第二章 贪心法

✓目录

- 概述
- 会场安排问题
- 单源最短路径问题
- 哈夫曼编码
- 最小生成树

教学目标

- 理解贪心法的概念
- 掌握贪心法的基本思想和要素
- 理解贪心法的正确性证明方法
- 通过对实例的学习，掌握贪心法的设计策略及解题步骤

学习贪心法的实际意义和学术价值

- 贪心法可以算得上是最接近人们日常思维的一种解题策略
- 简单、直接和高效
- 对范围相当广泛的许多实际问题它通常都能产生整体最优解，在一些情况下，即使采用贪心法不能得到整体最优解，但其最终结果却是最优解的很好近似解。
- 基于此，它在对NP完全问题的求解中发挥着越来越重要的作用。另外，近年来贪心法在各级各类信息学竞赛、ACM程序设计竞赛中经常出现，竞赛中的一些题目常常需要选手经过细致的思考后得出高效的贪心算法。

贪心法的基本思想

- 基本思想
  - 从问题的某一个初始解出发，在每一个阶段都根据贪心策略来做出当前最优的决策，逐步逼近给定的目标，尽可能快地求得更好的解。当达到算法中的某一步不能再继续前进时，算法终止。
- 得出的结论
  - 每个阶段面临选择时，贪心法都做出对眼前来讲是最有利的选择
  - 选择一旦做出不可更改，即不允许回溯
  - 根据贪心策略来逐步构造问题的解

贪心法的基本要素

- 最优子结构性质
  - 当一个问题的最优解一定包含其子问题的最优解时
  - 采用反证法证明
- 贪心选择性质
  - 所求问题的整体最优解可以通过一系列局部最优的选择获得，即通过一系列的逐步局部最优选择使得最终的选择方案是全局最优的

贪心法的解题步骤

- 分解：将原问题分解为若干个相互独立的阶段。
- 解决：对于每个阶段依据贪心策略进行贪心选择，求出局部的最优解。
- 合并：将各个阶段的解合并为原问题的一个可行解。影响时间复杂性的因素

会场安排问题

- 问题描述
  - 设有n个会议的集合C={1,2,…,n}，其中每个会议都要求使用同一个资源（如会议室），而在同一时间内只能有一个会议使用该资源。每个会议i都有要求使用该资源的起始时间b_i和结束时间e_i，且$b_i < e_i$。如果选择了会议i使用会议室，则它在半开区间$[b_i, e_i)$内占用该资源。如果$[b_i, e_i)$与$[b_j, e_j)$不相交，则称会议i与会议j是相容的。会场安排问题要求在所给的会议集合中选出最大的相容活动子集，也即尽可能地选择更多的会议来使用资源。

贪心策略

- 选择最早开始时间且不与已安排会议重叠的会议
- 选择使用时间最短且不与已安排会议重叠的会议
- 选择具有最早结束时间且不与已安排会议重叠的会议

算法设计

- 步骤1：初始化。开始时间存B；结束时间存E中且按照结束时间的非减序排序，B相应调整；集合A存储解，会议i如果在集合A中，当且仅当A[i]=true；
- 步骤2：根据贪心策略，首令A[1]=true；
- 步骤3：依次扫描每一个会议，如果会议i的开始时间不小于最后一个选入A中的会议的结束时间，则将会议i加入A中；否则，放弃，继续检查下一个会议与A中会议的相容性。

示例

【例2-1】设有11个会议等待安排，用贪心法找出满足目标要求的会议集合。这些会议按结束时间的非减序排列如表2-1所示。11个会议按结束时间的非减序排列表：

会议i	1	2	3	4	5	6	7	8	9	10	11
开始时间b_i	1	3	0	5	3	5	6	8	8	2	12
结束时间e_i	4	5	6	7	8	9	10	11	12	13	14

会议集合为{1,4,8,11}

算法分析

- 会场安排问题的主要步骤
  - 排序
  - 遍历
- 每个步骤的复杂度分析

算法正确性证明

- 问题存在从贪心选择开始的最优解
  - 贪心选择性质
    - 采用归纳法
- 一步一步的贪心选择能够得到问题的最优解
  - 最优子结构性质
    - 采用反证法

单源最短路径问题

- 问题描述

  - 给定一个有向带权图 G=(V，E)，其中每条边的权是一个非负实数。另外，给定V中的一个顶点，称为源点。现在要计算从源点到所有其它各个顶点的最短路径长度，这里的路径长度是指路径上经过的所有边上的权值之和。

Dijkstra算法思想

- 模拟法（拉绳子）
- 按各个顶点与源点之间路径长度的递增次序，生成源点到各个顶点的最短路径的方法，即先求出长度最短的一条路径，再参照它求出长度次短的一条路径，依此类推，直到从源点到其它各个顶点的最短路径全部求出为止。

算法设计

- 设u为源点。集合S中的顶点到源点的最短路径的长度已经确定，集合V-S中所包含的顶点到源点的最短路径的长度待定。
- 特殊路径：从源点出发只经过S中的点到达V-S中的点的路径。
- 贪心策略：选择特殊路径长度最短的路径，将其相连的V-S中的顶点加入到集合S中。

