第6章 分枝-限界法

6.1 分支限界法的基本思想

分支限界法与回溯法

（1）求解目标：回溯法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出在某种意义下的最优解。

（2）搜索方式的不同：回溯法以深度优先的方式搜索解空间树，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树。

6.1 分支限界法的基本思想

分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。

在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。

此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

6.1 分支限界法的基本思想

两类常用的方法选择下一个E-结点：

（1）先进先出(FIFO)：从活结点表中取出结点的顺序与加入结点的顺序相同。

后进先出(LIFO)：从活结点表中取出结点的顺序与加入结点的顺序相反。

（2）优先队列式分支限界法

按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

一 基本思想

1. 在e结点估算沿着它的各儿子结点搜索时，目标函数可能取得的“界”；
2. 把儿子结点和目标函数可能取得的“界”，保存在优先队列或堆中;
3. 从队列或堆中选取“界”最大或最小的结点向下搜索，直到叶子结点;
4. 若叶子结点的目标函数的值，是结点表中的最大值或最小值，则沿叶子结点到根结点的路径所确定的解，就是问题的最优解，否则转 3 继续搜索。

堆（Heap）

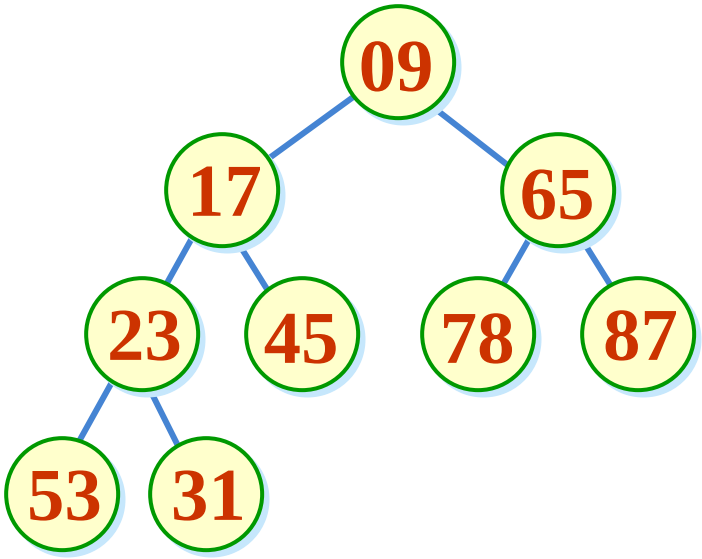
优先级队列

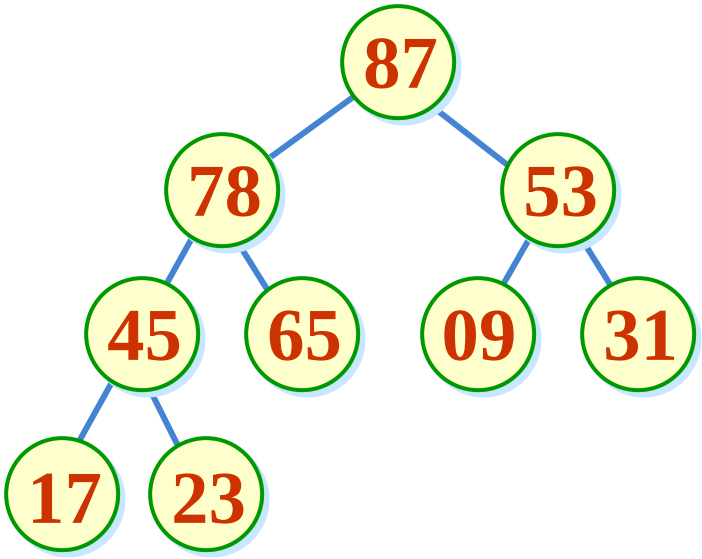
每次出队列的是优先权最高的元素

优先级队列支持以下操作：

找出一个具有最高优先级的元素
删除一个具有最高优先级的元素
添加一个元素到集合中

堆的定义





完全二叉树
数组表示

$K_i \le K_{2i+1}$ && $K_i \le K_{2i+2}$

9	17	65	23	45	78	87	53	31

完全二叉树
数组表示

$K_i \ge K_{2i+1}$ && $K_i \ge K_{2i+2}$

87	78	53	45	65	9	31	17	23

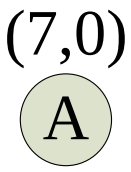
示例1：0-1背包问题

- 考虑实例n=4，w=[3,5,2,1]，v=[9,10,7,4]，C=7。
- 定义问题的解空间
  - 该实例的解空间为（x1,x2,x3,x4），xi=0或1(i=1,2,3,4)。
- 确定问题的解空间组织结构
  - 该实例的解空间是一棵子集树，深度为4。
- 搜索解空间
  - 约束条件 $\sum_{i=1}^{n} w_i x_i \le C$
  - 限界条件 cp+rp>bestp

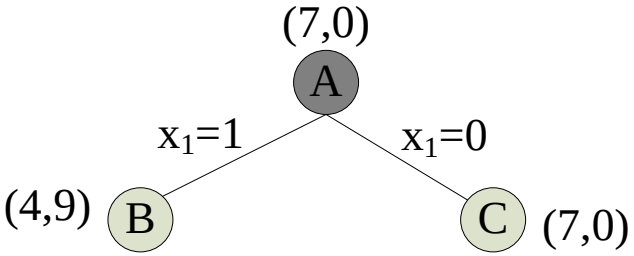
队列式分支限界法

n=4，w=[3,5,2,1]，v=[9,10,7,4]，C=7。

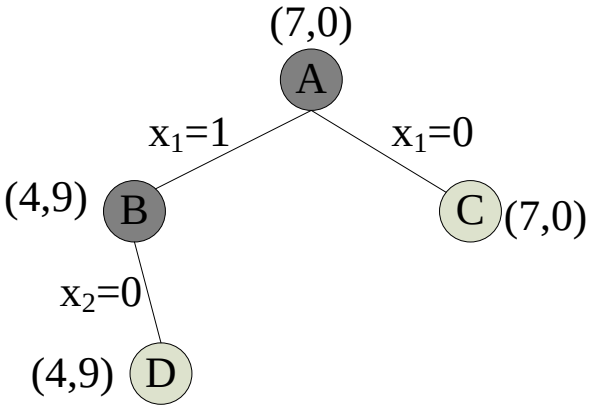
- **cp初始值为0；rp初始值为所有物品的价值之和；bestp表示当前最优解，初始值为0。**
- **当cp>bestp时，更新bestp为cp。**



