贪婪法/贪心法

Greedy method

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

贪婪法

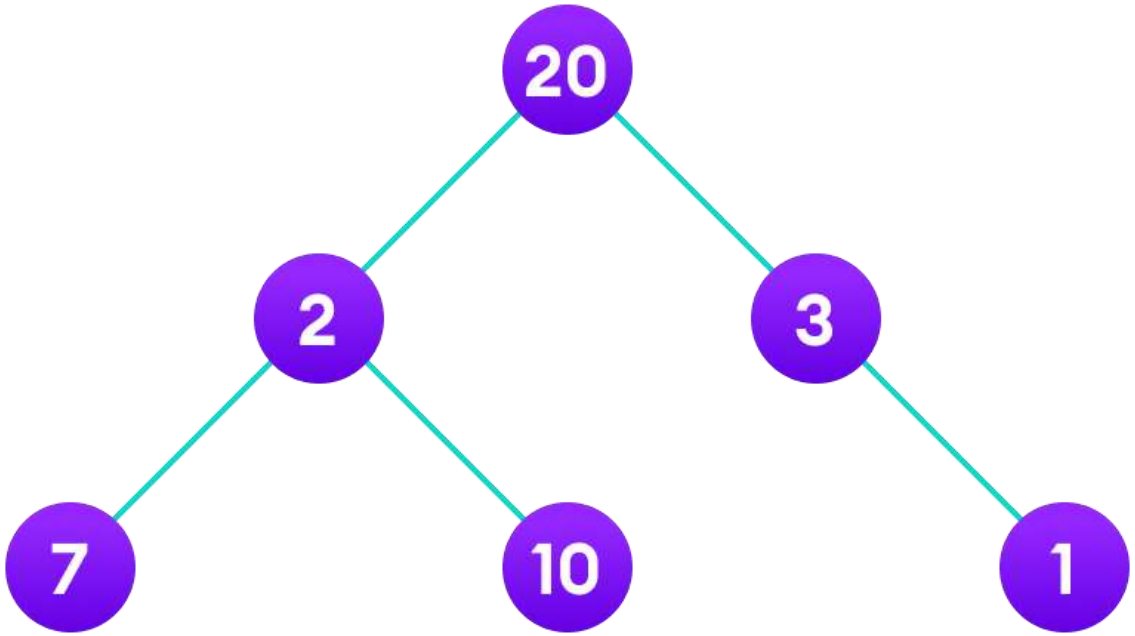
- 又称贪婪法，是求解最优化问题的常用策略，它将问题求解看成**多步决策过程**，在每个决策步骤时，都选取**当前最优**的选择（即局部最优的选择）。
- 爬山的目标是最高峰，有很多条路径，你总是选择最陡的路径。
- 人性也是贪婪的，贪婪的人总是作出眼前局部最有利的选择，而很难综合考虑长远全局的利益。他希望当前最佳选择会得到全局最终利益的最大化。

贪婪法

- 对于一些优化问题，贪婪法做出一系列局部最优选择，确实可以导致全局最优解。
- 例如，单源最短路径的Dijkstra算法从起点出发，每一步都选择距当前顶点距离最近的不会产生回路的顶点，从而获得从起点到各个顶点的最短路径。
- 哈夫曼编码、prim和kruskal最小生成树算法也都用贪婪法获得全局最优解。
- 还有一些问题，贪心法虽然不能获得最优解，但可以提供接近最优的近似解。

贪婪法

- 贪心法往往不能获得最优解。



贪心法/贪婪法的优缺点

- 提出一个贪婪算法是容易的
- 分析贪婪算法的时间复杂度比其他算法如分治递归要容易得多。
- 如何证明贪婪算法能得到全局最优，是很困难的。

适合贪婪法的问题的2个性质

当一个问题具有下面两个性质时，可考虑采用贪婪法。

- **1. 贪心选择性质**

　　指局部最优解导致全局最优解。即在解决问题过程的每一步，贪婪的选择将导致整体最佳解决方案。-证明或反证贪婪算法具有**贪心选择性质**往往很困难

适合贪婪法的问题的2个性质

- **2. 最优子结构**

　　最优子结构性质是指一个问题的最优解可以从子问题的最优解构造出来。换句话说，如果一个问题可以分解成更小的子问题，每个子问题都可以最优地解决，那么可以通过组合这些子问题的最优解来构造原始问题的最优解。

该性质有助于证明贪婪算法是否能得到全局最优解

找零钱

Coin Change 硬币找零

Youtube频道：**hwdong**

博客：**hwdong-net**.github.io

找零钱

- 给定1、5、10、25、100共5种硬币，如何给顾客找最少数目的硬币？
- 贪婪法：每次选择不超过当前金额的最大币值硬币!
- 如：34
- 答：　25+5+4*1
- 如: 279
- 答：2*100+3*25+4*1

贪婪法找零钱算法伪代码：

```
coin_change( x, c[ ] )
    对硬币数组c按价格从大到小排列c_n > c_{n-1}...>c_2 > c_1
    S ← empty
    while x>0:
        k ←小于当前价格的最大硬币c_k
        if 没找到k :
            break
        else:
            x= x- c_k
            S.push(c_k)
```

贪婪法能得到最优解吗？

- 当要支付的金额X介于c_k和c_{k+1}之间： $c_k <= X <= c_{k+1}$，贪婪法将支付c_k，
- 任何最优解也一定选择硬币c_k，
- 反证，如果不选择c_k，将要用面值小于c_k的硬币c_1、c_2、…、c_{k-1}支付X。

k	c_k	最优解	最优解$c_1, c_2, \ldots, c_{k-1}$构成的最大值
1	1	P ≤ 4	-
2	5	N ≤ 1	
3	10	N+D ≤ 2	
4	25	Q ≤ 3	
5	100	无限制	

贪婪法能得到最优解吗？

- 当要支付的金额X介于c_k和c_{k+1}之间： $c_k <= X <= c_{k+1}$，贪婪法将支付c_k，
- 任何最优解也一定选择硬币c_k，
- 反证，如果不选择c_k，将要用面值小于c_k的硬币c_1、c_2、…、c_{k-1}支付。

k	c_k	最优解	最优解$c_1, c_2, \ldots, c_{k-1}$构成的最大值
1	1	P ≤ 4	-
2	5	N ≤ 1	4
3	10	N+D ≤ 2	
4	25	Q ≤ 3	
5	100	无限制	

贪婪法能得到最优解吗？

- 当要支付的金额X介于c_k和c_{k+1}之间： $c_k <= X <= c_{k+1}$，贪婪法将支付c_k，
- 任何最优解也一定选择硬币c_k，
- 反证，如果不选择c_k，将要用面值小于c_k的硬币c_1、c_2、…、c_{k-1}支付。

k	c_k	最优解	最优解$c_1, c_2, \ldots, c_{k-1}$构成的最大值
1	1	P ≤ 4	-
2	5	N ≤ 1	4
3	10	N+D ≤ 2	4+5=9
4	25	Q ≤ 3	
5	100	无限制	

贪婪法能得到最优解吗？

- 当要支付的金额X介于c_k和c_{k+1}之间： $c_k <= X <= c_{k+1}$，贪婪法将支付c_k，
- 任何最优解也一定选择硬币c_k，
- 反证，如果不选择c_k，将要用面值小于c_k的硬币c_1、c_2、…、c_{k-1}支付。

k	c_k	最优解	最优解$c_1, c_2, \ldots, c_{k-1}$构成的最大值
1	1	P ≤ 4	-
2	5	N ≤ 1	4
3	10	N+D ≤ 2	4+5=9
4	25	Q ≤ 3	20+4=24
5	100	无限制	

贪婪法能得到最优解吗？

- 当要支付的金额X介于c_k和c_{k+1}之间： $c_k <= X <= c_{k+1}$，贪婪法将支付c_k，
- 任何最优解也一定选择硬币c_k，
- 反证，如果不选择c_k，将要用面值小于c_k的硬币c_1、c_2、…、c_{k-1}支付。

k	c_k	最优解	最优解$c_1, c_2, \ldots, c_{k-1}$构成的最大值
1	1	P ≤ 4	-
2	5	N ≤ 1	4
3	10	N+D ≤ 2	4+5=9
4	25	Q ≤ 3	20+4=24
5	100	无限制	75+24=99

贪婪法适用所有找零钱问题吗？

- 答案是否定的!
- 货币只有 25 分、20分、10 分、5 分和 1 分四种硬币。需要找给顾客41分钱。
- 贪婪法：**41= 25+10+5+1**。需要**4**枚硬币
- 而**41=20+20+1,**。只要**3**枚硬币

背包问题

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

背包问题

- n个物品的重量w_1、w_2、…、w_n，价值为v_1、v_2、…、v_n，背包限重为W。问：如何装使价值最大？

item	weight	value
1	1	1
2	2	5
3	5	18
4	6	22
5	7	28

W = 11

价值优先

(7,28), (2,5), (1,1)

重量优先

(1,1),(2,5), (5,18)

性价比优先

(7,28), (2,5), (1,1)

背包问题

- n个物品的重量w_1、w_2、…、w_n，价值为v_1、v_2、…、v_n，背包限重为W。问：如何装使价值最大？

item	weight	value
1	1	1
2	2	5
3	5	18
4	6	22
5	7	28

W = 11

价值优先

(7,28), (2,5), (1,1)

重量优先

(1,1),(2,5), (5,18)

性价比优先

(7,28), (2,5), (1,1)

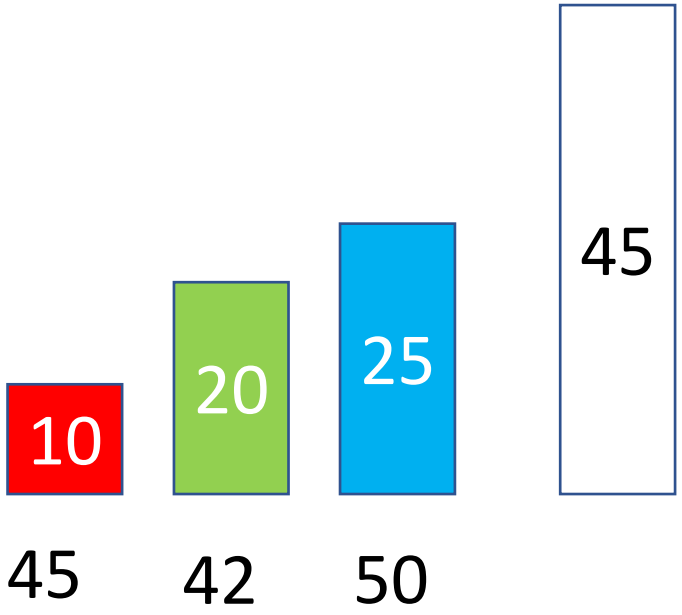
(6,22), (5,18)

分数背包问题

- n个物品的重量w_1、w_2、…、w_n，价值为v_1、v_2、…、v_n，背包限重为W。问：如何装使价值最大？
- 如何允许物品被任意分割，如饮料，这种背包问题称为**分数背包问题**。不允许对物品分割的，称为0-1**背包问题**。如果同一个物品有多个，称为**整数背包问题**。

分数背包问题

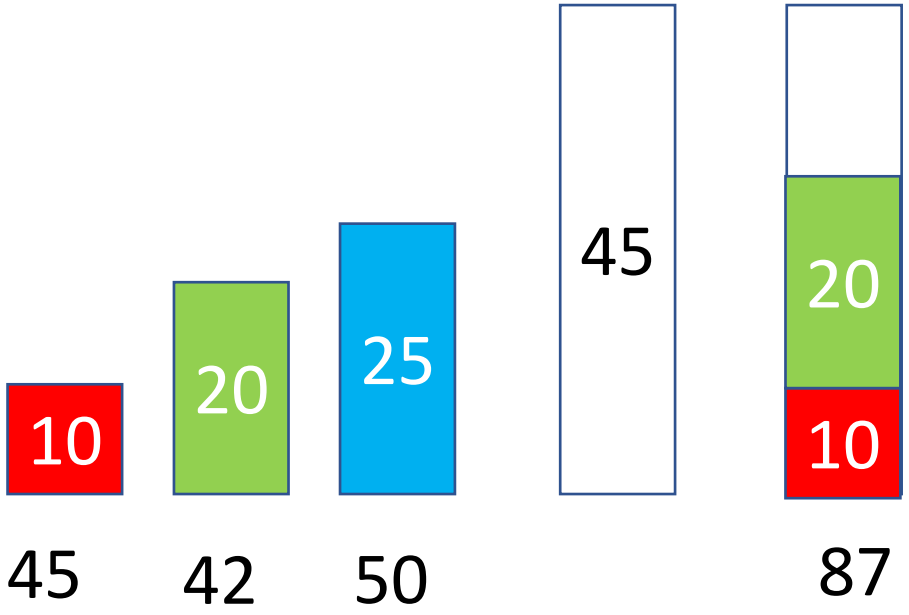
- 有3种饮料，其容量分别为10,20,30，价值分别为60,100,120，如何装满容量为40的杯子，使得总价值最大？



分数背包问题

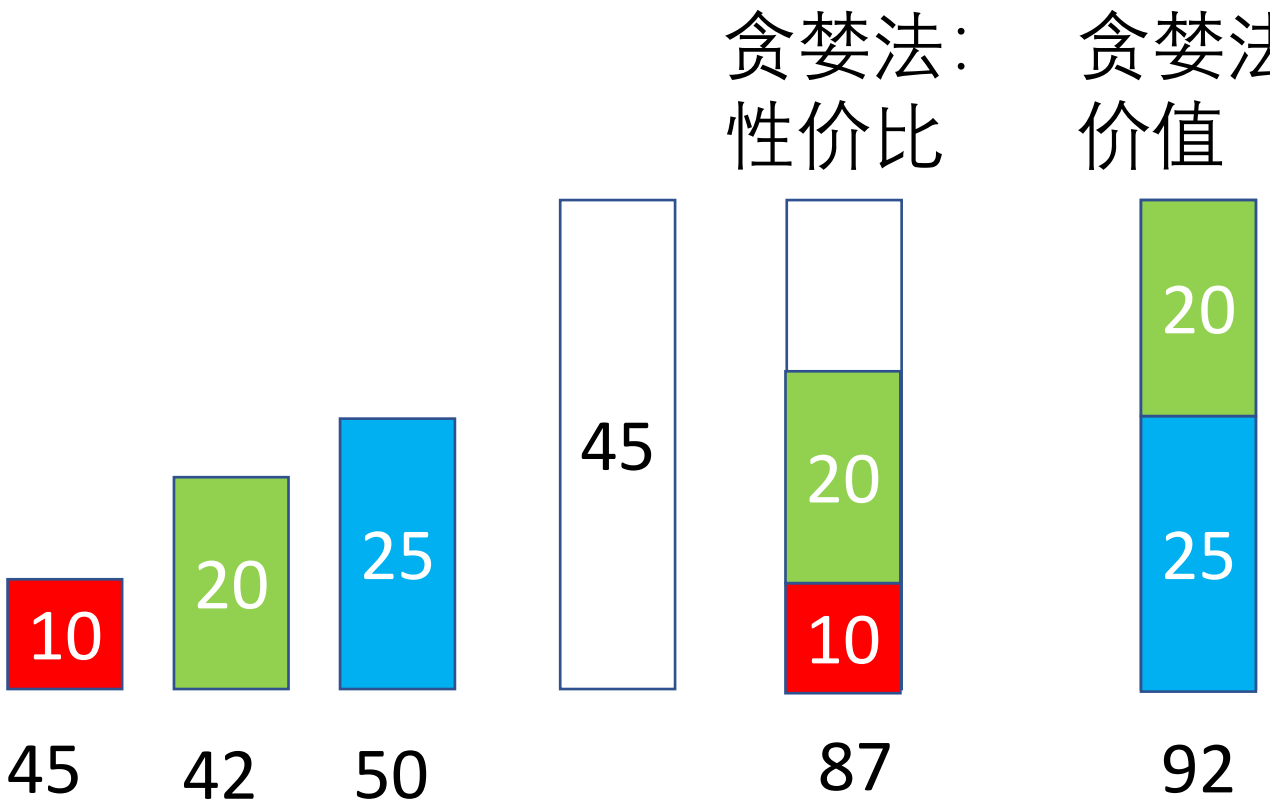
- 有3种饮料，其容量分别为10,20,30，价值分别为60,100,120，如何装满容量为40的杯子，使得总价值最大？

贪婪法：
性价比



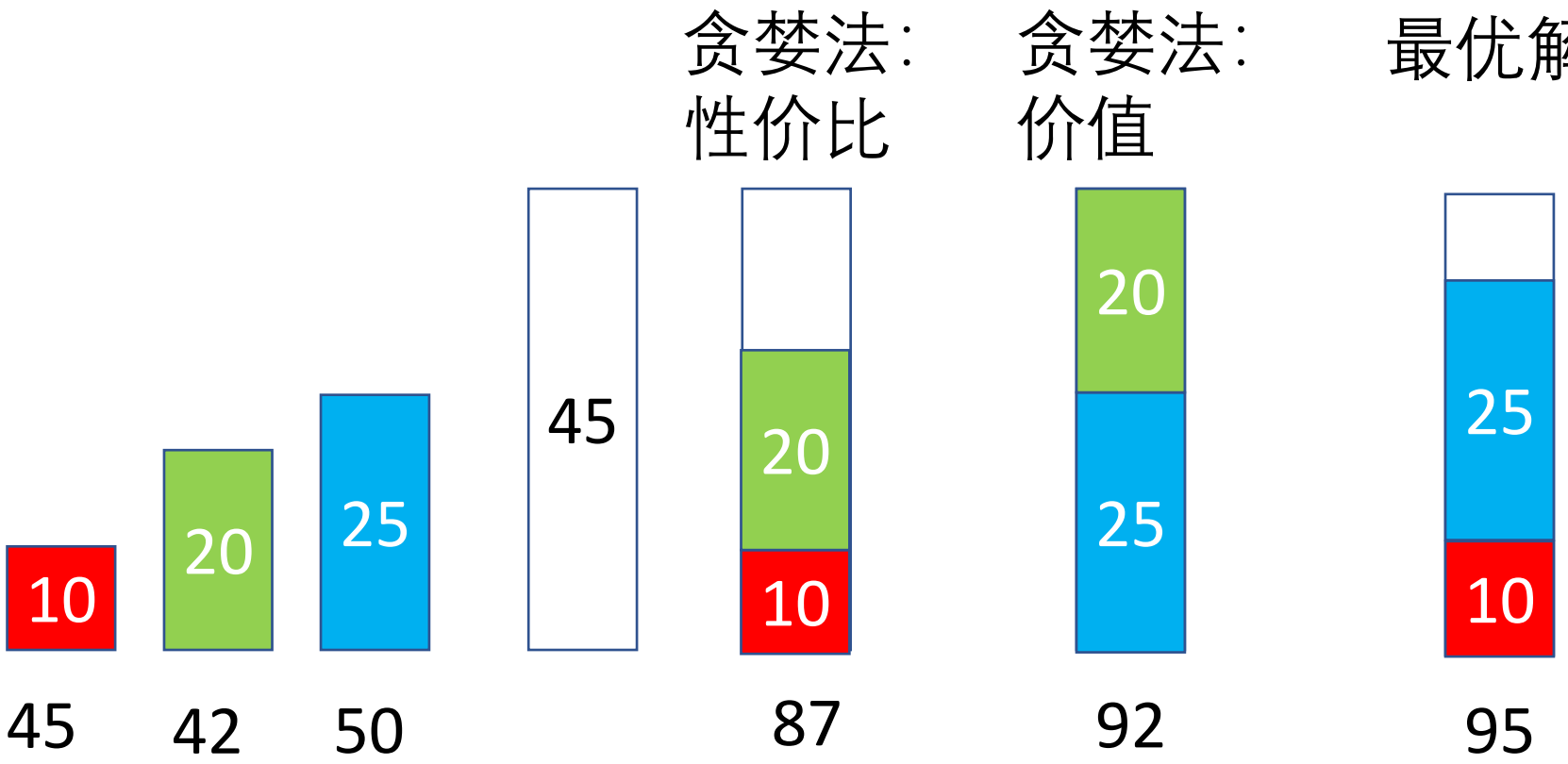
分数背包问题

- 有3种饮料，其容量分别为10,20,30，价值分别为60,100,120，如何装满容量为40的杯子，使得总价值最大？



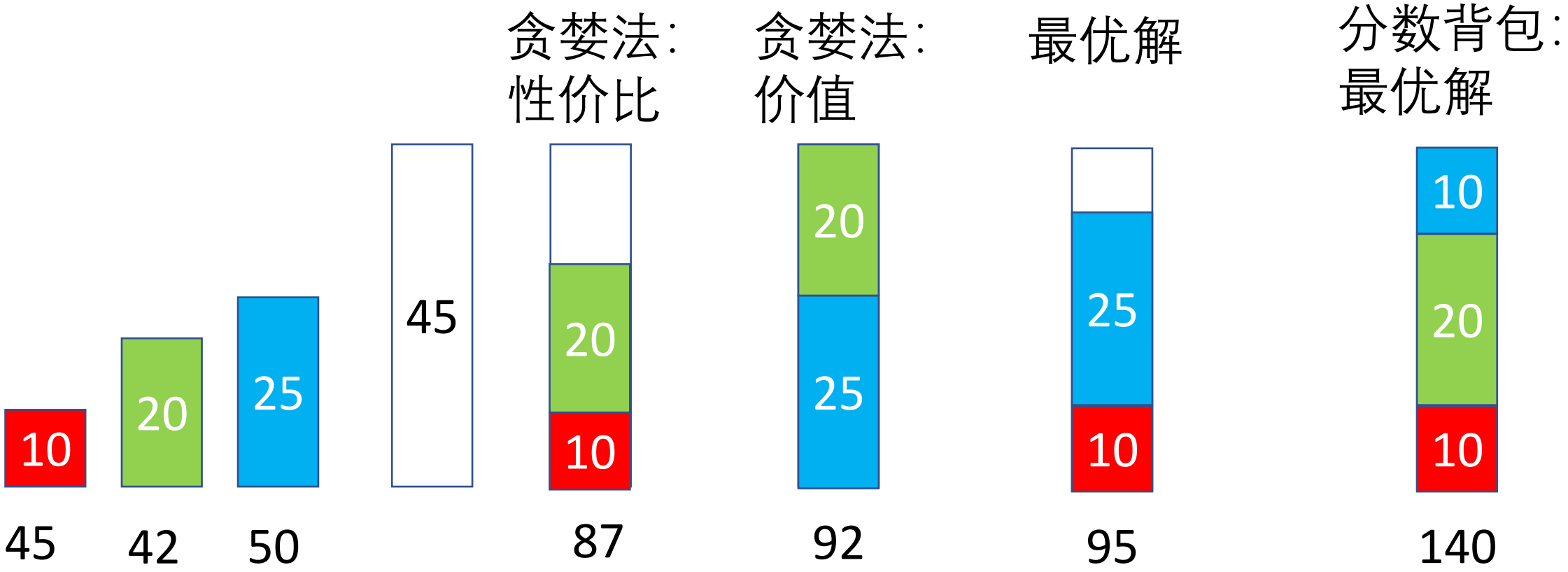
分数背包问题

- 有3种饮料，其容量分别为10,20,30，价值分别为60,100,120，如何装满容量为40的杯子，使得总价值最大？



分数背包问题

- 有3种饮料，其容量分别为10,20,30，价值分别为60,100,120，如何装满容量为40的杯子，使得总价值最大？



分数背包问题

- 贪婪法对于0-1背包问题不能得到最优解，是因为背包会有剩余容量，而分数背包肯定能装满整个背包，采用性价比优先的贪婪法肯定可以得到最优解。

最优解OPT为$\{q_1,q_2,\ldots,q_n\}$。

假设

$$\sum_{i=1}^{n} p_i v_i < \sum_{i=1}^{n} q_i v_i$$

设i是第一个 $p_i \neq q_i$ 的下标。

根据贪婪法策略，必定有： $p_i > q_i$

但OPT是最优解，必定有j>i，使得

$$p_j < q_j$$

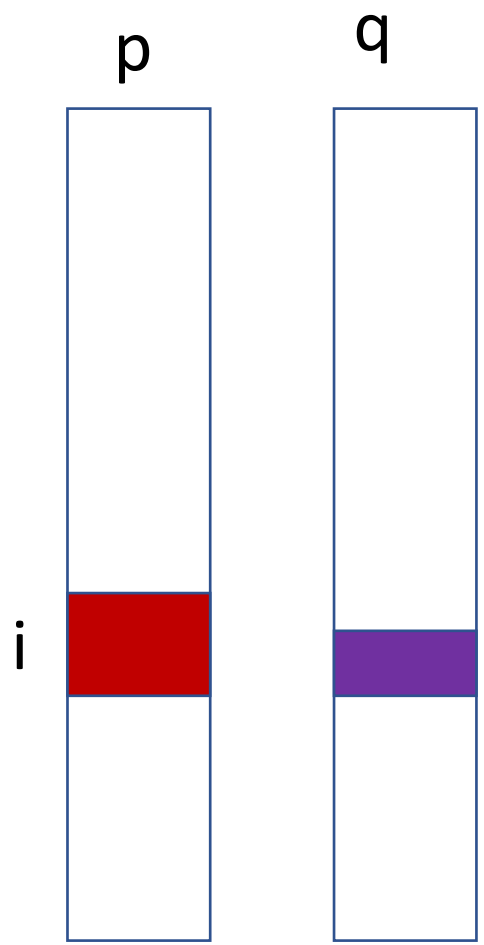
$$q' = \{q'_1, q'_2, \ldots, q'_n\} \text{ where } q'_k = q_k \text{ for all } k \neq i, j$$

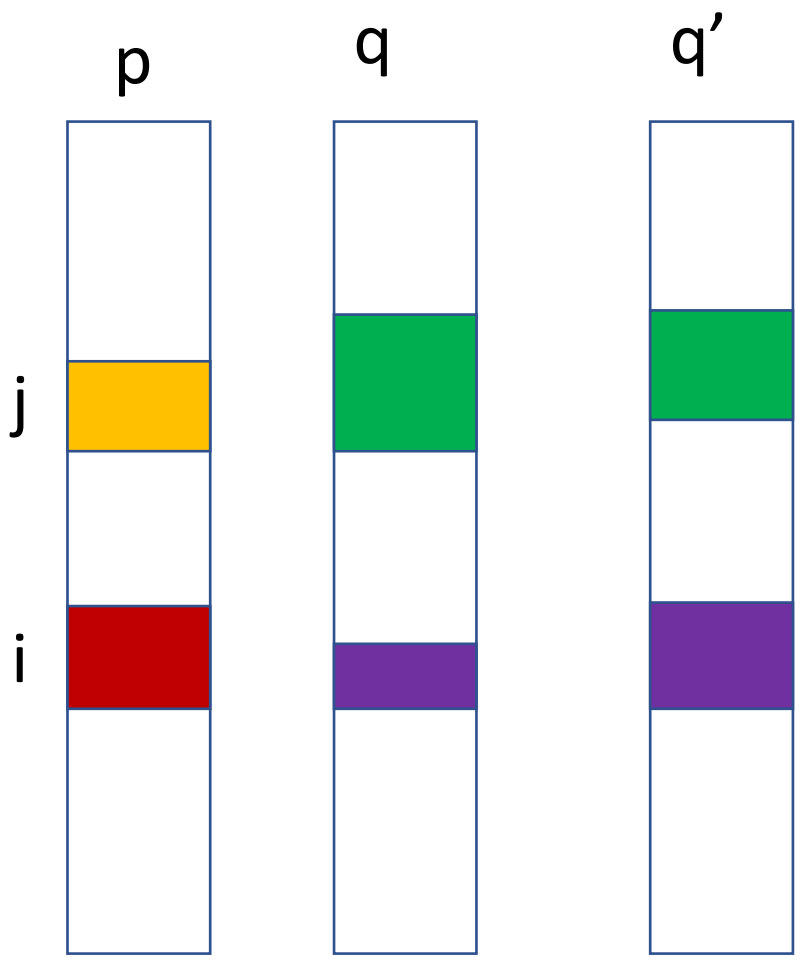
$$q'_i = q_i + \epsilon, \; q'_j = q_j - \epsilon \frac{w_i}{w_j}$$