求解步骤

- 步骤1：设计合适的数据结构。带权邻接矩阵C，即如果< u, x > Є E，令C[u][x]=<u, x >的权值，否则，C[u][x]=无穷；采用数组dist来记录从源点到其它顶点的最短路径长度；采用数组p来记录最短路径；
- 步骤2：初始化。令集合S={u}，对于集合V-S中的所有顶点x，设置dist[x]=C[u][x]；如果顶点i与源点相邻，设置p[i]=u，否则p[i]=-1；
- 步骤3：在集合V-S中依照贪心策略来寻找使得dist[x]具有最小值的顶点t，t就是集合V-S中距离源点u最近的顶点。
- 步骤4：将顶点t加入集合S中，同时更新集合V-S；
- 步骤5：如果集合V-S为空，算法结束；否则，转步骤6；
- 步骤6：对集合V-S中的所有与顶点t相邻的顶点x，如果dist[x]>dist[t]+C[t][x]，则dist[x]=dist[t]+C[t][x]并设置p[x]=t。转步骤3。

构造实例

【例2-2】在如图2-2所示的有向带权图中，求源点0到其余顶点的最短路径及最短路径长度。

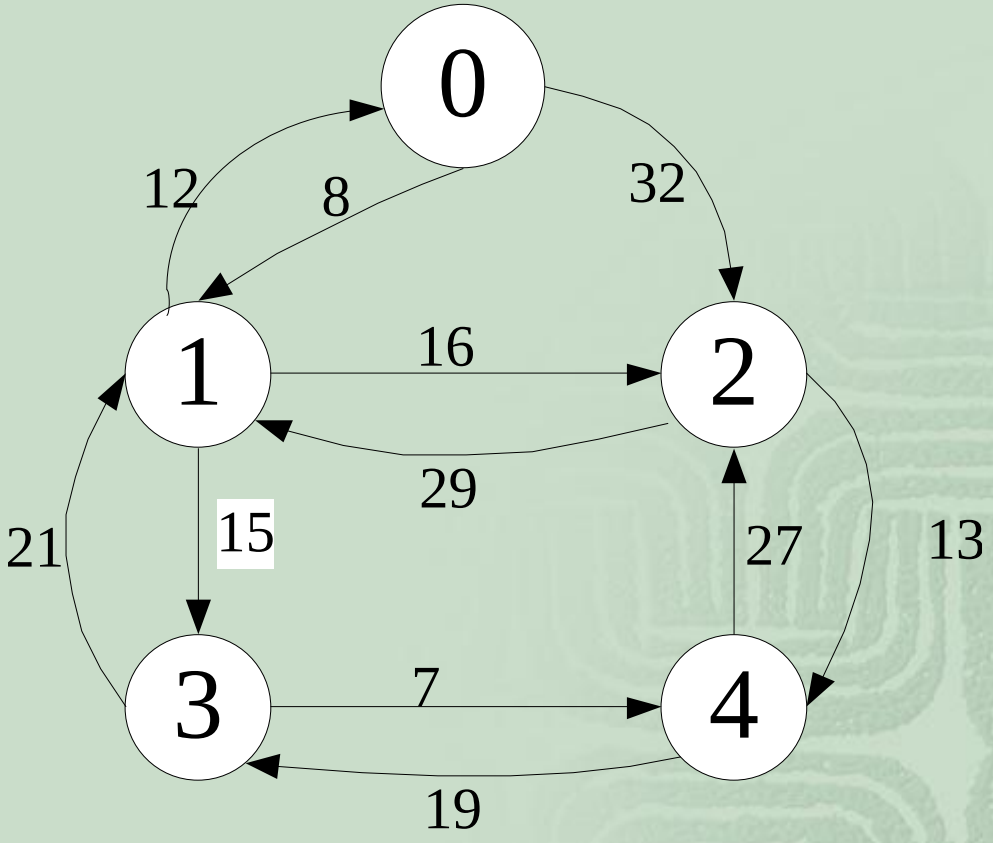
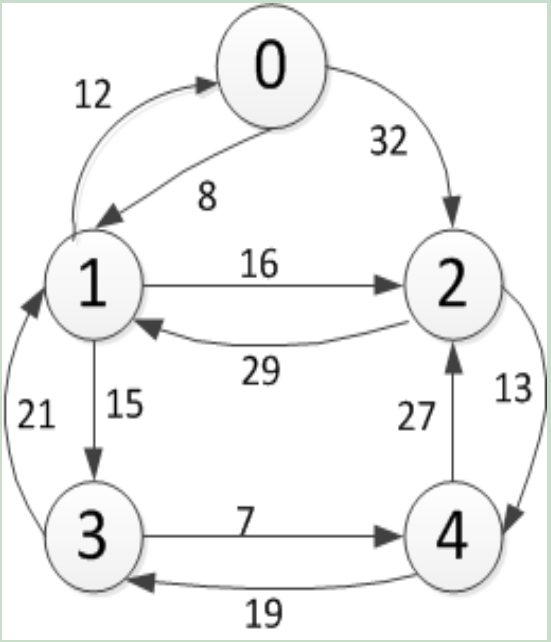