队列=[A]
bestp=-1

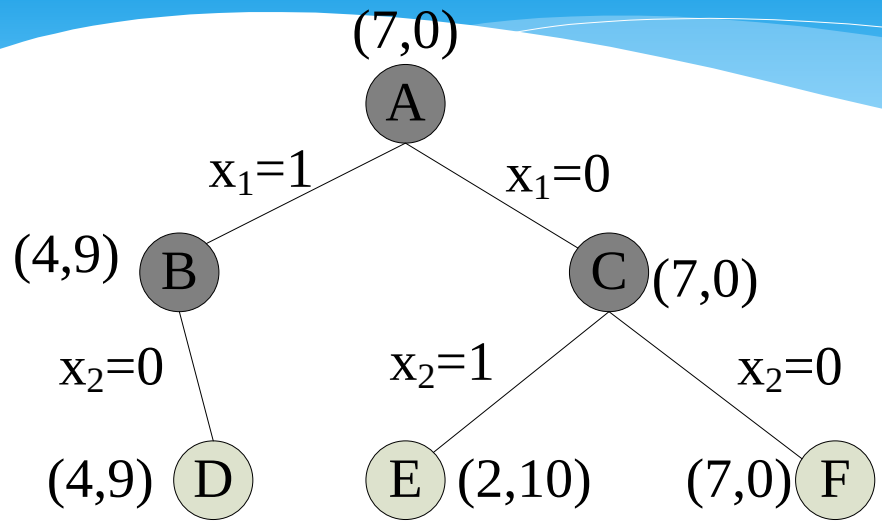


队列=[~~A~~，B，C]
bestp=9

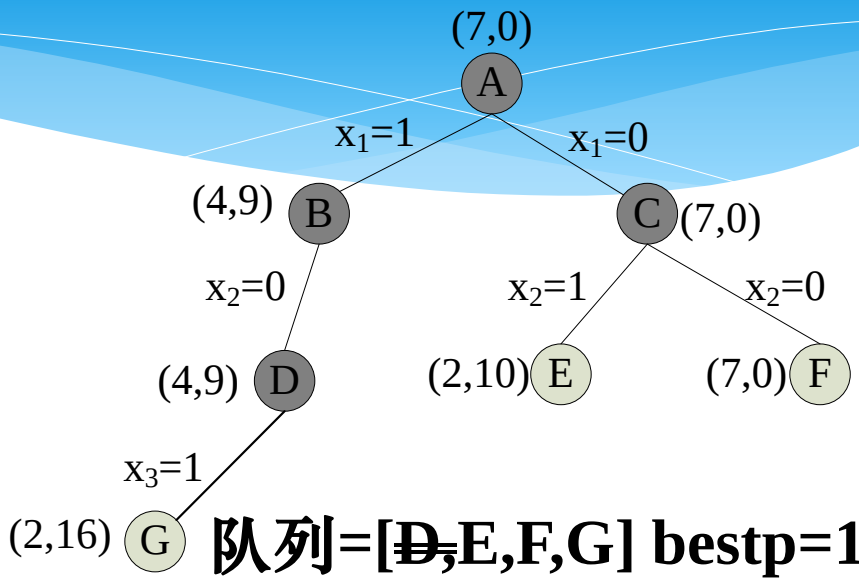


队列=[~~B~~，C，D]
bestp=9

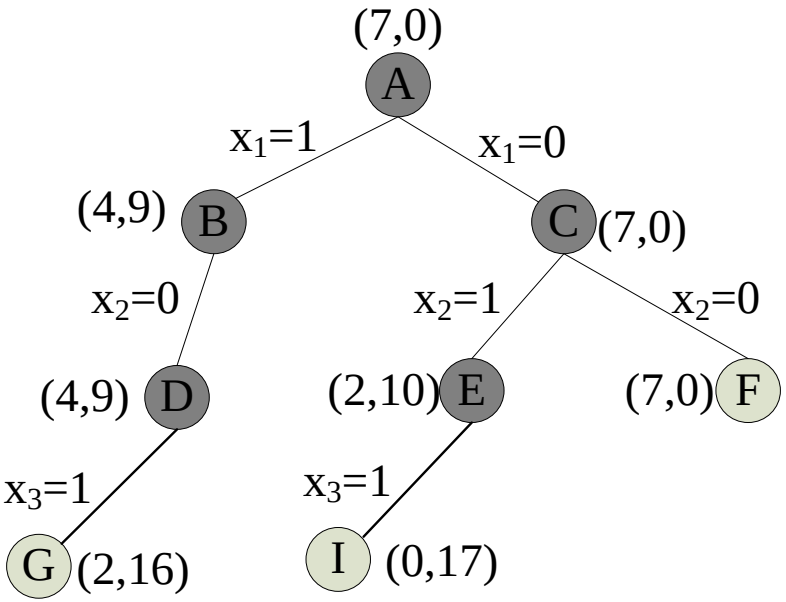
n=4，w=[3,5,2,1]，v=[9,10,7,4]，C=7。



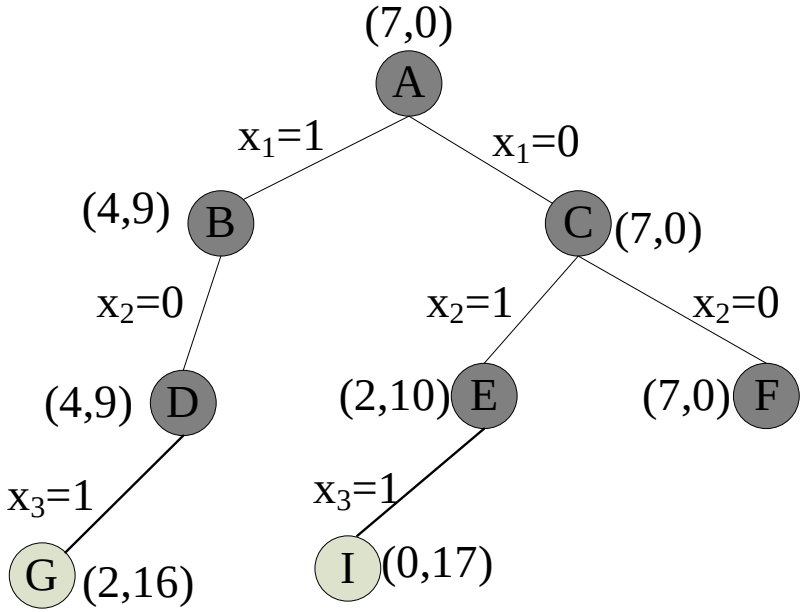
队列=[~~C~~,D,E,F] bestp=10



队列=[~~D~~,E,F,G] bestp=16

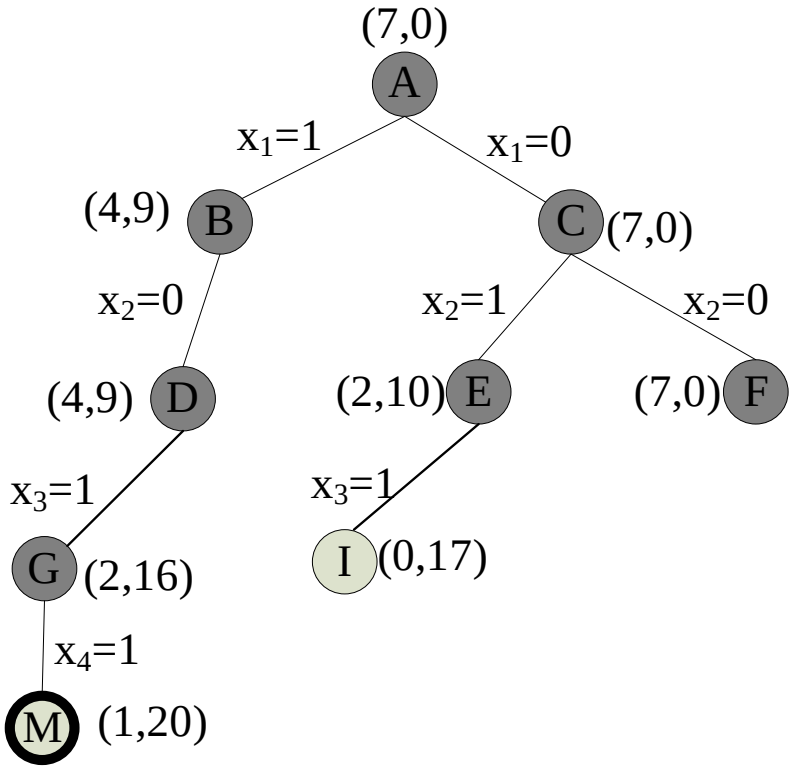