总重量没有改变： $\sum_{i=1}^{n} q'_i w_i = \sum_{i=1}^{n} q_i w_i$

$$\sum_{i=1}^{n} q'_i v_i = \sum_{i=1}^{n} q_i v_i + \epsilon v_i - \epsilon \frac{w_i}{w_j} v_j > \sum_{i=1}^{n} q_i v_i.$$

q’比OPT更优，产生矛盾!


p
q
i



$$q'_j = q_j - \epsilon \frac{w_i}{w_j}$$

$$q'_i = q_i + \epsilon,$$

贪婪法：

将物品按照性价比v_i/w_i从大到小排序

迭代地选择性价比高的物品放入背包

贪婪法：

将物品按照性价比v_i/w_i从大到小排序

for i = 1 to n:

将第i个物品尽可能多地放入背包

```cpp
struct Item {
    int value;
    int weight;
};

bool cmp(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}
```

```
double fractionalKnapsack(int W, vector<Item> items) {
    sort(items.begin(), items.end(), cmp);

    int currentWeight = 0;
    double finalValue = 0.0;

    for (int i = 0; i < items.size(); i++) {
        if (currentWeight + items[i].weight <= W) {
            currentWeight += items[i].weight;
            finalValue += items[i].value;
        } else {
            int remain = W - currentWeight;
            finalValue += items[i].value * ((double) remain / items[i].weight);
            break;
        }
    }
    return finalValue;
}
```

```cpp
int main() {
    int W = 50;
    vector<Item> items = {{60, 10}, {100, 20}, {120, 30}};

    cout << "Maximum value we can obtain: " << fractionalKnapsack(W, items);
    return 0;
}
```

区间调度问题

interval scheduling

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

区间调度问题

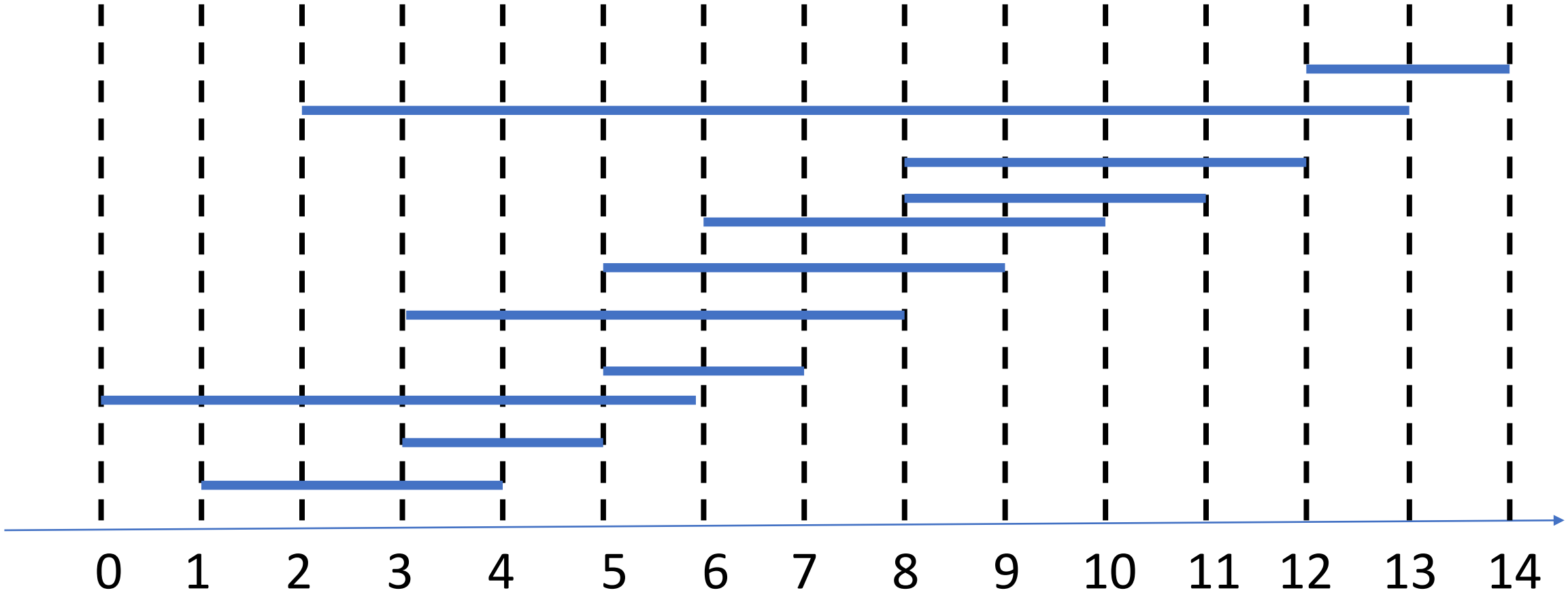
- 区间调度问题是计算机科学中的经典优化问题。给定一组具有开始时间和结束时间的间隔，目标是找到最大数量的非重叠间隔。
- 区间可以是一个任务，有一个开始和结束时间，两个任务是相容的，如果它们的执行区间不重叠。
- 如：有一些人需要使用某个房间，每个人的使用该房间的起始和结束时间为(s_i, e_i)，房间每次只能给一个人使用，问：最多可以给多少人使用？
- 或者有多个工件(任务)要用一台机器加工，每个工件（任务）有一个起始和结束时间为(s_i, e_i)。如何调度，使得尽可能多的工件被加工。

区间调度问题

- 有一组区间{(s_i, e_i)}要放到一个实数轴上，要求其中2个区间(s_i, e_i)、(s_j, e_{ij})的放置不能重叠（冲突），最多能放多少个这样的区间？
- 如：

(1, 4),(3, 5),(0, 6),(5, 7),(3, 8),(5, 9),(6, 10),(8, 11),(8, 12),(2, 13),(12, 14)

(1, 4),(3, 5),(0, 6),(5, 7),(3, 8),(5, 9),(6, 10),(8, 11),(8, 12),(2, 13),(12, 14)

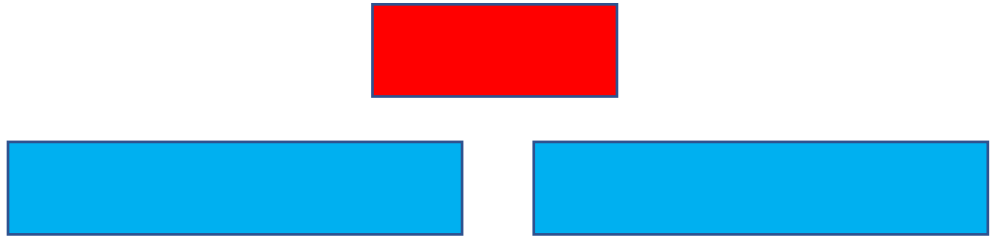


区间调度问题：贪婪法的价值标准

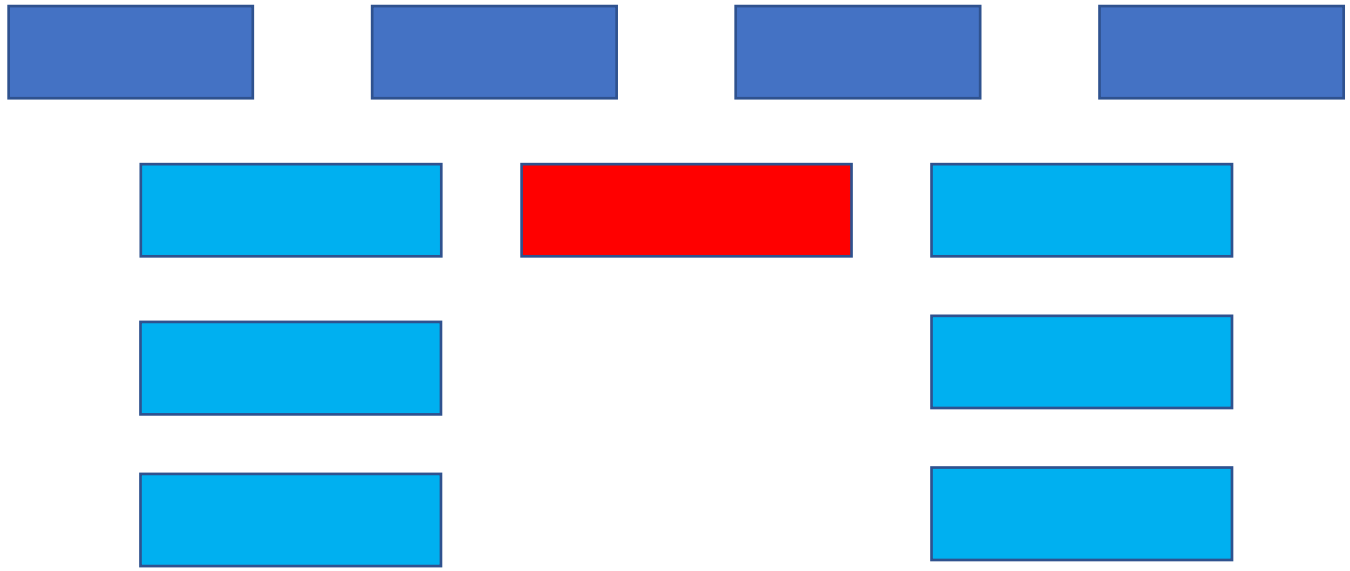
- 最早开始时间s_i排序
- 最早结束时间e_i排序
- 最短区间 （$e_i - s_i$）排序
- 最少冲突数c_i排序

最早开始时间

最短区间



最少冲突



结束时间优先的贪婪法：

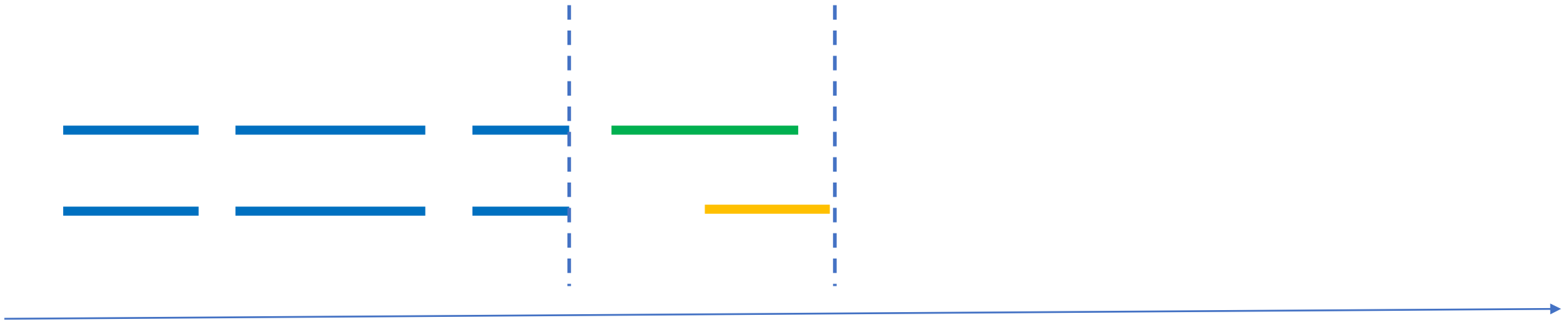
- 按照结束时间排序 O(nlogn)
- S ←空集
- for i = 1 to n: O(n)
- 　if J_i与已有选择S不冲突，则将J_i加入S O(1)

J_i需要与最后一个加入的那个工件的结束时间比较

证明（反证法）：

设$s_1,s_2,\ldots,s_k$贪婪法选择的任务，$t_1,t_2,\ldots,t_m$是最优解选择的任务，

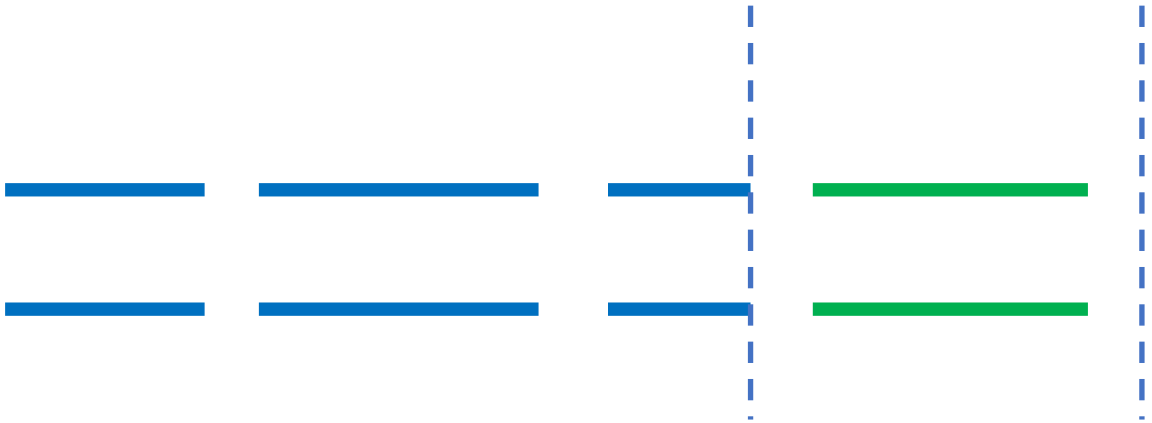
设$I = \max\{i|\ s_j = t_j, j<i;\ \ s_j != t_j\ , j==i\}$ 是所有最优解中第一个与贪婪解不同的任务序号的最大值。



证明（反证法）：

设$s_1,s_2,\ldots,s_k$贪婪法选择的任务，$t_1,t_2,\ldots,t_m$是最优解选择的任务，

设$I = \max\{i \mid s_j = t_j, j<i;\ s_j != t_j, j==i\}$ 是所有最优解中第一个与贪婪解不同的任务序号的最大值。



最优解的第I个替换为贪婪法的第I个，

仍然是最优解，但

贪婪法和它已经有I个相同，矛盾！

证明2

- 设ALG={$s_1,s_2,\ldots,s_k$}是算法的解，而Opt={$t_1,t_2,\ldots,t_m$}是最优解。目标是证明k=m。
- 引理1: 对所有的1<=r<=k，必有$e(s_r)<= e(t_r)$。
- 证：1) r=1时， 根据贪婪法，$e(s_1)$是左右区间结束时间最早的，因此，必有$e(s_1) <= e(t_1)$。

2) 假如r-1,r>=2时，结论成立，即$e(s_{r-1})<= e(t_{r-1})$

要证$e(s_r)<= e(t_r)$

2) 假如r-1,r>=2时，结论成立，即$e(s_{r-1}) <= e(t_{r-1})$

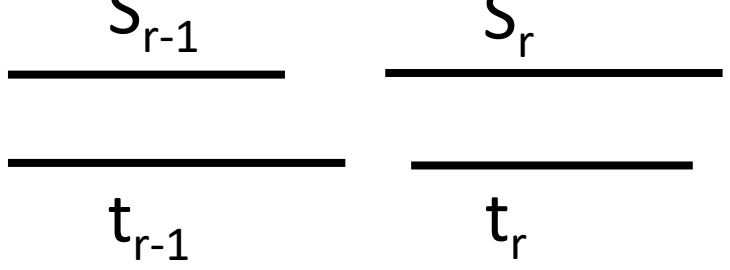
如图所示，因为OPT是相容区间，因此$e(t_{r-1}) <= s(t_r)$

从而，$e(s_{r-1}) <= e(t_{r-1}) <= s(t_r)$

因此 t_r是贪婪法选择s_{r-1}后的相容候选区间之一，即是第r步的候选区间之一。

根据贪婪法，第r步选择的s_r结束时间$e(s_r)$应该最小，因此：

$e(s_r) <= e(t_r)$



- 再证明：k=m
- 反证：如果k<m。OPT必然多出一个k+1的区间t_{k+1}，

从而，$e(t_k) <= s(t_{k+1}) <= s(t_{k+1})$

从而，$e(s_k) <= s(t_{k+1}) <= s(t_{k+1})$

说明贪婪法在第k步后，还存在一个相容的候选区间t_{k+1}，根据贪婪法，必然继续选择这个区间，从而贪婪法的区间个数至少k+1。与贪婪法只有k个区间的假设矛盾!

最优装载问题

Youtube频道： hwdong

博客： hwdong-net.github.io

最优装载问题

- n个集装箱要装上载重量为C的轮船，第i个集装箱的重量为w_i,问：如何装使装上船的集装箱数目最大？
- 这是0-1背包的特殊情形，每个箱子价值都为1。
- 0-1背包找不到多项式时间的解决方法，但这个特殊的背包问题可以用贪婪法在多项式时间里求解。

最优装载问题的贪婪法

- 贪心策略:　轻者优先
- 算法过程:
  - 先按重量从小到大排序
  - 迭代地将箱子装入轮船

伪代码

```
int Loading(T w[], T C, int n){
        sort(w)
        for i=1 to n:
            if   w_i<=C:
                count += 1;    C = C- w_i
            else:
                 break
        return count;
}
```

证明（反证法）：

设箱子按重量排好序：$w_1 \le w_2 \le \ldots \le w_n$

最优解选择的箱子按重量排序为：$t_1 \le t_2 \le \ldots \le t_m$

如果$t_1 != w_1$，将t_1替换为w_1，则不会超过载重量，且箱子数目不会减小，依次类推，t_2替换为w_2，…., t_m替换为w_m。

因为$w_1 + w_2 + \ldots + w_m \le t_1 + t_2 + \ldots + t_m$. 采用贪婪策略的解至少也有m个箱子，即是最优解。

哈夫曼编码

Huffman Coding

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

字符编码： ASCII编码

- 许多编程语言用ASCII编码，每个字符用8位二进制位表示。ASCII可以表示256个不同的字符。

Character	ASCII code	8-bit binary value
Space	32	00100000
e	101	01100101
g	103	01100111
h	104	01101000
o	111	01101111
p	112	01110000
r	114	01110010
s	115	01110011

字符编码： ASCII编码

- 使用 ASCII 编码（每个字符 8 位），13 个字符的字符串“go go gophers”需要 13 * 8 = 104 位。

01100111 01101111 00100000 01100111 01101111 00100000 01100111 01101111 01110000 01101000 01100101 01110010 01110011

通信系统中的字符编码编码

- 某系统在通信联络中只可能出现八种字符(A,B,C,D,E,F,G,H)，采用ASCII码会造成浪费。只要3位二进制就足以区分这8个字符。

Character	Code Value	3-bit binary value
g	0	000
o	1	001
p	2	010
h	3	011
e	4	100
r	5	101
s	6	110
Space	7	111

通信系统中的字符编码编码

- 现在字符串“go go gophers”将被编码为：000 001 111 000 001 111 000 001 010 011 100 101 110。只需要3*13=39位二进制位。

Character	Code Value	3-bit binary value
g	0	000
o	1	001
p	2	010
h	3	011
e	4	100
r	5	101
s	6	110
Space	7	111

这种等长编码方案是否最优呢?

：通信系统中的字符编码编码

- 如果这8种字符(A,B,C,D,E,F,G,H)的使用概率分别为0.05、0.29、0.07、0.08、0.14、0.23、0.03、0.11，如何设计这些字符的二进制编码，以使通信中总码长尽可能短?
- 解决思路：不等长编码

对于出现频率低的字符，可以用超过3位编码，而对于出现频率高的可用少于3位的编码。

- 平均编码长度可表示为：

$$\sum_{i=1}^{n} f_i l_i$$

f_i, l_i分别是第i个字符出现频率和编码长度

前缀码

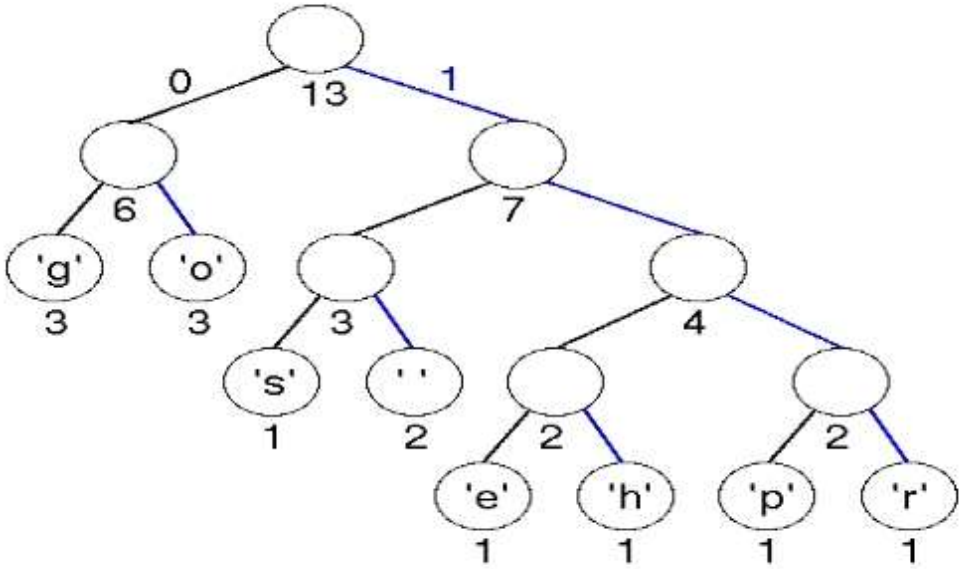
- 对每一个字符规定一个0,1串作为其代码，并要求任一字符的代码都不是其它字符代码的前缀。这种编码称为**前缀码**。
- 非**前缀码**的例子：

  A: 10　B: 01　C: 010　D: 001
- 编码报文：　0101001 表示的到底是 “BBD” 还是"CAB"？

用二叉树来构造前缀码

- 用二叉树可以构造前缀码，其中字符作为二叉树的结点，从根走到该叶子结点，左孩子、右孩子分别用0、1表示，则从根节点到叶子结点得到的0、1串就是该字符的编码。而且这种编码肯定是前缀码。

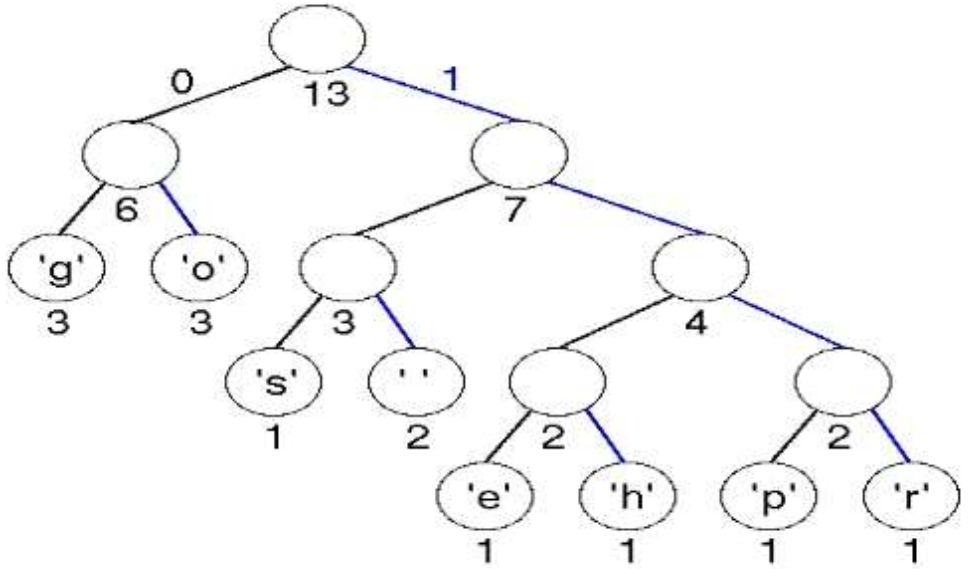
Character	Binary code
' '	101
'e'	1100
'g'	00
'h'	1101
'o'	01
'p'	1110
'r'	1111
's'	100



用二叉树来构造前缀码

- "go go gophers" is encoded (again, spaces would not appear in the bit-stream) as: 00 01 101 00 01 101 00 01 1110 1101 1100 1111 100.
- 一共37位。

Character	Binary code
' '	101
'e'	1100
'g'	00
'h'	1101
'o'	01
'p'	1110
'r'	1111
's'	100

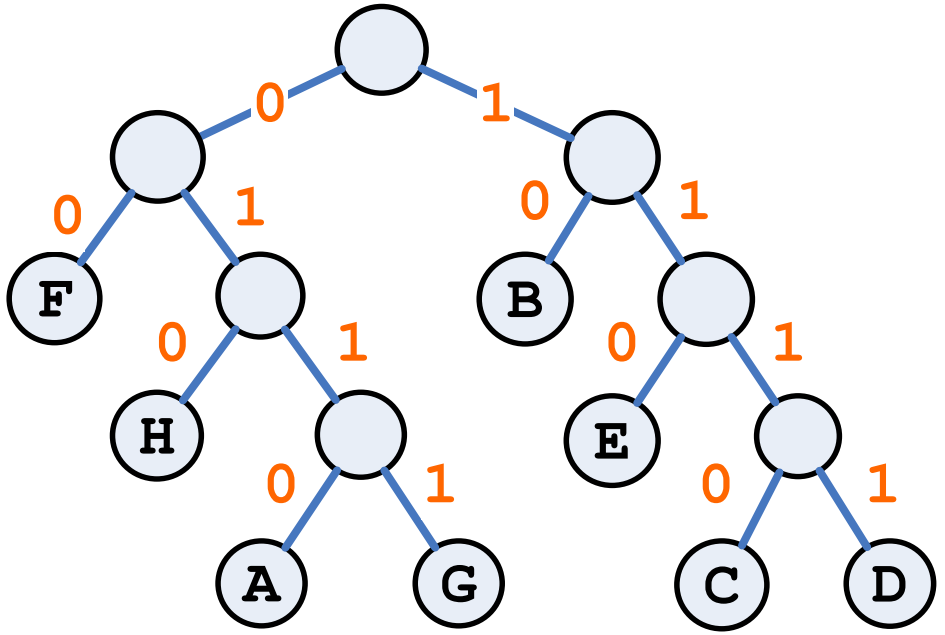


用二叉树来构造前缀码

- 不同的二叉树对应不同的前缀编码。哪种二叉树产生的编码方案最优（平均编码长度最短）？
- 答案：**哈夫曼编码**!