图2-2 有向带权图

实例的求解过程



S	V-S	dist[1]	dist[2]	dist[3]	dist[4]	t
{0}	{1,2,3,4}	8	32	max	max	1
{0,1}	{2,3,4}	-	24	23	max	3
{0,1,3}	{2,4}	-	24	-	30	2
{0,1,3,2}	{4}	-	-	-	30	4
{0,1,3,2,4}	{}					

实例的求解过程

	S	V-S	dist[1]	dist[2]	dist[3]	dist[4]	t
初始	{0}	{1,2,3,4}	8	32	max	max	1
1	{0,1}	{2,3,4}	-	24	23	max	3
2	{0,1,3}	{2,4}	-	24	-	30	2
3	{0,1,3,2}	{4}	-	-	-	30	4
4	{0,1,3,2,4}	{}					

前驱数组P变化情况表

	0	1	2	3	4
初始	-1	0	0	-1	-1
1	-1	0	1	1	-1
2	-1	0	1	1	3
3	-1	0	1	1	3
4	-1	0	1	1	3

依据此表，求得从源点0到顶点4的最短路径为0，1，3，4；从源点0到顶点3的最短路径为：0，1，3；而源点0到顶点2的最短路径为0，1，2；源点0到顶点1的最短路径为0，1。

算法描述及分析

- 从算法的描述中，不难发现语句
  if((!s[j])&&(dist[j]<temp))
  对算法的运行时间贡献最大，因此选择将该语句作为基本语句。当外层循环标号为1时，该语句在内层循环的控制下，共执行n次，外层循环从1~n，因此，该语句的执行次数为$n*n=n^2$，算法的时间复杂性为$O(n^2)$。

- 实现该算法所需的辅助空间包含为数组s和变量i、j、t和temp所分配的空间，因此，Dijkstra算法的空间复杂性为O(n)。

算法正确性证明

- 首先证明最优子结构性质
- 然后利用最优子结构证明贪心选择性
  - 贪心体现在：通过S集合搜索到达V-S集合的最短距离。t节点的上一个节点不是来自S，那么一定来自V-S，根据最优子结构性质，上一个节点一定是已算出的最短距离点，那么上一个点一定在S中，矛盾产生。

哈夫曼编码

- 哈夫曼编码问题的引出
  - 不等长编码方法出现的问题：任何一个字符的编码都不能是其它字符编码的前缀（即前缀码特性），否则译码时将产生二义性。那么如何来设计前缀编码呢？利用二叉树来进行设计。具体做法是：约定在二叉树中用叶子结点表示字符，从根结点到叶子结点的路径中，左分支表示“0”，右分支表示“1”。那么，从根结点到叶子结点的路径分支所组成的字符串做为该叶子结点字符的编码，可以证明这样的编码一定是前缀编码，这棵二叉树即为编码树。
  - 剩下的问题是怎样保证这样的编码树所得到的编码总长度最小？哈夫曼提出了解决该问题的方法，由此产生的编码方案称为哈夫曼算法。

算法的构造思想

- 以字符的使用频率做权构建一棵哈夫曼树，然后利用哈夫曼树对字符进行编码，俗称哈夫曼编码。具体来讲，是将所要编码的字符作为叶子结点，该字符在文件中的使用频率作为叶子结点的权值，以自底向上的方式、通过执行n-1次的“合并”运算后构造出最终所要求的树，即哈夫曼树，它的核心思想是让权值大的叶子离根最近。

- 采取的贪心策略：每次从树的集合中取出双亲为0且权值最小的两棵树作为左、右子树，构造一棵新树，新树根结点的权值为其左右孩子结点权之和，将新树插入到树的集合中。

算法的设计

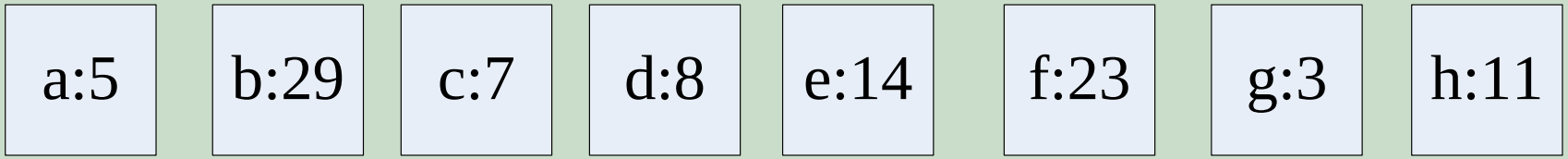
- 步骤1：确定合适的数据结构。
- 步骤2：初始化。构造n棵结点为n个字符的单结点树集合F={T1，T2,…，Tn}，每棵树中只有一个带权的根结点，权值为该字符的使用频率；
- 步骤3：如果F中只剩下一棵树，则哈夫曼树构造成功，转步骤6；否则，从集合F中取出双亲为0且权值最小的两棵树Ti和Tj，将它们合并成一棵新树Zk，新树以Ti为左儿子， Tj为右儿子（反之也可以）。新树Zk的根结点的权值为Ti与Tj的权值之和；
- 步骤4：从集合F中删去Ti、Tj，加入Zk；
- 步骤5：重复步骤3和4；
- 步骤6：从叶子结点到根结点逆向求出每个字符的哈夫曼编码（约定左分支表示字符“0”，右分支表示字符“1”）。则从根结点到叶子结点路径上的分支字符组成的字符串即为叶子字符的哈夫曼编码。算法结束。

