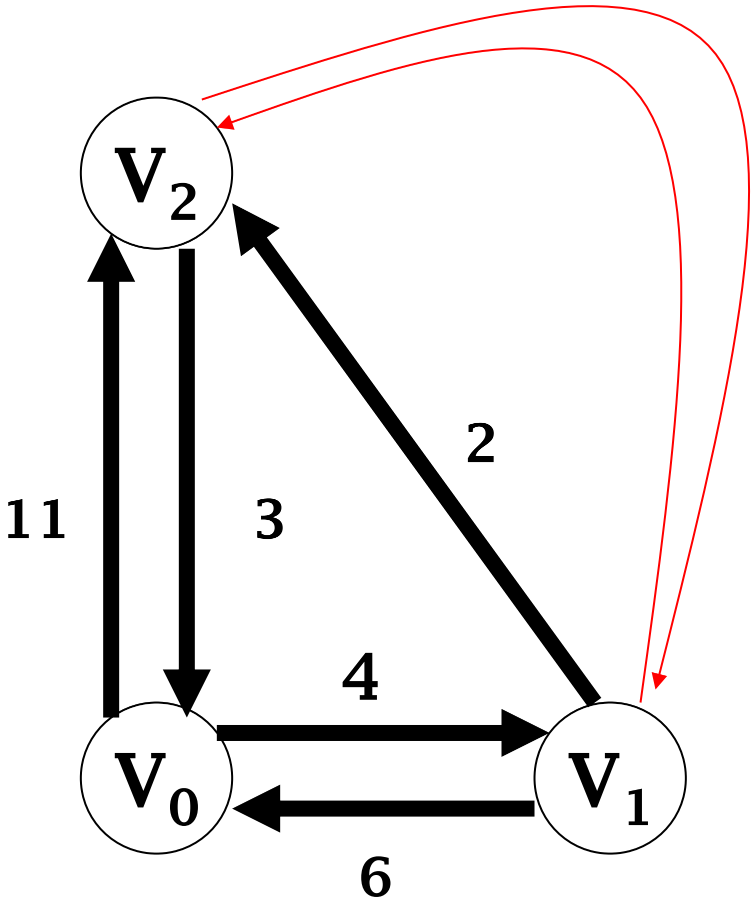
Floyd算法

- 进行第k次迭代时，矩阵$adj^{(k-1)}$已求得，故从顶点v_i到顶点v_j中间顶点的序号不大于k的最短路径分两种情况：
  1. 中间**不经过顶点**v_k，则有$adj^{(k)}[i, j] = adj^{(k-1)}[i, j]$
  2. 中间**经过顶点**v_k，此时有$adj^{(k)}[i, j] < adj^{(k-1)}[i, j]$，那么这条由顶点$v_i$经$v_k$到顶点$v_j$的中间顶点序号不大于k的最短路径由两段组成：
     - 其一是从顶点v_i到顶点v_k的中间顶点序号不大于k-1的最短路径
     - 另一是从顶点v_k到顶点v_j的中间顶点序号不大于k-1的最短路径

  即，$adj^{(k)}[i, j] = adj^{(k-1)}[i, k] + adj^{(k-1)}[k, j]$，因而

  $adj^{(k)}[i, j] = \min\{ adj^{(k-1)}[i, j],\ adj^{(k-1)}[i, k]+adj^{(k-1)}[k, j]\}$