哈夫曼编码(David Huffman发明的)

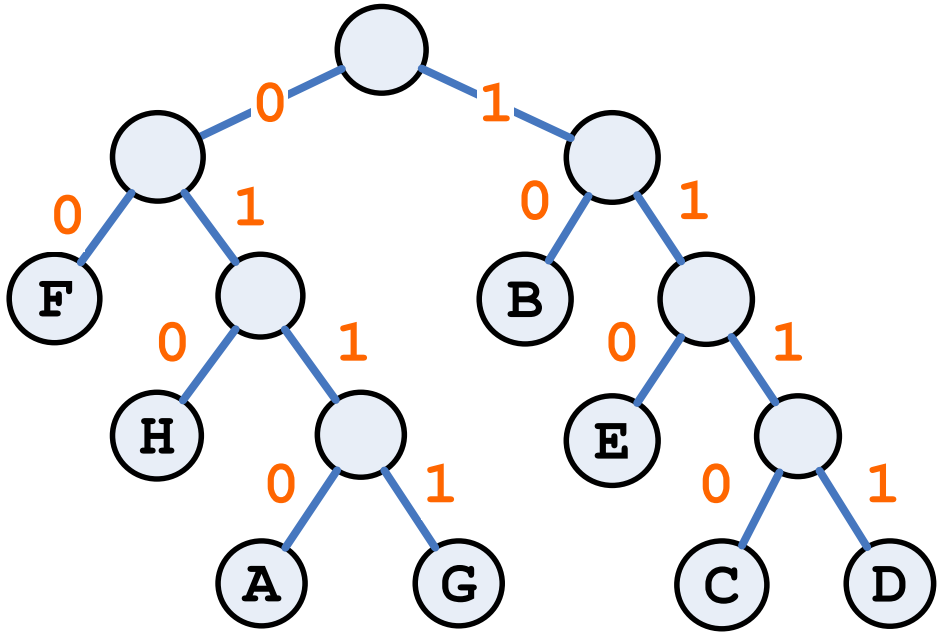
- 每个字符作为一颗二叉树，权值是该字符地出现频率；通过不断地两两合并权值最小的二叉树，最终得到一个二叉树，称为哈夫曼树。



哈夫曼编码(David Huffman发明的)

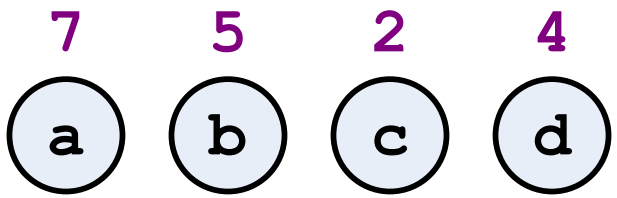
- 左分支表示0，右分支表示1，把从根到叶子的路径上所有分支构成的0,1作为叶子的二进制编码，即为**赫夫曼编码**

- 比如
- A：0110
- B：10
- C：1110
- …



- 设结点a,b,c,d的权值分别为7,5,2,4，试构造赫夫曼树

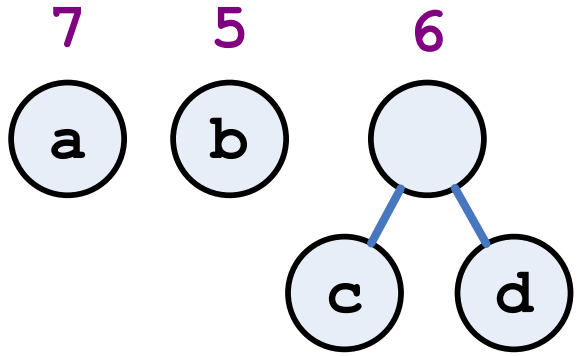
(1)F = {



}

构造叶子结点为根的n棵树

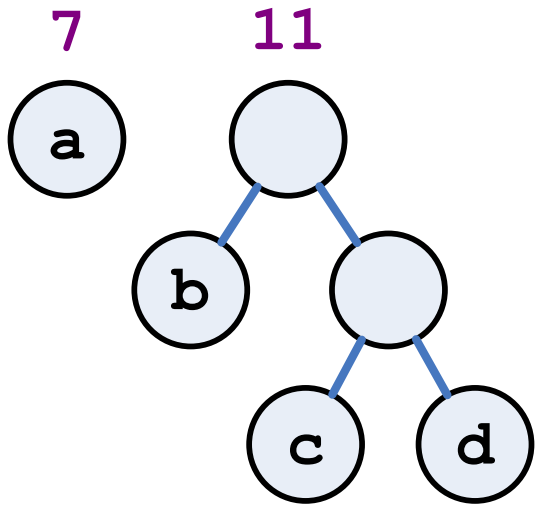
(2)F = {



}

选择权最小的2个棵树两两合并

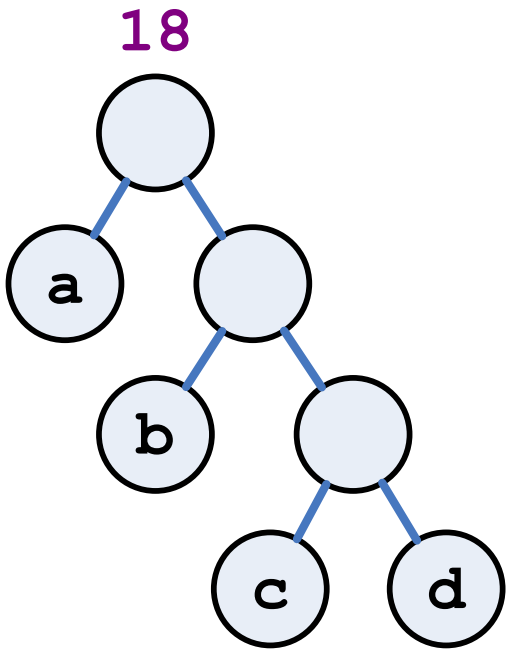
(3)F = {



}

选择权最小的2个棵树两两合并

(4)F = {



}

选择权最小的2个棵树两两合并

哈夫曼树的构造过程

- (1)根据给定的n个权值{w1,w2,…,wn}构成n棵二叉树的集合F={T1,T2,…,Tn}，其中每棵二叉树只含一个带权的根结点，其左右子树均空
- (2)在F中选取两棵根结点的权值最小的树作为左右子树，构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树根结点的权值之和
- (3)在F中删除这两棵二叉树，同时将新得到的二叉树加入F
- (4)重复(2)和(3)，直到F只含一棵二叉树

哈夫曼树构造算法的伪代码

```
huffman(C, prob) {                          // C = chars, prob = probabilities
    for each (x in C) {
        add x to Q sorted by prob[x]        // add all to priority queue
    }
    for (i = 1 to |C| - 1) {                // repeat until only 1 item in queue
        z = new internal tree node
        left[z]  = x = extract-min from Q   // extract min probabilities
        right[z] = y = extract-min from Q
        prob[z]  = prob[x] + prob[y]        // z's probability is their sum
        insert z into Q                     // z replaces x and y
    }
    return the last element left in Q as the root
}
```

详细代码实现

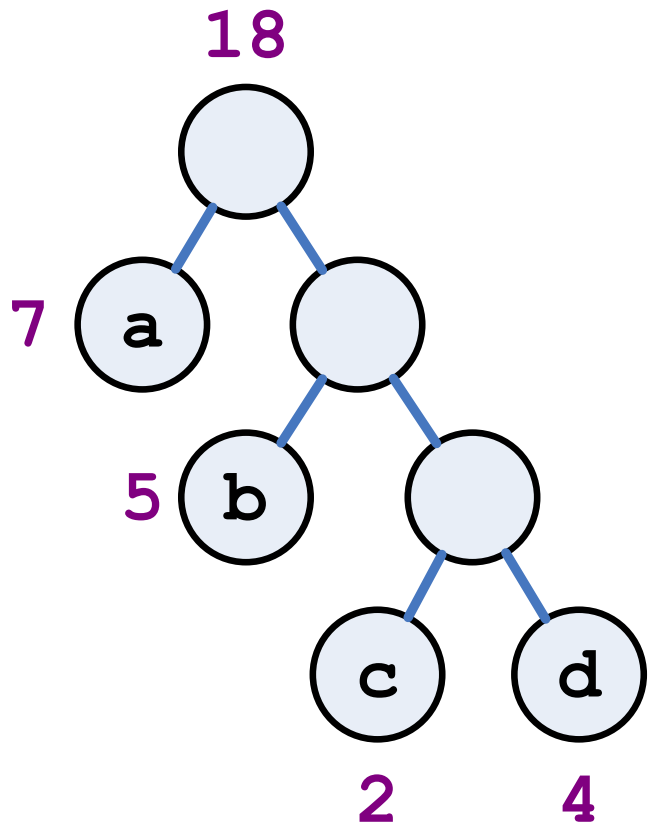
- 网上搜索或参考网易云课堂hwdong的数据结构课程

静态三叉链表

为什么哈夫曼编码的码长最短?

$$B(T) = \sum_{c \epsilon C} p(c) d_T(c)$$

$$B(T) = \frac{7}{18} * 1 + \frac{5}{18} * 2 + \frac{2}{18} * 3 + \frac{4}{18} * 3$$



证明：哈夫曼编码是最优的前缀编码

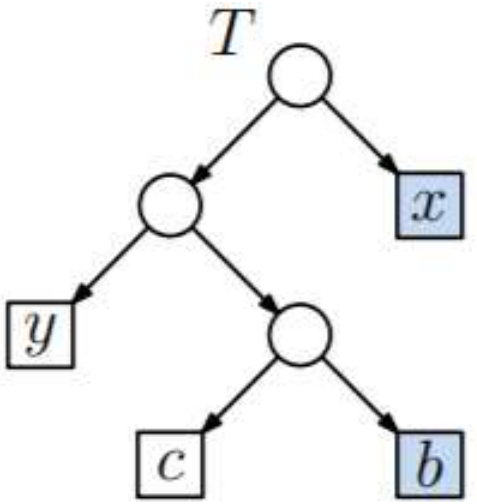
引理1：设x,y是字符集C中概率最低的2个字符，那么必定存在最优的前缀编码使得x, y具有相同最大长度编码且它们仅最后一个编码位不同（它们是具有最大深度的兄弟）。

- 证明：设T是任一最优前缀编码树，{b,c}是最大深度的2个兄弟，如果{x,y}={b,c}，结论成立。
- 如果{x,y}!={b,c}, 假设p(b)≤p(c), p(x)≤p(y)，那么

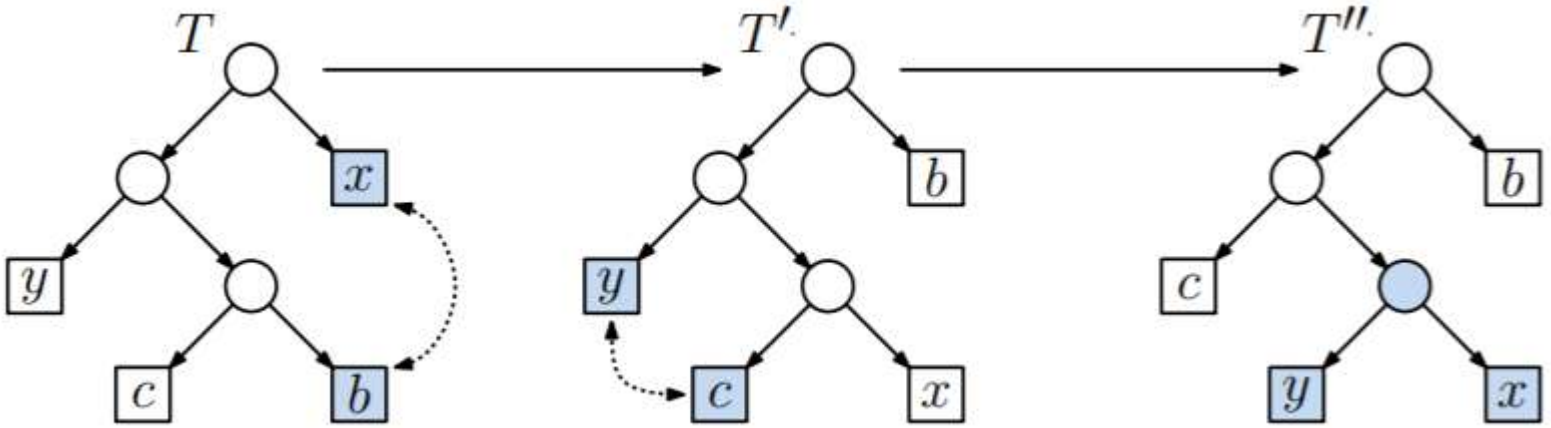
  p(x)≤p(b), p(y)≤p(c),

  因为b,c具有最大深度，

  因此：$d_T(b) \geq d_T(x)$, $d_T(c) \geq d_T(y)$，



- 交换x和b，得到T‘，继续交换y和c，得到T”



T”使命题成立

$$
\begin{aligned}
B(T') &= B(T) - (\text{old cost for } b \text{ and } x) + (\text{new cost for } b \text{ and } x) \\
&= B(T) - (p(x)d_T(x) + p(b)d_T(b)) + (p(x)d_T(b) + p(b)d_T(x)).
\end{aligned}
$$

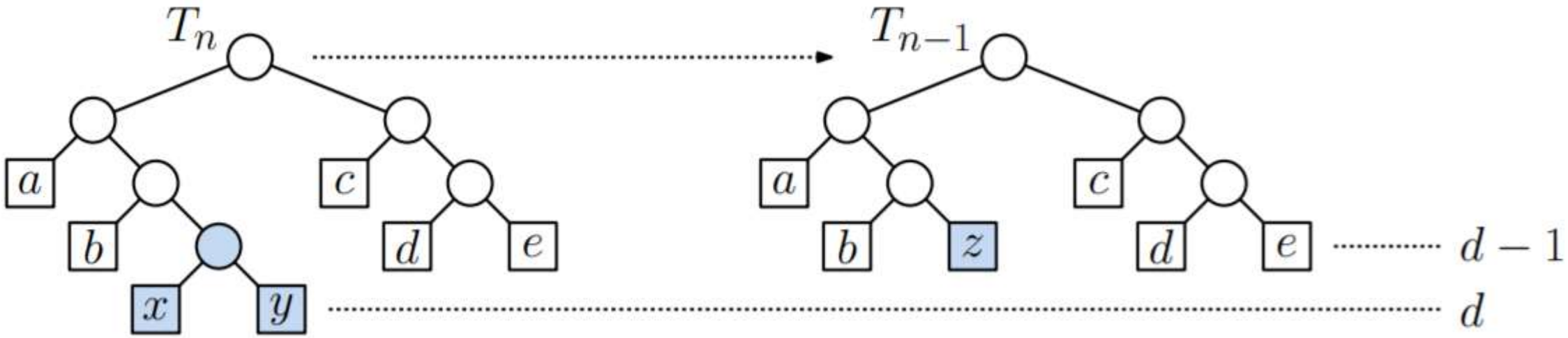
$$
\begin{aligned}
B(T') &= B(T) + p(x)(d_T(b) - d_T(x)) - p(b)(d_T(b) - d_T(x)) \\
&= B(T) - (p(b) - p(x))(d_T(b) - d_T(x)) \\
&\leq B(T),
\end{aligned}
$$

因为T是最优的，
所以T’也是最优的

同理： B(T’’)≤B(T’)

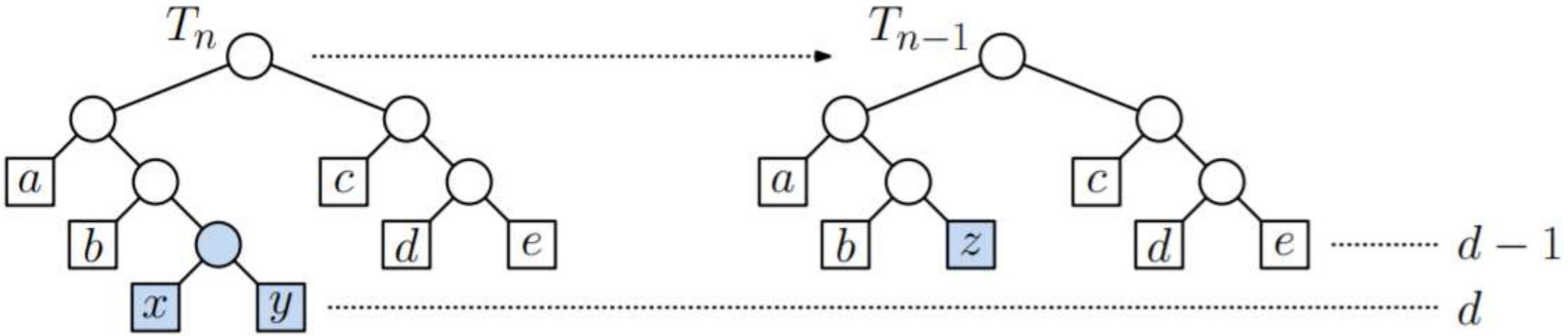
所以T’’也是最优的

- **引理2**：设T_n是满足引理1的前缀编码树（即最低概率x,y是具有最大深度的兄弟），设T_{n-1}是用一个单个叶子节点z代替x,y的父节点的树（p(z) = p(x)+p(y)）,那么$B(T_n) = B(T_{n-1}) + p(z)$



证：设d是x,y的深度，则z的深度是d-1，那么：

$$\begin{aligned} B(T_n) &= B(T_{n-1}) - (z\text{'s cost in } B(T_{n-1})) + (x \text{ and } y\text{'s costs in } B(T_n)) \\ &= B(T_{n-1}) - p(z)(d-1) + (p(x)\,d + p(y)\,d) \end{aligned}$$



$$
\begin{aligned}
B(T_n) &= B(T_{n-1}) - (z\text{'s cost in } B(T_{n-1})) + (x \text{ and } y\text{'s costs in } B(T_n)) \\
&= B(T_{n-1}) - p(z)(d-1) + (p(x)\,d + p(y)\,d) \\
&= B(T_{n-1}) - p(z)(d-1) + p(z)\,d \\
&= B(T_{n-1}) + p(z).
\end{aligned}
$$

这说明最小化树T_n等价于最小化树T_{n-1}.

- **引理3**：哈夫曼算法产生的编码是最优前缀

证明（数学归纳法）：

n=1时是成立的，因为只有唯一的一个顶点的树。

当n>=2时，根据引理1，最低概率的2个字符{x,y}是最大深度的兄弟，哈夫曼算法将它们用z代替，从而转化为n-1个字符的问题，而根据归纳假设，哈夫曼代码可以得到这n-1个字符的最优前缀编码树T_{n-1}，将T_{n-1}的z用{x, y}代替产生了一个树T_n，$B(T_n) = B(T_{n-1}) + p(z)$，因为$T_{n-1}$是最优的编码树，z替换为{x,y}不依赖于树的结构，因此，T_n是n个字符的最优的编码树。

图的基本概念

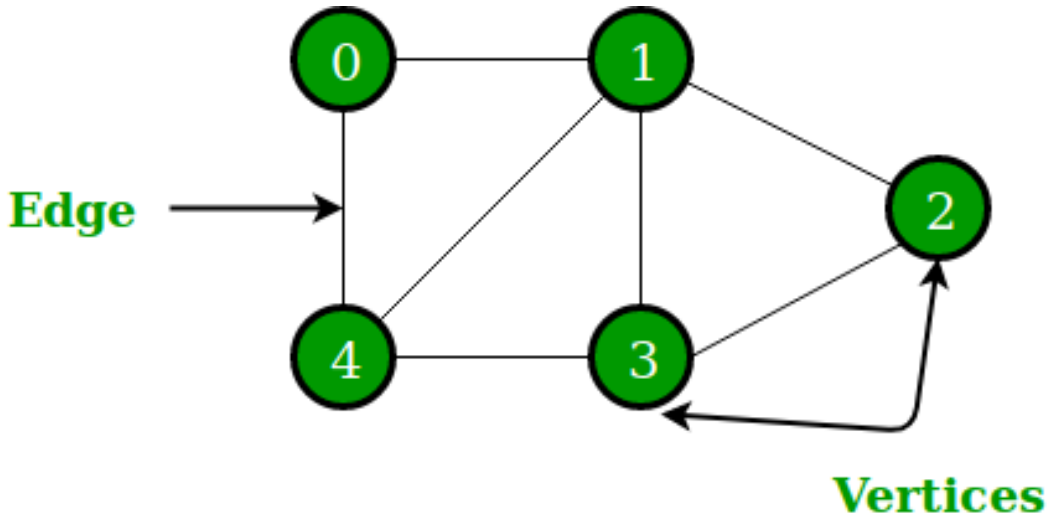
Graph

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

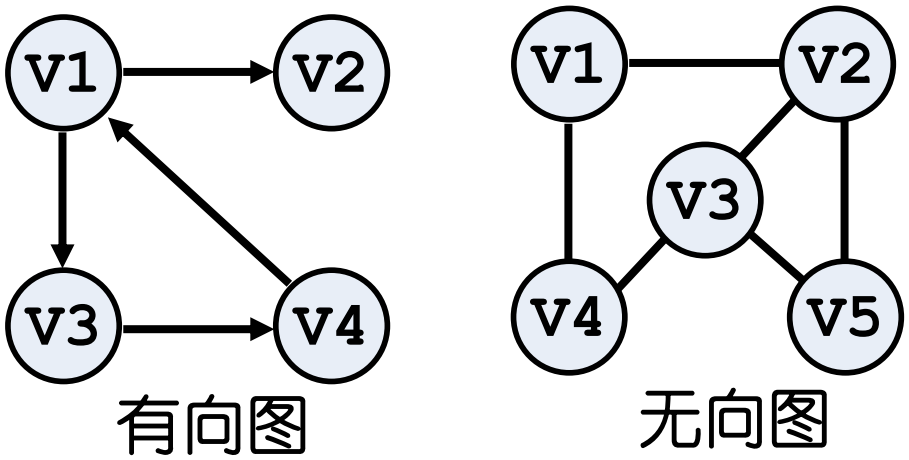
图

- 图G(V,E)是一个顶点集合V和边集合E。



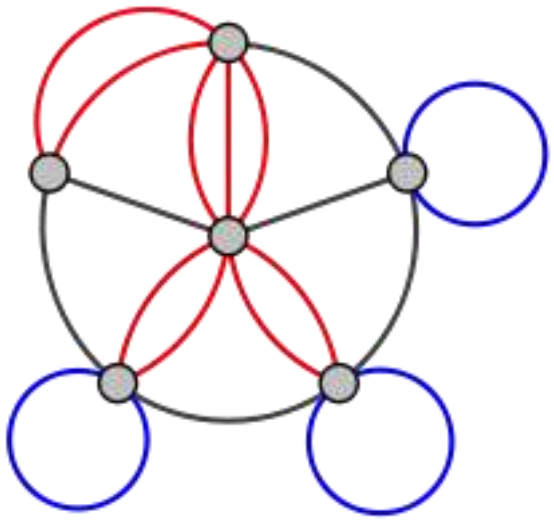
图

- 图G(V,E)是一个顶点集合V和边集合E。
- 无向图(undirected graph)中，边是一个无序顶点对:e=(u,v)
- 有向图(directed graph)中，边是一个有序顶点对:e=<u,v>



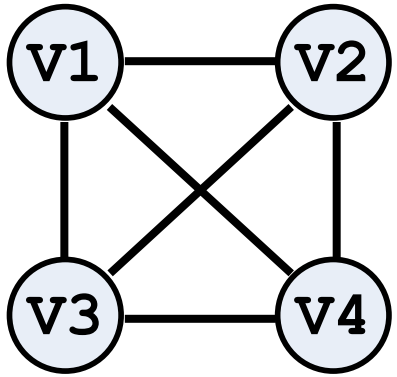
图

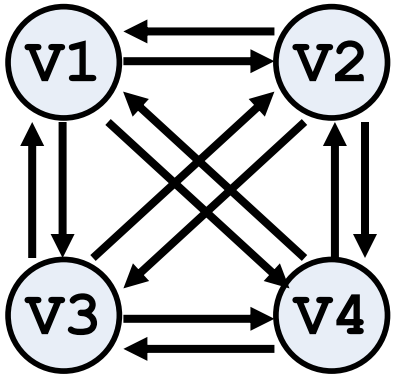
- 图G(V,E)是一个顶点集合V和边集合E。
- 无向图(undirected graph)中，边是一个无序顶点对:e=(u,v)
- 有向图(directed graph)中，边是一个有序顶点对:e=<u,v>
- 多边图(a multigraph): 2个顶点之间可以有多个边。
- self-loop：一个边的2个顶点是同一个顶点。
- 简单图：没有self-loop边的图。



完全图

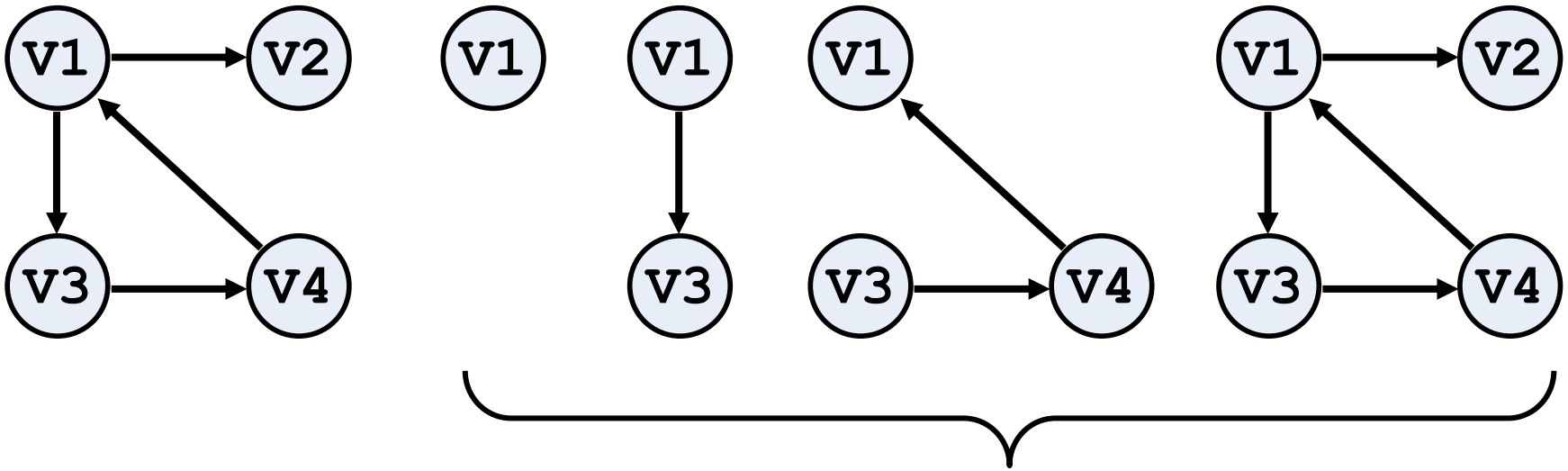
- 若无向图任意2个顶点之间都有一条边，则此图为完全无向图。
- 若有向图任意2个顶点u,v都有2条反向的弧<u,v>和<v,u>，则此图为完全有向图。
- 完全图其实就是边/弧的数量达到最大值





