贪心算法的实例

哈夫曼编码

Huffman Codes – 用于文件压缩

【Example】 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

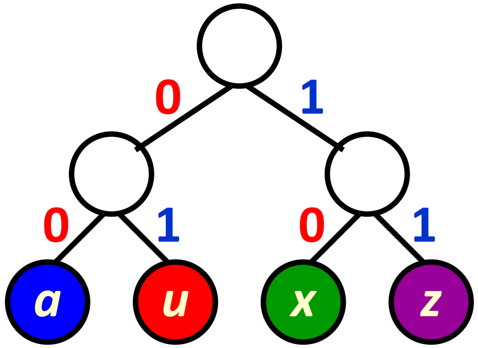
- frequency ::= number of occurrences of a symbol.

In string *aaaxuaxz* , *f*(*a*) = 4, *f*(*u*) = 1, *f*(*x*) = 2, *f*(*z*) = 1.

The size of the coded string can be reduced using variable-length codes, for example, *a* = 0, *u* = 110, *x* = 10, *z* = 111. ➡ 00010110010111

Note: If all the characters occur with the same frequency, then there are not likely to be any savings.

Representation of the original code in a binary tree /* trie */

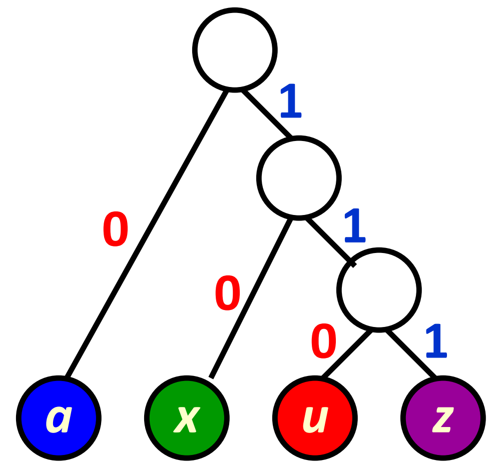


Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

- If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\sum d_i f_i$.

Cost (*aaaxuaxz* → 00000010001001011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



Any sequence of bits can always be decoded unambiguously if the characters are placed only at the leaves of a *full tree* – such kind of code is called *prefix code*.

Find the full binary tree of minimum total cost where all characters are contained in the leaves.