Floyd算法示例



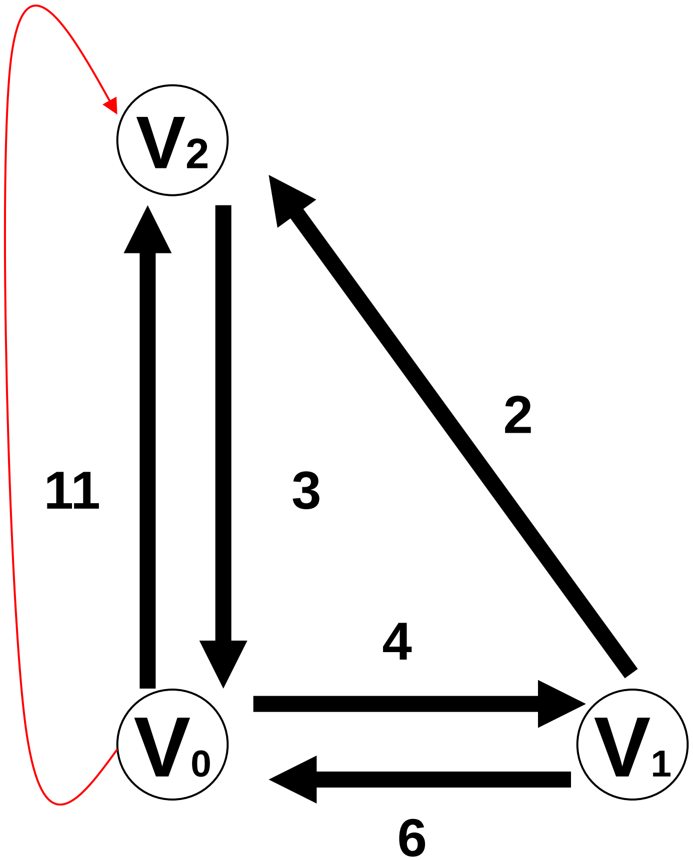
$$adj = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix} \qquad Path= \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & -1 & 2 \end{pmatrix}$$

$$adj^{(0)}= \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \qquad Path= \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$2 \quad < \quad 6+11$

$\infty \quad > \quad 3+4$

Floyd算法示例



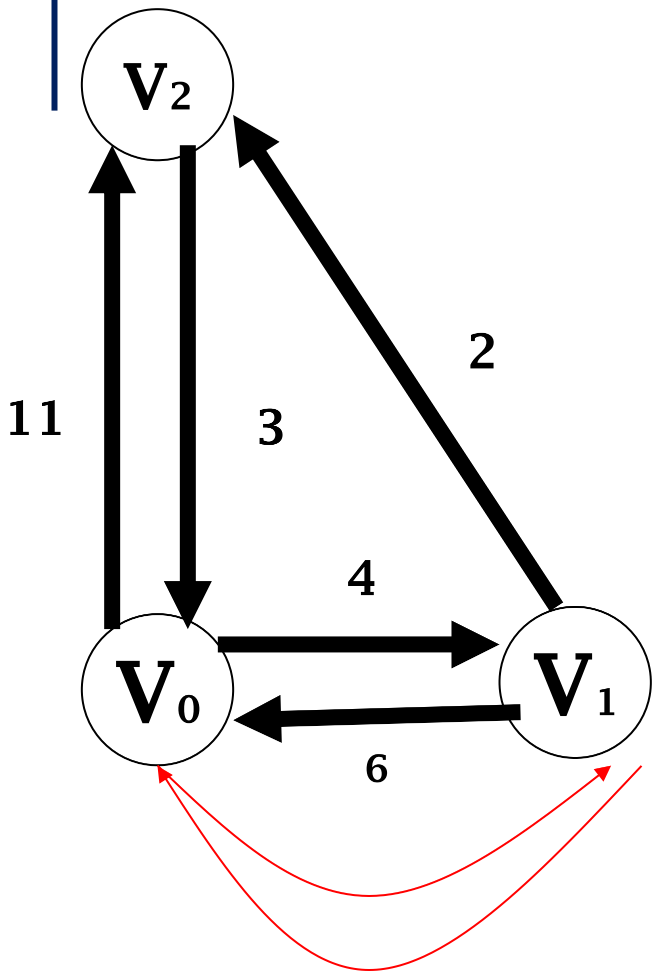
$$adj = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \boxed{\infty} & 0 \end{pmatrix} \quad Path= \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & -1 & 2 \end{pmatrix}$$

$$adj^{(0)}= \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path= \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$$adj^{(1)}= \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path= \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$11 \quad > \quad 4+2$

$3 \quad < \quad \infty(V_2V_1, V_1V_0)$



$4 < \infty(V_0V_2, V_2V_1)$

$6 > 2+3$

$$adj = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \boxed{\infty} & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 2 & -1 & 2 \end{pmatrix}$$

$$adj^{(0)} = \begin{pmatrix} 0 & 4 & \boxed{11} \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$$adj^{(1)} = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 2 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$$adj^{(2)} = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 1 \\ 2 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