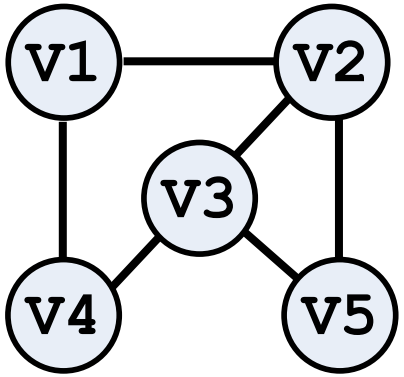
子图

- 有两个图G={V, E},G'={V',E'}
- 如果V'⊆V, E'⊆E，则称G'为G的子图



邻接点、度

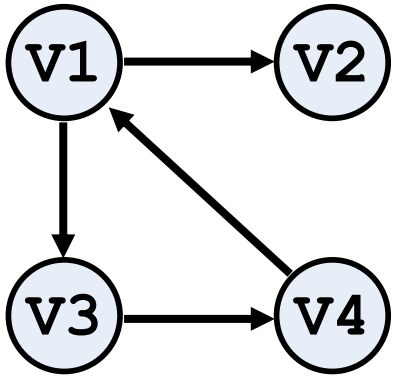
- u, v之间如果有一条边(u,v)或弧<u,v>,就称它们互为邻接点。
- 一个顶点的邻接点或边的个数，称为该顶点的度。



- TD(V1)=2
- TD(V2)=3
- TD(V3)=3
- TD(V4)=2
- TD(V5)=2

入度和出度

- 对有向图： 顶点的出度是以它为弧尾的弧的个数，而入度是以它为弧头的弧的个数 。度= 入度+出度



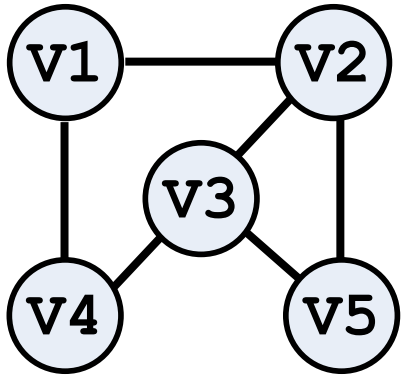
- ID(V1)=1
- OD(V1)=2
- TD(V1)=ID(v1)+OD(v1)=3

度和边数的关系

- 一个有n个顶点，e条边或弧的图满足：

$$e = \frac{1}{2}\sum_{i=1}^{n} TD(v_i)$$

- 即边（或弧）的总数 = 各个顶点的度的总数的一半



- TD(V1)=2
- TD(V2)=3
- TD(V3)=3
- TD(V4)=2
- TD(V5)=2

- e=6

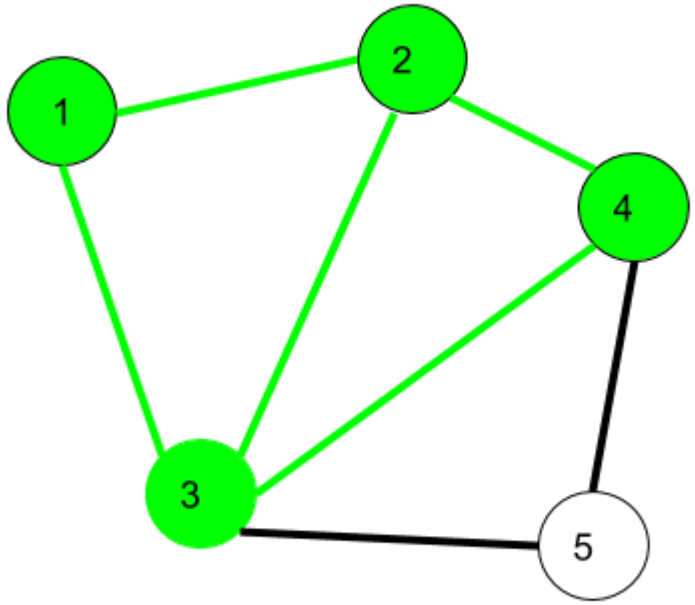
路径、简单路径、**trail**、回路，简单回路

- **Walk（路径）**：从一个顶点经过一系列边到达另外一个顶点，所经过的顶点和边的序列。
- **Trail** ：没有重复边的路径
- **Path（简单路径）**：没有重复顶点和边的路径。
- 回路(**Circuit**): 没有重复边（可能有重复顶点）、起点终点相同的路径
- **Cycle(简单回路)**：除起点终点相同、没有重复顶点和重复边的回路

Walk (路径)：从一个顶点经过一系列边到达另外一个顶点，所经过的顶点和边的序列。

Vertices may repeat. Edges may repeat (Closed or Open)

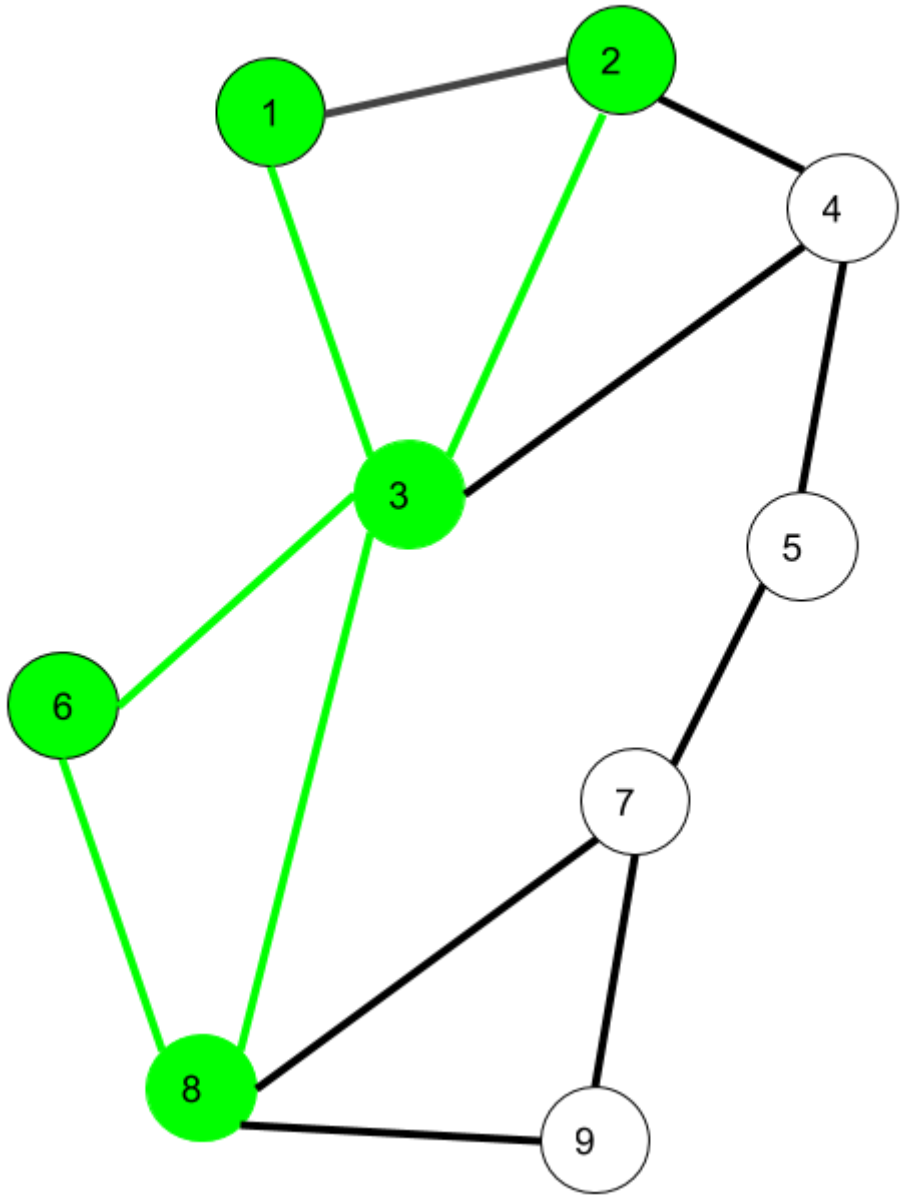
如：1-2-3-4-2-1



Trail： Vertices may repeat. Edges cannot repeat (Open)

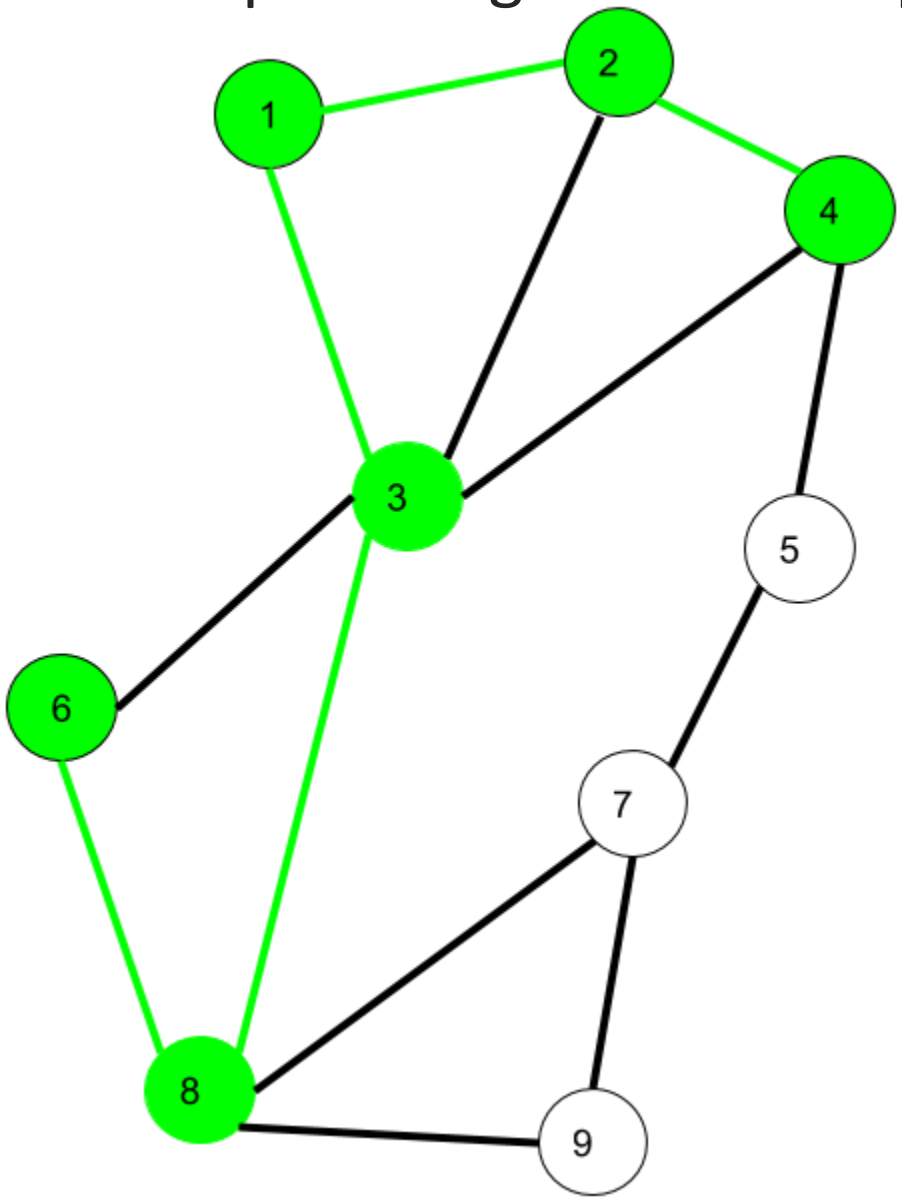
如： 1-3-8-6-3-2

1-3-8-6-3-2-1



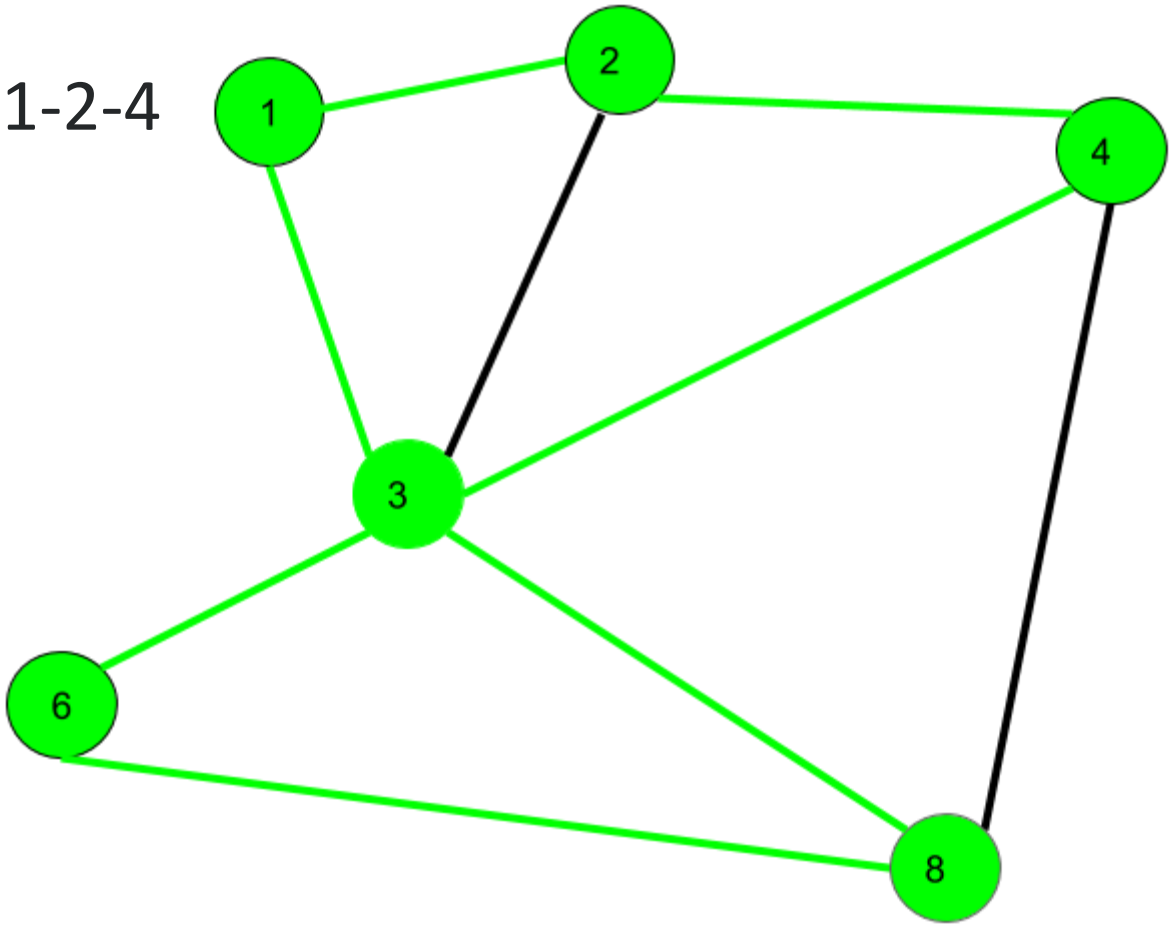
path : Vertices cannot repeat. Edges cannot repeat (Open)

如： 6-8-3-1-2-4



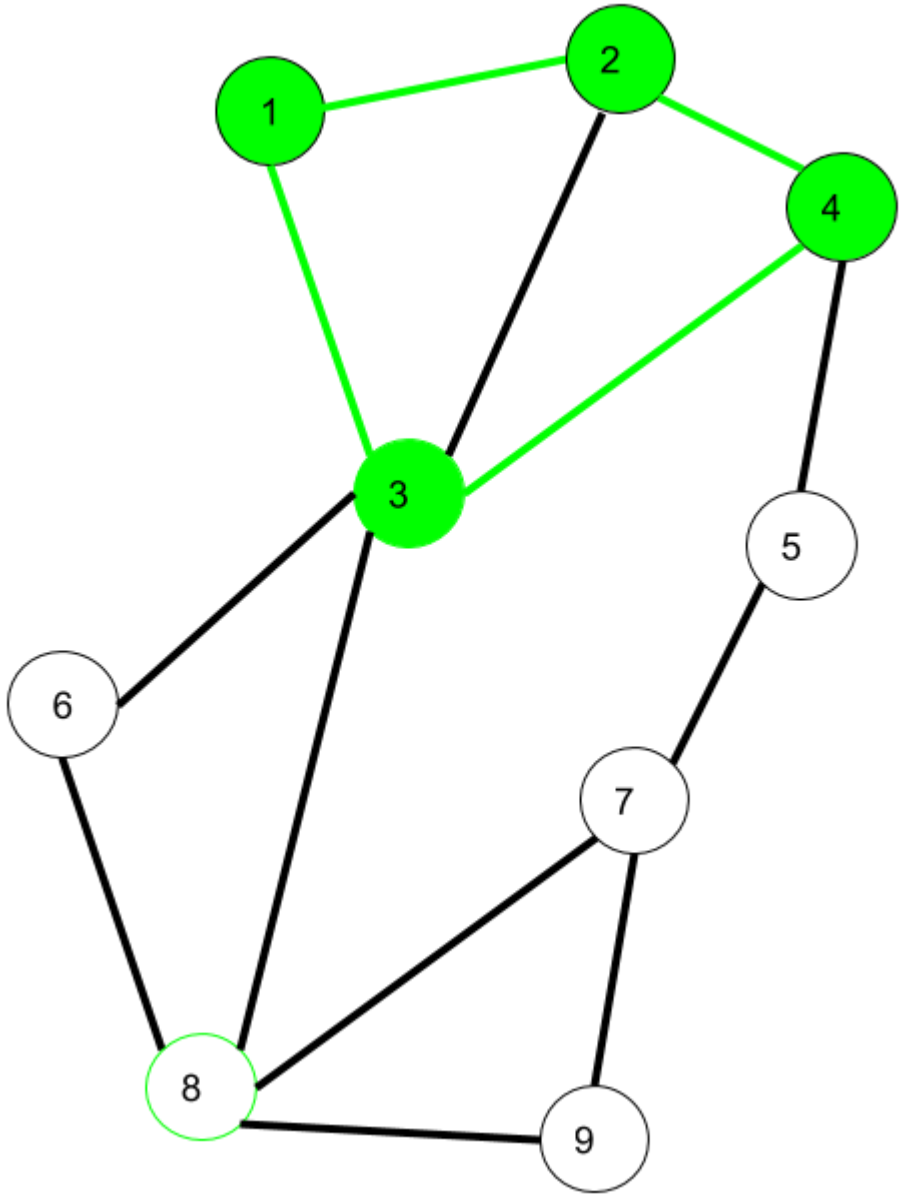
circuit : Vertices may repeat. Edges cannot repeat (Closed)

如： 6-8-3-1-2-4



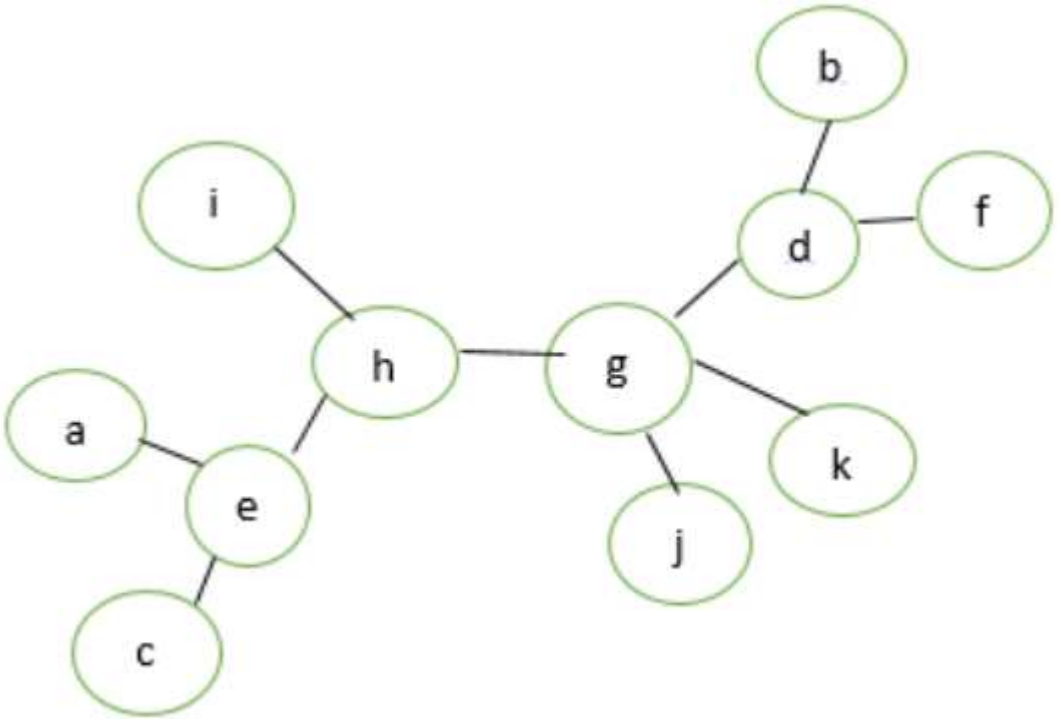
cycle：Vertices cannot repeat. Edges cannot repeat (Closed)

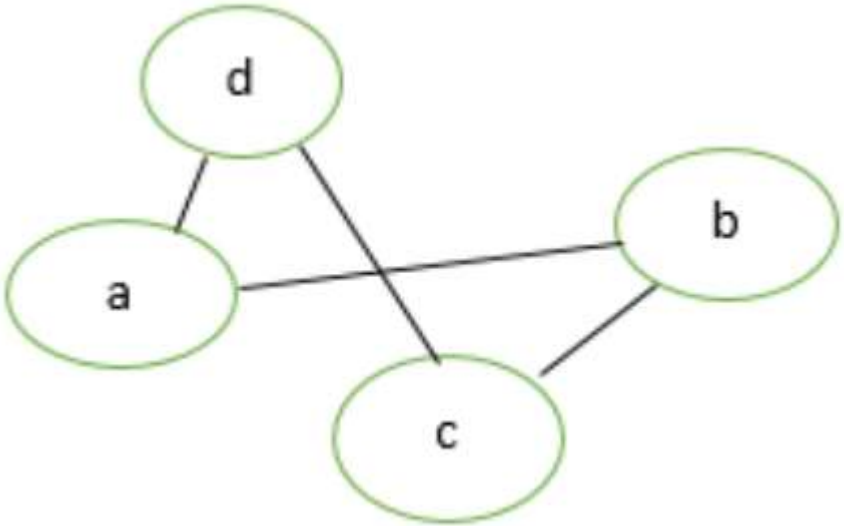
如： 1-2-4-3-1



树tree

- 没有（简单）回路（cycle）的连通图称为**树**。





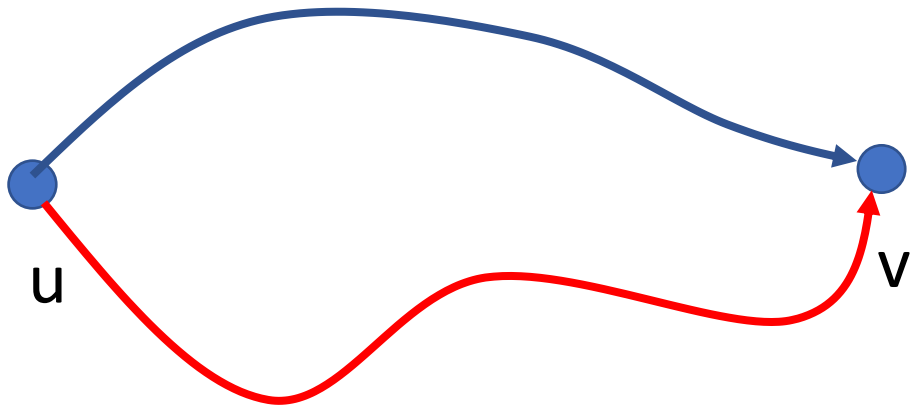
Not a tree

树tree的性质

- 任意2个顶点有且只有一条路径。
- 树中添加1个边就会形成一个（简单）回路
- 树是连通无回路的
- 删除树的一条边，就会不连通

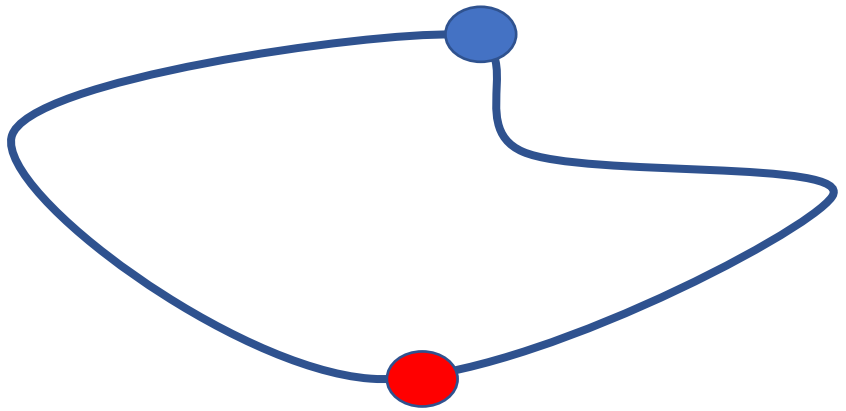
树tree的性质

- 定理1：任意2个顶点有且只有一条路径。
- 证明（反证）：如果由2个顶点存在2条不一样的路径，则这2顶点之间就构成了一个回路。

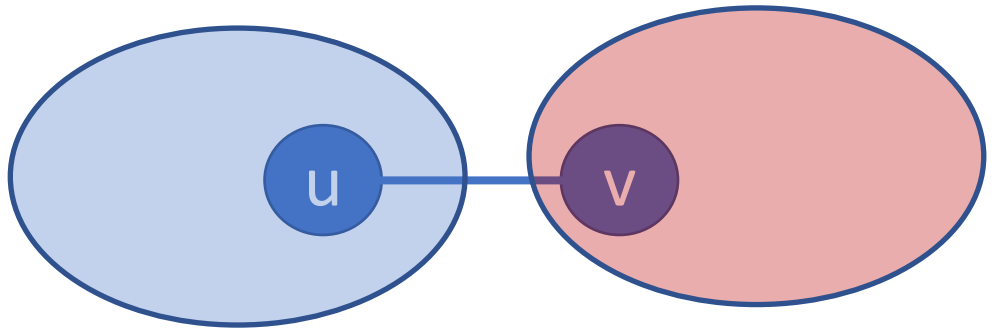


- 定理2：如果图的任意2个顶点之间都有且只有一条路径，则这个图必然是一个树
- 证明：任意2个顶点存在路径，说明图是连通的。

任意2个顶点只有唯一的路径，说明没有回路，因为如果有回路，说明由2个顶点之间至少有2个不一样的路径



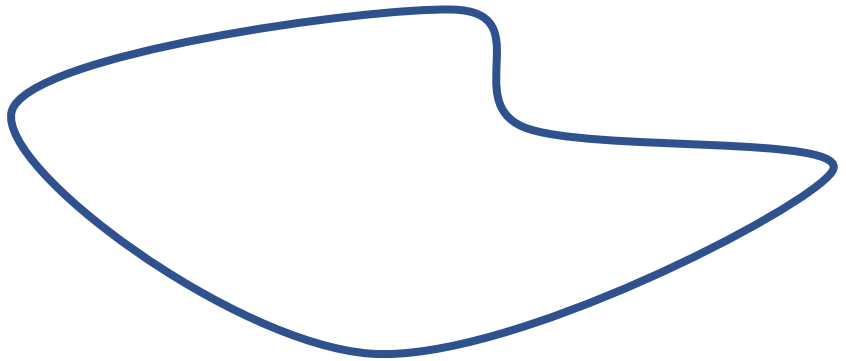
- 定理3：n个顶点的树的边的数目一定是n-1。
- 证明（数学归纳法）：
- 当n=1时，只有一个顶点的树，边数目为0
- 当n=2时，只有2个顶点的树，边数目为1
- 假设当n=m时命题成立，要证n=m+1也成立
- 设(u,v)是一条边，删除这个边，图G被分割为2个不连通的子图G1、G2，因为u,v之间只有唯一路径。