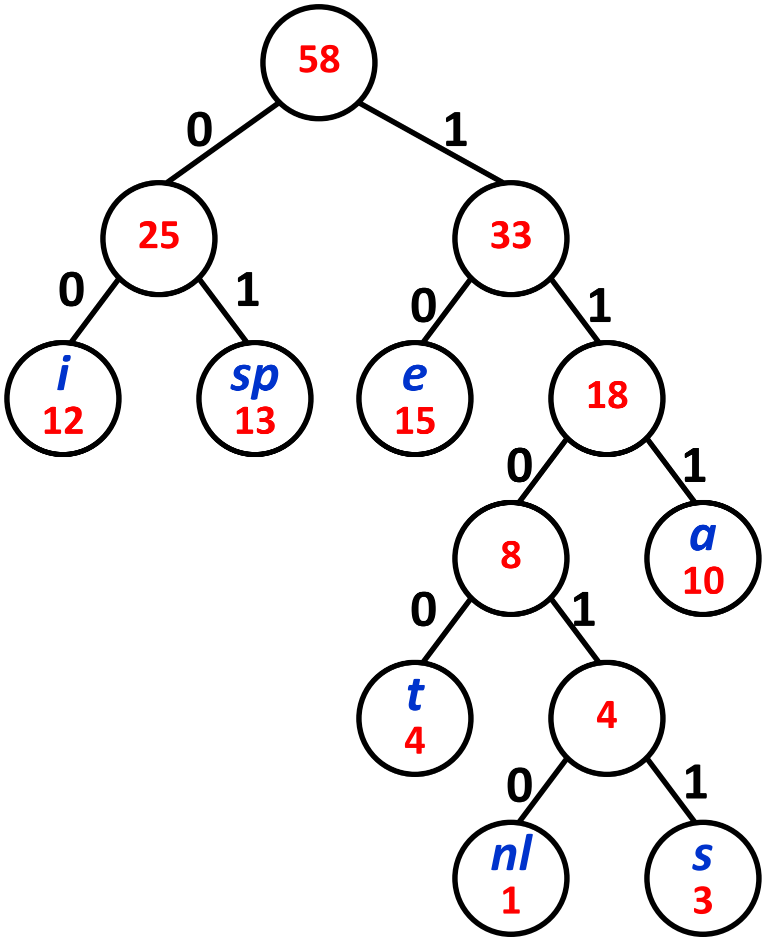
Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{
    consider the C characters as C single node binary trees,
    and initialize them into a min heap;
    for ( i = 1; i < C; i++ ) {
      create a new node;
      /* be greedy here */
      delete root from min heap and attach it to left_child of node;
      delete root from min heap and attach it to right_child of node;
      weight of node = sum of weights of its children;
      /* weight of a tree = sum of the frequencies of its leaves */
      insert node into min heap;
   }
}
```

$T = O(\ C \log C\ )$

【Example】

C_i	a	e	i	s	t	sp	nl
f_i	10	15	12	3	4	13	1



a : 111

e : 10

i : 00

s : 11011

t : 1100

sp : 01

nl : 11010

$$
\begin{aligned}
Cost &= 3\times 10 + 2\times 15 \\
&\quad + 2\times 12 + 5\times 3 \\
&\quad + 4\times 4 + 2\times 13 \\
&\quad + 5\times 1 \\
&= 146
\end{aligned}
$$

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。

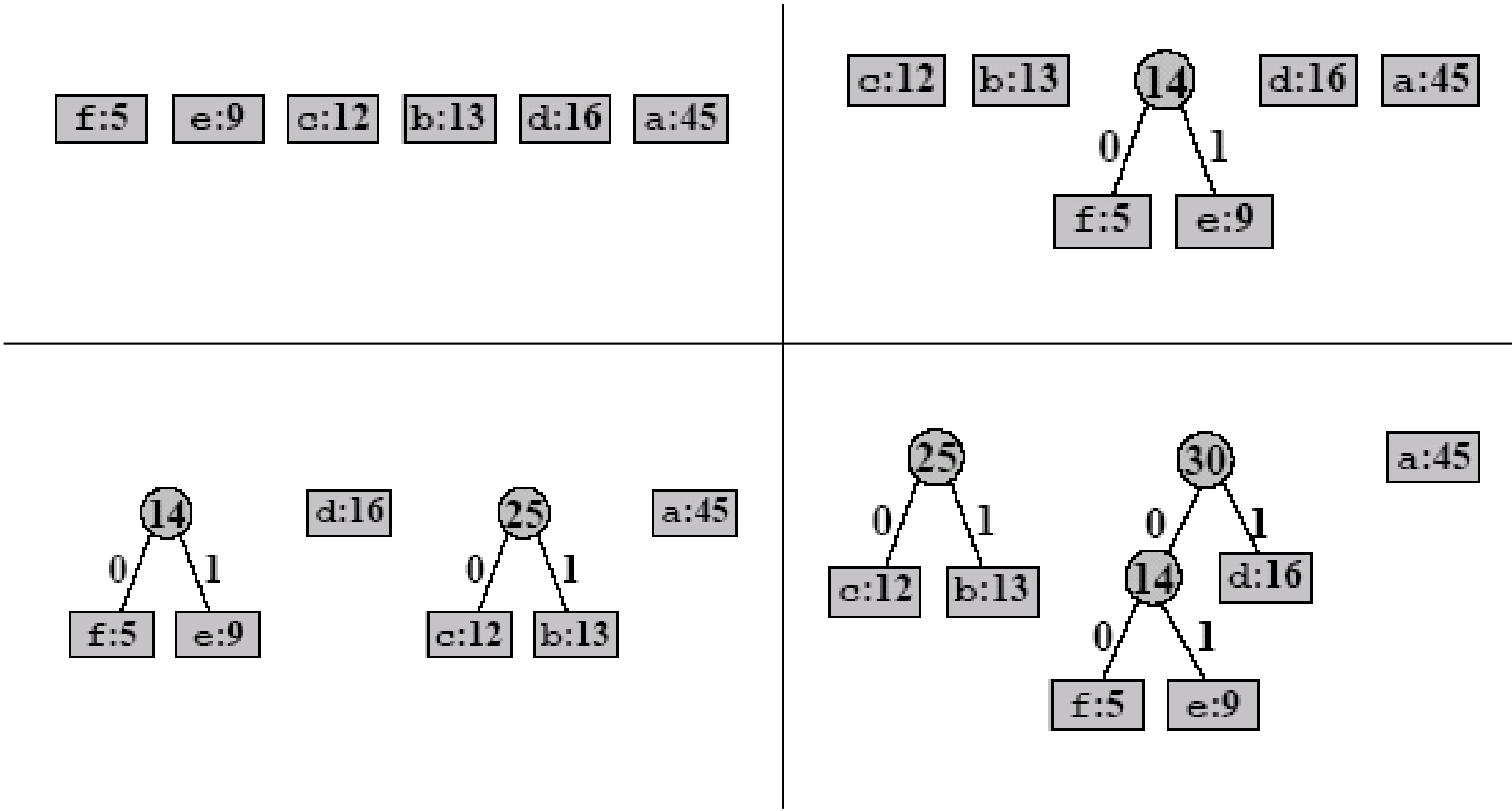
某一数据文件中字符出现的频率表如下：

	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

使用定长码：则表示6个不同的字符需要3位：a=000,b=001,…,f=101。用这种方法对整个文件进行编码需要300,000位。

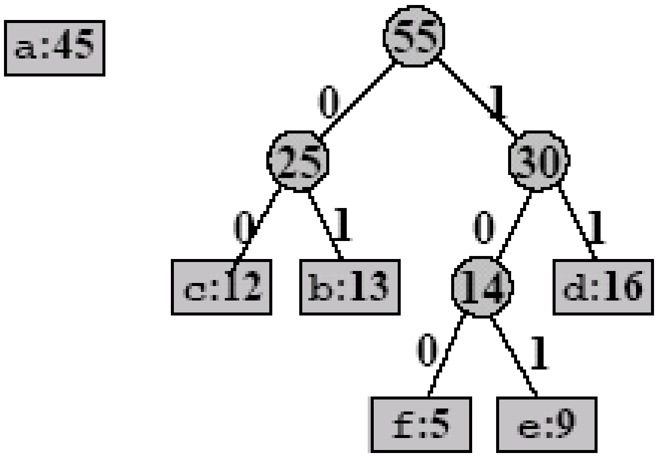
使用变长码：字符a用1位串0表示，而字符f用4位串1100表示。整个文件的总码长为：(45×1+13×3+12×3+16×3+9×4+5×4)×1000=224,000位。它比用定长码方案好，总码长减少约25%。

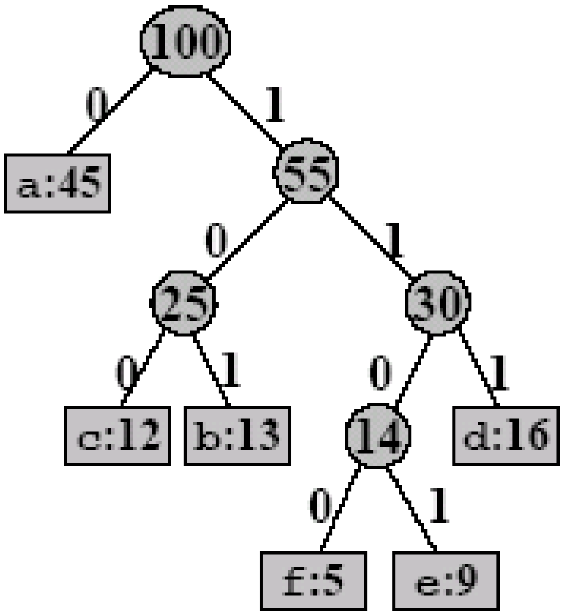
Example of Huffman codes



Example of Huffman codes

- (cont.)





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

(2) 最优子结构性质

- 证明思路:

  设字符集C的1个最优前缀码表示为二叉树T 。