Floyd算法的实现

```
void Floyd(Graph& G, Dist** &D)  {
    int i,j,v;
    D = new Dist*[G.VerticesNum()];              // 申请空间
    for (i = 0; i < G.VerticesNum(); i++)
        D[i] = new Dist[G.VerticesNum()];
    for (i = 0; i < G.VerticesNum(); i++)        // 初始化数组D
        for (j = 0; j < G.VerticesNum(); j++) {
            if (i == j) {
                D[i][j].length = 0;
                D[i][j].pre = i;
            } else {
                D[i][j].length = INFINITE;
                D[i][j].pre = -1;
            }
        }
```

Floyd算法的实现

```
for (v = 0; v < G.VerticesNum(); v++)
    for (Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e)) {
        D[v][G.ToVertex(e)].length = G.Weight(e);
        D[v][G.ToVertex(e)].pre = v;
    }
// 加入新结点后，更新那些变短的路径长度
for (v = 0; v < G.VerticesNum(); v++)
    for (i = 0; i < G.VerticesNum(); i++)
        for (j = 0; j < G.VerticesNum(); j++)
            if (D[i][j].length > (D[i][v].length+D[v][j].length)) {
                D[i][j].length = D[i][v].length+D[v][j].length;
                D[i][j].pre = D[v][j].pre;
            }
}
```

Floyd算法的时间代价

- 三重循环检查了所有的可能性
  - 在算法结束时，D数组存储了所有成对顶点间的最短路径，pre存储了路径的有关信息
- 时间代价为O(|V|^3)

Floyd算法实现讨论

将 “D[i][j].pre = D[v][j].pre” 改为

“ D[i][j].pre = v”

是否可以?

- 上述两种方案不影响 D[i][j].length 的求解
- 对于恢复最短路径，策略有何不同?
  那种更优?