$num = (m_1-1)+(m_2-1)+1 = n-1$

- 定理4：n个顶点连通图如果有n-1条边，则是一个树。
- 证明（反证）：假如G不是一个树，即存在回路，则可以将回路的边不断删除，直到得到一个没有回路的连通图G'，G'是一个树，而边的数目小于n-1，与定理3矛盾。

- 也可以用数学归纳法证明



- 定理5：n(n≥2)个顶点的树至少有2个叶子节点。

注：叶子节点是只有一个邻接边（度为1）的顶点。

- 证明：至少有2个顶点，每个顶点都和其他顶点存在连通路径，因此，每个顶点的度不可能是0。

  边数目e = n-1，和顶点度的关系为：$2e = \sum_{i=1}^{n} degree(vi)$

  即$2n-2 = \sum_{i=1}^{n} degree(vi)$

  $= n_1+2n_2+3n_3+\ldots \geq n_1+2(n-n_1) = 2n-n_1$

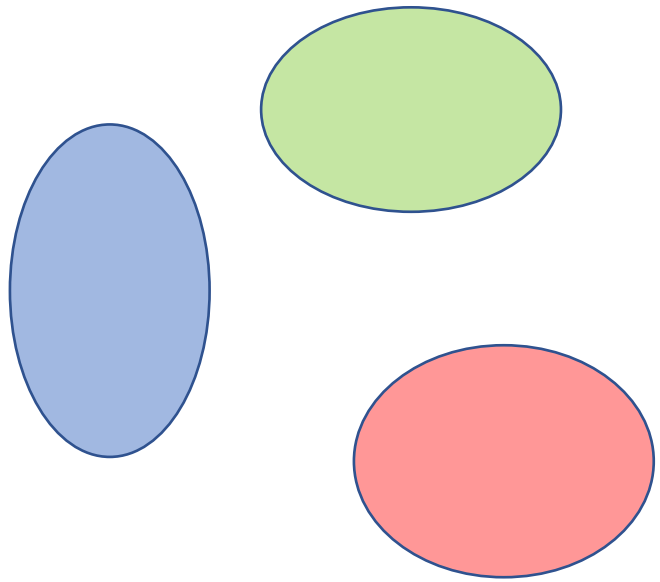
- 即$n_1 \geq 2$

- 定理6： n个顶点的图G(V,E)如果有n-1条边且没有回路（cycle），则它是一个树。
- 证明（反证）：假设G不连通，则V可以分成k>1个互不连通的子集$V_1,V_2,\ldots V_k$，V_i内的顶点可以通过边集合E_i连通,但V_i、V_j间任何2个顶点没有边相连。

得到k个子图$G_i=G(V_i, E_i)$,每个子图是没有回路的连通图，即是一个树。

从而

$$|E| = \sum_{i=1}^{k} |E_i| = |V| - k < |V| - 1 = |E|$$



- 定理7：如果图G(V,E)具有下面3个性质的2个，必然具有所有3个性质：

(1) 图是连通的

(2) 图没有回路

(3) |E| = |V|-1

因此，具有其中2个性质的图，必然是一个树。

当然，一个树也必然具有其中3个性质。

最小生成树

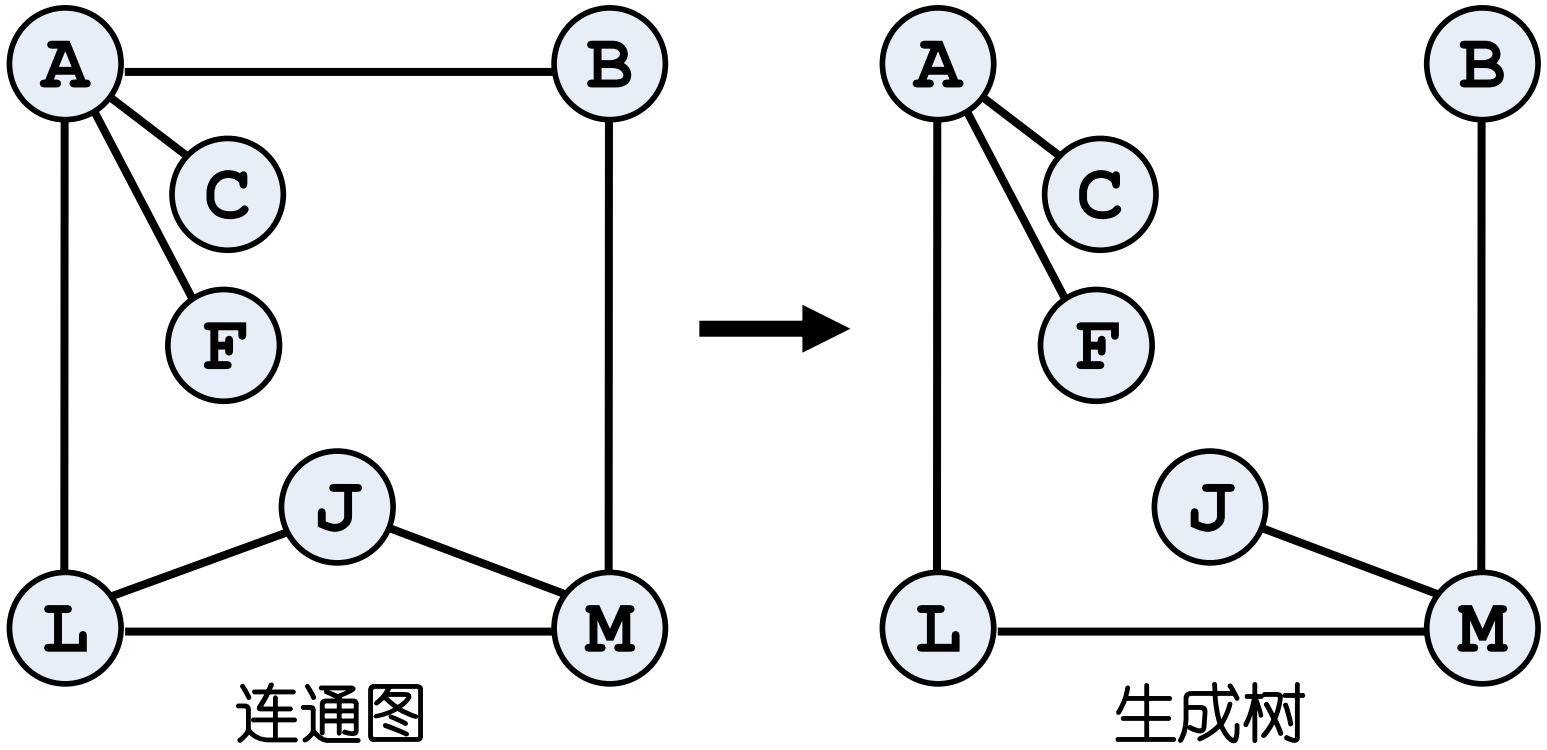
Minimum Spanning Tree

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

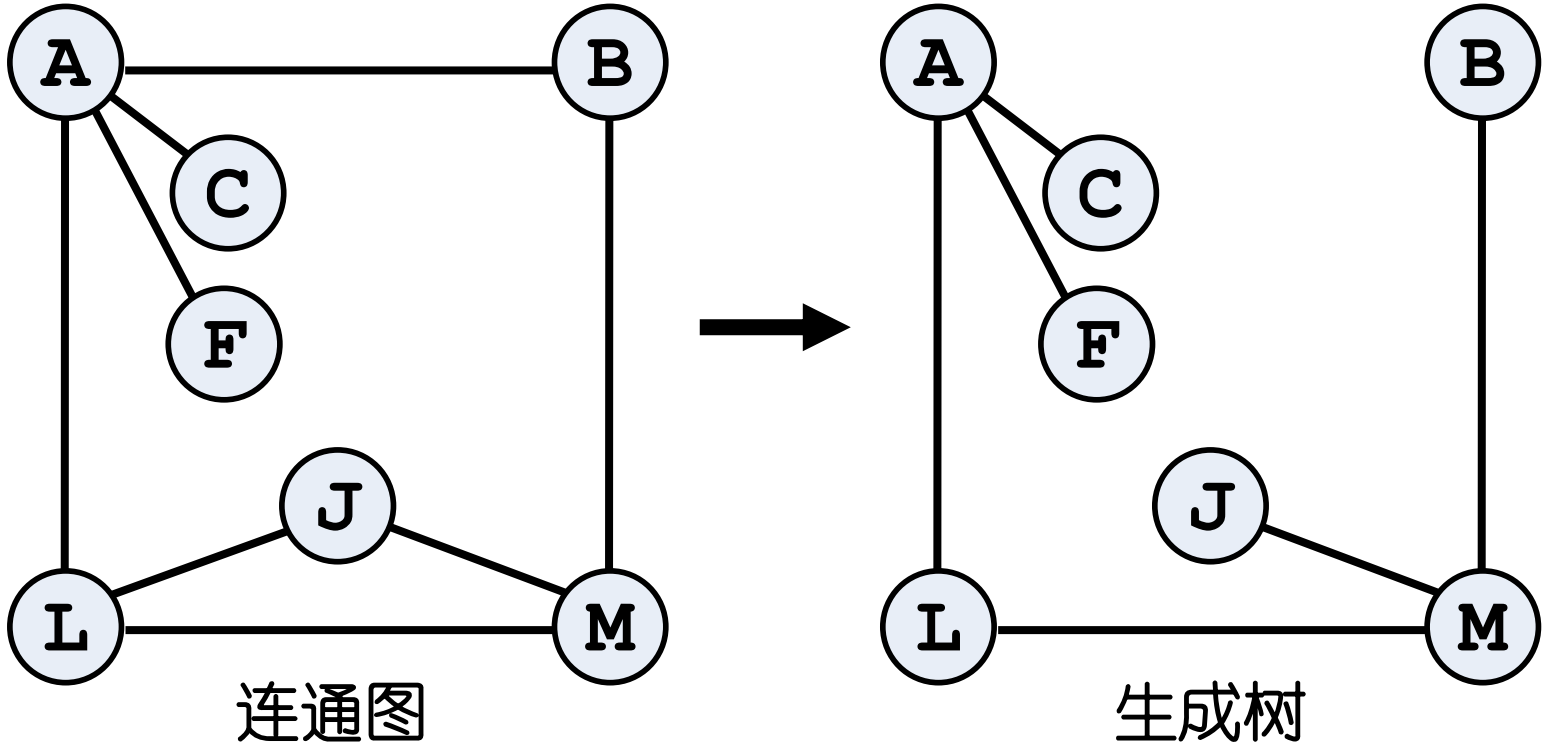
生成树(Spanning Tree)

- 一个**连通图**的包含所有顶点的**极小连通子图. (树)**



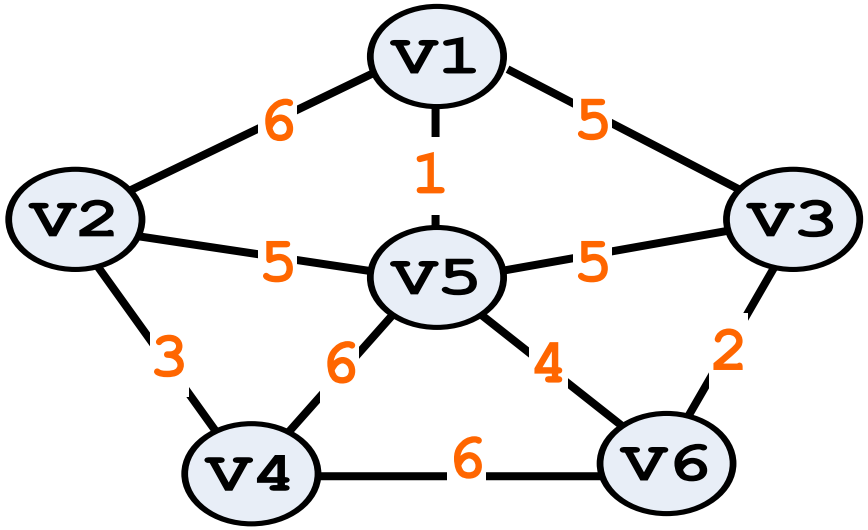
生成树(Spanning Tree)

- 包含原连通图的所有n个顶点
- 包含原有的其中n-1条边
- 且保证连通



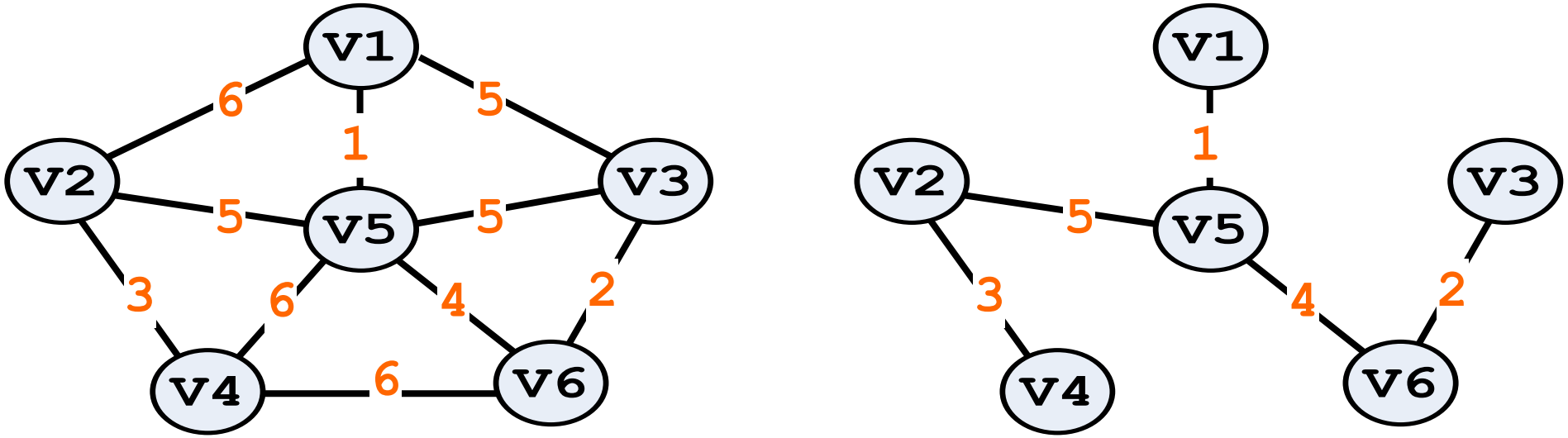
带权连通图

- 图的边有一个权值。



最小生成树(minimum Spanning Tree)

- 带权的连通图的**各边权值之和最小**的生成树

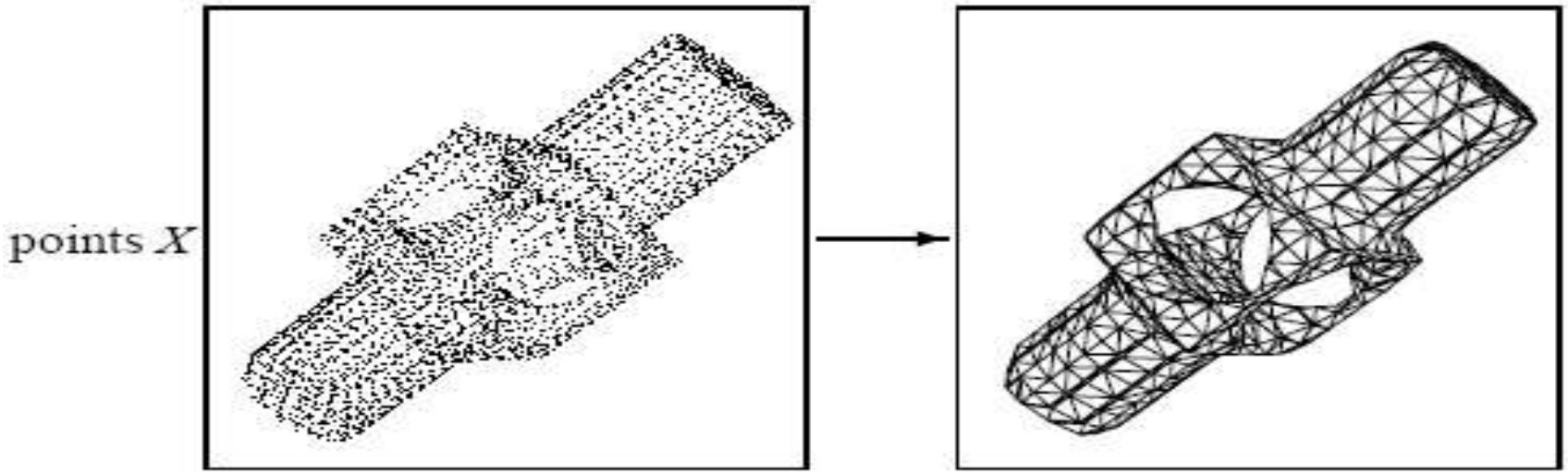


最小生成树的应用

- 网络设计：计算机网络、道路网络、电路布线等
- NP难问题的近似解：如旅行商问题
- 聚类分析
- 图形学

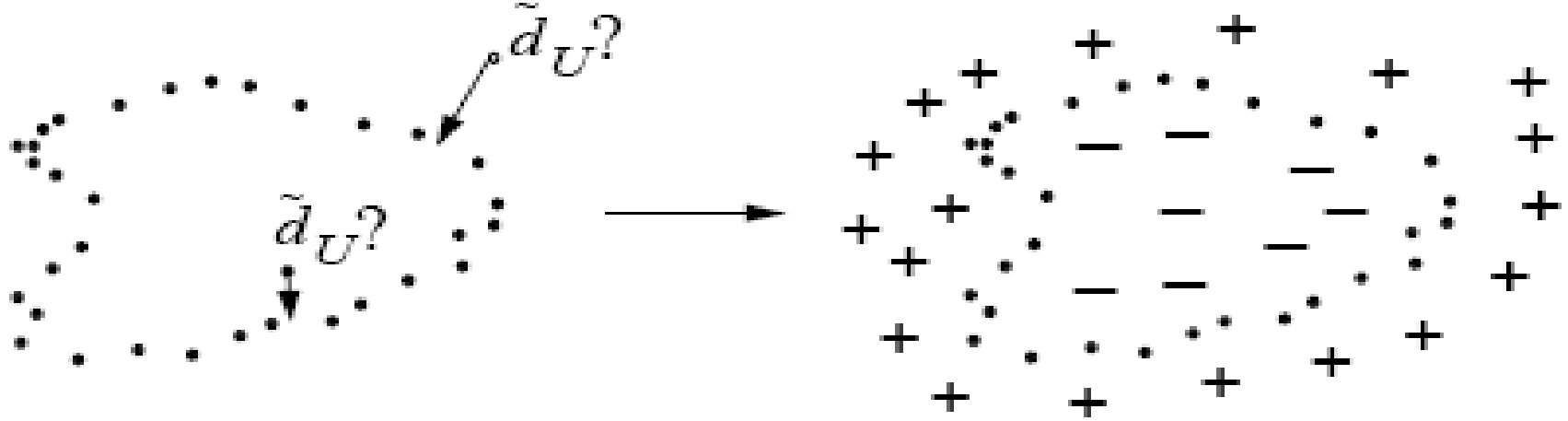
曲面重建

- Hooppe曲面重建算法(Surface reconstruction from unorganized points.H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle. *ACM SIGGRAPH 1992*, 71-78.)

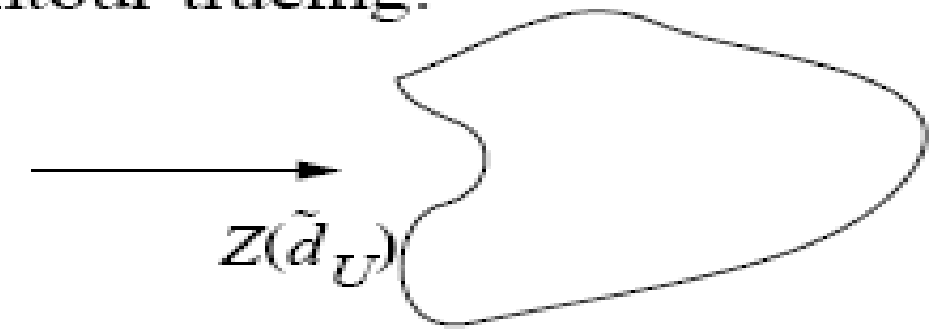


曲面重建

1. Estimate d_V from data points:



2. Use contour tracing:



曲面重建

步骤:

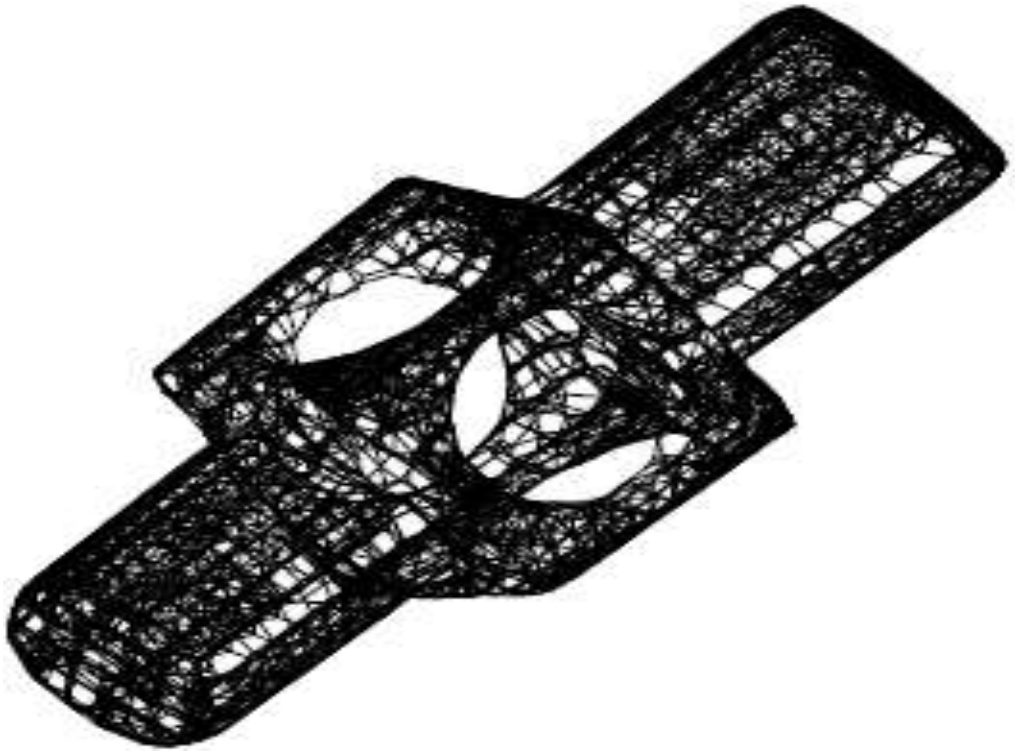
1) 构造Riemannian图
2) 计算各点附近的切平面
3) 切平面法矢量的一致化(基于最小生成树的遍历定向)
4) Marching Cube等值面抽取网格生成

曲面重建

步骤:

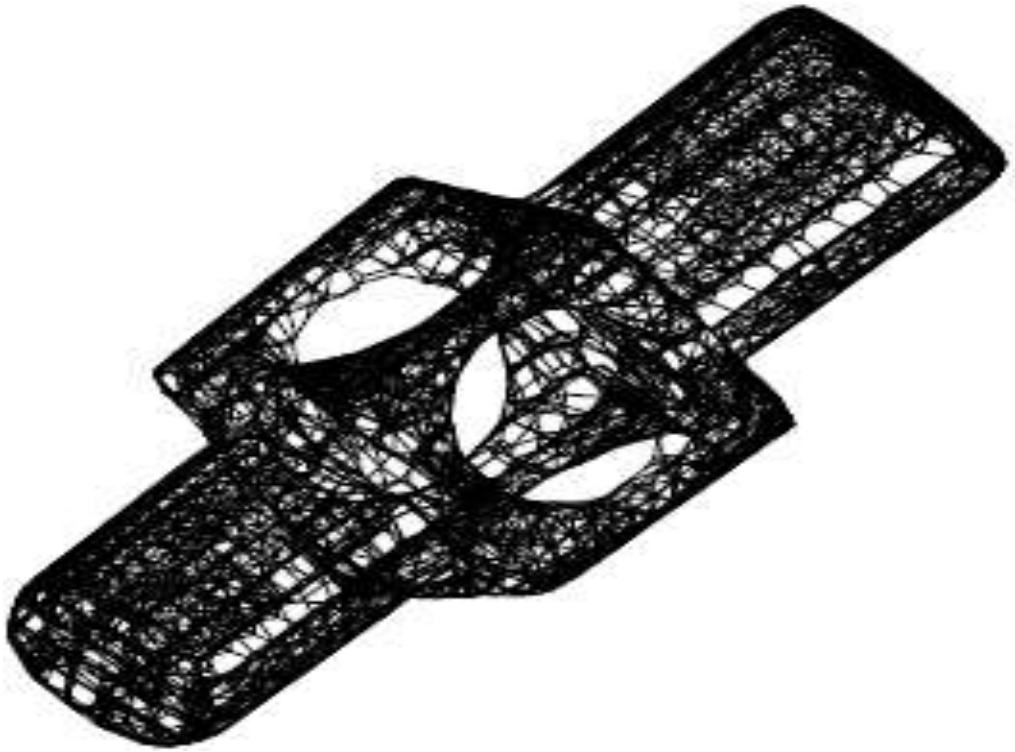
1) 构造Riemannian图
2) 计算各点附近的切平面
3) 切平面法矢量的一致化(基于最小生成树的遍历定向)
4) Marching Cube等值面抽取网格生成