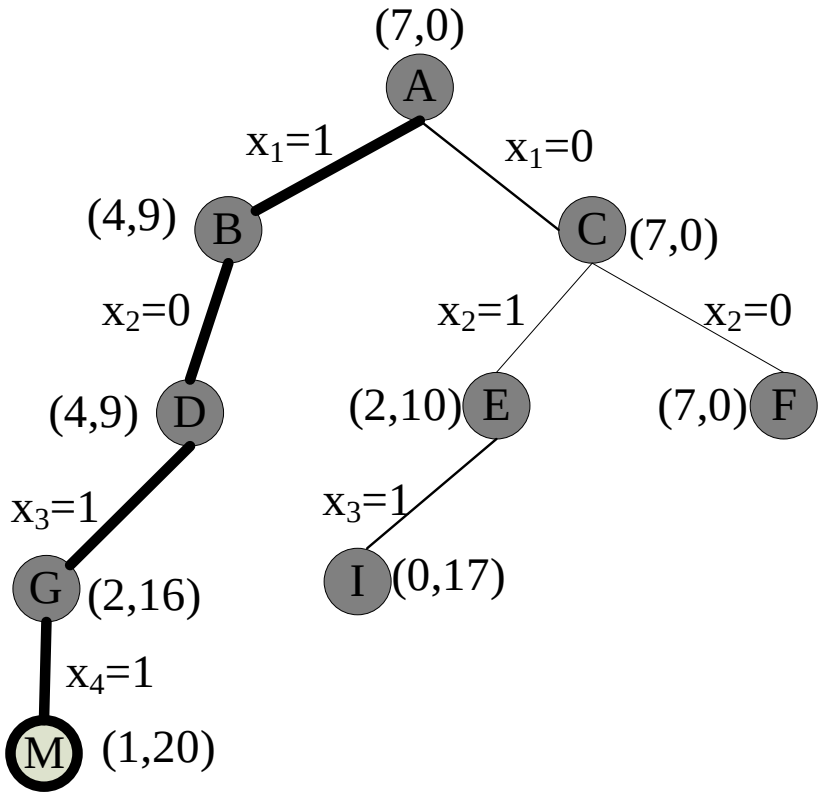
队列=[~~E~~,F,G,I] bestp=17



队列=[~~F~~,G,I] bestp=17

n=4，w=[3,5,2,1]，v=[9,10,7,4]，C=7。





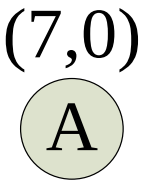
队列=[~~G~~,I, M] bestp=20

队列=[~~I~~,M] bestp=20

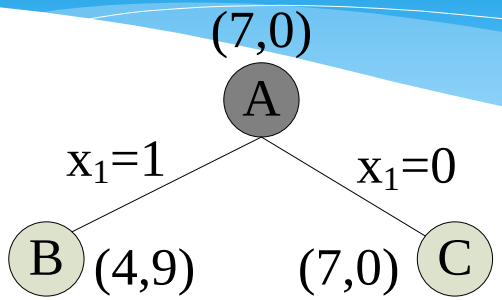
优先队列式分支限界法

* 优先级：活结点代表的部分解所描述的装入背包的物品价值上界，该价值上界越大，优先级越高。活结点的价值上界up=cp+rp。
* 约束条件：同队列式
* 限界条件：up=cp+rp>bestp。
* rp 剩余物品装满背包的价值