构造实例

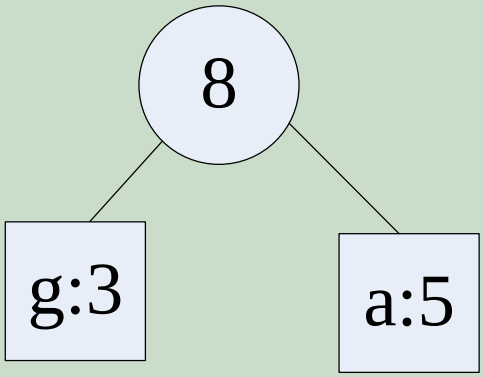
- 【例2-3】已知某系统在通信联络中只可能出现8种字符，分别为a，b，c，d，e，f，g，h，其使用频率分别为0.05，0.29，0.07，0.08，0.14，0.23，0.03，0.11，试设计哈夫曼编码。

- 设权w=(5，29，7，8，14，23，3，11)，n=8，按哈夫曼算法的设计步骤构造一棵哈夫曼编码树，具体过程如下：

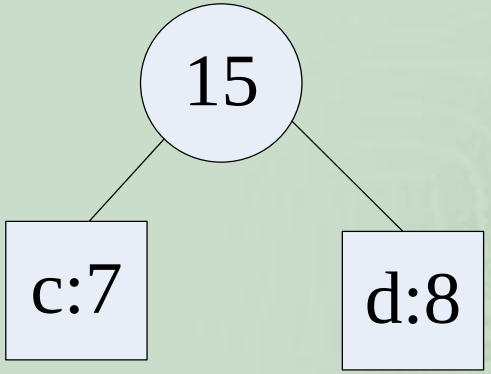
实例构造



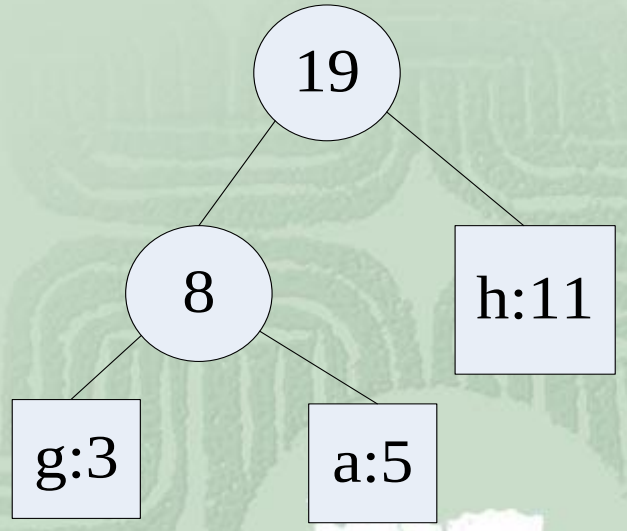
（1）



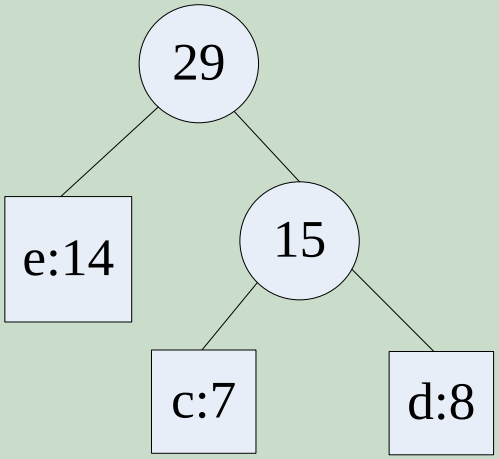
（2）



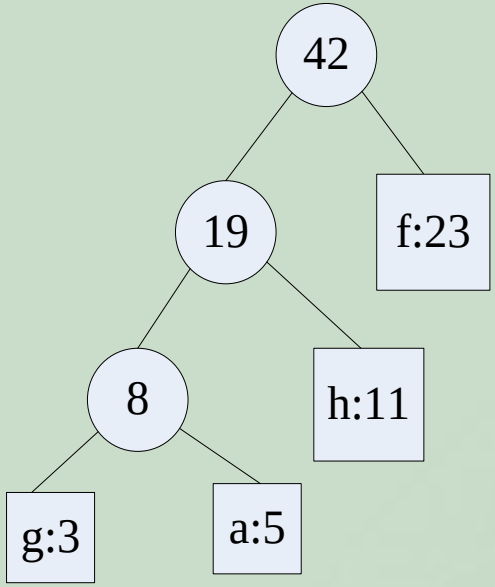
（3）



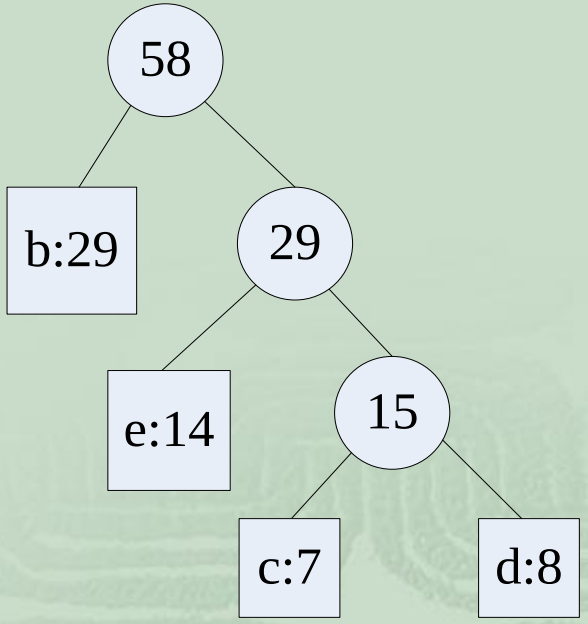
（4）



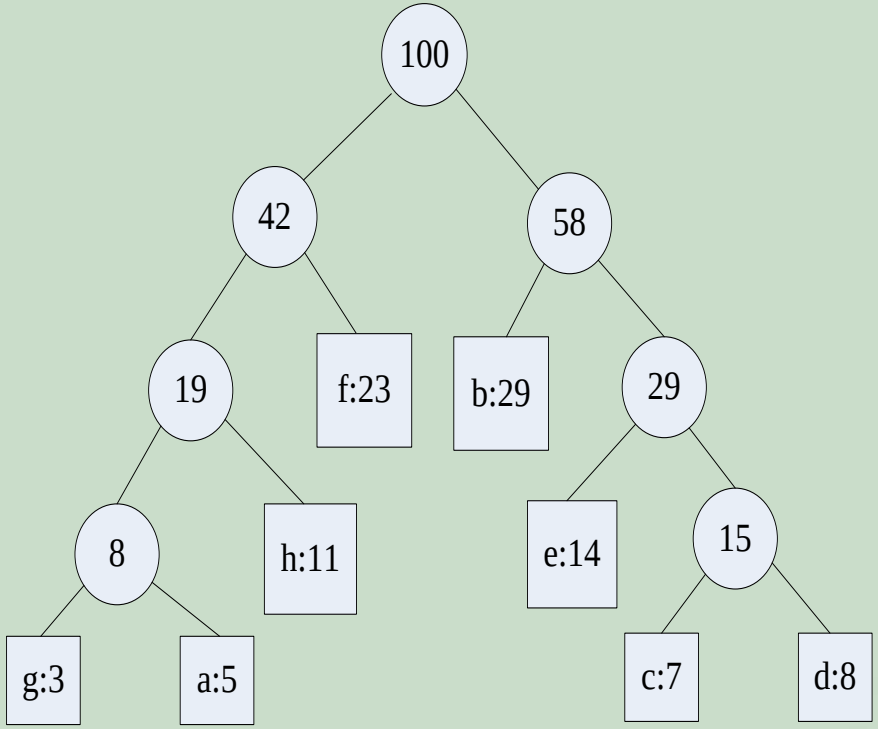
（5）



（6）



（7）



（8）

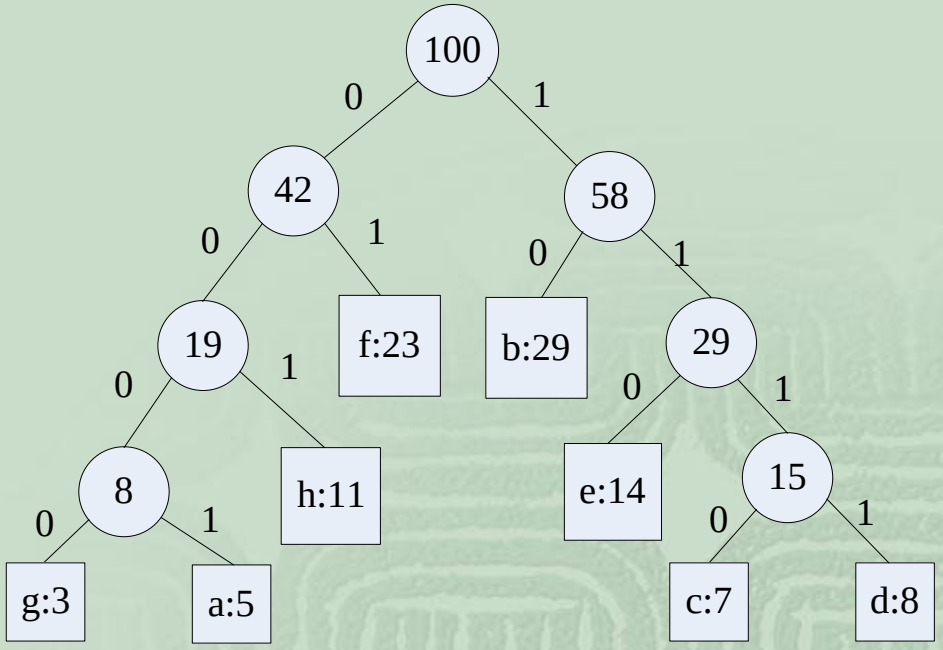


图2-11 哈夫曼编码树

算法描述及分析

- 采用线性结构实现的算法，其复杂性为$O(n^2)$。
- 算法的改进：采用优先队列实现，其复杂性为O(nlogn)。

最小生成树

- 设G=(V,E)是无向连通带权图。E中每条边(v,w)的权为c[v][w]。
- 生成树：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。
- 耗费：生成树上各边权的总和
- 最小生成树：在G的所有生成树中，耗费最小的生成树
- 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。

Prim算法

- 算法思想