  采用一定方法，将T修改新树T’，使得

1）在T’中具有最小频率的x和y是最深叶子，且互为兄弟

2）T’还是C的最优前缀。

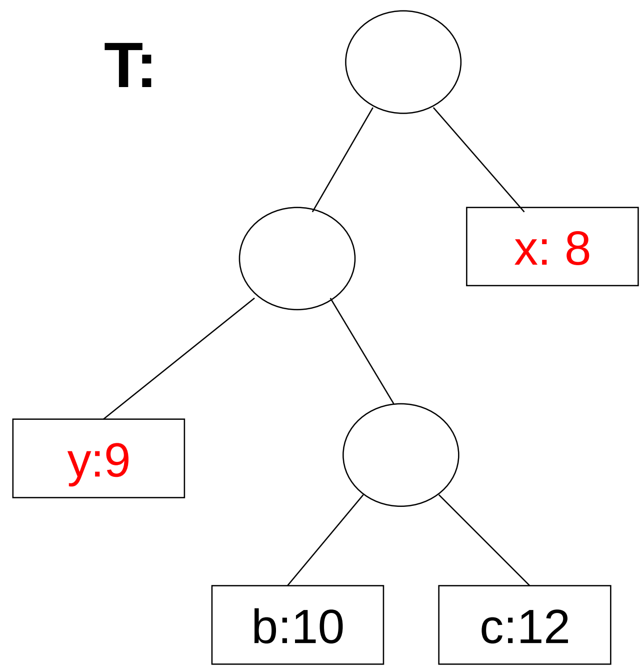
  这样x、y在最优前缀码T’中只有最后一位不同。

假设：b、c是T中最深叶子且互为兄弟，f(b)<=f(c);

已知：C中2个最小频率字符f(x)<=f(y),

但在T中，x、y有可能并非最深结点！！

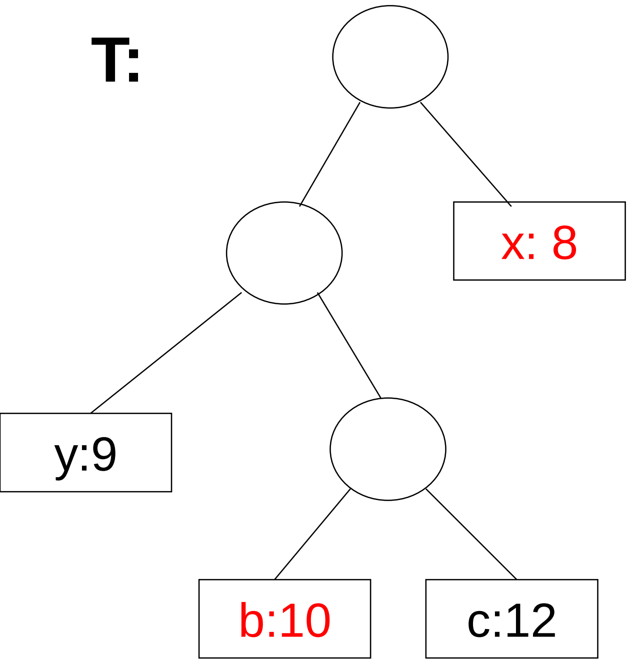
T:

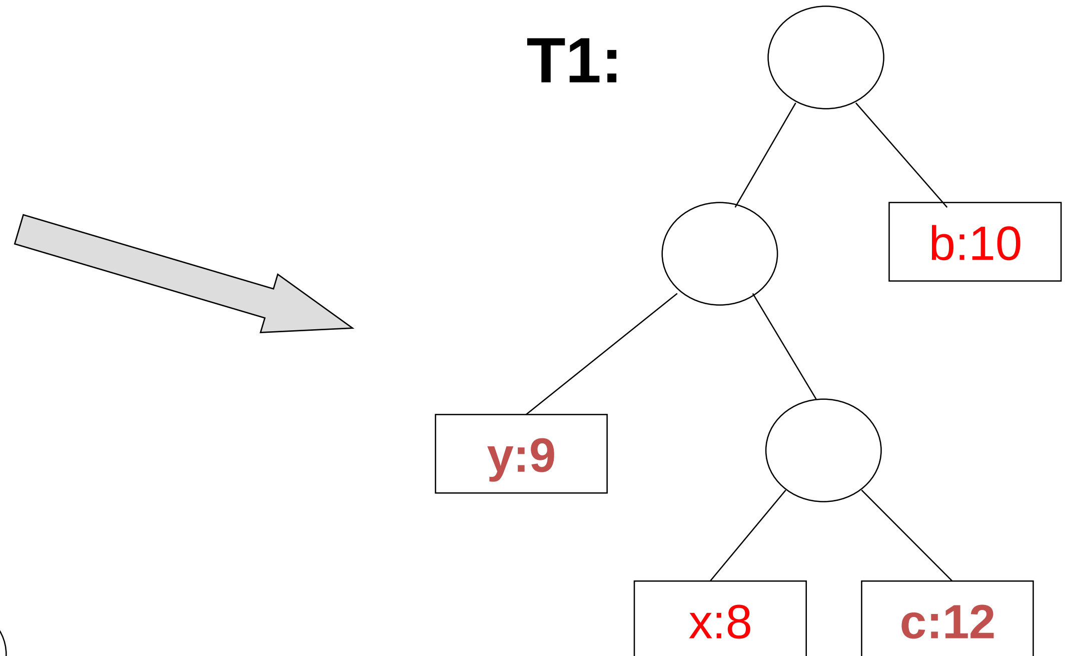


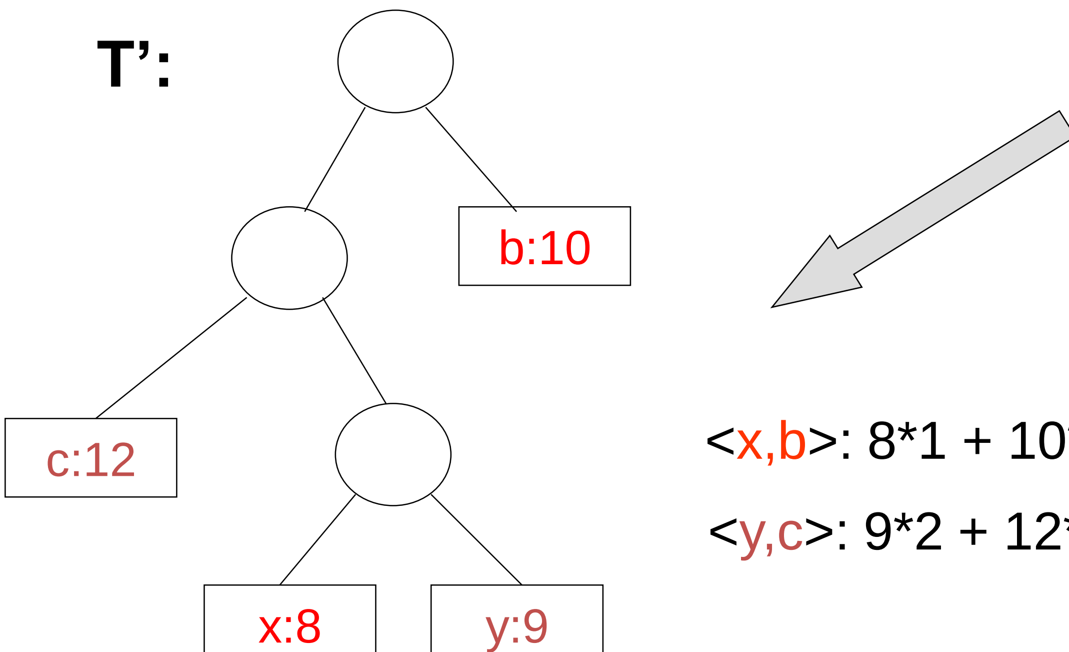
由于x、y具有最小频率,

故f(x)<=f(b)， f(y)<=f(c)

Step1.在T中交换b和x的位置得到T1
Step2. 在T1中交换c和y的位置，得到T’







<x,b>: 8*1 + 10*3=38 > 8*3 + 10*1=34

<y,c>: 9*2 + 12*3=54 > 9*3 + 12*2=51

- 可以证明:

1) B(T)—B(T1) ≤0，即第一步交换不会增加平均码长
2) B(T1)—B(T’) ≤0，即第二步交换也不会增加平均码长

故T’的码长仍然是最短的，即T’是最优前缀码，并且其最小频率的x、y具有最深的深度（最长的编码），且只有最后一位不同。

二、最优子结构性质

需要证明:

给定字符集C和其对应的最优前缀码T，可以从中得到子问题C’ (C的子集)及其对应的最优前缀子树T’

- 构造性证明:

对T中2个互为兄弟的叶节点x、y，z为其父节点。

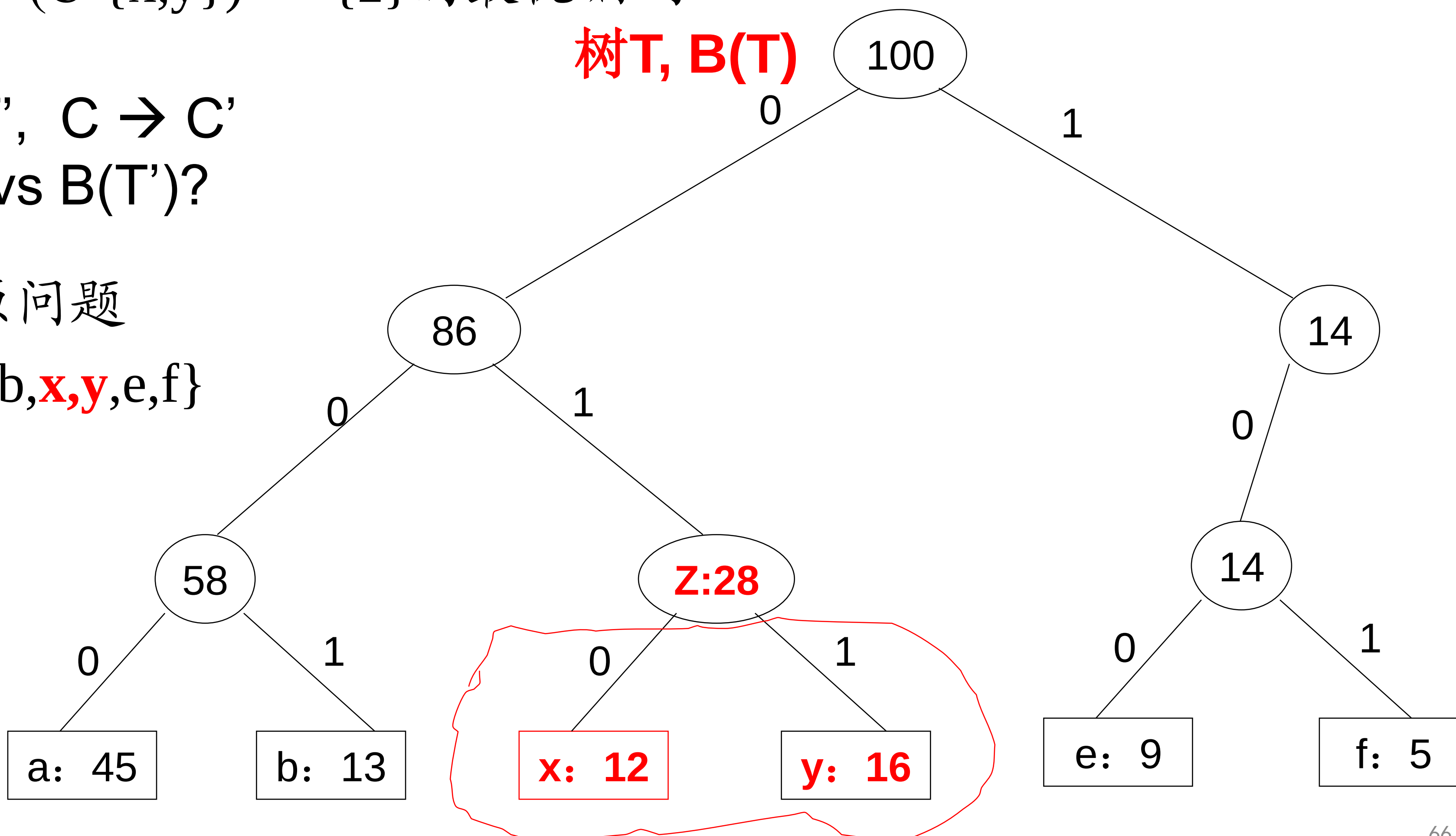
将z看做频率为f(z)=f(x)+f(y)的字符，则T' =T-{x, y}是子问题 C'= (C-{x,y}) ∪ {z}的最优编码

T→ T'，C → C'

B(T) vs B(T')?

e.g. 原问题

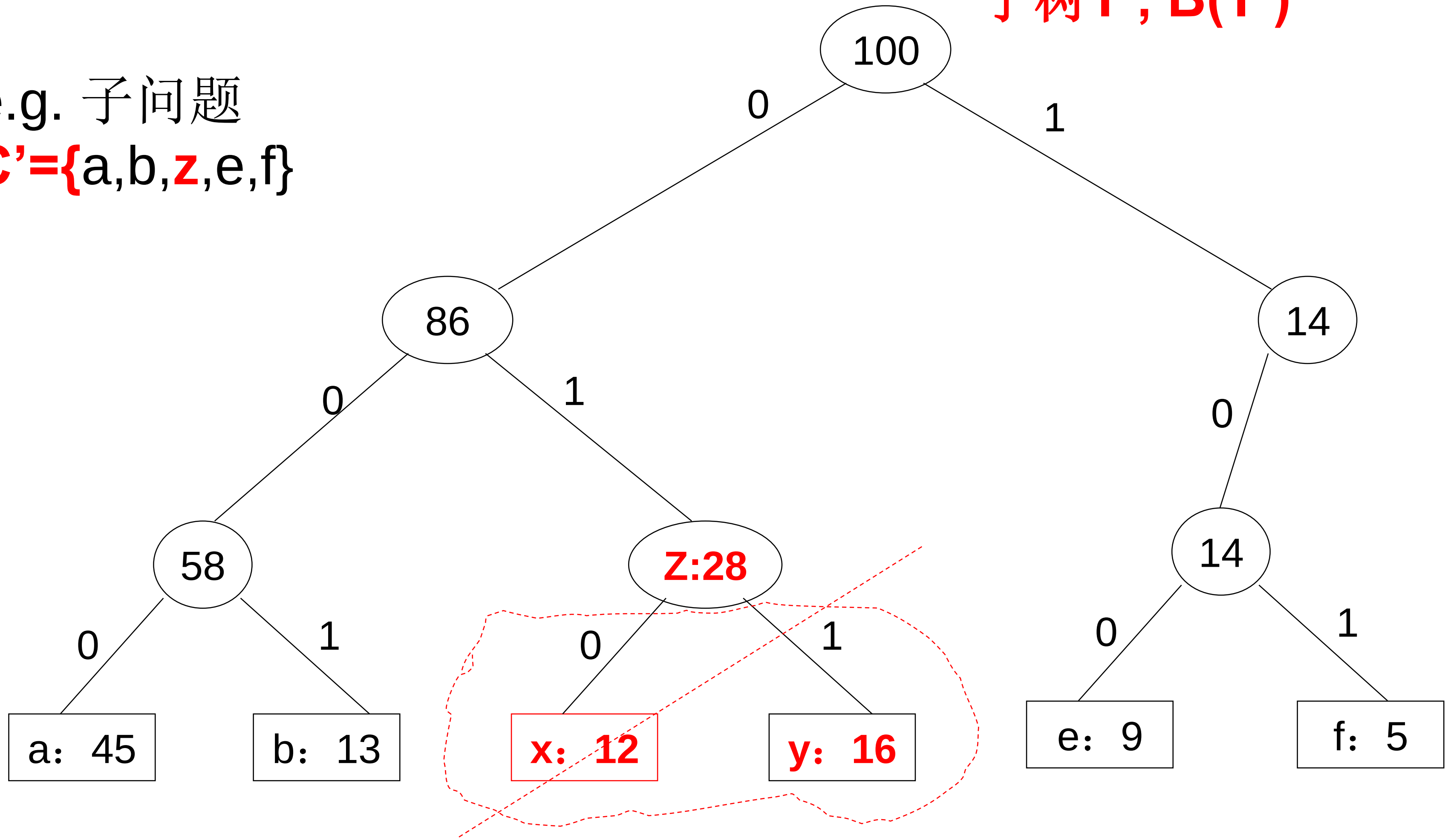
C={a,b,**x,y**,e,f}



e.g. 子问题

C'={a,b,**z**,e,f}

子树T', B(T')



- 证明关键点:

1）T的平均码长B(T)可用子树T’的平均码长B(T’) 表示

$$B(T) = B(T') + 1*f(x) + 1*f(y)$$

上式：递推表达式，表示原问题最优值与子问题最优值之间的关系

2) T’所表示的C’的前缀码的码长B(T’)是最短/最优的

反证法证明:

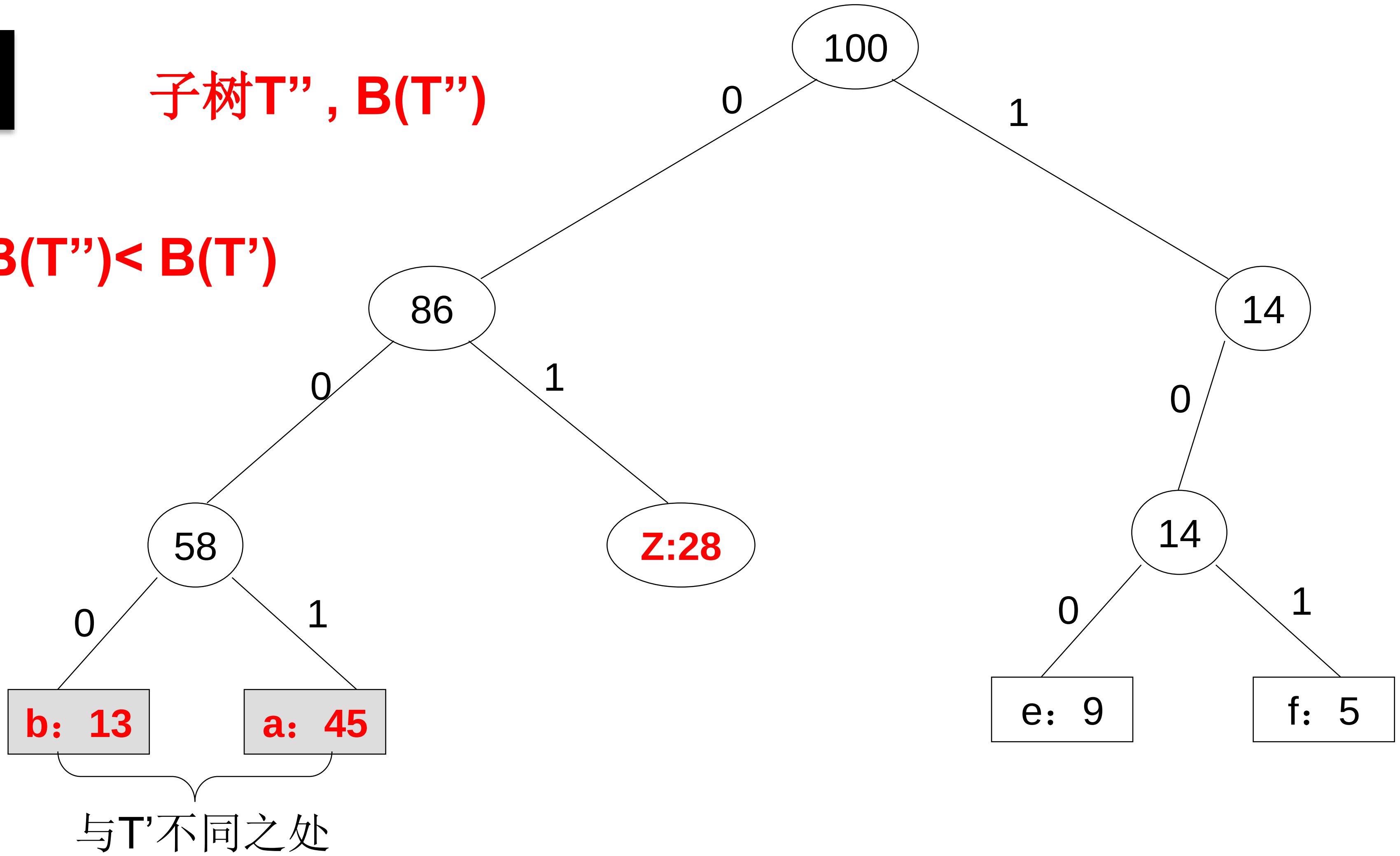
假设有另一个T”，是子问题C’的最优前缀码，即B(T’)>B(T”)。

节点z在T”还是一个叶节点，在T” 中将z替换为其子节点x、y，得到T’”。

e.g 见下页:

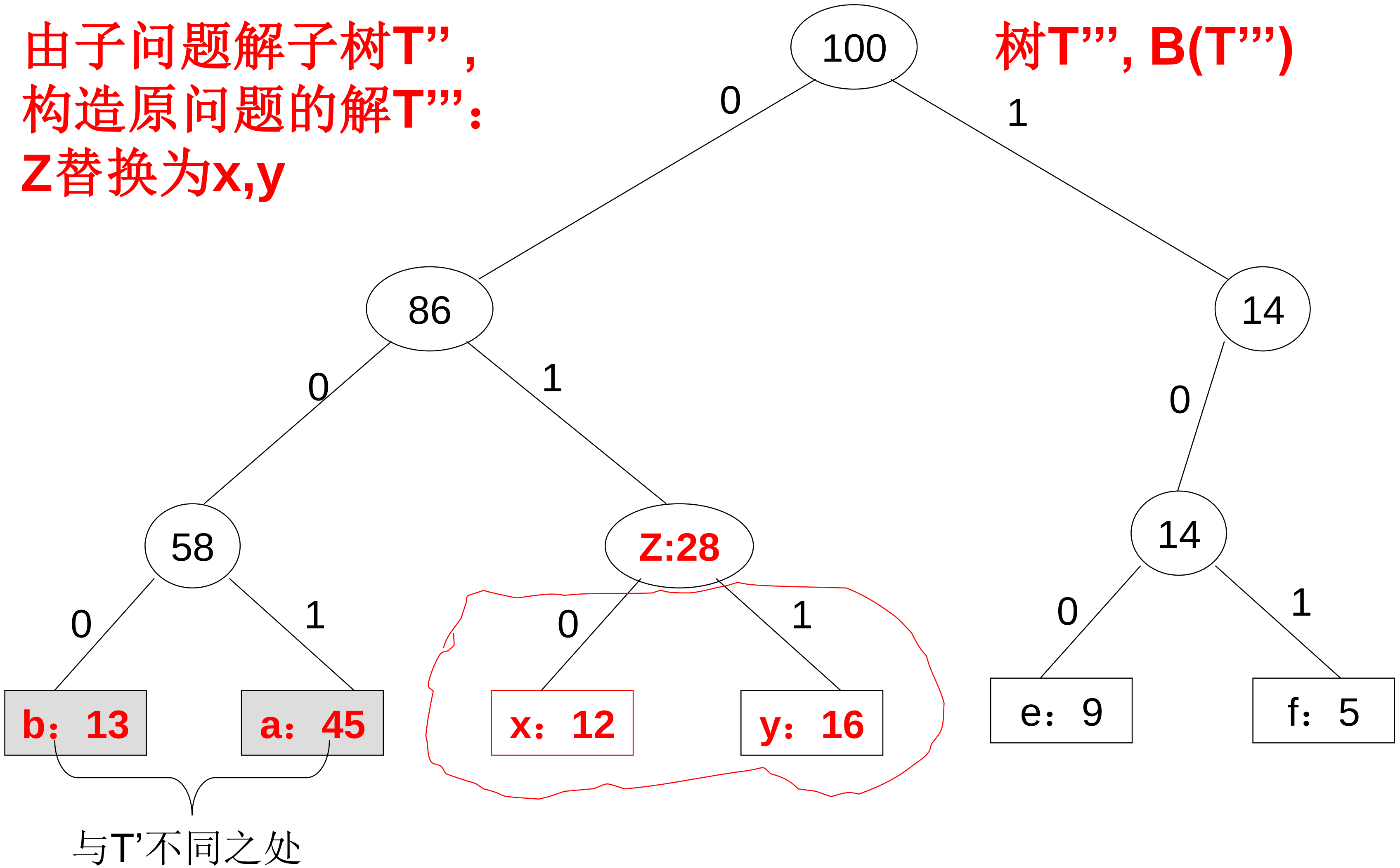
子树T”, B(T”)

B(T”)< B(T’)



由子问题解子树T”，
构造原问题的解T’”：
Z替换为x,y

树T’”, B(T’”)



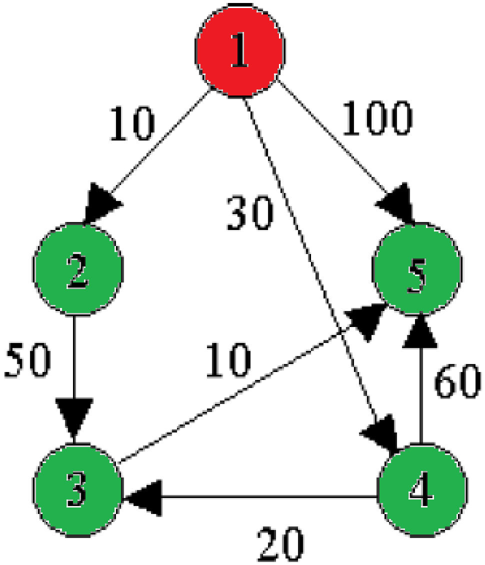
T’”是关于原问题C的1个解，同时B(T’”)<B(T)，与T是最优解矛盾。

贪心算法应用