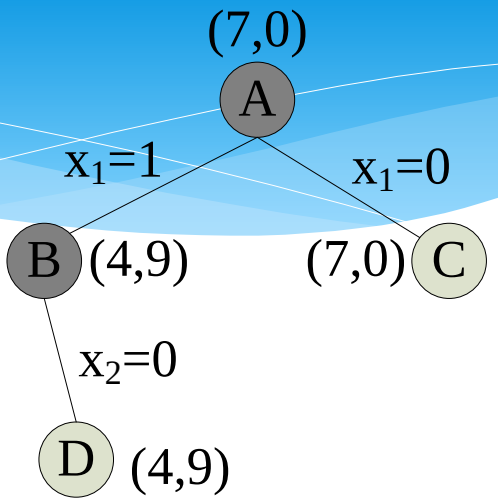
n=4，w=[3,5,2,1]，v=[9,10,7,4]，C=7。



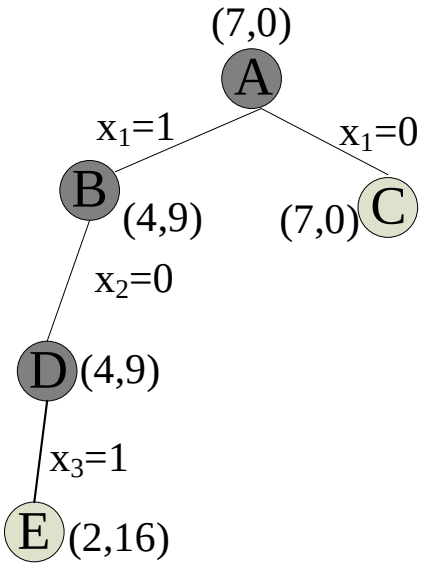
优先队列=[A]



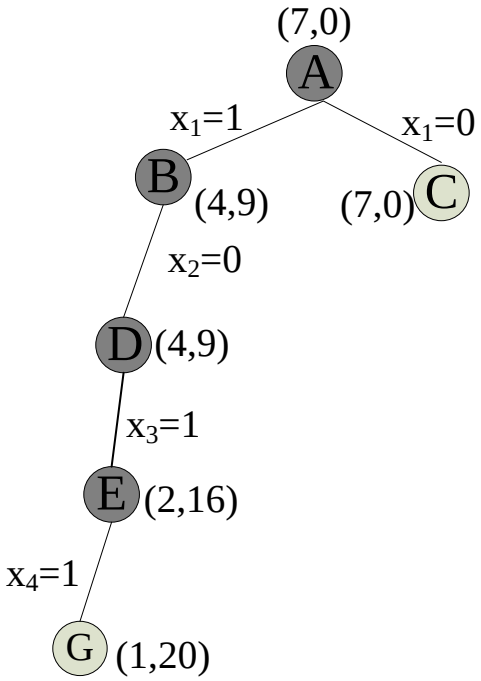
优先队列=[~~A~~,B,C]



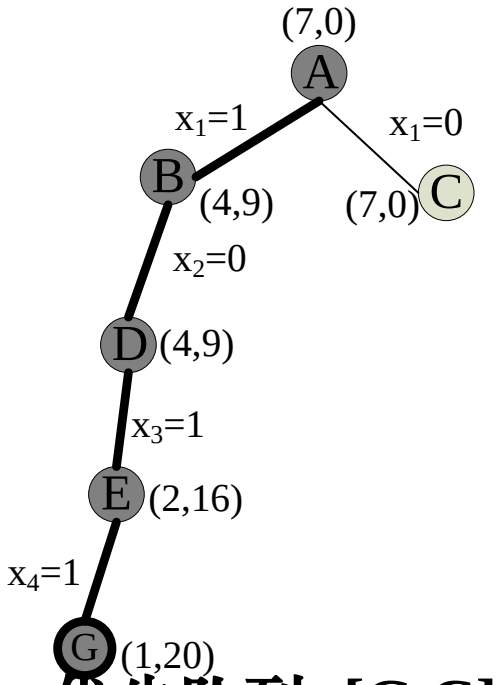
优先队列=[~~B~~,D,C]



优先队列=[~~D~~,E,C]

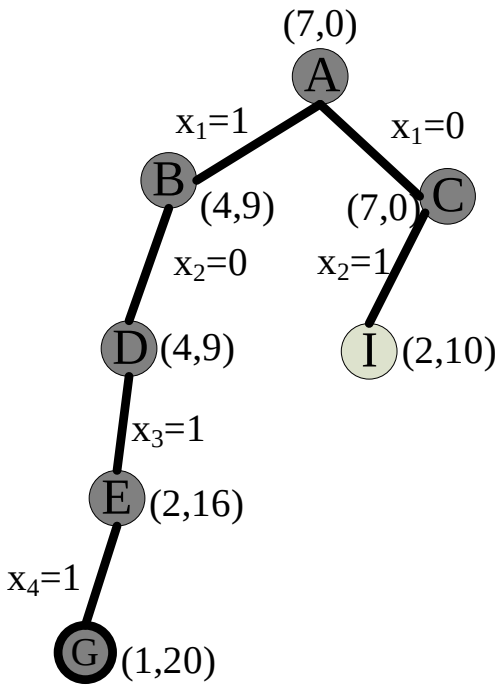


优先队列=[~~E~~,G,C]