  - 设G=(V，E)是无向连通带权图，V={1,2,…,n}；设最小生成树T=(U，TE)，算法结束时U=V，TE⊆ E 。
  - 首先，令U={u0}，TE={}。然后，只要U是V的真子集，就做如下贪心选择：选取满足条件 i ∈ U，j ∈ V-U，且边(i，j)是连接U和V-U的所有边中的最短边，即该边的权值最小。然后，将顶点j加入集合U，边(i，j)加入集合TE。继续上面的贪心选择一直进行到U=V为止，此时，选取到的所有边恰好构成G的一棵最小生成树T。需要注意的是，贪心选择这一步骤在算法中应执行多次，每执行一次，集合TE和U都将发生变化，即分别增加一条边和一个顶点。

算法设计的关键

- 实现Prim算法的关键
  - 找到连接U和V-U的所有边中的最短边！为此必须知道V-U中的每一个顶点与它所连接的U中的每一个顶点的边的信息。
  - 该信息可通过设置两个数组closest和lowcost来体现。其中，closest[j]表示V-U中的顶点j在集合U中的最近邻接顶点，lowcost[j]表示V-U中的顶点j到U中的所有顶点的最短边的权值，即边(j, closest[j])的权值。

算法设计步骤

- 步骤1：确定合适的数据结构。设置带权邻接矩阵C，bool数组s[]，如果s[i]=true，说明顶点i已加入集合U；设置两个数组closest[]和lowcost[]；
- 步骤2：初始化。令集合U={u0}，TE={}，并初始化数组closest、lowcost和s；
- 步骤3：在集合V-U中寻找使得lowcost具有最小值的顶点t，t就是集合V-U中连接集合U中的所有顶点中最近的邻接顶点；
- 步骤4：将顶点t加入集合U，边(t，closest[t])加入集合TE；
- 步骤5：如果集合V=U，算法结束，否则，转步骤6；
- 步骤6：对集合V-U中的所有顶点k，更新其lowcost和closest，用下面的公式更新：if (C[t][k]<lowcost[k]) {lowcost[k]=C[t][k];closest[k]=t;}，转步骤3。