曲面重建

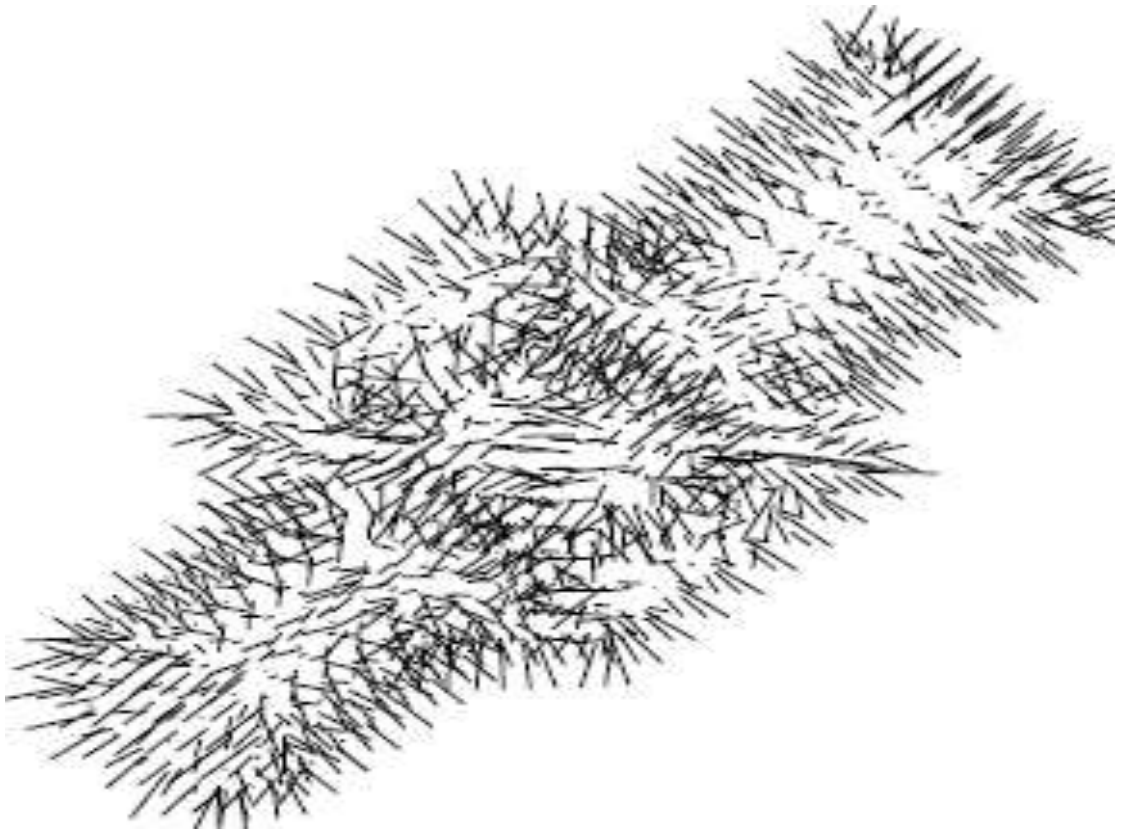


Riemannian图

曲面重建

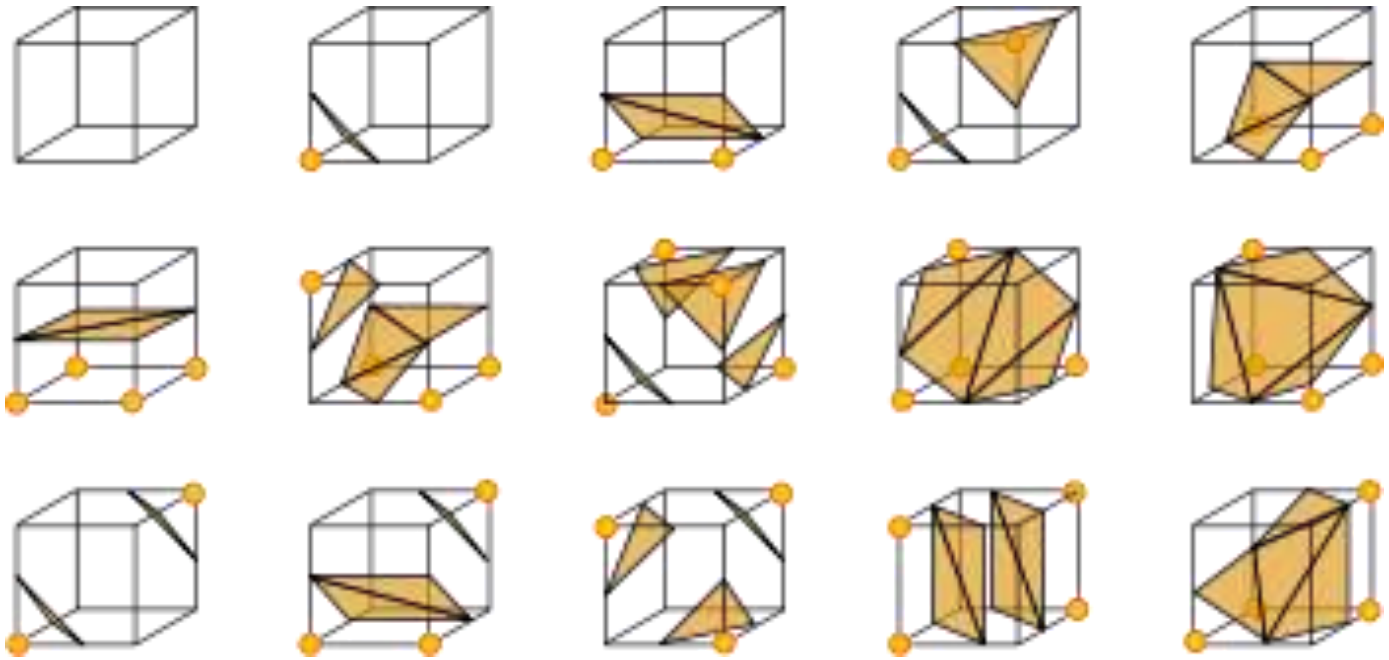


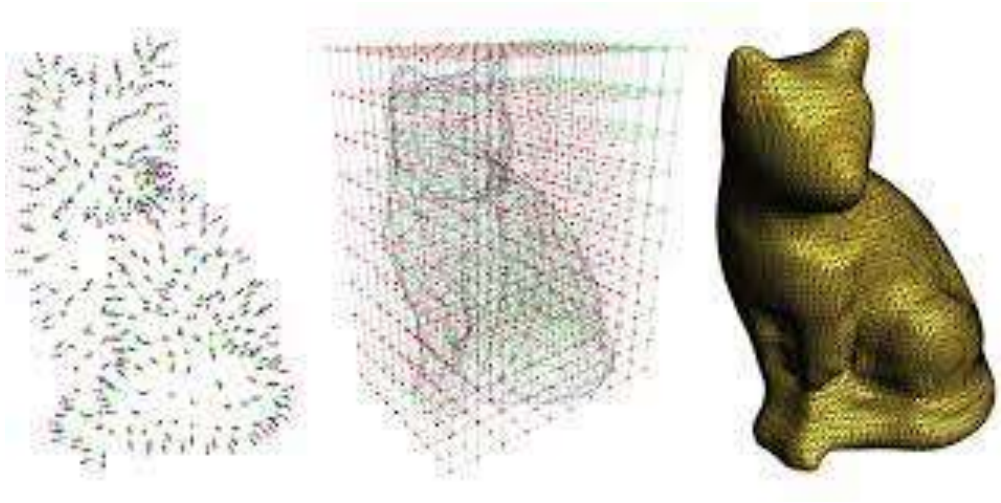
Riemannian图



一致的法矢量

Marching cube





最小生成树的2个贪婪法算法

- Prim算法：

  每次增加一个顶点到构建中的最小生成树中

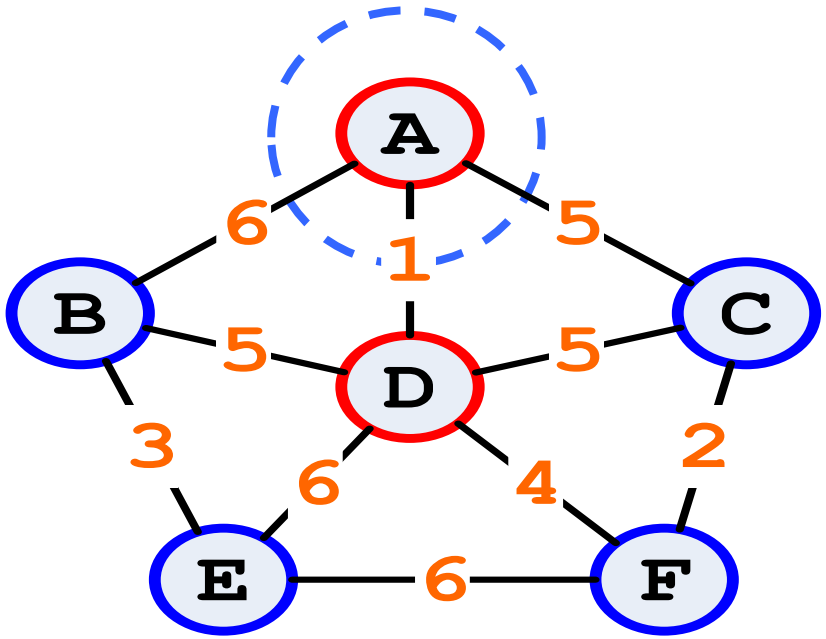
  每次选择构成**最短距离**的顶点加入到U中

- Kruskal算法：

  每次增加一条最小权值的边到构建中的最小生成树中，只要不形成回路

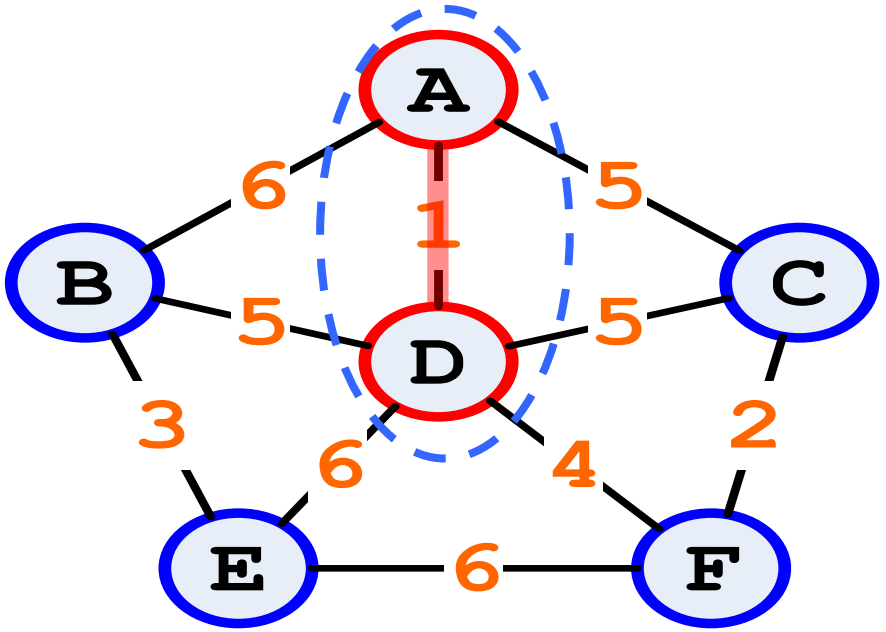
Prim算法

- U为最小生成树中顶点的集合，初始时，U={u0}
- 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U



Prim算法

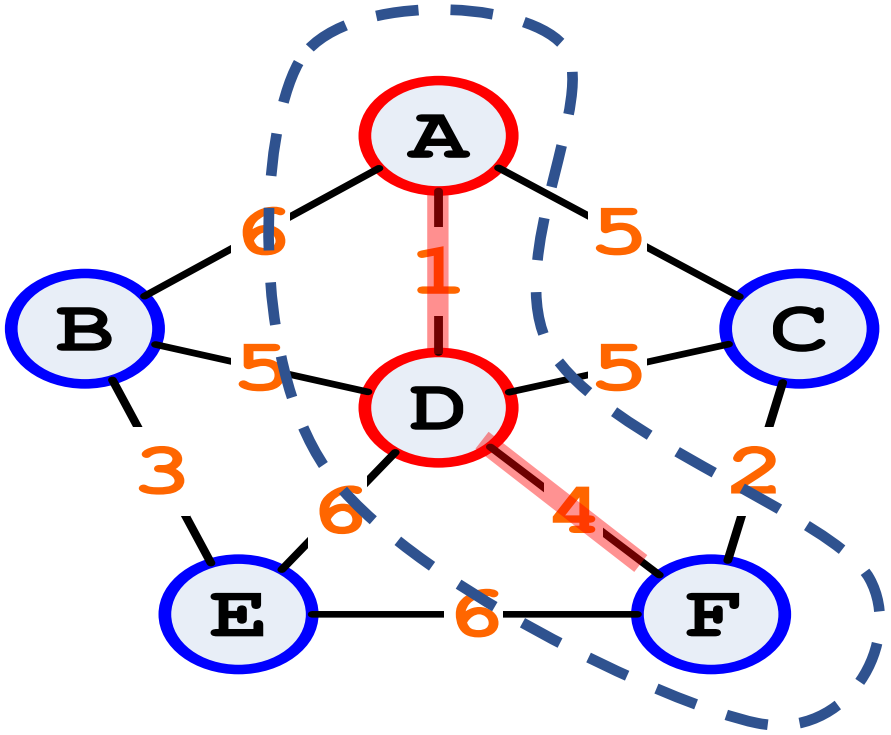
- U为最小生成树中顶点的集合，初始时，U={u0}
- 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U



- 重复，直到所有的顶点都加入到U中

Prim算法

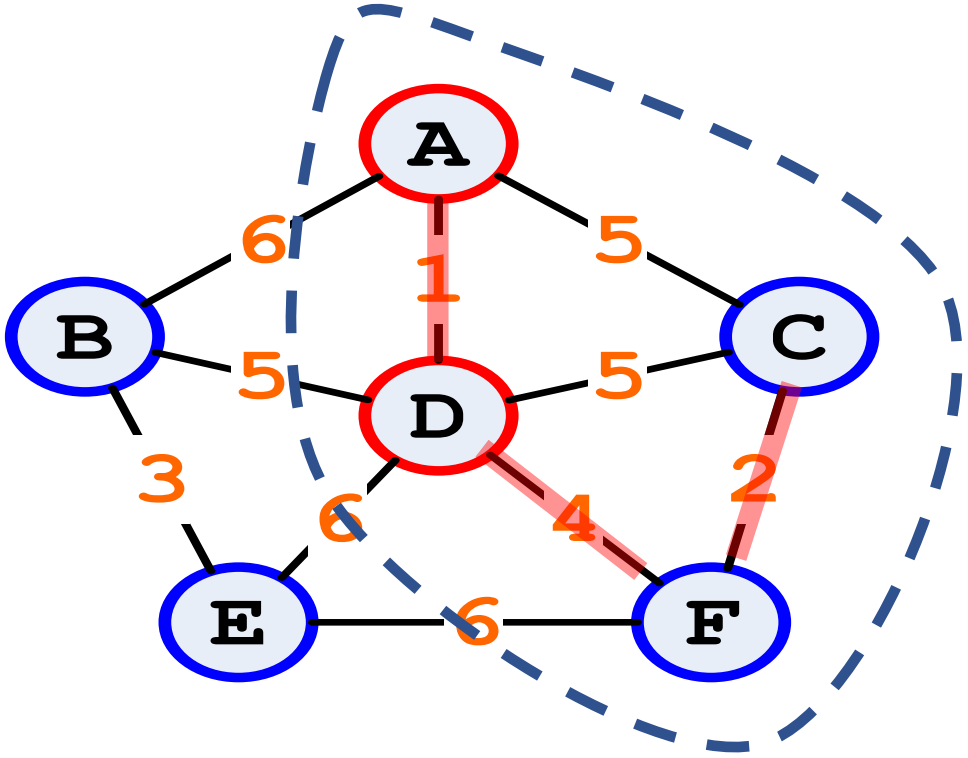
- U为最小生成树中顶点的集合，初始时，U={u0}
- 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U



- 重复，直到所有的顶点都加入到U中

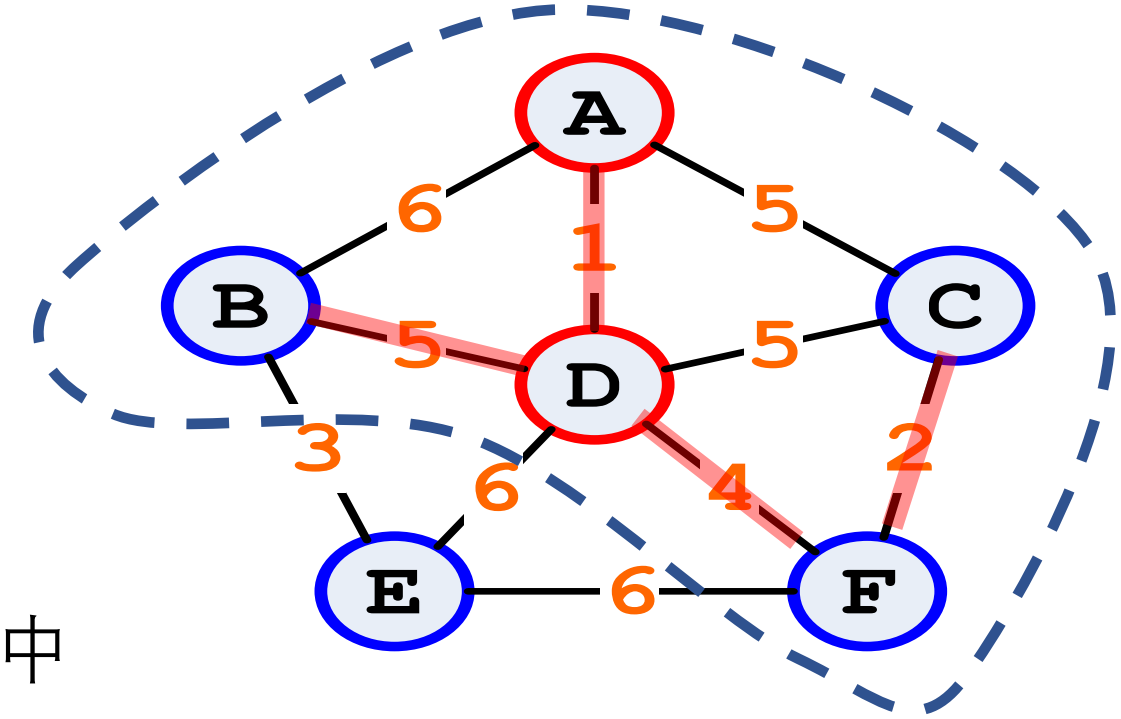
Prim算法

- U为最小生成树中顶点的集合，初始时，U={u0}
- 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
- 重复，直到所有的顶点都加入到U中



Prim算法

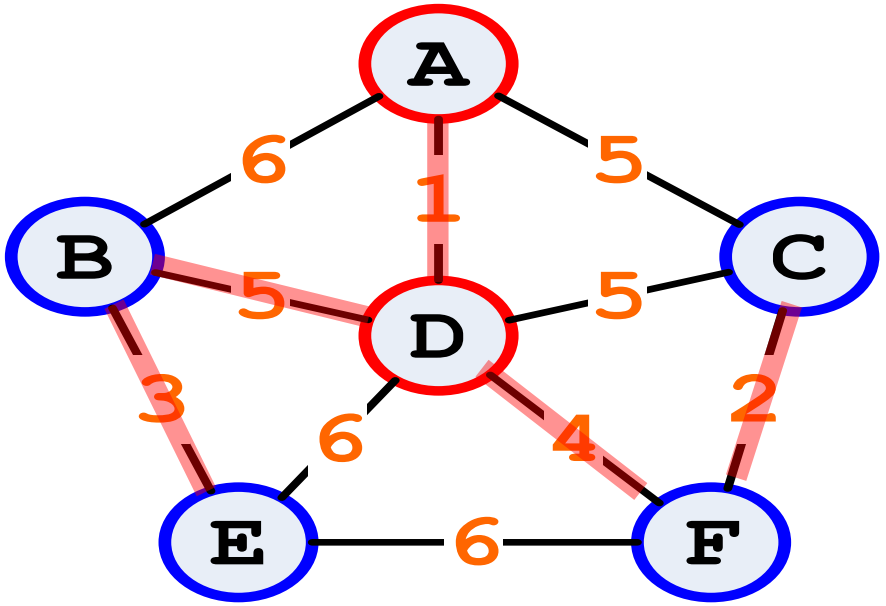
- U为最小生成树中顶点的集合，初始时，U={u0}
- 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U



- 重复，直到所有的顶点都加入到U中

Prim算法

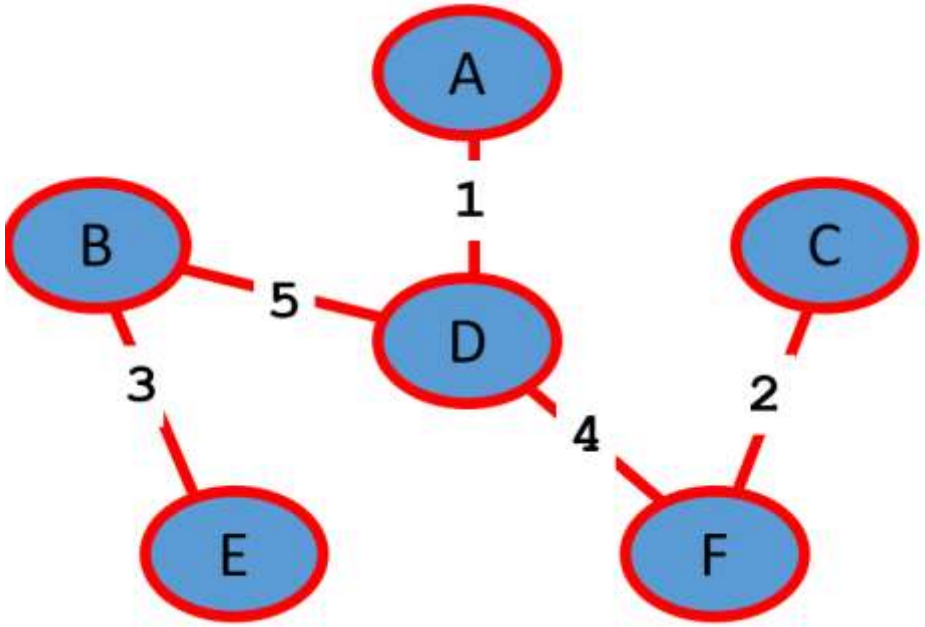
- U为最小生成树中顶点的集合，初始时，U={u0}
- 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U



- 重复，直到所有的顶点都加入到U中

Prim算法

- U为最小生成树中顶点的集合，初始时，U={u0}
- 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U



- 重复，直到所有的顶点都加入到U中

Prim算法伪代码

U ← u0

for i=1 to n-1:

    在V-U中选择一个到U距离最短的边(u_i,v_j)

    将v_j加入到U中

时间复杂度

U ← u0

for i=1 to n-1: O(n)

在V-U中选择一个到U距离最短的边(u_i,v_j) O(n)

将v_j加入到U中 O(1)

$T(n) = O(n^2)$

正确性证明（数学归纳法）

证明：prim算法构造过程的T_k是总是最小生成树的一部分。

证明过程：

1） 证明k=1时，即第一条边必是某个最小生成树的一条边。

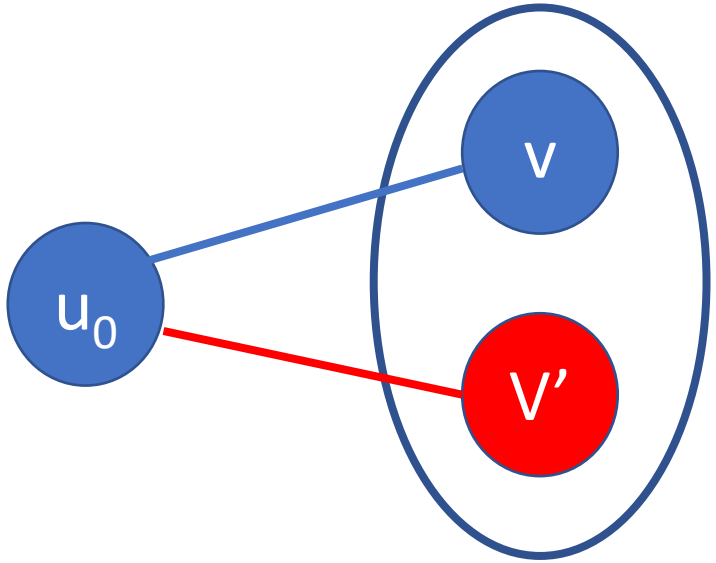
2) 假设当k时成立（T_k是某个最小生成树的一部分）

要证明，prim算法构造的T_{k+1}也必是最小生成树的一部分

最小生成树性质

1） 证明k=1时，即第一条边必是某个最小生成树的一条边。

反证： 假如(u_0,v)是和u_0相连边中的最小权值的边，但没有一个最小生成树T包含它。



T是任意一个最小生成树，但u_0,和v’相连(u_0,v').
必然有$w(u_0,v) \leq w(u_0,v')$。
用(u_0,v)代替(u_0,v')，也构成一个生成树T*，

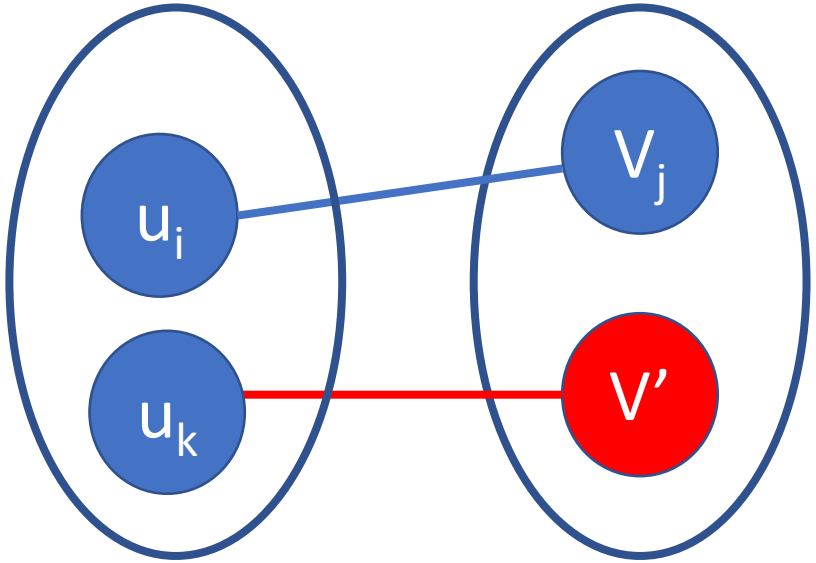
$$w(T^*) \leq w(T)$$

即T*必然是最小生成树，从而产生矛盾!