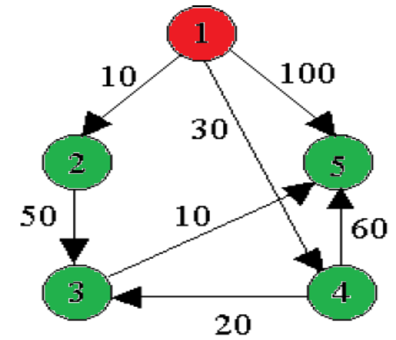
单源最短路径

单源最短路径

给定带权有向图G =(V, E)，其中每条边的权是非负实数。另外，还给定V中的一个顶点，称为**源**。现在要计算从源到所有其它各顶点的**最短路长度**。这里路的长度是指路上各边权之和。这个问题通常称为**单源最短路径问题**。



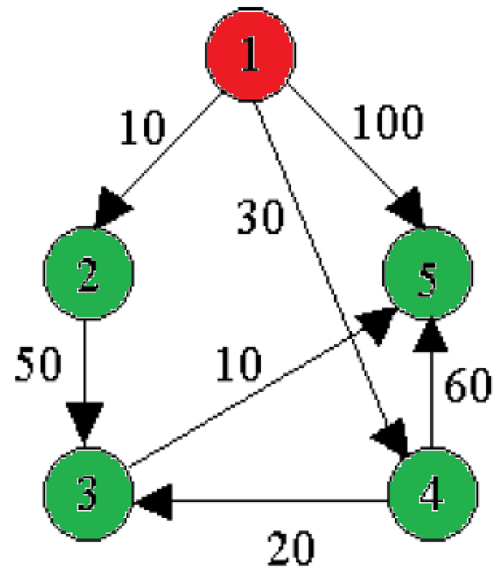
贪心算法的求解



其基本思想是，设置顶点集合S并不断地作贪心选择来扩充这个集合。一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。

初始时，S中仅含有源。设u是G的某一个顶点，把从源到u且中间只经过S中顶点的路称为从源到u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。Dijkstra算法每次从V-S中取出具有最短特殊路长度的顶点u，将u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。

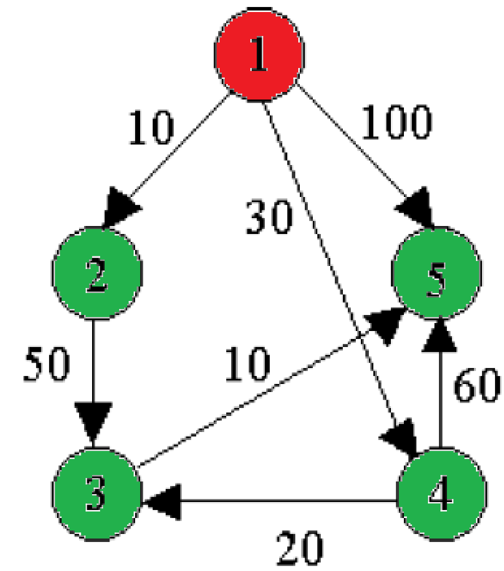
例如，对右图中的有向图，应用Dijkstra算法计算从源顶点1到其它顶点间最短路径的过程列在下表中。



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

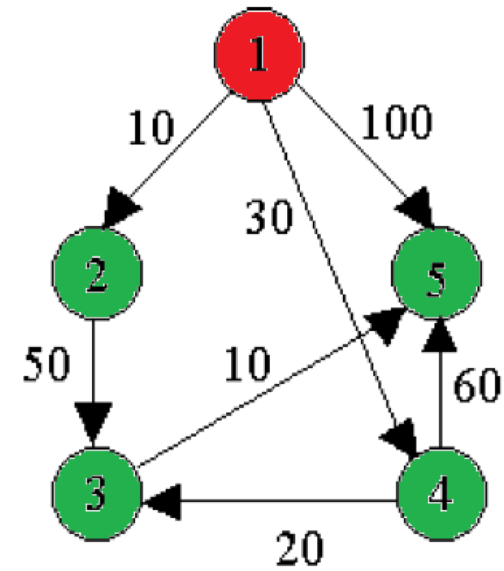
例如，对右图中的有向图，应用Dijkstra算法计算从源顶点1到其它顶点间最短路径的过程列在下表中。



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

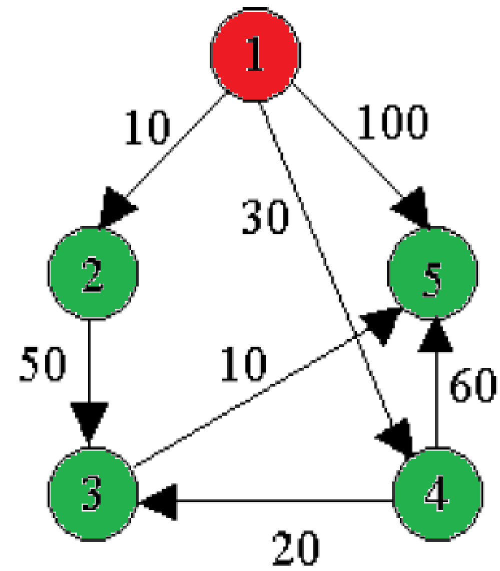
例如，对右图中的有向图，应用Dijkstra算法计算从源顶点1到其它顶点间最短路径的过程列在下表中。



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

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表中。



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

给定带权有向图G =(V, E)，V={1,2,3,..., n}, 其中每条边的权c[i][j]是非负实数。

给定V中的一个顶点v，称为**源**。

要求: 计算从源v到所有其它各顶点的**最短路长度**。

路长度：路上各边权之和。

一、Dijkstra算法: 贪心算法

1. 设置集合S，记录组成最短路径的顶点

1) 初始时S中只有源点v。不断地作贪心选择，扩充S

2) 1个顶点属于集合S，当且仅当从源到该顶点的最短路径长度已知

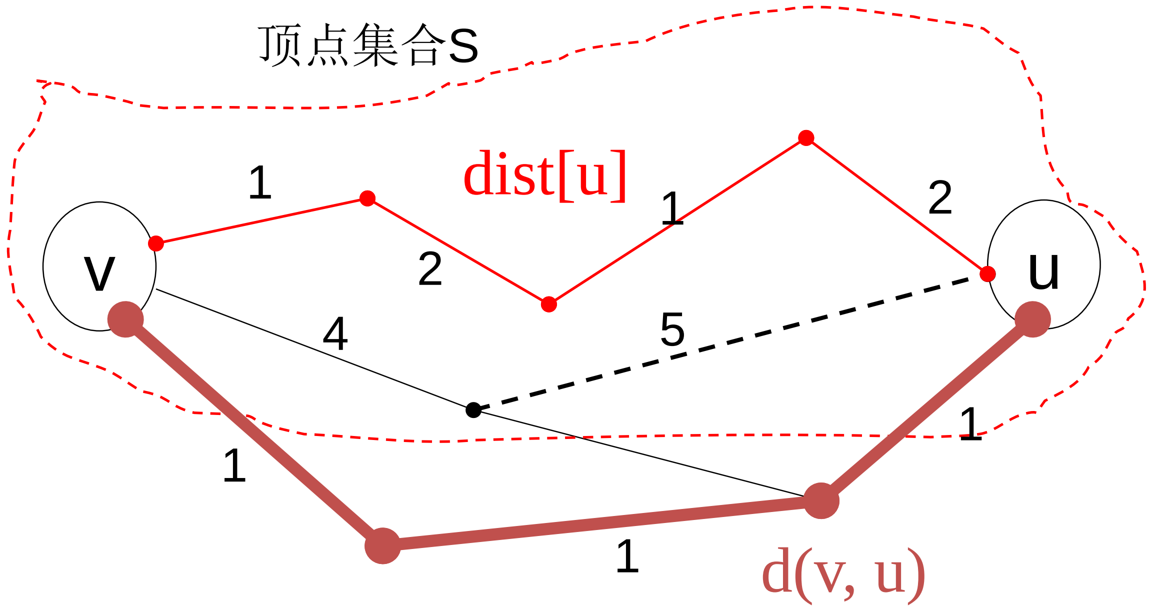
2. 设u是G的某一个顶点，

1）从源v到u的**全局最短路径**长度为d(v, u)，此最短路径有可能经过不在S中的顶点

2）从源v到u的、且**中间只经过S中顶点的路**称为从源到u的特殊路径。

用数组dist[u]记录当前S中每个顶点u所对应的最短特殊路径长度。

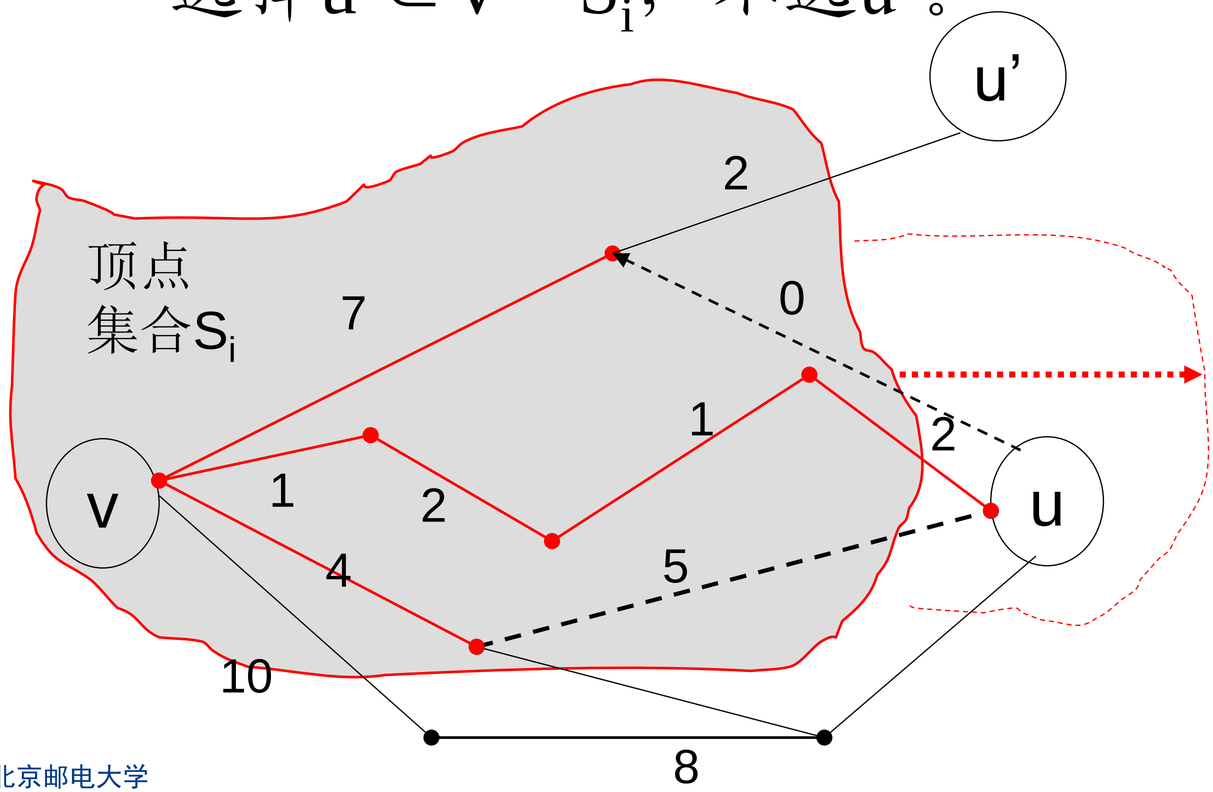