构造实例

按Prim算法对如图2-12所示的无向连通带权图构造一棵最小生成树。

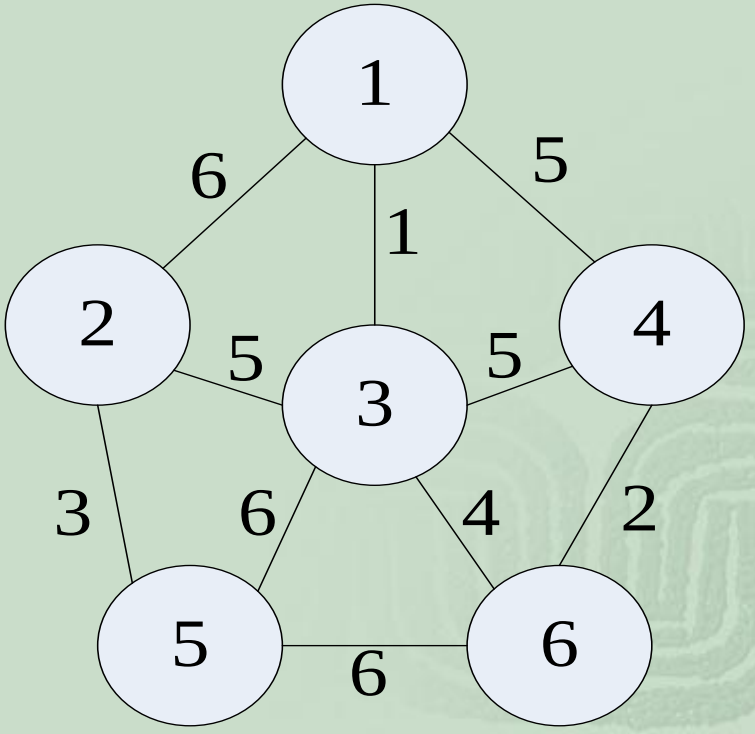


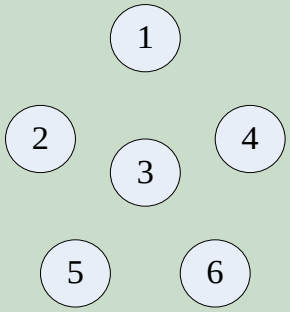
图2-12 无向连通带权图

构造过程中辅助变量值的变化

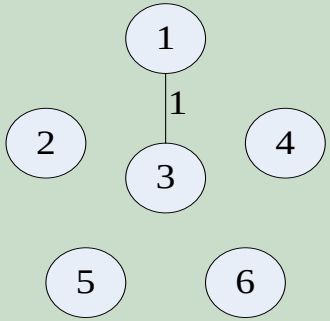
- 假定初始顶点为顶点1，即U={1}；t：集合V-U中距离集合U最近的顶点。

	U	V-U	顶点编号					t
			2	3	4	5	6	
closest lowcost	{1}	{2,3,4,5,6}	1 6	1 1	1 5			3
closest lowcost	{1,3}	{2,4,5,6}	3 5	- -	1 5	3 6	3 4	6
closest lowcost	{1,3,6}	{2,4,5}	3 5	- -	6 2	3 6	- -	4
closest lowcost	{1,3,4,6}	{2,5}	3 5	- -	- -	3 6	- -	2
closest lowcost	{1,2,3,4,6}	{5}	- -	- -	- -	2 3	- -	5
closest lowcost	{1,2,3,4,5,6}	{}	- -	- -	- -	- -	- -	

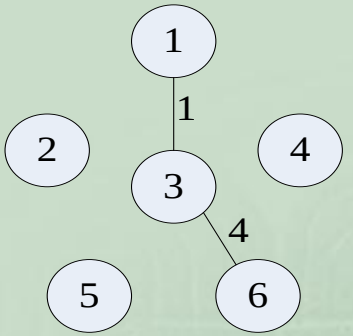
构造过程图示法



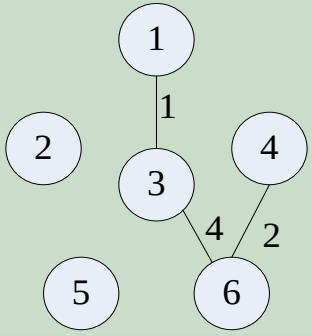
（1）



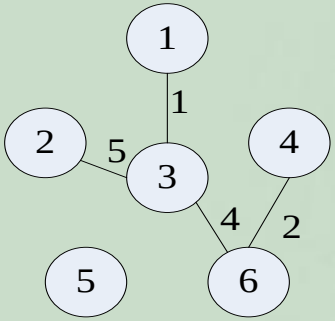
（2）



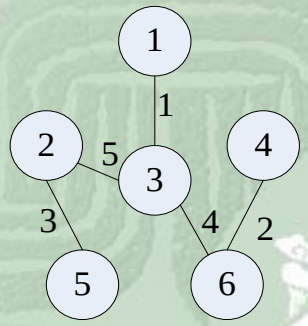
（3）



（4）



（5）



（6）

算法分析