最短路径的应用

- 地图集
- 交通网络
- ？？
- ．．．

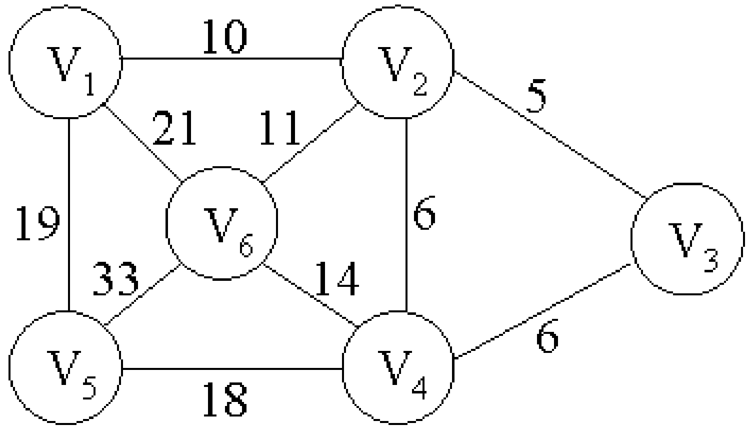
图的运算

- 图的周游
- 最短路径
- 最小生成树
- 关键路径

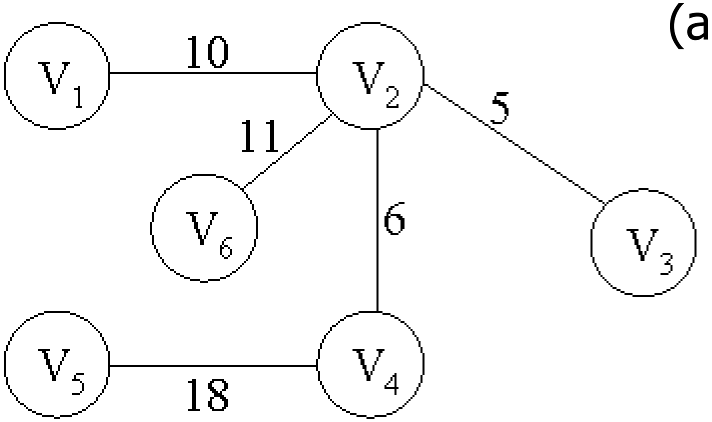
最小生成树

- 给定一个连通的无向带权图G（即每条边带有相应的长度或权值），G的**最小生成树**（Minimum-cost Spanning Tree，简称MST）是一个包括G的 所有顶点和一个边的子集 的图，边的子集满足下列条件：
  - 子集中所有边的权之和在类似子集中**最小**
  - 子集中的边保证**图的连通**
- 也称**最小支撑树**（图的周游中可以生成支撑树）
- MST的应用
  - 连接电路板上一系列接头所需焊接的线路最短
  - 城市间建立电话网所需的线路最短
  - ……

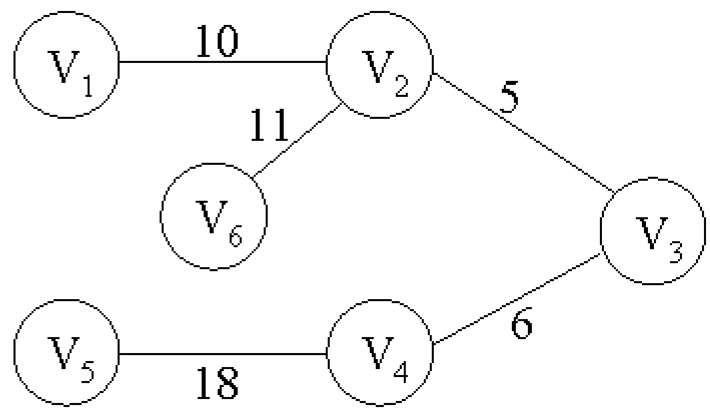
最小生成树



(a)



(a)的MST



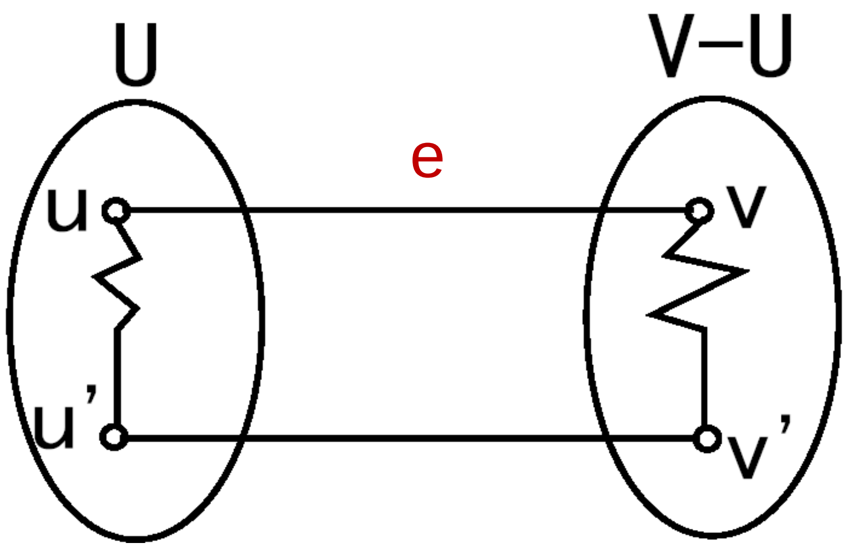
(a)的MST

MST的性质

- **最小生成树的含义**
  - 满足MST 要求的边集所构成的树支撑起了图的所有的顶点
  - 此边集的代价最小
- **MST 不存在回路**
  - 若MST 的边集中有回路，显然可通过去掉回路中某条边而得到边集代价更小的MST
- **MST 是一棵有|V|-1条边的自由树**

MST的性质

- 假设G=(V,E) 是一个带权连通图，U是顶点集V的一个非空子集；若 e=(u, v) 是一条具有**最小权值的边**，其中u∈U，v∈V-U，则必存在一棵包含边 e 的最小生成树