因此，$d(v,u) \le dist[u]$



3. 贪心策略:

每次从V-S中取出具有(只经过S中顶点)的最短特殊长度dist[u]的顶点u，将u添加到S中， 同时对数组dist作必要的修改

e.g. 下图中， 对S_i外的顶点u、u’，从当前顶点集合S_i，
选择$u \in V - S_i$， 不选u’。



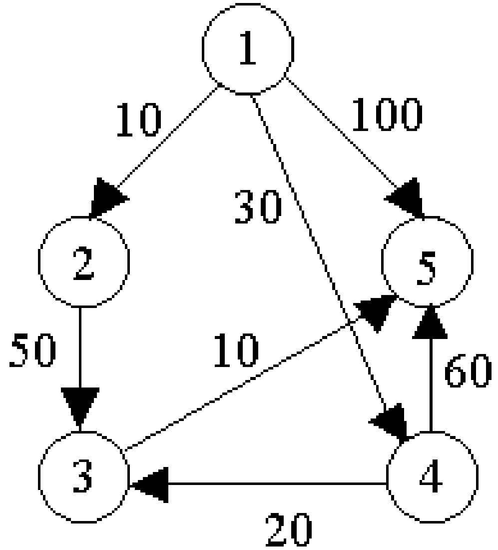
扩展后的顶点集合

$S_{i+1} = S_i \cup \{u\}$

4. 当集合S包含V中所有顶点，dist[u]就记录了从源v到所有其它顶点u之间的最短路径长度

实例

E.g. 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。



Dijkstra算法的迭代过程:

算法特点：迭代过程中，每个节点u的dist[u]值是非递增的

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

计算复杂性

用带权邻接矩阵表示具有n个顶点和e条边的带权有向图G(V, E).

Dijkstra算法的主循环体需要O(n) 时间。这个循环需要执行n-1次，所以完成循环需要$O(n^2)$时间。

算法的其余部分所需要时间不超过 $O(n^2)$

算法正确性——问题及其子问题描述!

对图G(V, E), 源点v，问题从以下三方面描述:

1）源点v，图中顶点集合V, 算法迭代过程中保持不变

2）用于构造最短路径的当前顶点集合S_i，不断增加，定义不同规模的子问题

3）指标：相对于现有S_i，对各顶点u，dist[u]

不同的S_i，定义了不同规模的子问题;

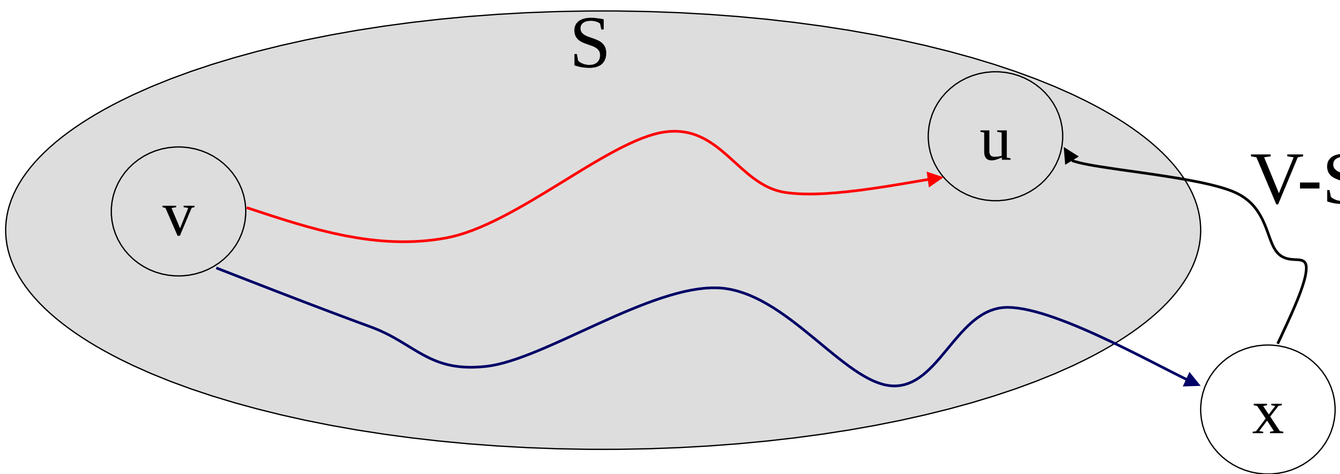
当S_i=V时，算法迭代结束，最短路径考虑了图中全部顶点

S_i越小，问题越简单，自下而上求解

算法正确性——贪心选择性质

贪心选择策略：在每步迭代时，从V-S中选择具有最短特殊路径dist[u]的顶点u，加入S

贪心策略正确性：需证明对顶点u，从v开始、经过G中任意顶点到达u的全局最短路径的长度d(v, u) = 从v开始、只经过S中顶点到达u的最短路径的长度dist(u), 即不存在另一条v到u的最短路径，该路径上某些节点x在V-S 中，且该路径的长度d(v,u)<dist[u].

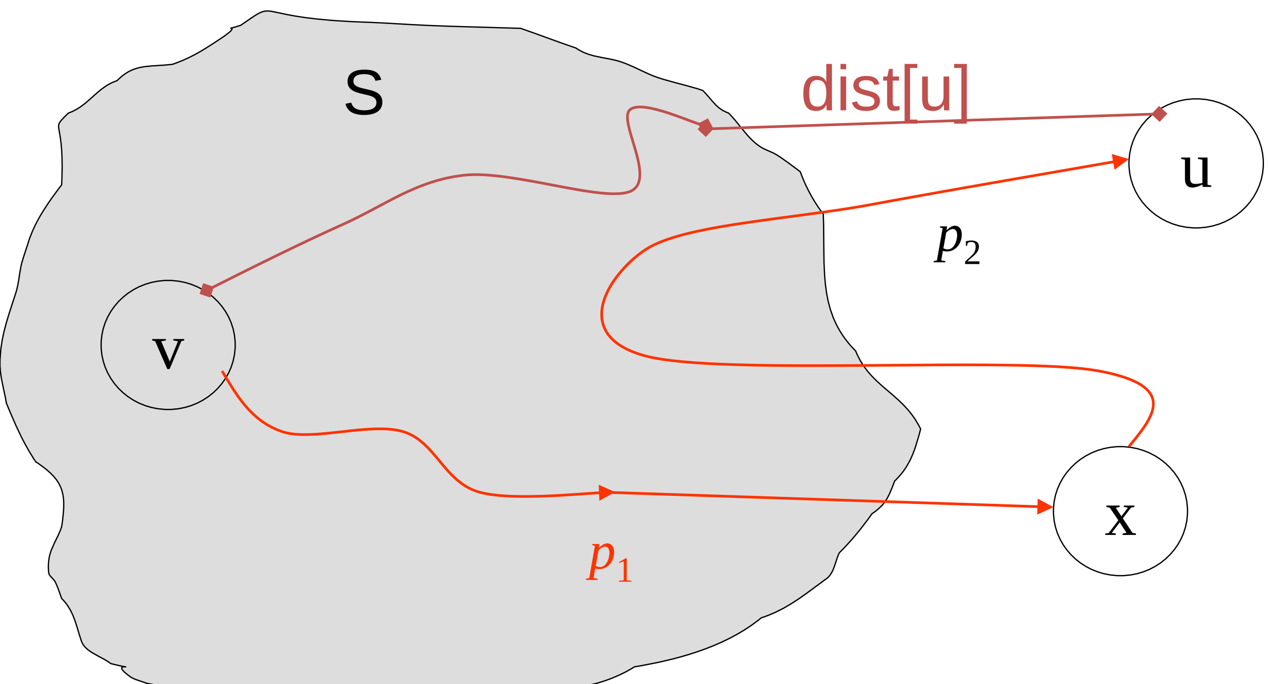


反证法

假设:

（1）在迭代求解过程中，顶点u是遇到的第1个满足: $d(v,u) < dist[u]$的顶点

（2）从v到u的全局最短路径上，第1个属于V-S的顶点为x



u到v的全局最短路径$d(v,u)$由2段组成:

(1) $p_1 = dist[x]=d(v,x)$, $v \longrightarrow x$

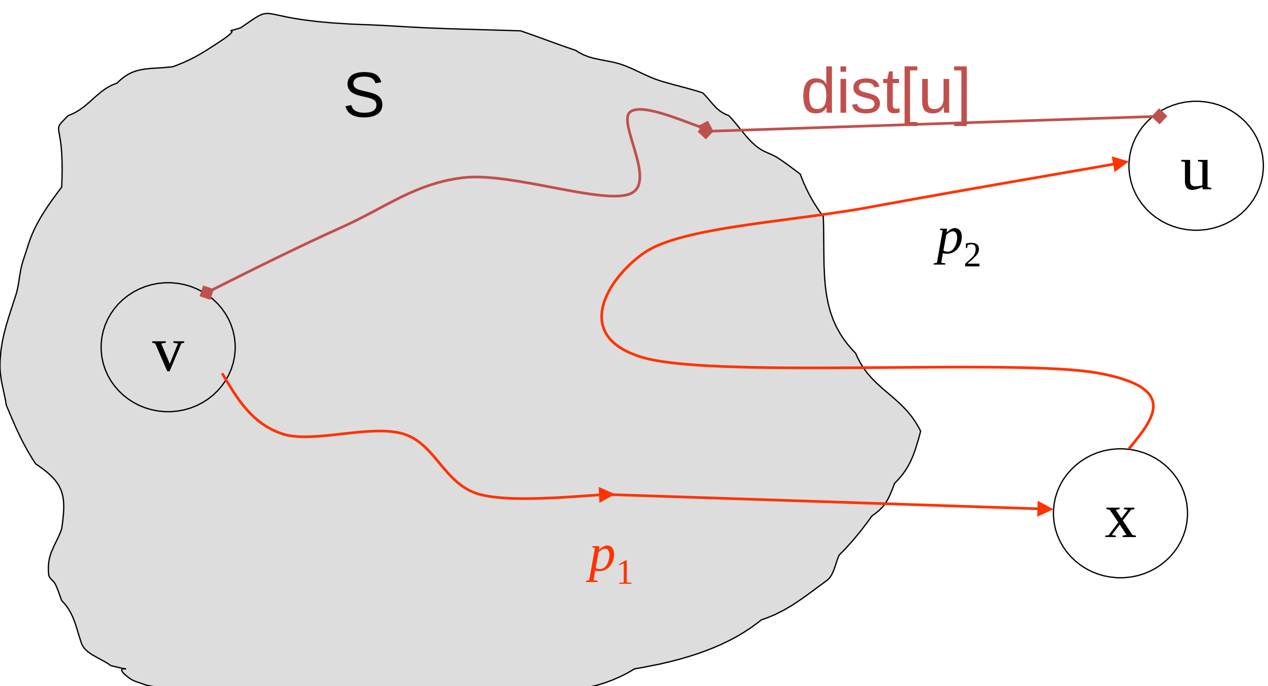
(2) p_2 $x \longrightarrow u$

首先，因为u是第一个满足全局最短路径不完全在S集合中的顶点，即

$$d(v, u) < dist[u],$$

而x是在u之前遇到的顶点，x的最短路径完全在S中，因此，

$$dist[x] = d(v,x) \le d(v, u)$$



u到v的全局最短路径$d(v,u)$由2段组成：

(1) $p_1 = dist[x]=d(v,x)$, $v \rightarrow x$