2）假设当k时成立，即 T_k是某个最小生成树T的一部分
　　要证明，prim算法构造的T_{k+1}也是某个最小生成树的一部分

设(u_i,v_j)是和U相连边中的最小权值的边， T_{k+1}的边将由T_k的边和(u_i,v_j)构成。

A）如果(u_i,v_j)属于T，则T_{k+1}是T的一部分。
B）否则，将(u_i,v_j)加到T中，必然形成回路，这条回路必有一条边(u_k,v')连通U和V-U。
显然有$w(u_i,v_j) \leq w(u_k,v')$。
用(u_i,v_j)代替(u_k,v')，也构成一个生成树T*，
　　$w(T^*) \leq w(T)$
即T*必然是最小生成树，且包含T_{k+1}的所有边



Kruskal算法

Minimum Spanning Tree

Youtube频道： **hwdong**

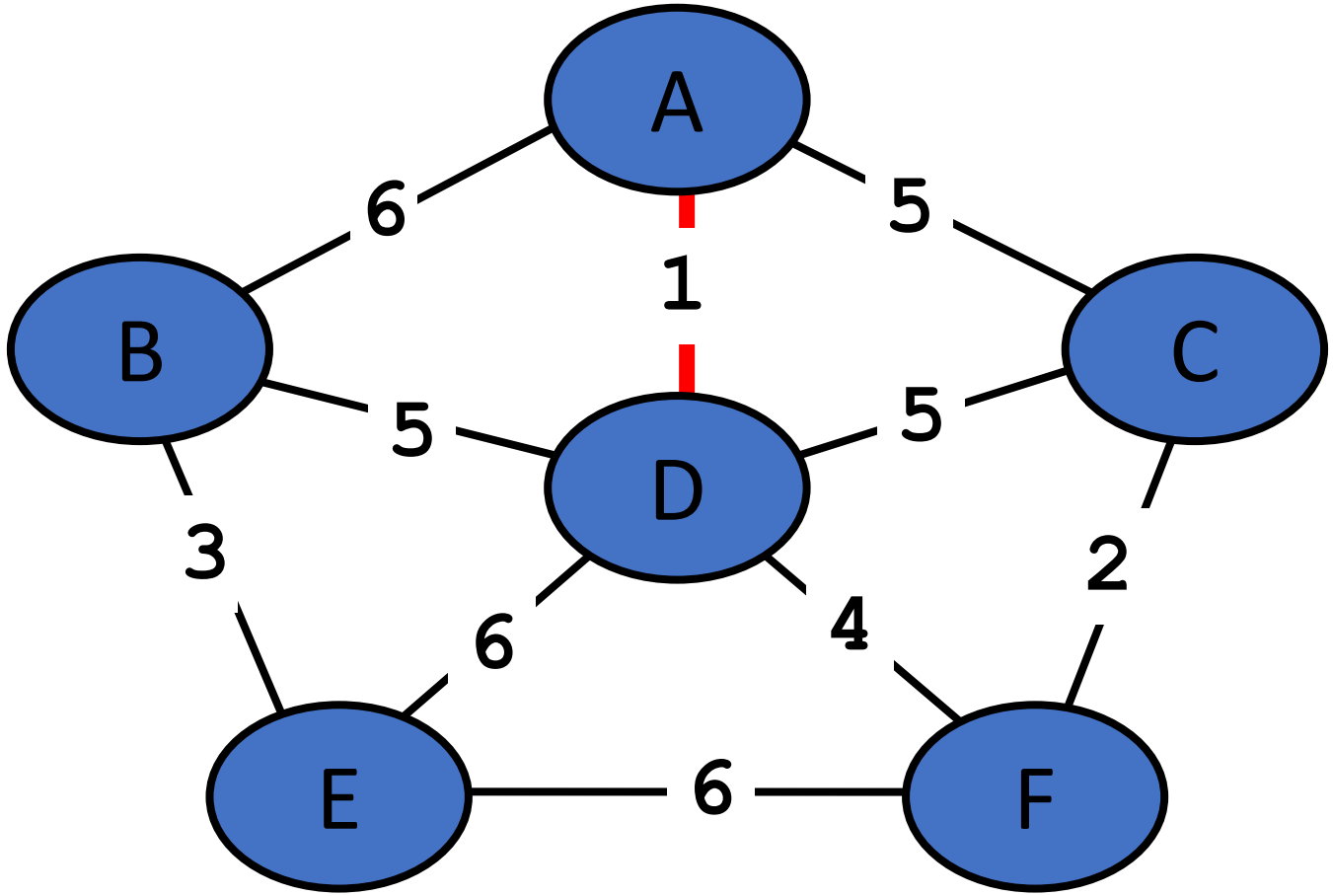
博客： **hwdong-net**.github.io

Kruskal算法

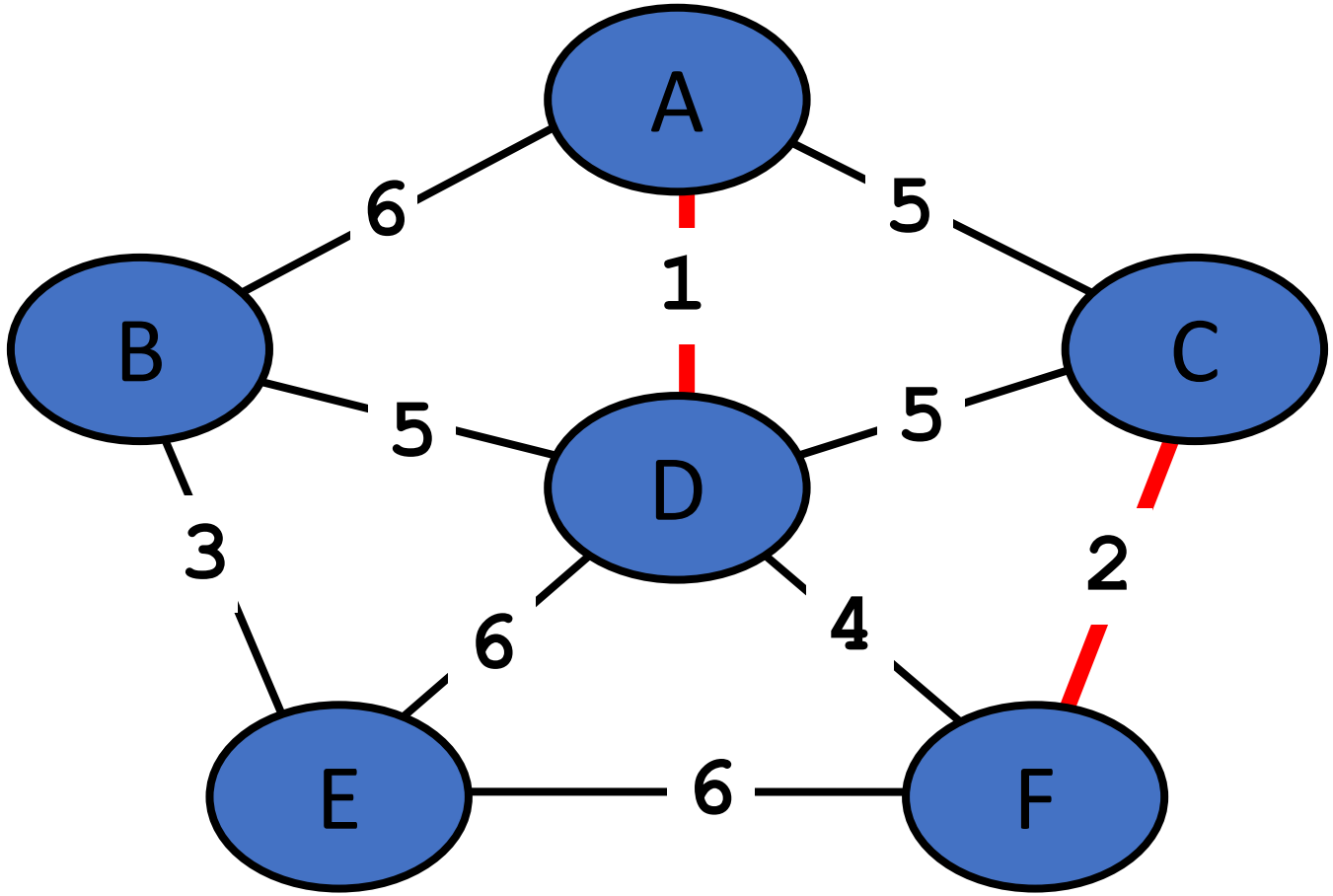
- 对图G(V, E)的所有边按权重从小到大排序： $e_1 \le e_2 \le \ldots \le e_m$
- 设T为构建中的最小生成树，E(T)={}, V(T) = V。
- While |E(T)|< |V|-1:

  将最小的不会导致回路的最小权边e_i添加到T

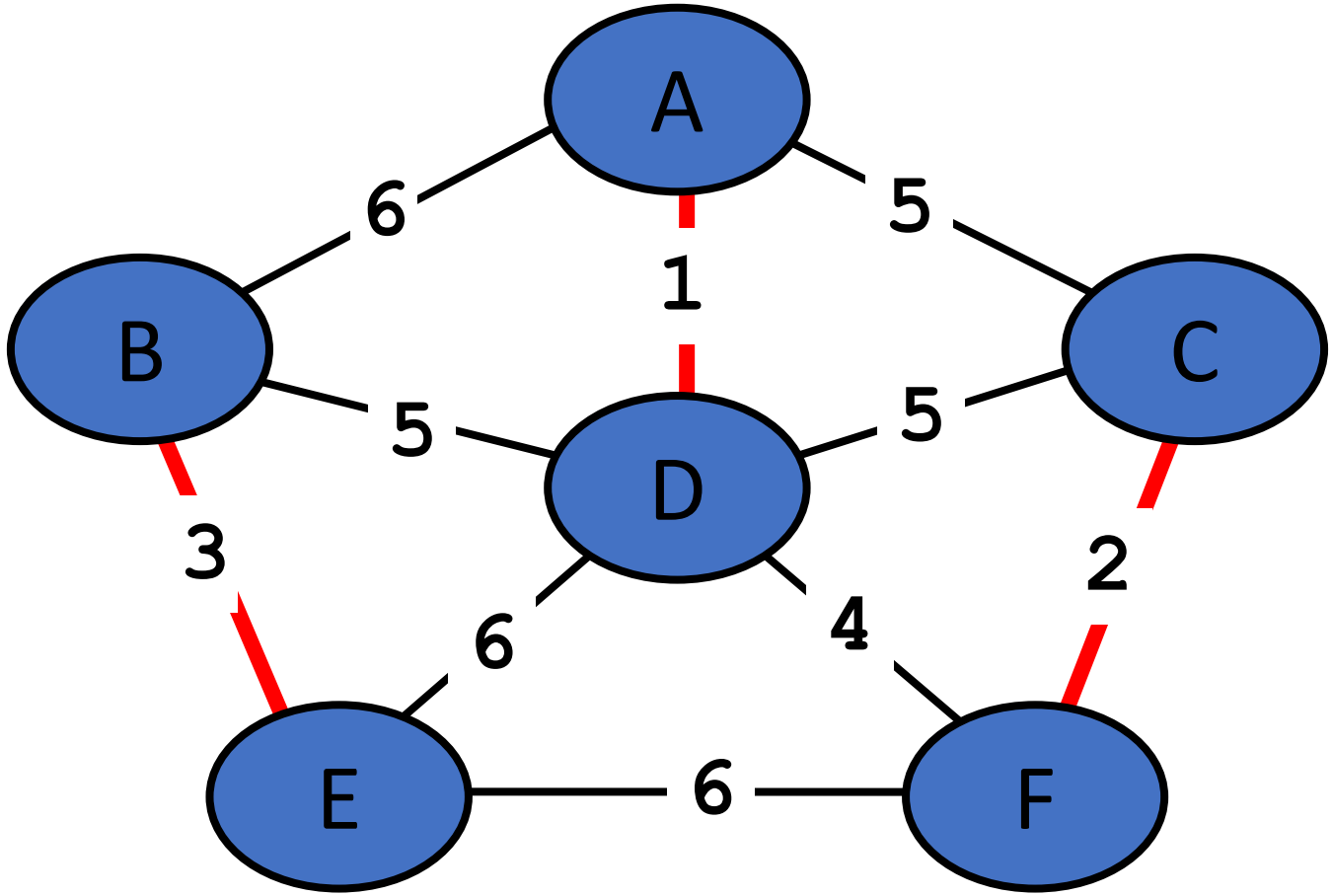
Kruskal算法



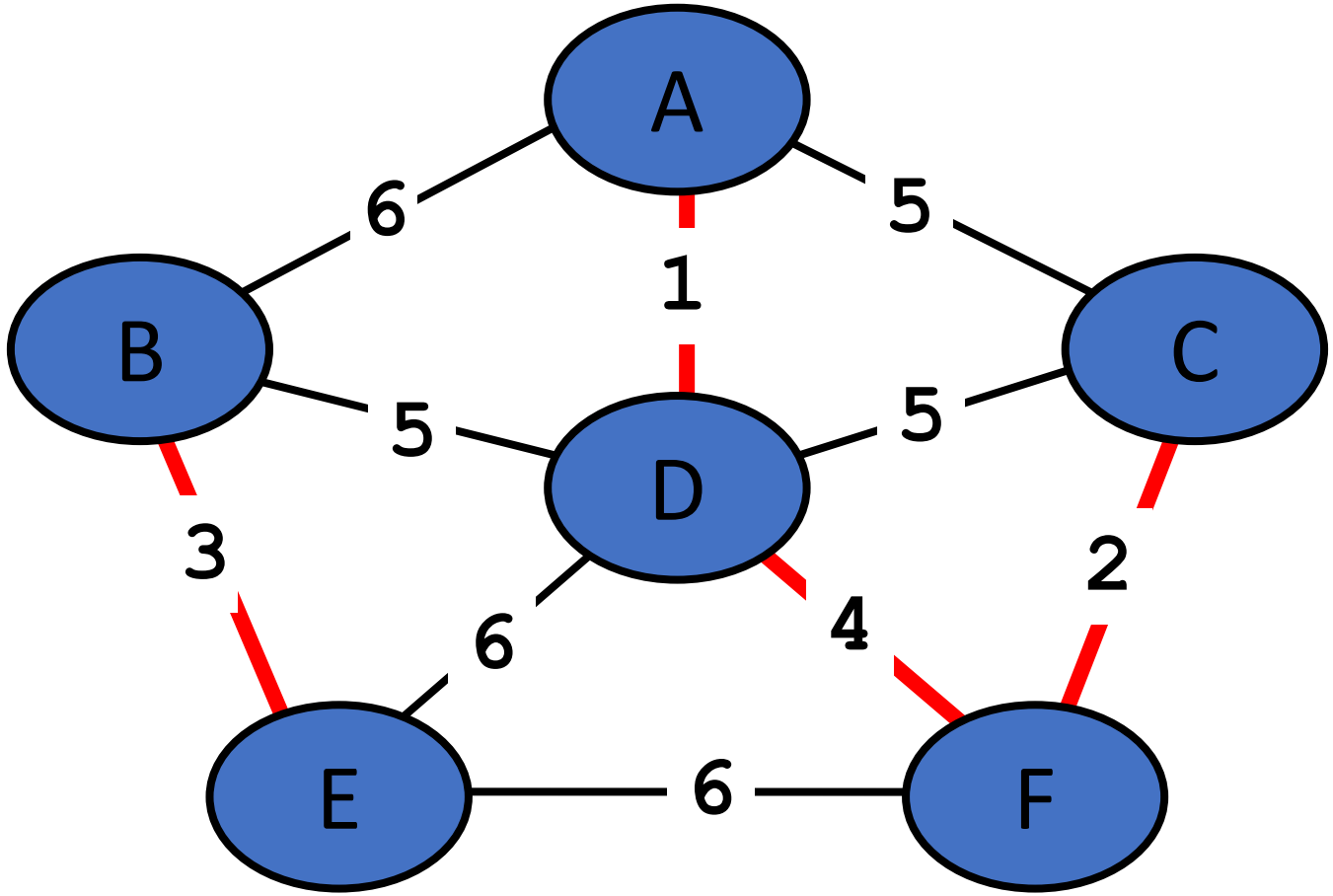
Kruskal算法



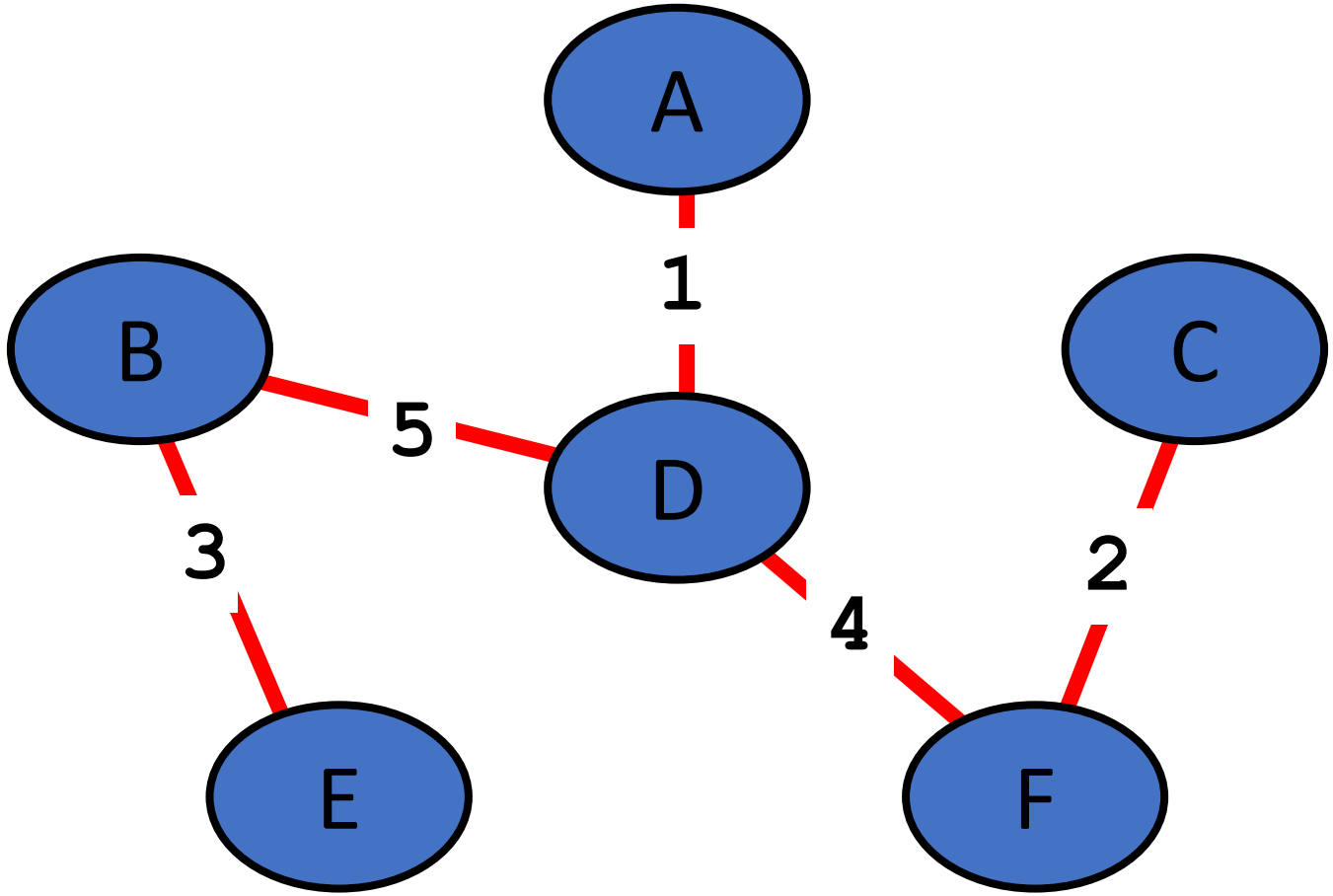
Kruskal算法



Kruskal算法



Kruskal算法



时间复杂度

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v∈V[G]                                   V
3       do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w   O(ElogE)
5  for each edge(u, v)∈E, taken in nondecreasing order by weight   E
6       do if FIND-SET(u)≠FIND-SET(v)                          logV
7            then A ← A∪{(u, v)}
8                 UNION(u, v)
                                          O(ElogE)+O(ElogV)
9  return A
```

时间复杂度

- 因为E不超过V^2, 因此，O(LogV) 和O(LogE)是一样的。
- 因此，可以说时间复杂度是O(ElogE) 或 O(ElogV)

Kruskal算法在每一步选择一条不导致回路的最小边添加到构建中的最小生成树的边集合中

证明：kruskal得到的是一个最小生成树

1. 先证明kruskal得到的是一个生成树
2. 再证明这是一个最小生成树

1. 先证明kruskal得到的T是一个生成树

证明： G'是kruskal算法得到的n-1条边及邻接点构成的子图G'(E(T), V')。

如果V'=V，则G' = T就是包含所有顶点的没有回路的n-1条边的图，根据定理6，它是一个树，即T是G的一个生成树。

因此，只要证明V'至少n个顶点（即包含了图G的所有顶点，即V' = V）。

命题： 不包含回路的n-1条边的邻接点数目至少为n。

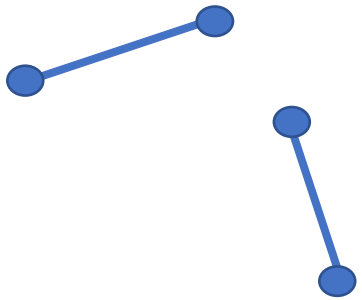
命题： 不包含回路的n-1条边的邻接点数目至少为n。

证明(数学归纳法)：

当n=1时，只有一条边，该边邻接点数目至少是2.

当n=2时，即有2条边，它们的邻接点数目至少是3. 因为如果只有2个顶点的话，这2个顶点之间就有2条边了，既不是简单图，也存在回路。

当然也可能有4个顶点



命题： 不包含回路的n-1条边的邻接点数目至少为n。

证明(数学归纳法):

设当n=k时，它们的邻接点数目至少是k+1.

当n=k+1时，如果它们的邻接点数目只有k+1，

删除任意一条边e，只剩余k个边，根据假设，必然也包含这k+1个顶点，说明这是一个树。

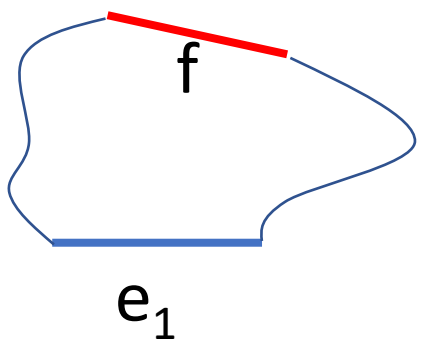
那么再将这个边添加到这个树中，必然形成回路，矛盾!

因此，它们的邻接点数目至少是k+2。得证!

2. 再证明这是一个最小生成树

证明：设T是kruskal算法得到的生成树。第一个边e_1是第一个加入到T中的边， e_1必然属于某个最小生成树。

设T*是一个最小生成树，将e_1添加到T*必然导致一个cycle，该cycle上的所有边的权值肯定不小于$w(e_1)$，删去该cycle上除e_1外的任何一条边f得到的仍然是一个生成树$T' = T + e_1 - f$，且$w(T') \leq w(T^*)$。因此,T'是一个最小生成树，而$e \epsilon T'$



设T_k是kruskal前k步构成的图，假设T_k是某个最小生成树T*的子图，
要证：第k+1步得到的T_{k+1}也是某个最小生成树T’的子图。

设第k+1步添加的边是$e_{i,k+1}$，如果$e_{i,k+1} \in T^*$，则得证。
否则，将$e_{i,k+1}$添加到T*必然产生回路。这个回路中必然有边f不属于T_k，T_k+f是T*的一部分。
根据kruskal的贪婪选择，则$w(e_{i,k+1}) \leq w(f)$
从T*删除f而添加$e_{i,k+1}$，得到一个生成树T’，且$w(T') \leq w(T^*)$。
说明T’也是最小生成树，且$e_{i,k+1} \in T'$，即T_{k+1}是最小生成树T’的子图。

- 根据上面2步数学归纳法，证明了kruskal算法得到的生成树T是最小生成树。

单源最短路径

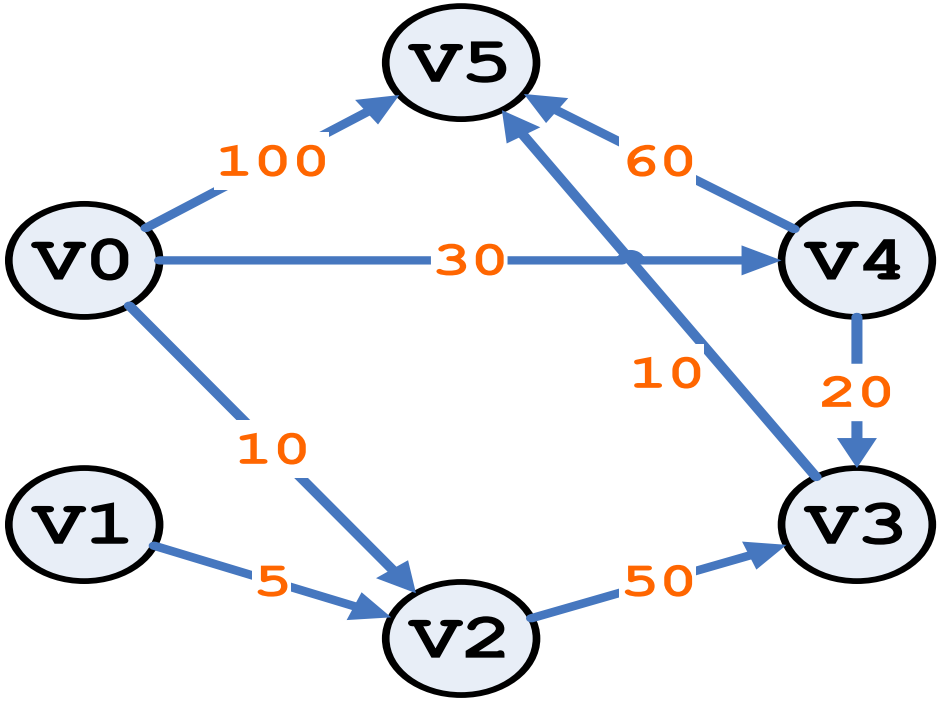
Dijkstra

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

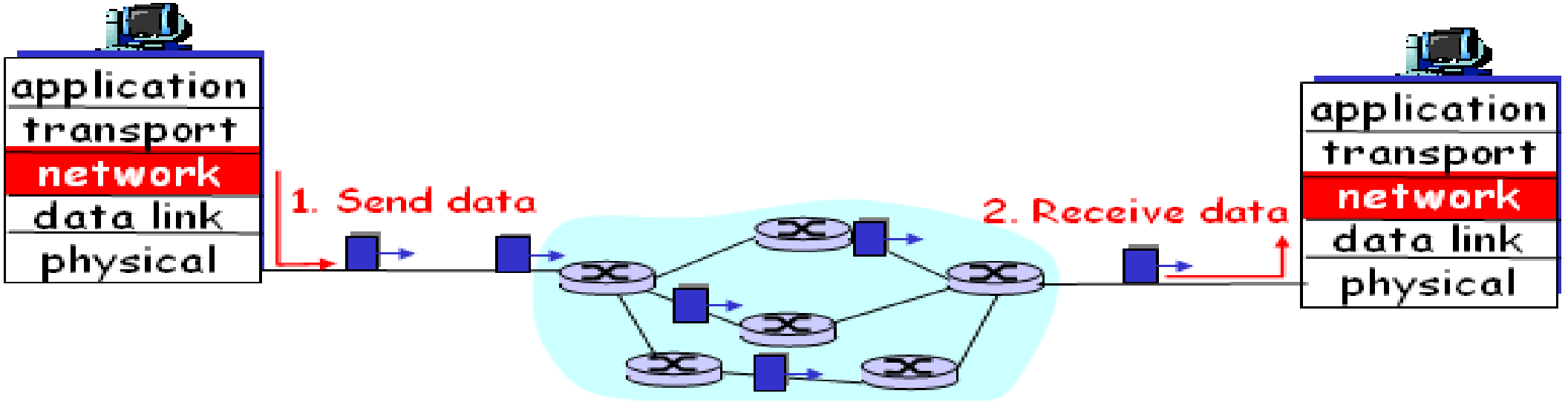
单源最短路径问题

- 从一个顶点出发，求去往其它所有顶点的最短路径及其长度。



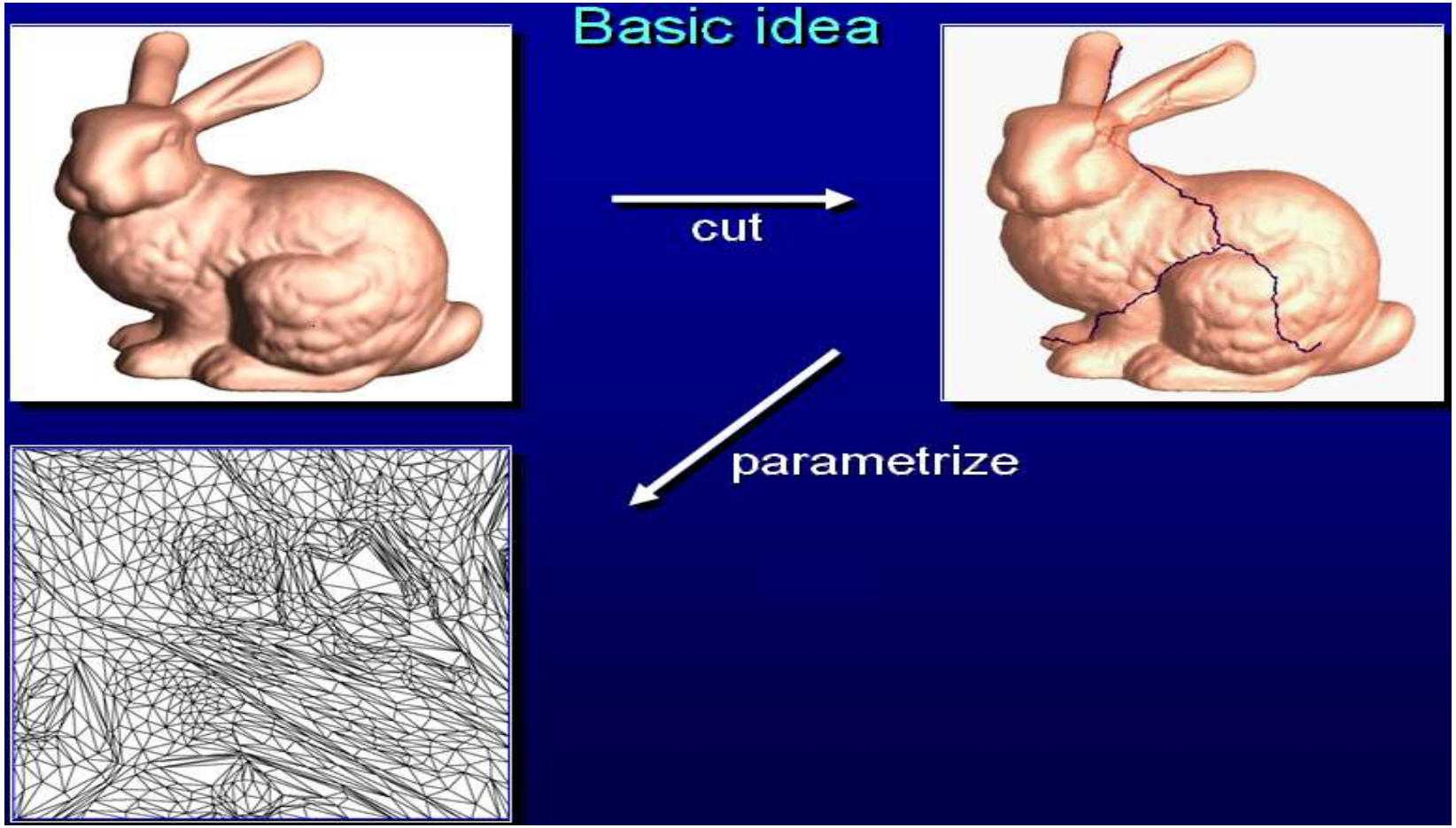
应用

- 网络的最短路径问题：如计算机网络、交通网络等



应用

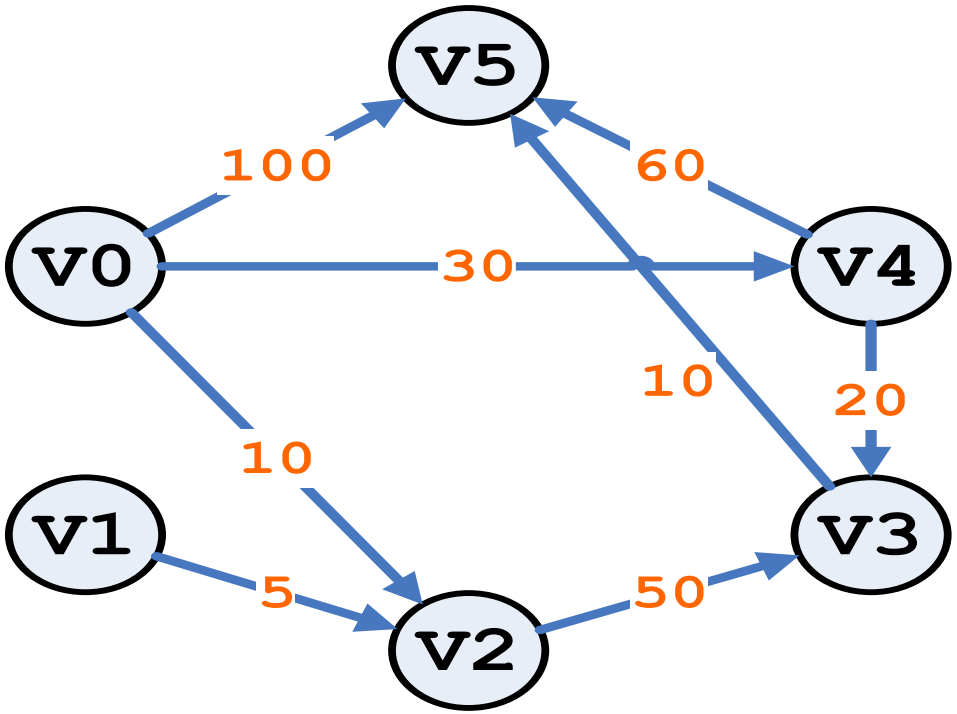
- 图形学：测地线



Dijkstra算法

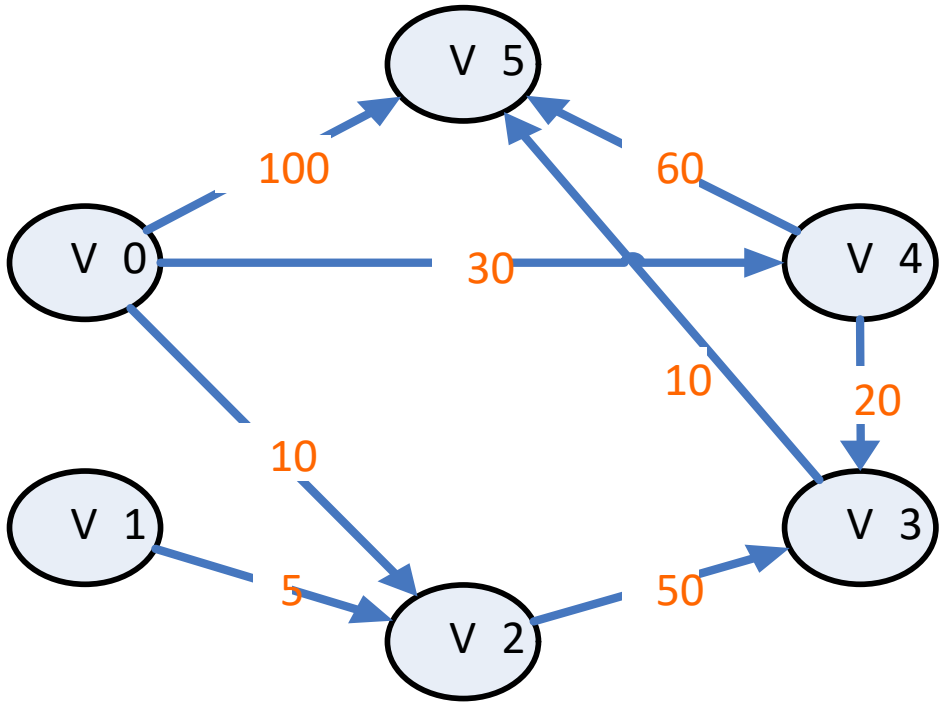
- 求从一个顶点出发到其它所有顶点的最短路径及其长度
  - 按最短路径长度递增的次序产生各个最短路径

$D(V_2) < D(V_4) < D(V_3) < \ldots$



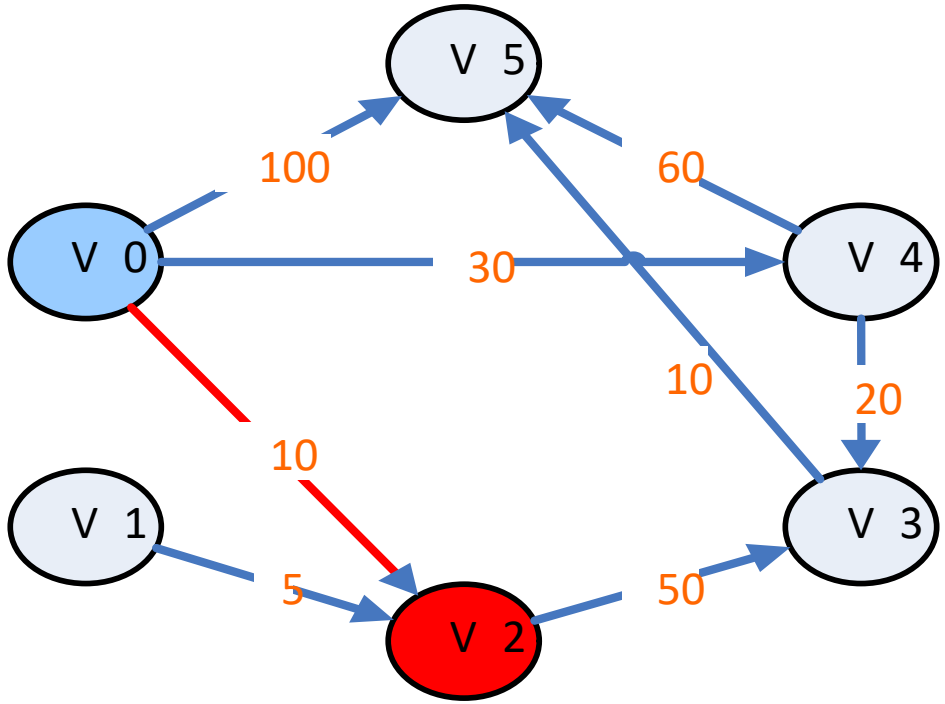
- 最短路径长度最短的最短路径的特点：

  在这条路径上，必定只含一条弧，并且这条弧的权值最小。



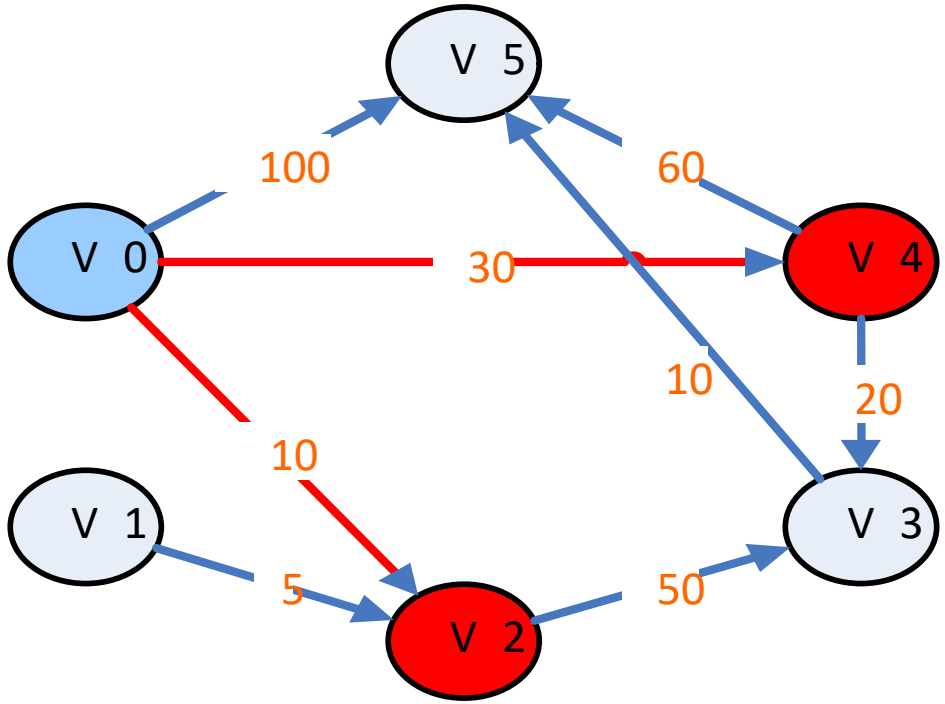
- 最短路径长度次短的最短路径的特点：

  它只可能有两种情况：或者是直接从源点到该点（只含一条弧）；或者是，从源点经过顶点v2，再到达该顶点（由两条弧组成）。



- 第3条最短路径的特点：

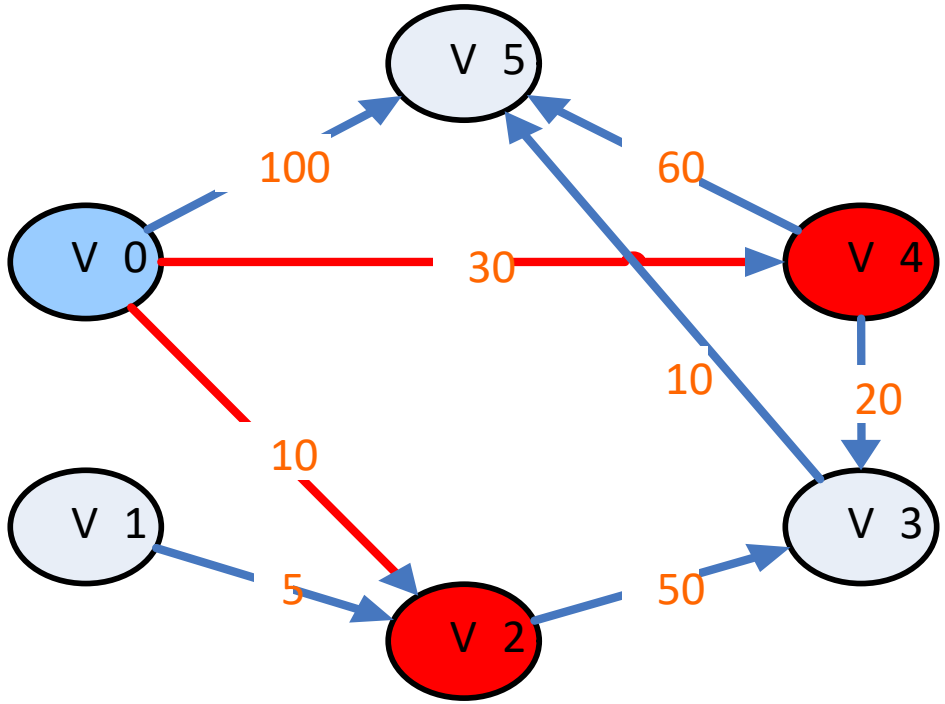
  它只可能有两种情况:或者是直接从源点到该点(只含一条弧);
  或者是，从源点经过顶点(v2,v4)，再到达该顶点(由两条弧组成)。



- 第3条最短路径的特点：

  它只可能有两种情况:或者是直接从源点到该点(只含一条弧);
  或者是，从源点经过顶点(v2,v4)，再到达该顶点(由两条弧组成)。

v4的加入，是否会改变其他未知最短距离的顶点的最短距离呢?



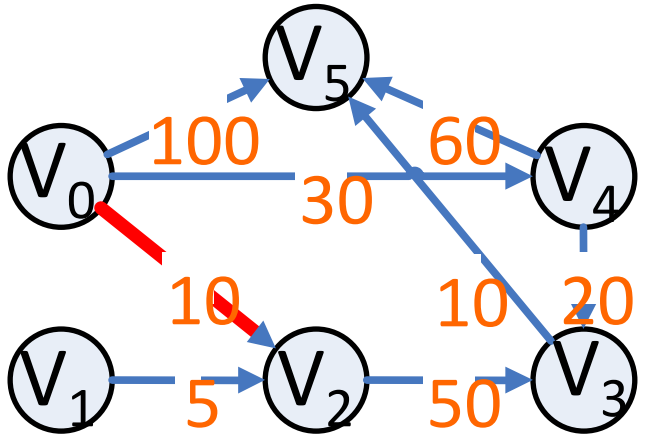
Dijkstra算法

- 数据结构
  - D = Distance：起点到终点的当前距离
  - P = Path：终点的前一个顶点
  - S = Set：已知最短路径的目的顶点集合
- 初始化
  - S = {V0}
  - 对于所有顶点v
    - 若存在弧<V0,V>，D(v) = arcs(V0,V)
    - 否则 D(v) = ∞

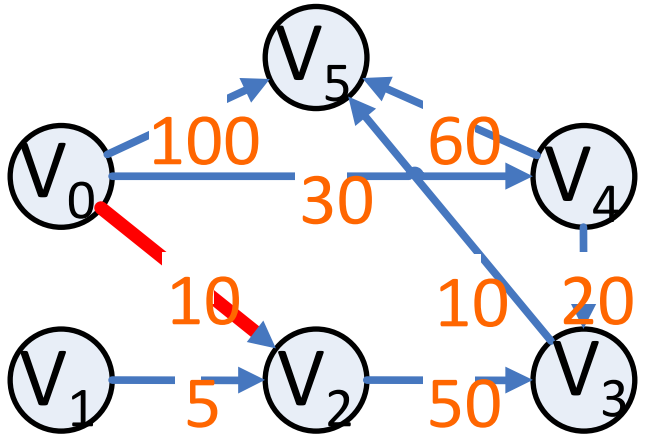
Dijkstra算法

```
循环(直到找不到从v0可以达到的顶点){
        在不在S中的顶点中查找D(v)最小的顶点v
        把v加入S，即v已知最短路径
        更新v的所有邻接点w的D(w)和P(w)
        if( D(v)+c(v,w) < D(w) ){
            D(w) = D(v)+arcs(v,v);
            P(w) = v;
        }
}
```

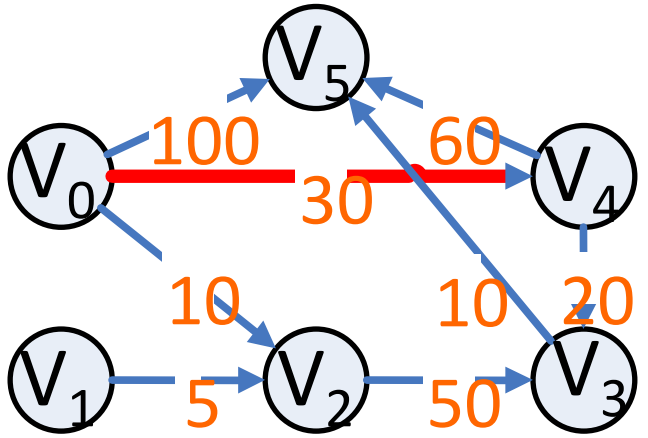
V_1	∞	0
V_2	10	0
V_3	∞	0
V_4	30	0
V_5	100	0



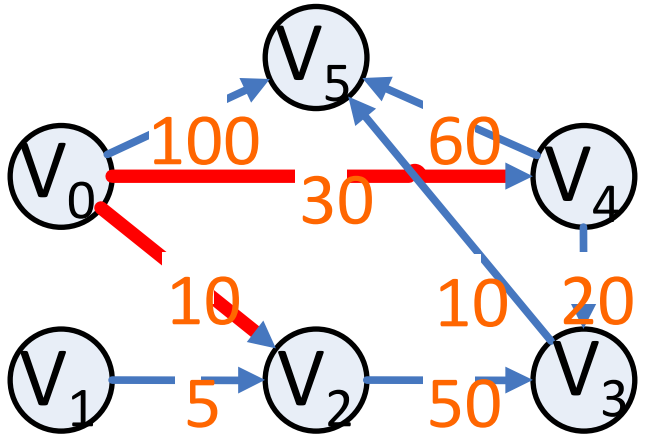
V_1	∞	0	∞	0
V_2	10	0	10	0
V_3	∞	0	**60**	2
V_4	30	0	30	0
V_5	100	0	100	0



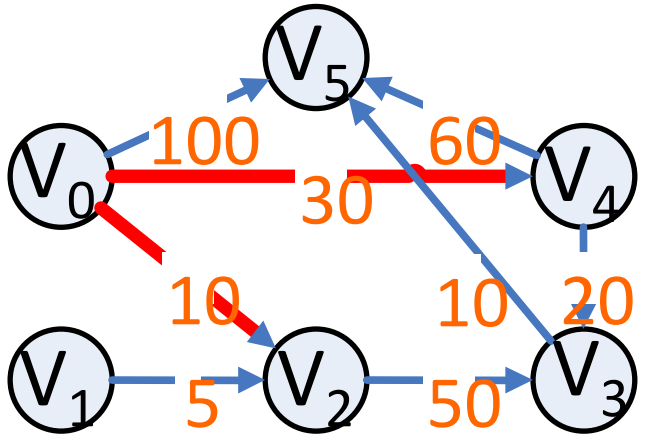
V_1	∞	0	∞	0
V_2	10	0	10	0
V_3	∞	0	60	2
V_4	30	0	30	0
V_5	100	0	100	0



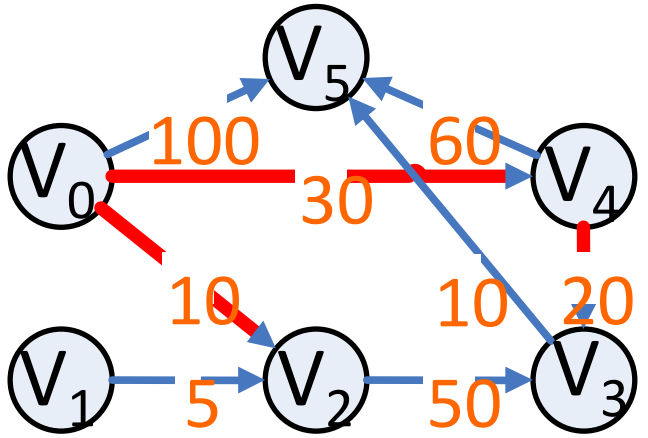
V_1	∞	0		∞	0		∞	0
V_2	10	0		10	0		10	0
V_3	∞	0		60	2		**50**	4
V_4	30	0		30	0		30	0
V_5	100	0		100	0		**90**	4



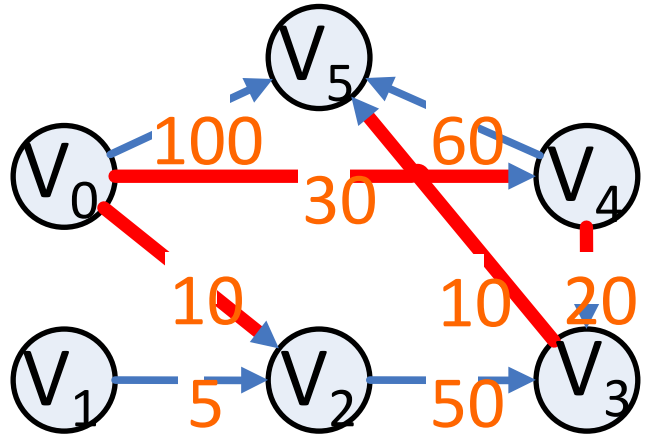
V_1	∞	0		∞	0		∞	0
V_2	10	0		10	0		10	0
V_3	∞	0		60	2		50	4
V_4	30	0		30	0		30	0
V_5	100	0		100	0		90	4



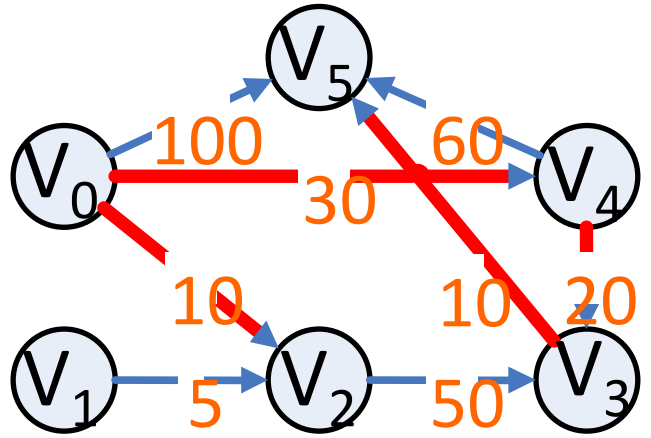
V_1	∞	0	∞	0	∞	0	∞	0
V_2	10	0	10	0	10	0	10	0
V_3	∞	0	60	2	50	4	50	4
V_4	30	0	30	0	30	0	30	0
V_5	100	0	100	0	90	4	**60**	3



V_1	∞	0	∞	0	∞	0	∞	0
V_2	10	0	10	0	10	0	10	0
V_3	∞	0	**60**	2	50	4	50	4
V_4	30	0	30	0	30	0	30	0
V_5	100	0	100	0	90	4	60	3



V_1	∞	0	∞	0	∞	0	∞	0	∞	0
V_2	10	0	10	0	10	0	10	0	10	0
V_3	∞	0	**60**	2	50	4	50	4	50	4
V_4	30	0	30	0	30	0	30	0	30	0
V_5	100	0	100	0	90	4	60	3	60	3



Dijkstra算法

对于所有顶点v
  P(v) = v0;
  若存在弧<V0,V>，D(v) = arcs(V0,V)
  否则 D(v) = ∞

n次

循环(直到找不到可加入S的顶点){
  在不在S中的顶点中查找D(v)最小的顶点v
  把v加入S，即v已知最短路径
  更新v的所有邻接点w的D(w)和P(w)
    if(D(v)+c(v,w) < D(w)){
      D(w) = D(v)+arcs(v,w); P(w) = v; }
}

n-1次
n次
n次?

时间复杂度 = $O(e+n^2)$

时间复杂度 = $O(e+n\log_2 n)$

堆

证明：数学归纳法

- 对每个uϵS,证明d[u]就是从起点到u的最短距离
- 对S中元素个数进行归纳法。
- 1) |S|=2，是成立的。
- 2)对|S|=k成立，v是下一个加入到S中的顶点，设d[v]路径的前一个顶点为u，即通过u到达v。