- 从算法Prim的描述可以看出，if((!s[j])&&(lowcost[j]<temp))是算法的基本语句，该语句的执行次数为n^2，由此可得Prim算法的时间复杂度为$O(n^2)$。显然该复杂性与图中的边数无关，因此，Prim算法适合于求边稠密的网的最小生成树。
- 容易看出，Prim算法和Dijkstra算法的用法非常相似，它们都是从余下的顶点集合中选择下一个顶点来构建一棵扩展树，但是千万不要把它们混淆了。由于解决的问题不同，计算的方式也有所不同，Dijkstra算法比较路径的长度，因此必须把边的权值相加，而Prim算法则直接比较给定的边的权值。

Kruskal算法

- 算法思想
  - 设G=(V，E)是无向连通带权图，V={1,2,…,n}；设最小生成树T=(V，TE)，该树的初始状态为只有n个顶点而无边的非连通图T=(V，{})，Kruskal算法将这n个顶点看成是n个孤立的连通分支。它首先将所有的边按权从小到大排序。然后，只要T中的连通分支数目不为1，就做如下的贪心选择：在边集E中选取权值最小的边(i，j)，如果将边(i，j)加入集合TE中不产生回路（或环），则将边(i，j)加入边集TE中，即用边(i，j)将这两个连通分支合并连接成一个连通分支；否则继续选择下一条最短边。在这两种情况下，都把边(i，j)从集合E中删去。继续上面的贪心选择直到T中所有顶点都在同一个连通分支上为止。此时，选取到的n-1条边恰好构成G的一棵最小生成树T。

算法设计的关键——避开环路

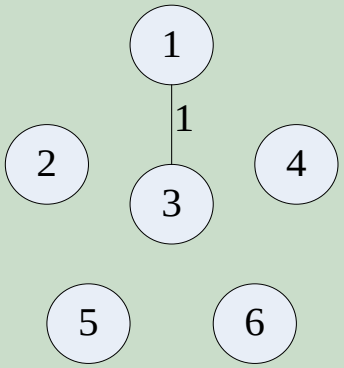
- 那么，怎样判断加入某条边后图T会不会出现回路呢？该算法对于手工计算十分方便，因为用肉眼可以很容易看到挑选哪些边能够避免构成回路，但使用计算机程序来实现时，还需要一种机制来进行判断。**Kruskal**算法用了一个非常聪明的方法，就是运用集合的性质进行判断：如果所选择加入的边的起点和终点都在T的集合里，那么就可以断定一定会形成回路。因为，根据集合的性质，如果两个点在一个集合里，就是包含的关系，那么反映到图上就一定是包含关系。