(2) p_2 $x \rightarrow u$

对v到u的全局最短路径，有

d(v, x) + distance(x, u) = $\underline{\text{d(v ,u) < dist[u]}}$

根据假设

由于distance(x, u) >0, 因此

$\underline{\text{dist[x]= d(v, x)}}$ < d(v ,u) < dist[u], 即

X仍然满足贪心性质

dist[x]< dist[u]

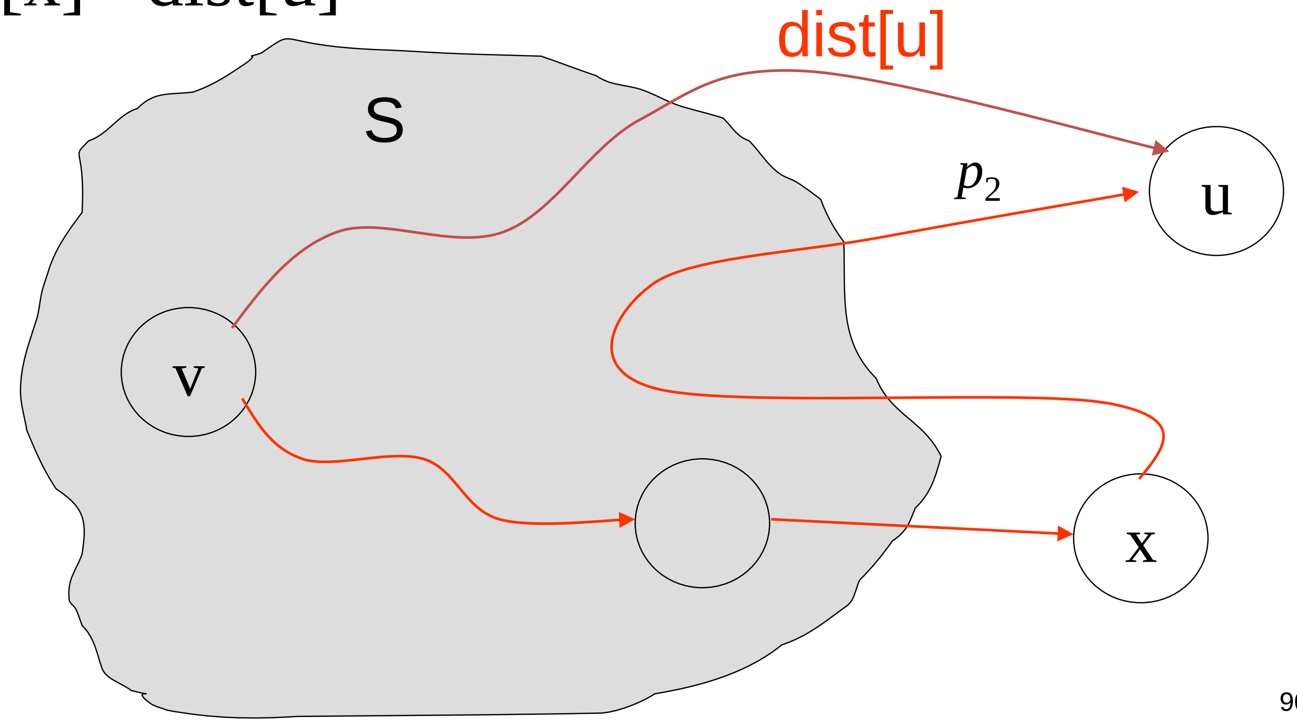
但是根据路径p构造方法，在下图所示情况下，*u*、*x*都在集合S之外，即*u*、*x*都属于V-S，但*u*被选中时，并没有选x，根据扩展S的原则——选dist最小的顶点加入S，说明此时:

$$dist[u] \le dist[x]$$

这与前面推出的

$$dist[x] < dist[u]$$

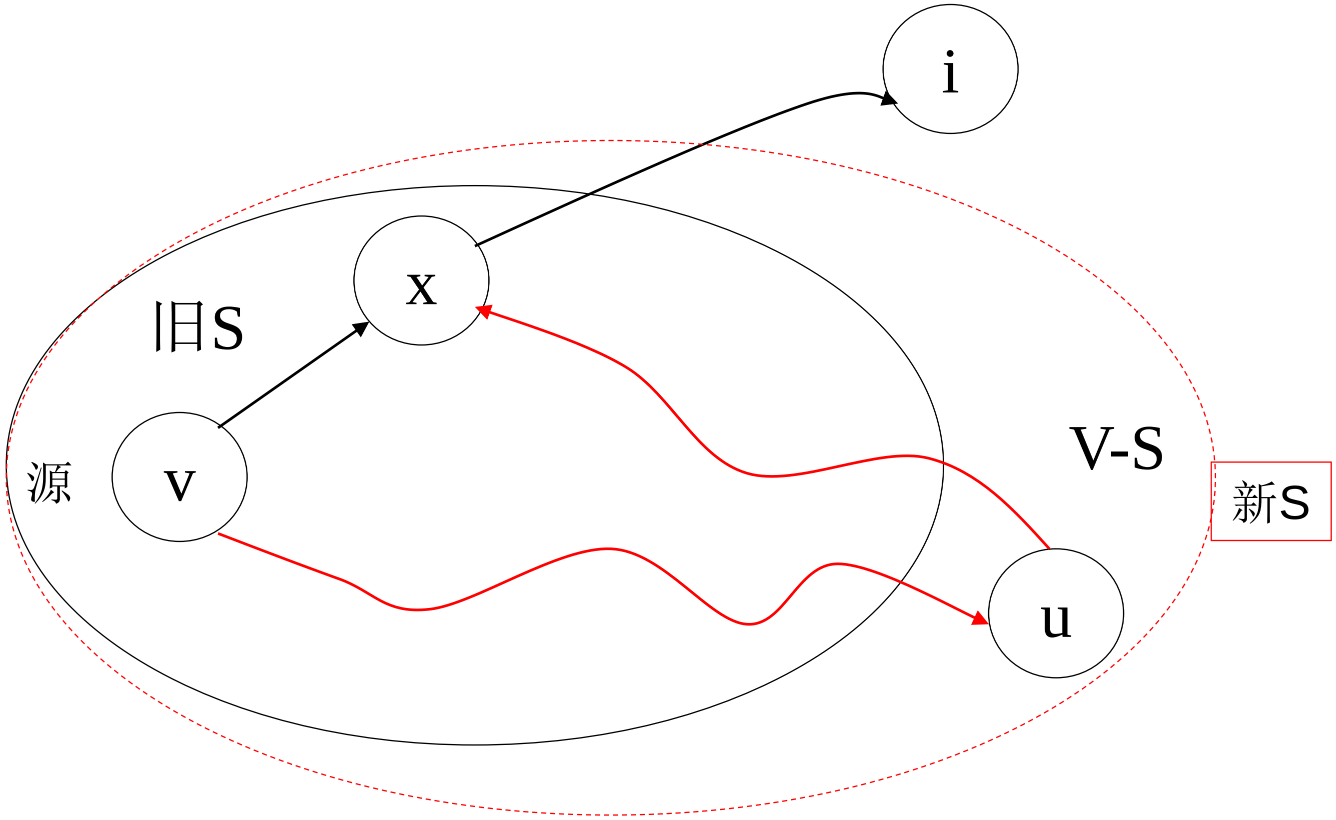
相矛盾。



最优子结构性质

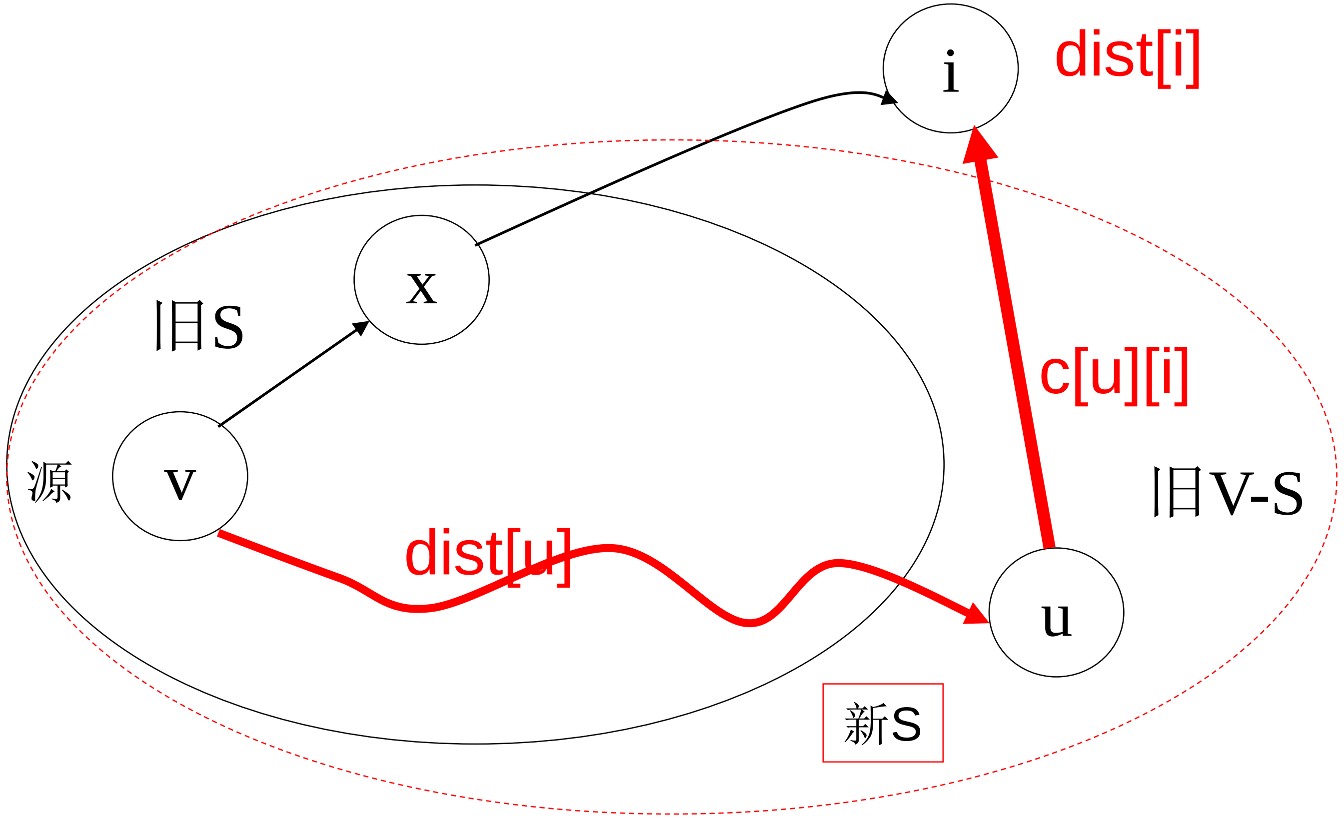
对顶点u，考察将u加到S之前和之后，dist[u]的变化，添加u之前的S称为旧S，加入u之后的S称为新S。

要求：u加到S中后，dist[u]不变化。



对另外1个节点i，考察u的加入对dist[i]的影响：

1）假设添加u后，出现1条从v到i的新路，该路径先由v经过旧S中的顶点到达u，再从u经过一条直接边到达i

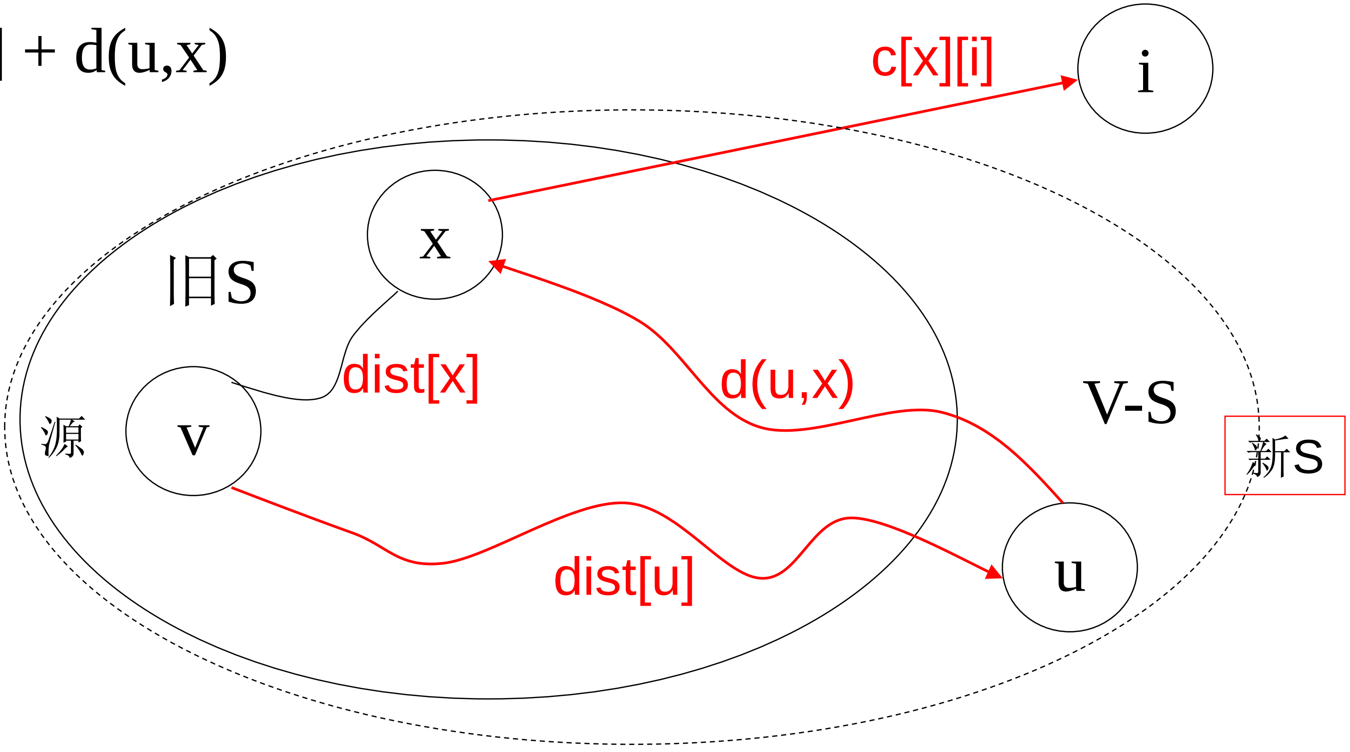


该路径的最短长度=dist[u] + c[u][i]

如果dist[u] + c[u][i] <原来的dist[i]，则算法用dist[u] + c[u][i]替代dist[i]，得到新的dist[i]。否则， dist[i]不更新。

2）如果新路径如下图所示，先经过u，再回到S中的x，由x直接到达i。x处于老的S中， 故dist[x]已经是最短路径，x比u先加入S，因此

$$\text{dist}[x] \leq \text{dist}[u] + d(u,x)$$



此时，从原v到i的最短路径dist[i]小于路径（v, u, x, i）的长度，因此算法更新dist[i]时不需要考虑该路径，u的加入对dist[i]无影响。

因此，无论算法中dist[u]的值是否变化，它总是关于当前顶点集合S到顶点u的最短路径。

也就是说：对于加入u之前、之后的新旧S所对应的2个子问题，算法执行过程保证了dist[u]始终是u的最优解

贪心算法的应用

最小生成树算法

最小生成树

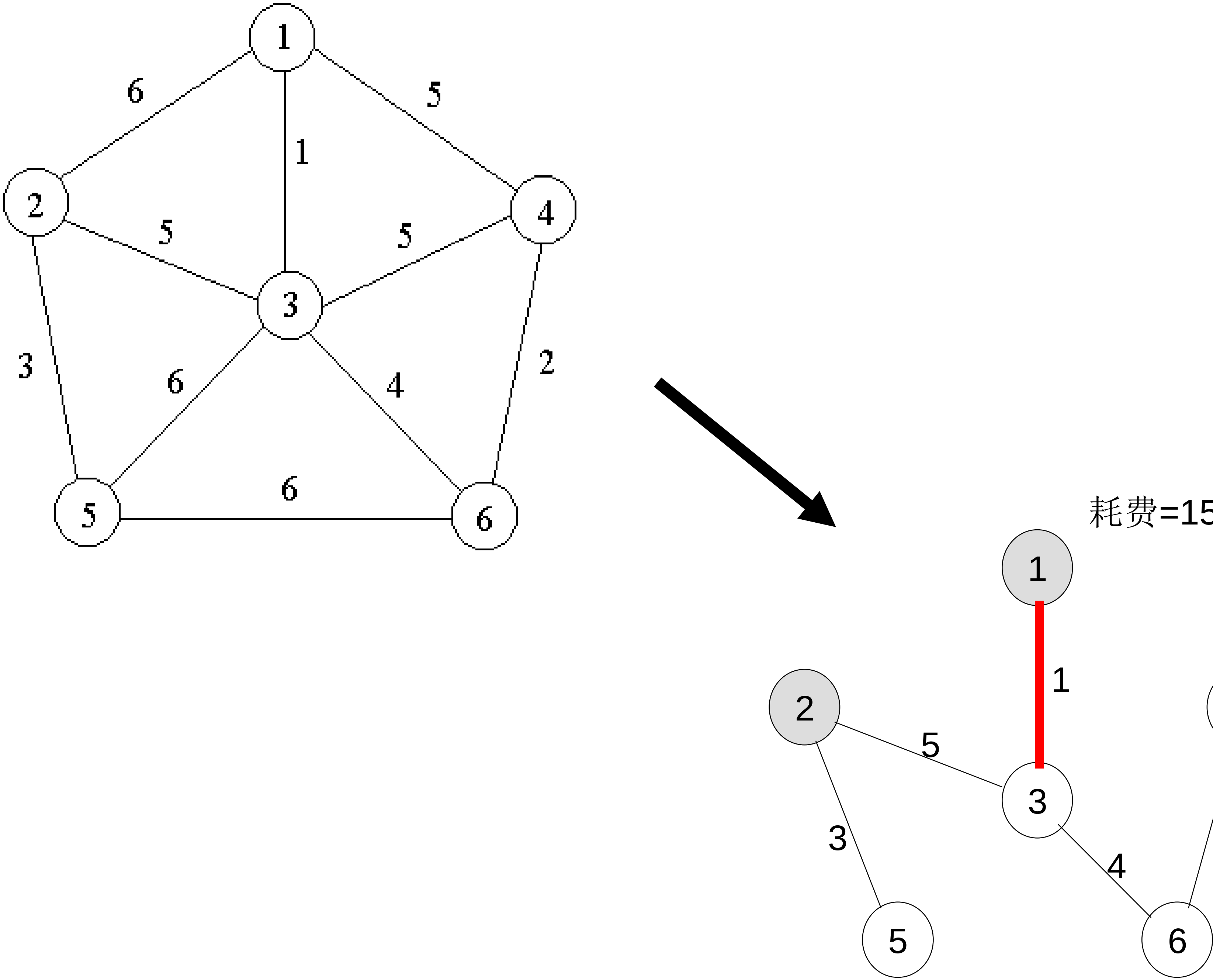
生成树: 设一个网络表示为无向连通带权图G =(V, E) , E中每条边(v,w)的权为c[v][w]。如果G的子图G'是一棵包含G的所有顶点的树，则称G'为G的生成树。

生成树的**成本/代价/耗费（cost）**： 生成树上各边权的总和。

G的**最小生成树**：在G的所有生成树中，耗费最小的生成树。

应用举例:

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，最小生成树给出建立通信网络的最经济方案。


耗费=15

最小生成树性质

基于贪心选择策略, 构造最小生成树算法