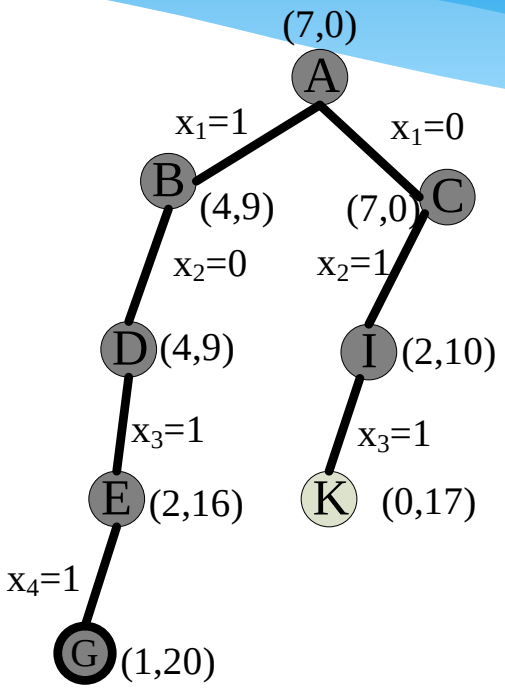


优先队列=[~~G~~,C]

n=4，w=[3,5,2,1]，v=[9,10,7,4]，C=7。



优先队列=[~~G~~,I]



优先队列=[~~I~~]

最优解：[x1,x2,x3,x4]=[1,0,1,1]

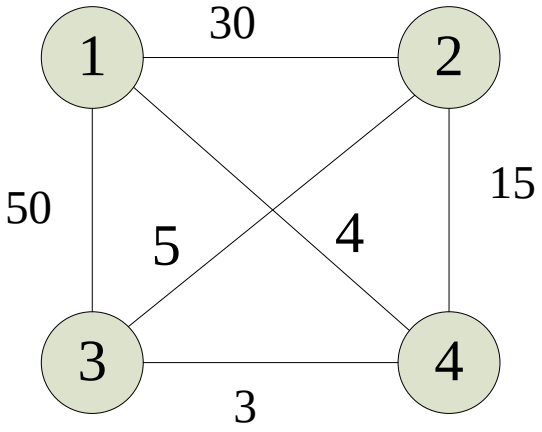
分支限界搜索过程算法描述

```
while (i != n+1)
{
        Typew wt = cw + w[i];
        if (wt <= c) {
                if (cp+p[i] > bestp)
                        bestp = cp+p[i];
                AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);
        }
        up = Bound(i+1);
        if (up > bestp) // 右子树可能含最优解
                AddLiveNode(up, cp, cw, false, i+1);
                 //取下一个扩展节点
}
```

示例2： 旅行售货员问题

问题描述

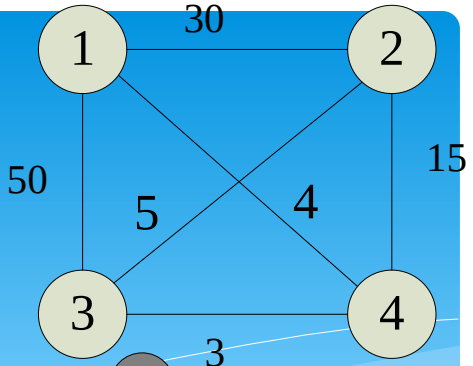
某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小。

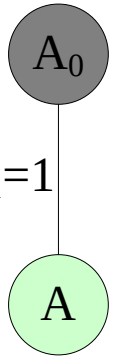


示例2： 旅行售货员问题

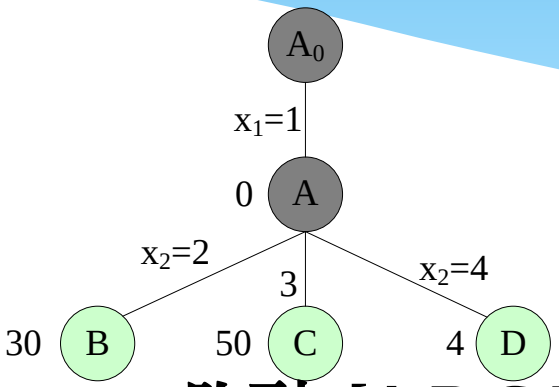
* 问题的解空间（x1,x2,x3,x4），其中令S={1,2,3,4}， x1=1，x2∈S-{x1}，x3∈S-{x1,x2}，x4 ∈ S-{x1,x2,x3}。
* 解空间的组织结构是一棵深度为4的排列树。
* 搜索
  * 约束条件g[i][j]!=∞，其中g是该图的邻接矩阵；
  * 限界条件：cl<bestl，其中cl表示当前已经走的路径长度，初始值为0；bestl表示当前最短路径长度，初始值为∞。