MST的性质

- **证明（反证法）：**
  - 假设连通图G的所有最小生成树都不包含(u, v)
  - 设T是G上的一棵最小生成树，若把边(u, v) 加入到T，由于T是包含图G所有顶点的自由树，加入边(u, v) 后必然存在一条包含(u, v)在内的回路
  - 由于T是生成树，则在T上必存在另一条边(u', v')，其中u' ∈ U，v' ∈ V - U；且u和u' 之间， v和v' 之间有路径相通。删去边(u', v') 就可以消除回路，同时得到另一棵生成树T' 。因为(u, v) 的权不高于(u', v')，则T' 的代价也不高于T。因此T' 是包含(u, v) 的一棵最小生成树

由此与假设产生矛盾。

最小生成树的构造

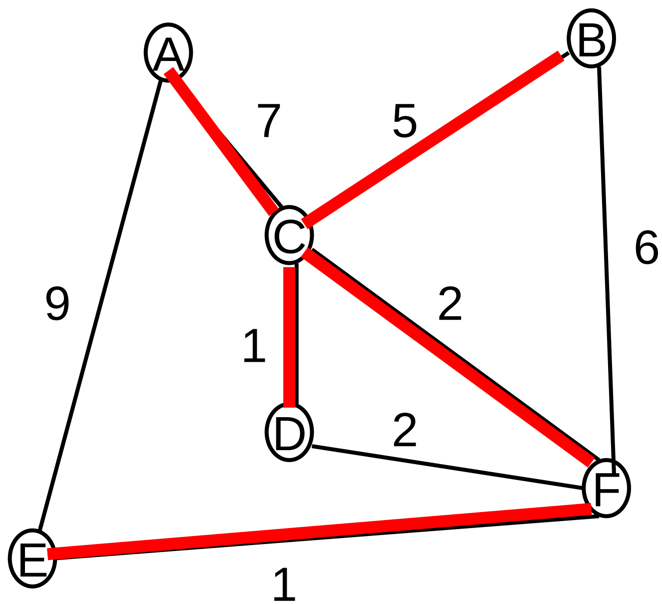
- 利用 最小生成树的性质 构造最小生成树，两种经典算法:
  - Prim算法
  - Kruskal 算法

  均为贪心算法

Prim 算法

- 主要步骤
  1. 由图中任一顶点 S 开始，初始化MST 为 S
  2. 选出与 S 相关联的边中**权最小**的一条，设其连接 S 与另一顶点 W，将顶点 W 和边 (S, W) 加入MST;
  3. 选出与 S 或 W 相关联的边中**权最小**的一条，设其连接另一新顶点 V，将此边和新顶点V 加入 MST 中;
  4. 如此反复处理，每一步都通过选出连接目前已在 MST 中的某个顶点及另一不在 MST 中顶点的**权最小**的边而扩展MST
- **注意：prim算法不是寻找下一个离起始点最近的顶点，而是下一个与MST 中任意顶点相距最近的顶点**

Prim算法示例



Prim算法实现

- 取权值最小的边
  - 首先对边进行完全排序?
  - 使用最小值堆来实现!
  - 一次取一条边。实际上在完成MST 前仅需访问一小部分边

Prim算法的一种实现

```
void Prim(Graph& G, int s, Edge* &MST)  {
    // s是开始顶点，数组MST用于保存最小生成树的边
    int MSTtag = 0;                              // 最小生成树的边计数
    MST = new Edge[G.VerticesNum()-1];           // 为数组MST申请空间
    Dist *D;
    D = new Dist[G. VerticesNum()];              // 为数组D申请空间
    for (int i = 0; i < G.VerticesNum(); i++) {  // 初始化Mark数组、D数组
        G.Mark[i] = UNVISITED;
                D[i].index = i;
                D[i].length = INFINITE;
                D[i].pre = s;
    }
    D[s].length = 0;
    G.Mark[s]= VISITED;                          // 开始顶点标记为VISITED
    int v = s;
```