算法设计——求解步骤

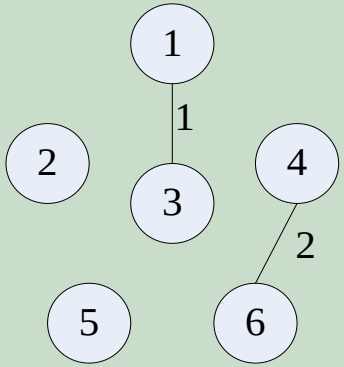
- 步骤1：初始化。将图G的边集E中的所有边按权从小到大排序，边集TE={}，把每个顶点都初始化为一个孤立的分支，即一个顶点对应一个集合；
- 步骤2：在E中寻找权值最小的边(i，j)；
- 步骤3：如果顶点i和j位于两个不同连通分支，则将边(i，j)加入边集TE，并执合并操作将两个连通分支进行合并；
- 步骤4：将边(i，j)从集合E中删去，即E=E-{(i，j)}；
- 步骤5：如果连通分支数目不为1，转步骤2；否则，算法结束，生成最小生成树T。

构造实例——图2-12所示

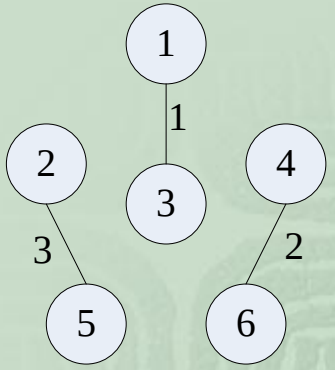
- 首先，将图的边集E中的所有边按权从小到大排序为：1(1，3)、2(4，6)、3(2，5)、4(3，6)、5(1，4) 和(2，3)和(3，4)、6(1，2)和(3，5)和(5，6)。



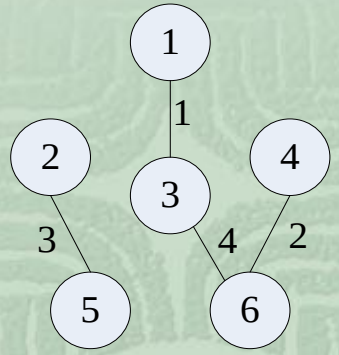
（1）



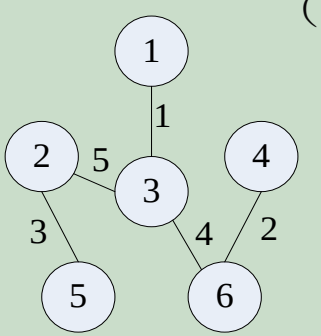
（2）



（3）



（4）



（5）

算法描述及分析

- 从算法的基本思想和设计步骤可以看出，要实现Kruskal算法必须解决以下两个关键问题：（1）选取权值最小的边的同时，要判断加入该条边后树中是否出现回路；（2）不同的连通分支如何进行合并。
- 为了便于解决这两大问题，必须了解每条边的信息。那么，在Kruskal算法中所设计的每条边的结点信息存储结构如下：
- **struct edge{**
- **double weight; //边的权值**
- **int u，v; // u为边的起点，v为边的终点。**
- **}bian[];**
- **sort函数耗时最大，为此算法的时间复杂性为O(eloge)。**

Prim和Kruskal算法的比较

- （1）从算法的思想可以看出，如果图G中的边数较小时，可以采用Kruskal，因为Kruskal算法每次查找最短的边；边数较多可以用Prim算法，因为它是每次加一个顶点。可见，Kruskal适用于稀疏图，而Prim适用于稠密图。
- （2）从时间上讲，Prim算法的时间复杂度为$O(n^2)$，Kruskal算法的时间复杂度为O(eloge)。
- （3）从空间上讲，显然在Prim算法中，只需要很小的空间就可以完成算法，因为每一次都是从个别点开始出发进行扫描的，而且每一次扫描也只扫描与当前顶点集对应的边。但在Kruskal算法中，因为时刻都得知道当前边集中权值最小的边在哪里，这就需要对所有的边进行排序，对于很大的图而言，Kruskal算法需要占用比Prim算法大得多的空间。

思考题

- 0-1背包问题能否用贪心法实现？
- 有足够多的会场来安排一批活动，希望尽可能少用会场。
- 一辆汽车加满油后可行驶n公里，旅途中有若干加油站，如何用最少的加油次数到达目的地。
- 给定n位正整数，去掉k位数，剩下的数字按原先次序排列，如果让生成的数最小。
- 有n位顾客同时等待一项服务(比如理发)，顾客i需要的服务时间是t_i，如何让顾客总等待时间最少？