队列式分支限界法

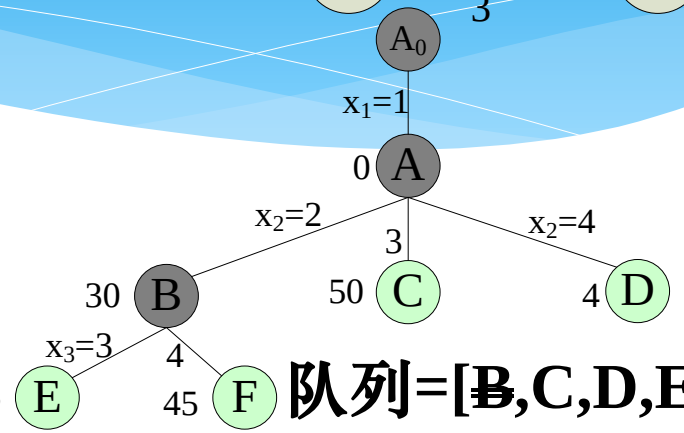




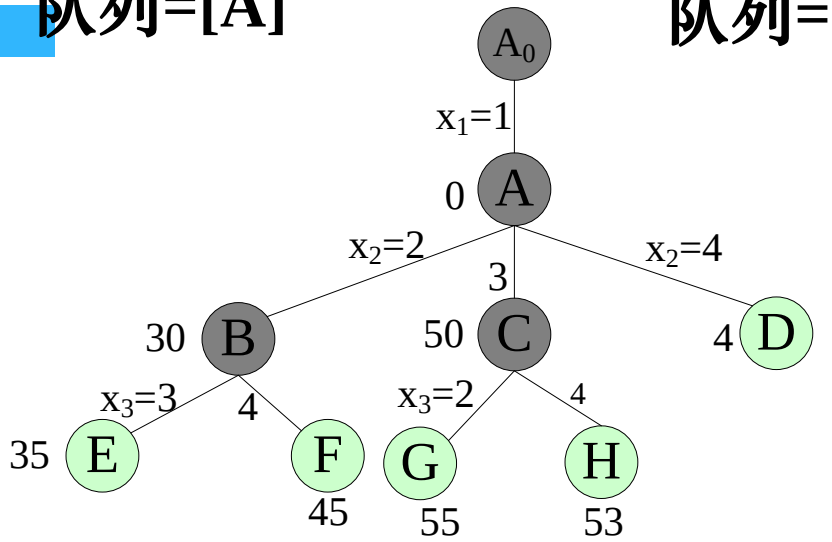
队列=[A]



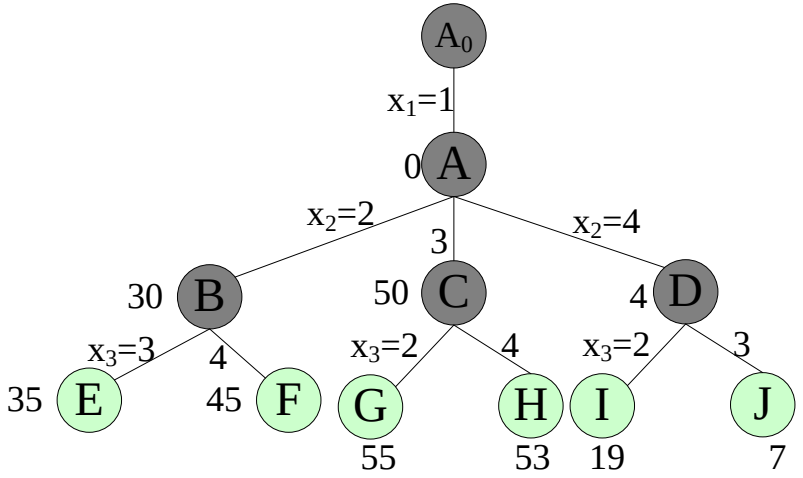
队列=[~~A~~,B,C,D]



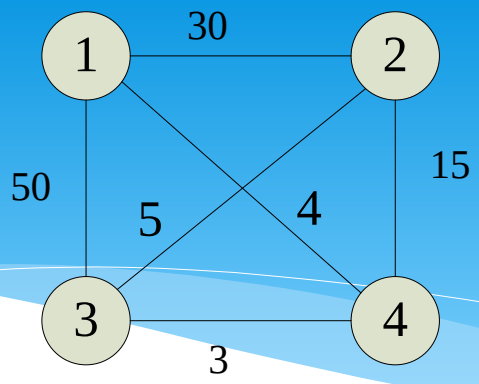
队列=[~~B~~,C,D,E,F]

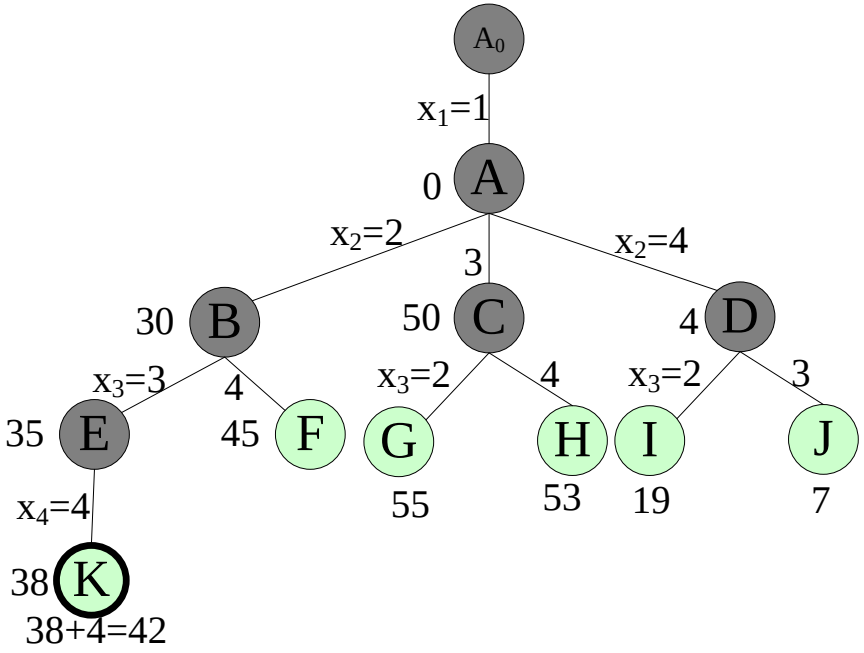


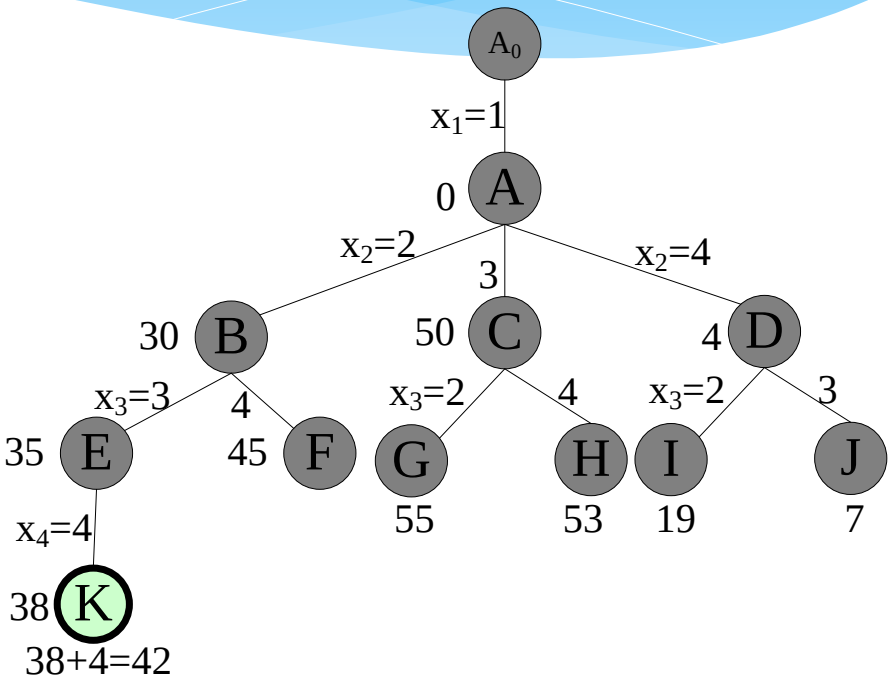
队列=[~~C~~,D,E,F,G,H]