Prim算法的一种实现

```
for (i = 0; i < G.VerticesNum()-1; i++) {
    if (D[v] == INFINITY)  return;          // 非连通，有不可达顶点
    // 因为v的加入，需要刷新与v相邻接的顶点的D值
    for (Edge e = G.FirstEdge(v); G.IsEdge(e);e = G.NextEdge(e))
        if (G.Mark[G.ToVertex(e)] != VISITED &&
                    (D[G.ToVertex(e)].length > e.weight)) {
            D[G.ToVertex(e)].length = e.weight;
            D[G.ToVertex(e)].pre = v;
        }
    v = minVertex(G, D);                    // 在D数组中找最小值记为v
    G.Mark[v] = VISITED;                    // 标记访问过
    Edge edge(D[v].pre, D[v].index, D[v].length); // 保存边
    AddEdgetoMST(edge,MST,MSTtag++);        // 将边edge加到MST中
}
}
```

Prim算法的一种实现

```
int minVertex(Graph& G, Dist* & D)  {          // 在D数组中找最小值
    int i, v;
    for (i = 0; i < G.VerticesNum(); i++)
        if (G.Mark[i] == UNVISITED) {
            v = i;                             // 令v为任一未访问的顶点
            break;
        }
    for (i = 0; i < G.VerticesNum(); i++)
        if ((G.Mark[i] == UNVISITED) && (D[i] < D[v]))
            v = i;                     // 保存当前发现的具有最小距离的顶点
    return v;
```

Prim算法的代价

- 代价主要体现在？
  - 采用相邻矩阵，则时间代价为$O(n^2)$
  - 采用邻接表算法的时间代价为

    $O(n^2) + O(|E|) = O(n^2)$

  适用于密集图

- 优化可能?

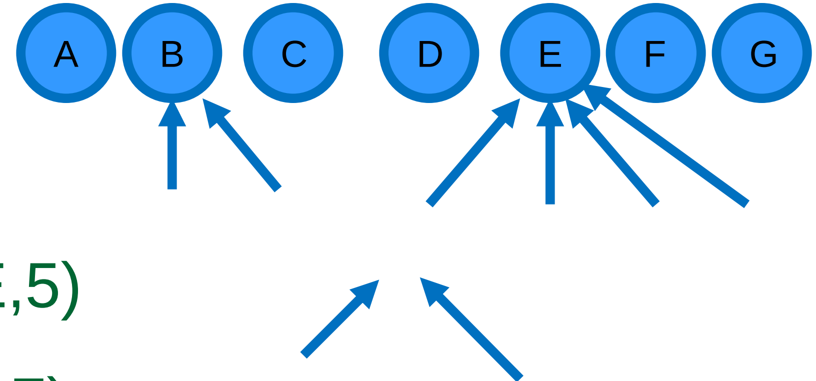
Prim算法的代价

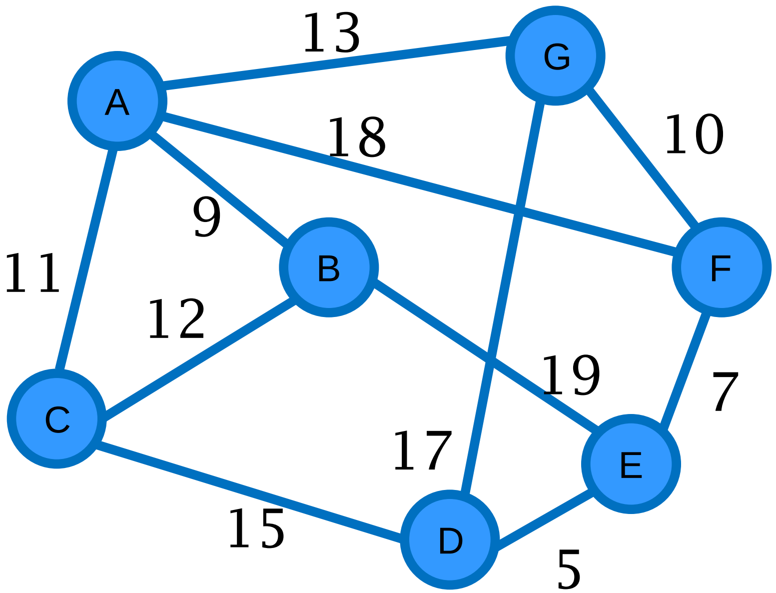
- Prim算法非常类似于Dijkstra算法，区别在于：
  - Prim算法中的距离值不需累积，直接采用离集合最近的边距
  - Dijkstra寻找与固定顶点距离最近的顶点
- Prim算法的时间复杂度也与Dijkstra算法相同