1） **Kruskal算法**

2） **Prim算法**

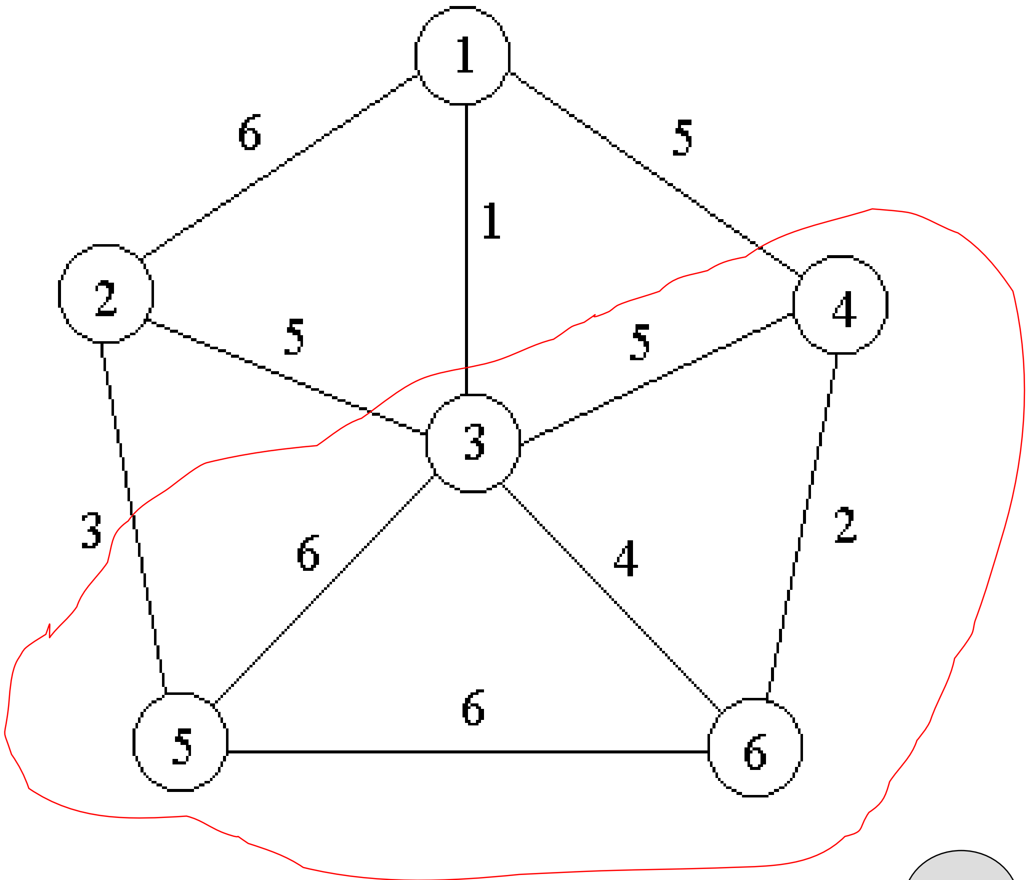
这2个算法贪心选择的方式不同， 都利用了**最小生成树(MST)性质**:

设G=(V, E)是连通带权图，顶点集U是V的真子集。 如果:

1）(u,v)∈E为横跨点集U和V-U的边，即u∈U，v∈V-U，并且

2）在所有这样的边中，(u,v)的权c[u][v]最小, 则一定存在G的一棵最小生成树，它以(u,v)为其中一条边，即(u,v)出现在最小生成树中

说明：真子集U可以任意选取

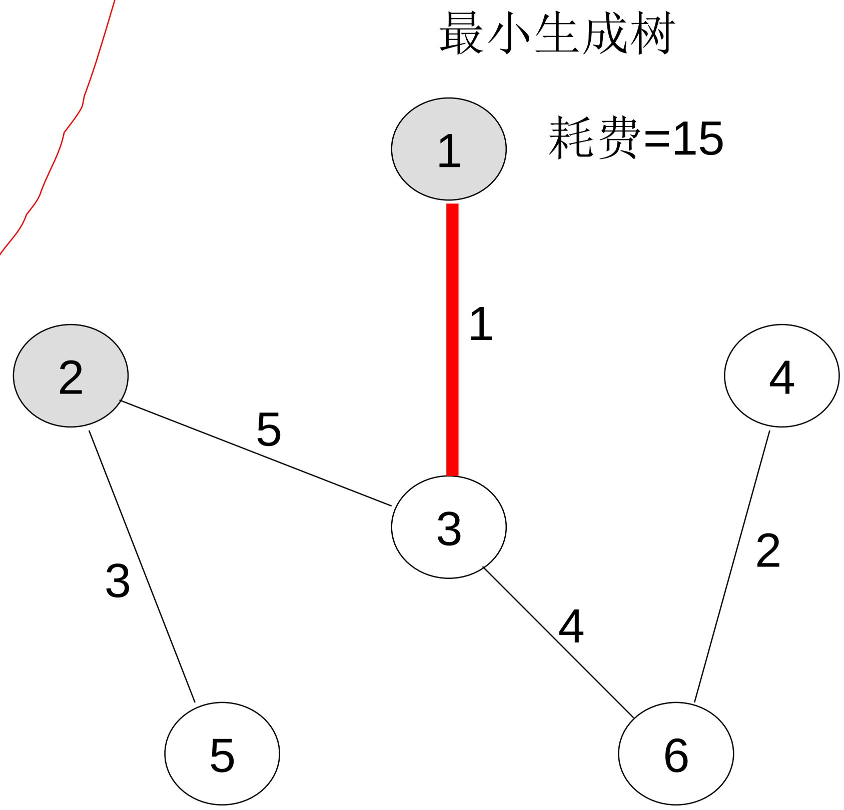


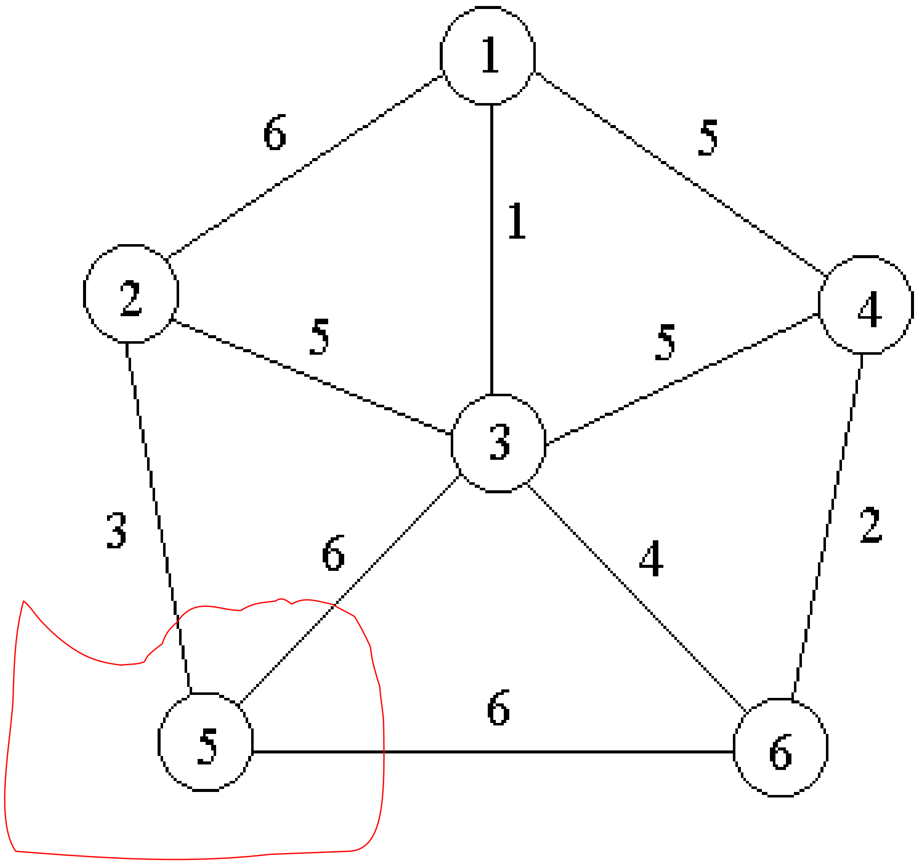
U={3, 4, 5, 6},

V-U={1,2}

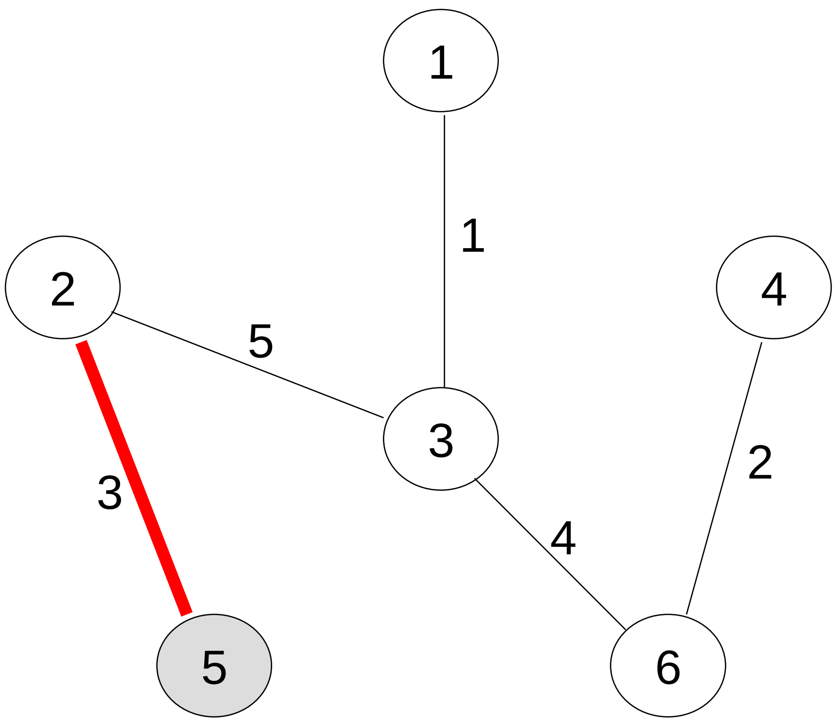
边集={<2,5>, <2,3>,

<1,3>, <1,4>}





耗费=15



U={5}，V-U={1,2,3,4,6}

边集={ <5,2>，<5,3>，<5.6>}

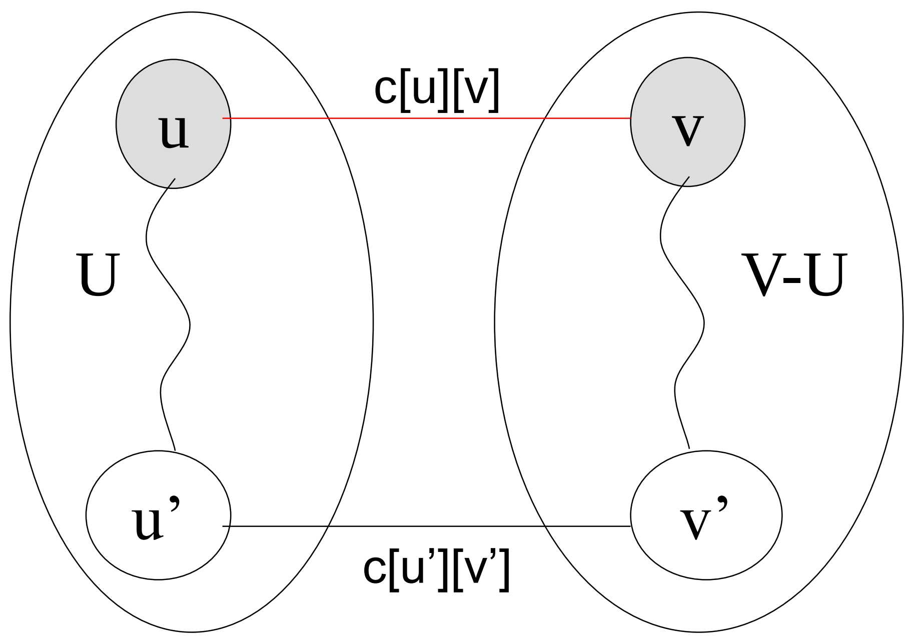
MST性质证明

反证法:

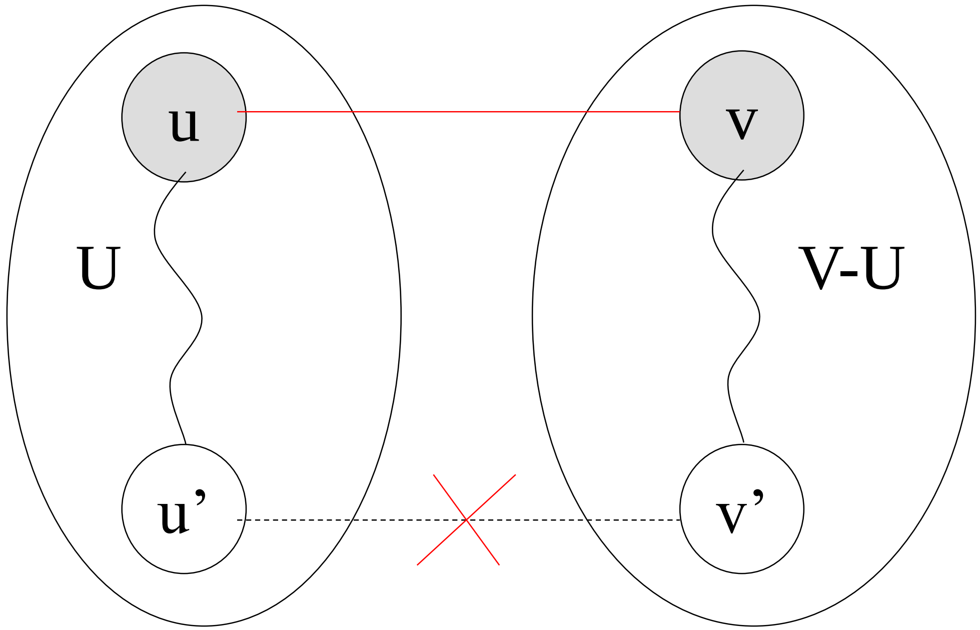
假设对G的任意一个最小生成树T，针对点集U和V-U，(u,v)∈E为横跨这2个点集的最小权边， T不包含该最小权边<u, v>，但T包括节点u和v。

将<u, v>添加到树T中， 树T将变为**含回路的子图**，并且该回路上有一条不同于<u, v> 的边<u', v'>, u' ∈U, v' ∈V-U

最小生成树T,
图T∪ {<u,v>},
c[u][v] ≤c[u'][v']



生成树T’,



将<u’, v’>删去，得到另一个树T’，即树T’是通过将T中的边<u’, v’>替换为<u,v> 得到的。

由于这2条边的耗费满足c[u][v] ≤c[u’][v’]，因此用较小耗费的边<u,v>替换后得到的树T’的耗费更小，即:

T’耗费≤ T的耗费

这与T是任意最小生成树的假设相矛盾

Prim算法

设G=(V, E)是连通带权图，V={1,2,…,n}。

Prim算法的基本原理:

Step1. 首先置顶点集合S={1}

Step2. 当S是V的真子集时，作如下的**贪心选择**:

选取满足条件i∈S，j∈V-S，且c[i][j]最小的边<i, j>，将顶点j添加到S中，边<i, j>加到边集T中。

Step3. 重复上述过程，直到S=V为止，此时边集T就是最小生成树

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**。

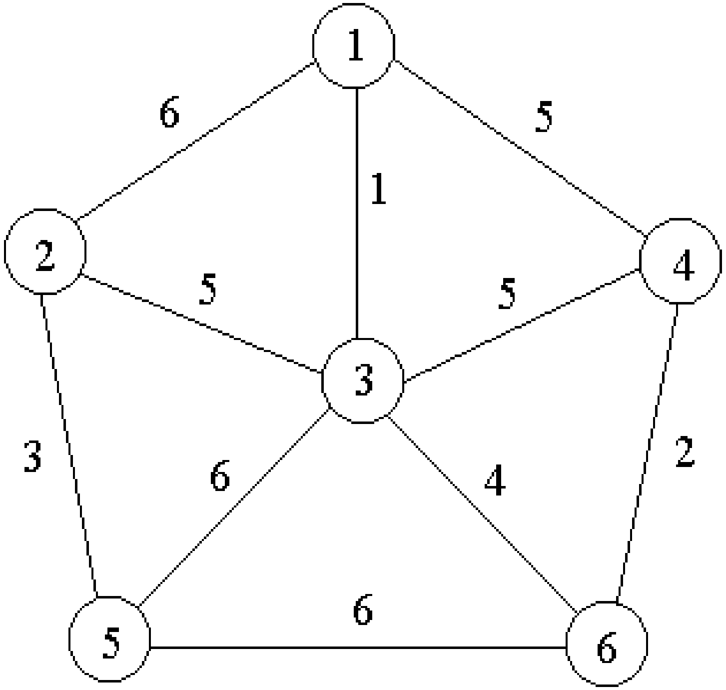
算法复杂性：$O(n^2)$

Prim算法正确性说明

利用最小生成树性质和数学归纳法（对顶点集合S归纳）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**:

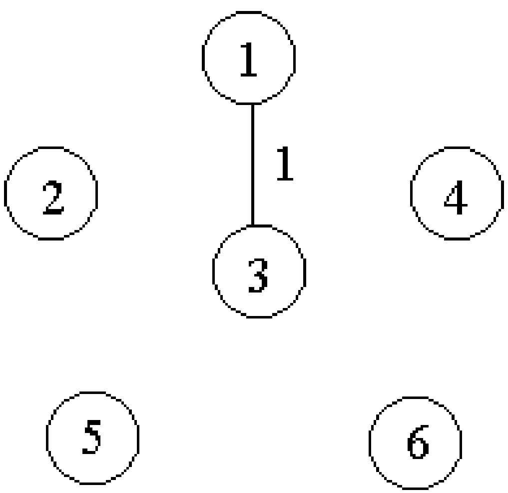
贪心选择性质、最优子结构性质

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。