队列=[~~D~~,E,F,G,H,I,J]





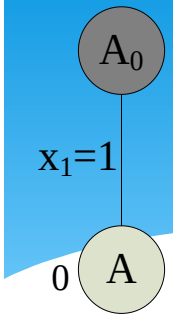


队列=[~~E~~, F,G,H,I,J,K]

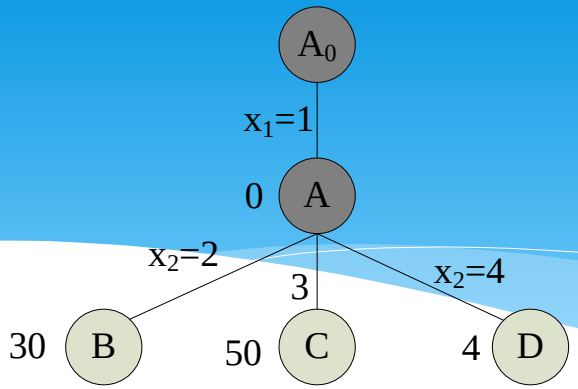
队列=[~~F~~,~~G~~,~~H~~,~~I~~,~~J~~,K]

优先队列式分支限界法

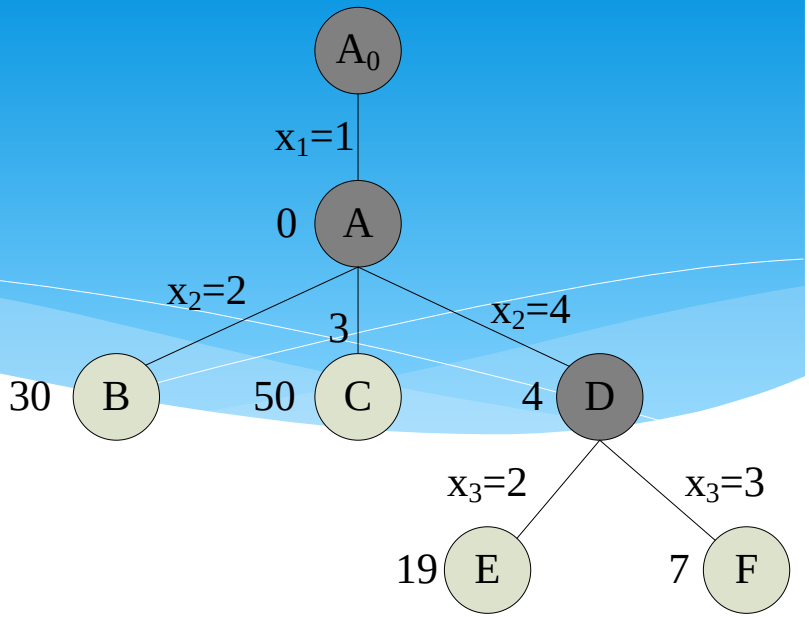
* 优先级：活结点所对应的已经走过的路径长度cl，长度越短，优先级越高。



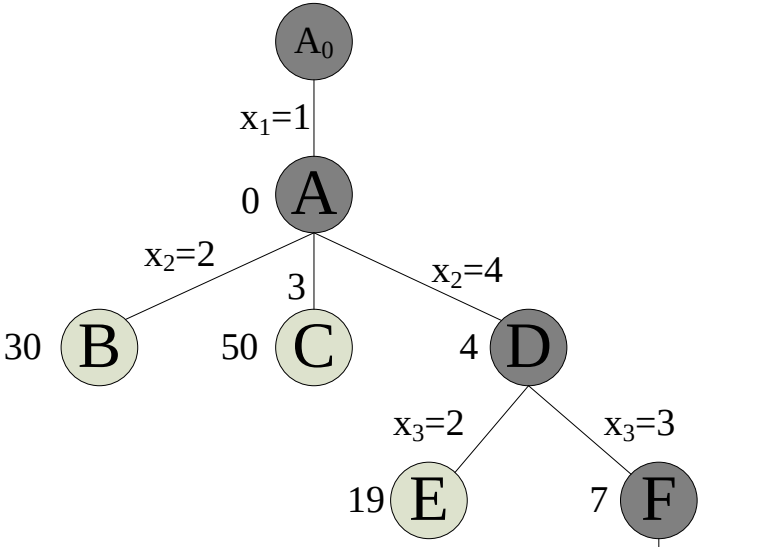
优先队列=[A]



优先队列=[~~A~~,D,B,C]