Kruskal 算法

- **一个基于贪心的构造方法：**
  1. 首先，将 G 中的 n 个顶点看成是 n 个独立的连通分量，此时状态为 n 个独根树 组成的森林，可记为 T = <V，{}>；
  2. 在边集 E 中选择代价最小的边，若该边关联的顶点分属 两个不同的连通分支，那么将此边加入到 T 中，否则舍去此边而选择下一条代价最小的边；
  3. 重复第 2 步，直到T中所有顶点都在同一个连通分量中为止，此时就得到图G的一棵最小生成树

Kruskal算法示例





1	4	1	4		4	5 4
A	B	C	D	E	F	G
0	1	2	3	4	5	6

- (D,E,5)
- (F,E,7)
- (A,B,9)
- (F,G,10)
- (A,C,11)
- (B,C,12)
- (A,G,13)
- (C,D,15)

....

Kruskal算法的一种实现

```
void Kruskal(Graph& G, Edge* &MST)  {                          // 数组MST用于保存最小生成树的边
    ParTree<int> A(G.VerticesNum());                                 // 等价类
    MinHeap<Edge> H(G.EdgesNum());                                   // 最小堆
    MST = new Edge[G.VerticesNum()-1];                               // 为数组MST申请空间
    int MSTtag = 0;                                            // 最小生成树的边计数
    bool heapEmpty;
    for (int i = 0; i < G.VerticesNum(); i++)                  // 将图的所有边插入最小堆H中
        for (Edge e = G. FirstEdge(i); G.IsEdge(e);e = G. NextEdge(e))
            if (G.FromVertex(e) < G.ToVertex(e))               // 对于无向图，防止重复插入边
                H.Insert(e);
    int EquNum = G.VerticesNum();                              // 开始n个顶点分别作为一个等价类
    while (EquNum > 1) {                                       // 当等价类的个数大于1时合并等价类
        heapEmpty = H.isEmpty();
        if (!heapEmpty)
            Edge e = H.RemoveMin();                            // 获得一条权最小的边
        if (heapEmpty || e.weight == INFINITY) {
            cout << "不存在最小生成树." <<endl;
            delete [] MST;                  // 释放空间
            MST = NULL;                     // MST赋为空
            return;
        }
        int from = G.FromVertex(e);         // 记录该条边的信息
        int to = G.ToVertex(e);
        if (A.Different(from,to)) {                            // 边e的两个顶点不在一个等价类
            A.Union(from,to);                                  // 将边e的两个顶点所在的等价类合并为一个
            AddEdgetoMST(e,MST,MSTtag++);                      // 将边e加到MST
            EquNum--;                                          // 等价类的个数减1
        }
    }
}
```

Kruskal 算法的关键

- 取权值最小的边
  - 首先对边进行完全排序?
  - 使用最小值堆来实现！ 一次取一条边。实际上在完成MST 前仅需访问一小部分边。
- 确定两个顶点是否属于同一等价类
  - 可基于树的父指针表示法中的UNION/FIND算法
    - 判断两个结点是否在同一集合中
    - 归并两个集合
  - UNION/FIND算法用一可棵树代表一个集合，若两个结点在同一棵树中，则认为它们在同一集合中

Kruskal 算法的时间代价

- 使用了路径压缩，Different() 和 Union() 函数几乎是常数
- 假设可能对几乎所有边都判断过了
  - 则最坏情况下算法时间代价为 Θ (elog e)，即堆排序的时间
- 通常情况下只找了略多于 n 次，MST 就已经生成
  - 时间代价接近于 Θ (nlog e)

最小生成树讨论

- 最小生成树是否唯一?
  - 不一定
  - 试设计算法生成所有的最小生成树
- 若边的权都不相等
  - 一定是唯一的，试证明之
- Prim与Kruskal均为贪心法
  - 正确性证明
  - 各自的适用场景