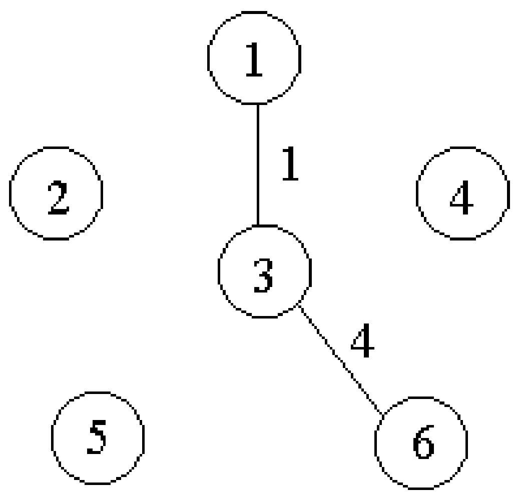
例如，对于左图中的带权图，按Prim算法选取边的过程如下页图所示。

S={1,3}



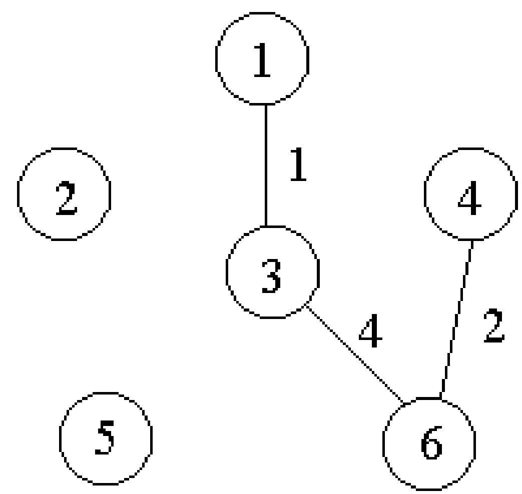
(a)

S={1,3,6}

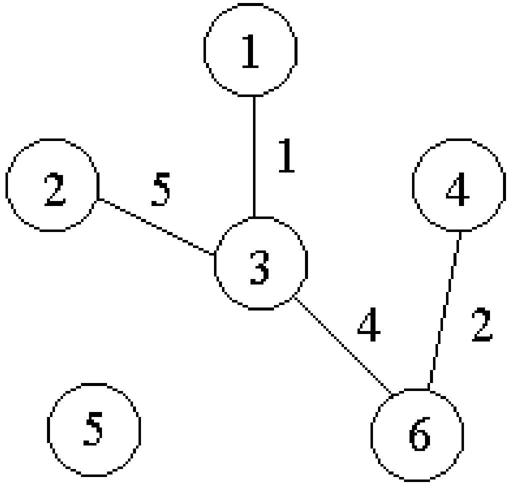


(b)

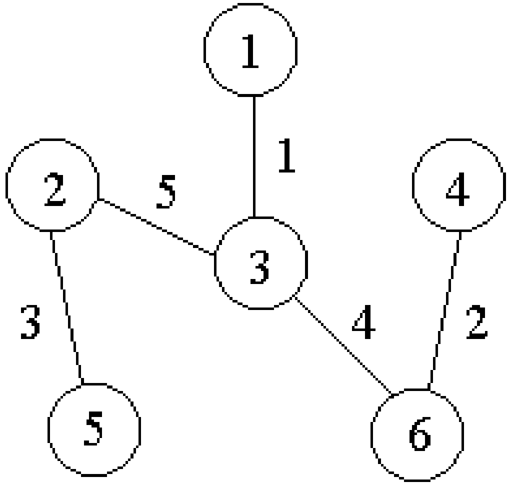
S={1,3,6,4}



(c)



(d)



(e)

S={1,3,6,4,2}

S={1,3,6,4,2, 5}

实现Prim算法时，应当考虑**如何有效地找出满足条件i∈S, j∈V-S，且权c[i][j]最小的边(i,j)**。

方法：设置2个数组closest和lowcost。

closest[j]: 记录j在S中与其最近的邻接顶点, j∈ $V-S$。

lowcost[j]: 记录j与S中的邻接顶点的最小权重。

在Prim算法执行过程中，先找出V-S中使lowcost值最小的顶点j，然后根据数组closest选取边(j, 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算法 -- 基本原理

Step1. 将G的n个顶点看成n个孤立的连通分支

Step2. 将所有的边按权从小到大排序

Step3. 从权最小的第一条边开始，依边权递增的顺序查看每一条边<v,w>，并按下述方法连接2个不同的连通分支：

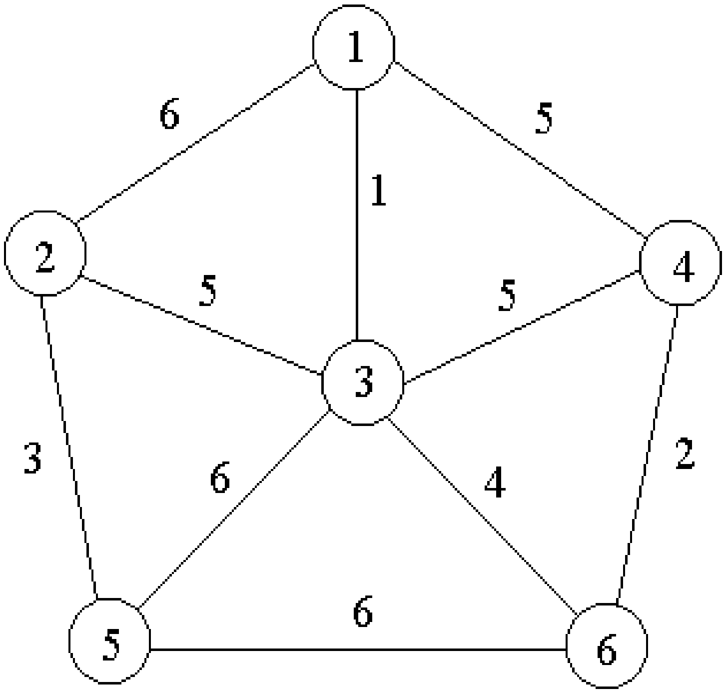
当查看到第k条边(v,w)时，

1) 如果端点v和w分别是当前2个不同的连通分支T1和T2中的顶点时，用边(v,w)将T1和T2合并成一个连通分支，然后继续查看后续第k+1条边

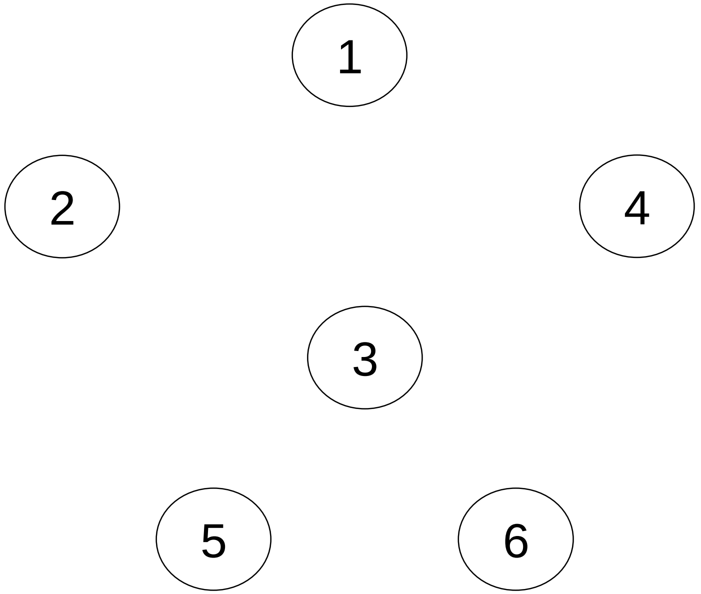
2) 如果端点v和w已经属于当前的同一个连通分支中，不允许将(v,w)加入，否则会产生回路。此时，直接再查看后续第k+1条边

Kruskal算法

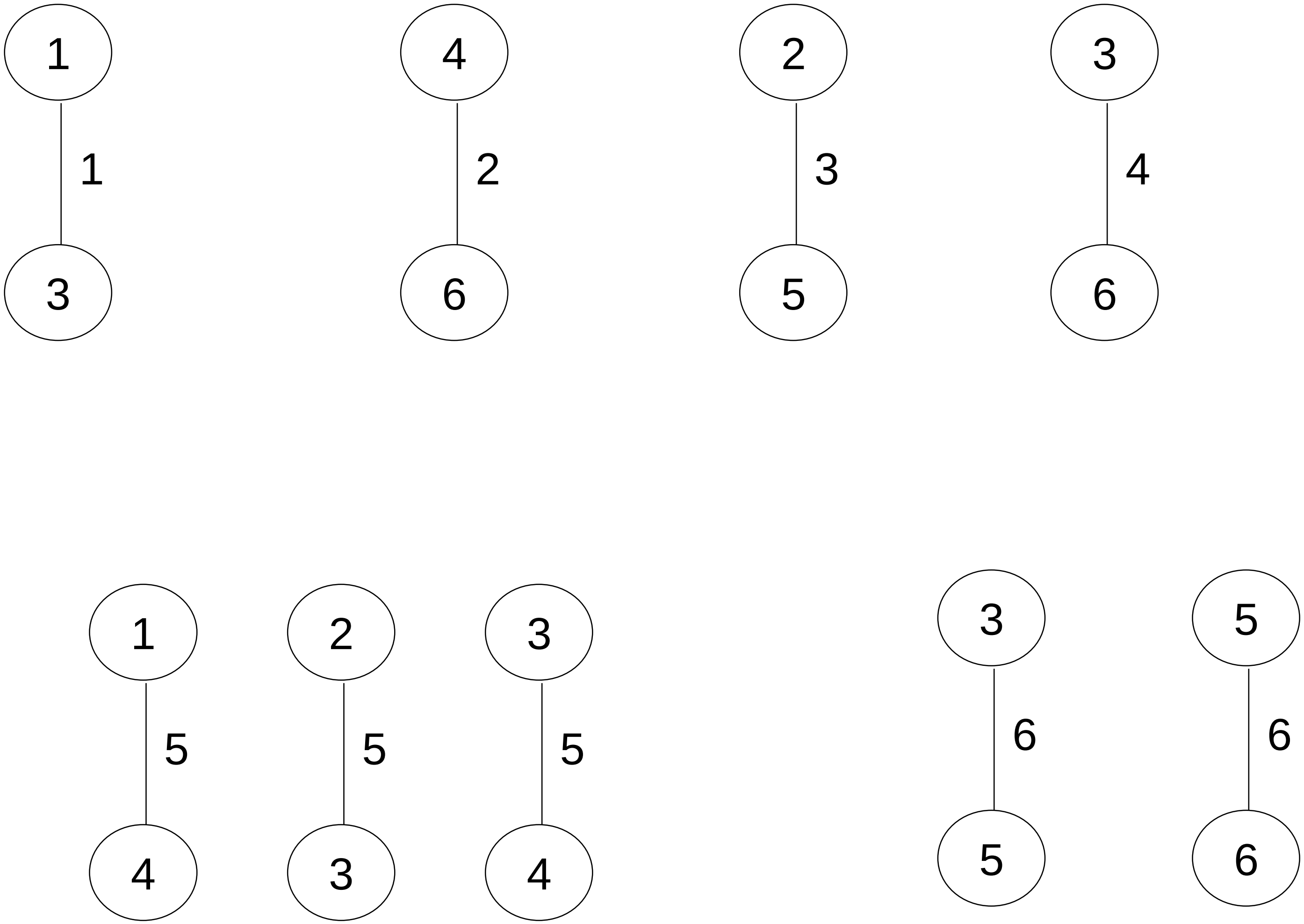
Step4. 持续上述过程，直到只剩下一个连通分支——最小生成树



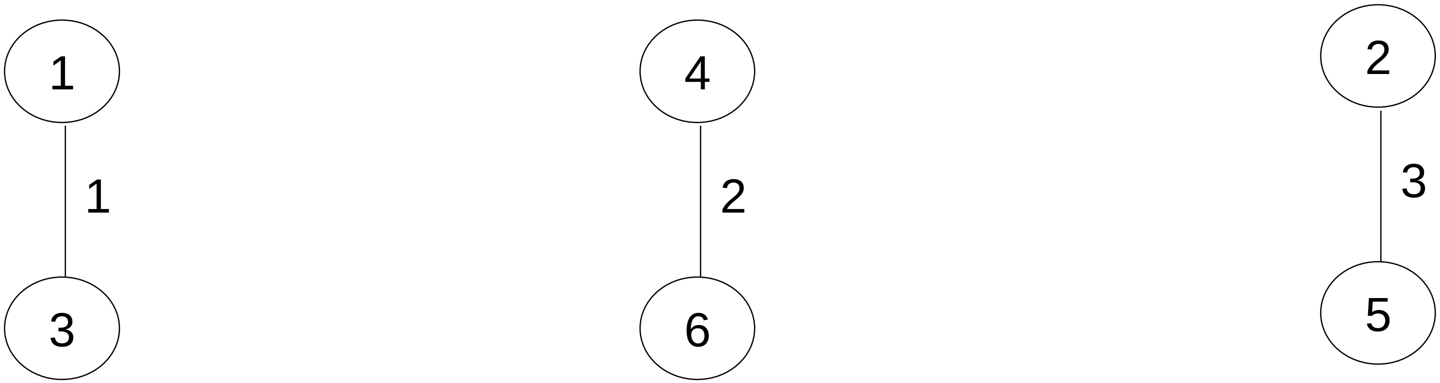
Step1.

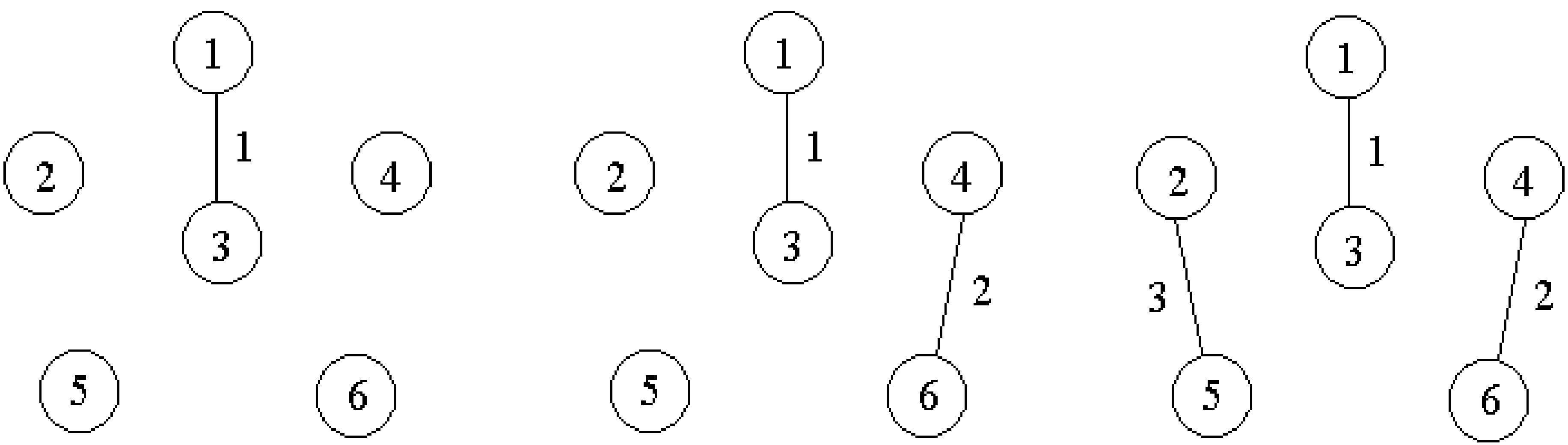


Step2. 边排序

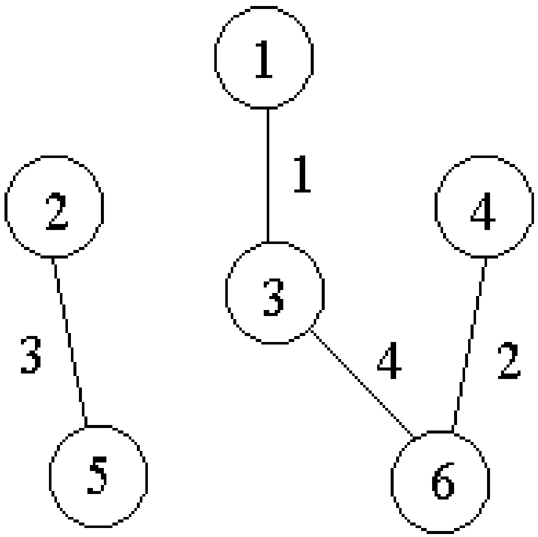


边排序：

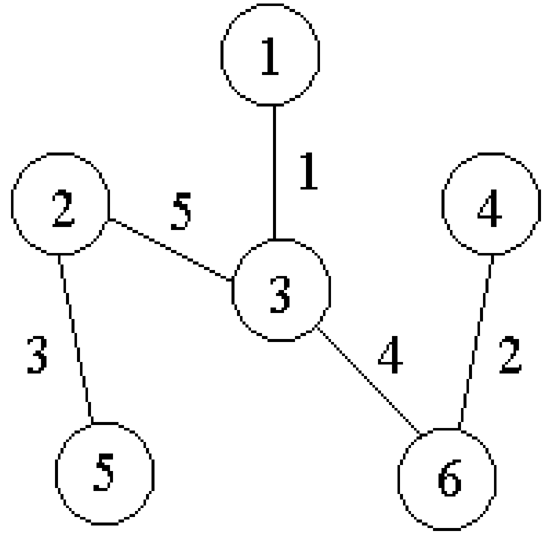




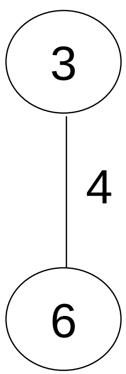
(a) (b) (c)

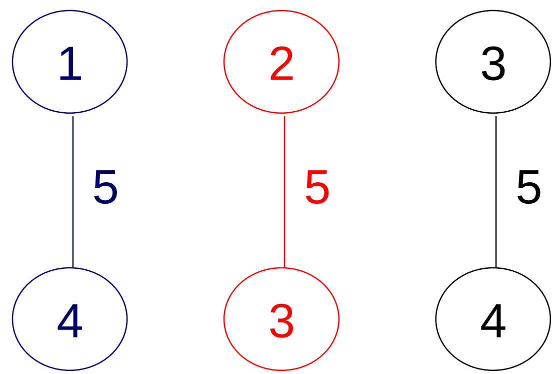


(d)



(e)





说明：
1）节点1与4、3与4已经在同一连通分量中
2）节点2、3分属于不同连通分量

Kruskal算法时间复杂性

当图的边数为e时，Kruskal算法所需的**计算时间**

$$O(e\log e)$$

与Prim算法比较

1. 当 $e=\Omega(n^2)$ 时，Kruskal算法比Prim算法差，
2. 当 $e=o(n^2)$ 时，Kruskal算法比Prim算法好得多