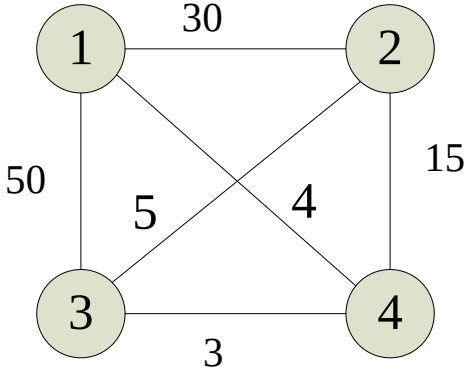


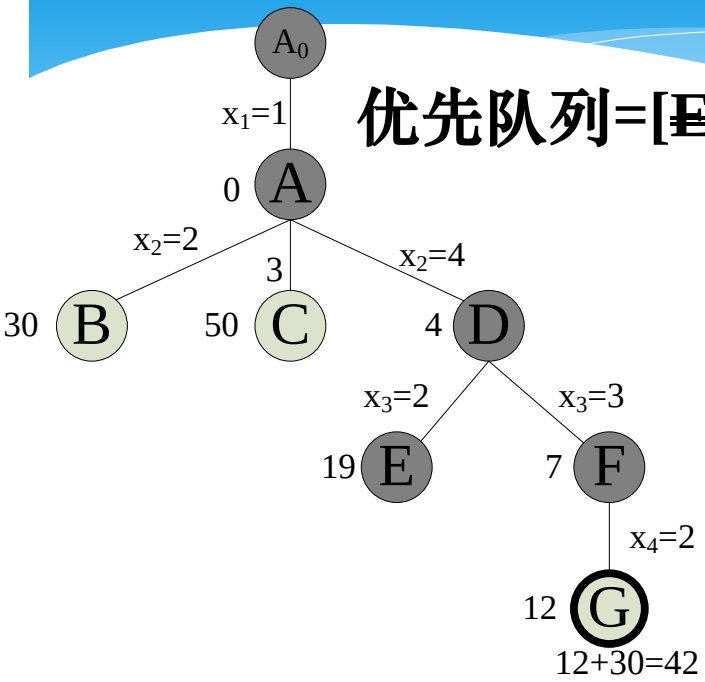
优先队列=[~~D~~,F,E,B,C]



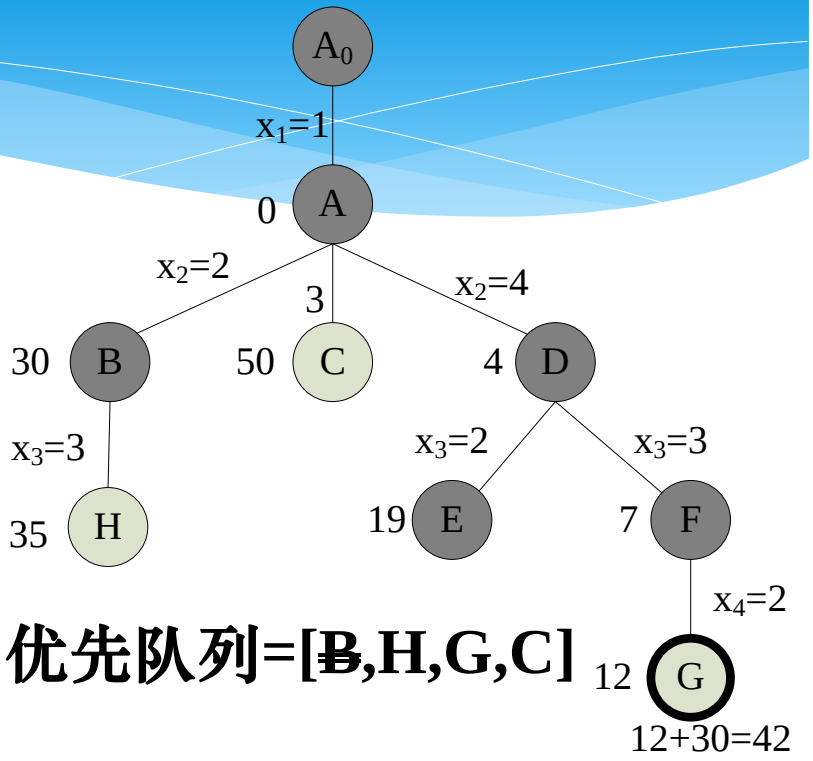
优先队列=[~~F~~,E,B,G,C]

12+30=42

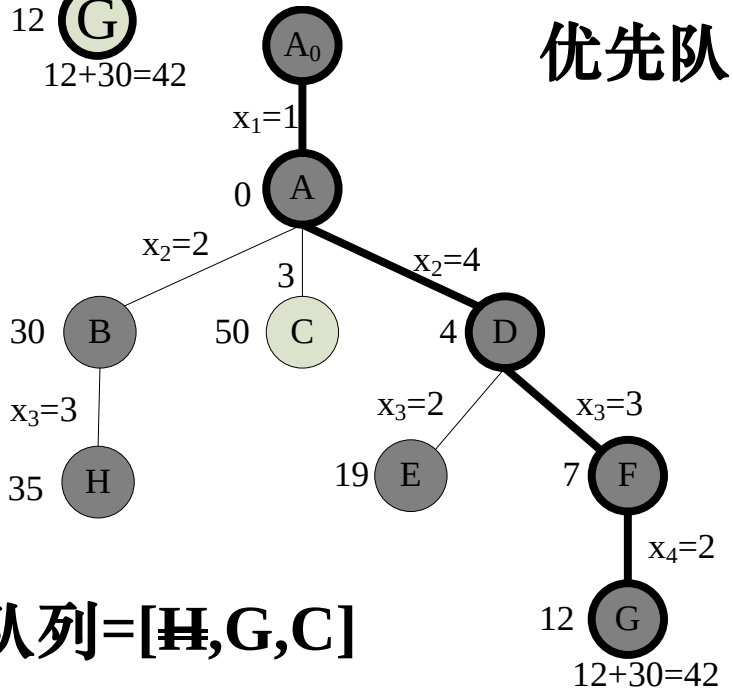




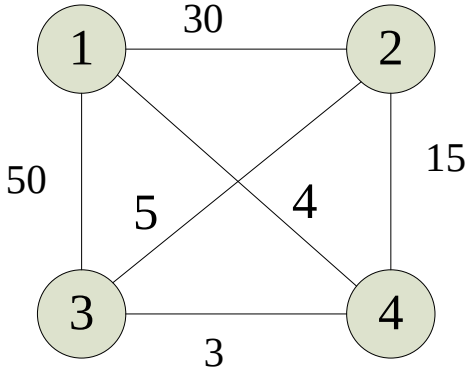
优先队列=[E,B,G,C]



优先队列=[B,H,G,C]



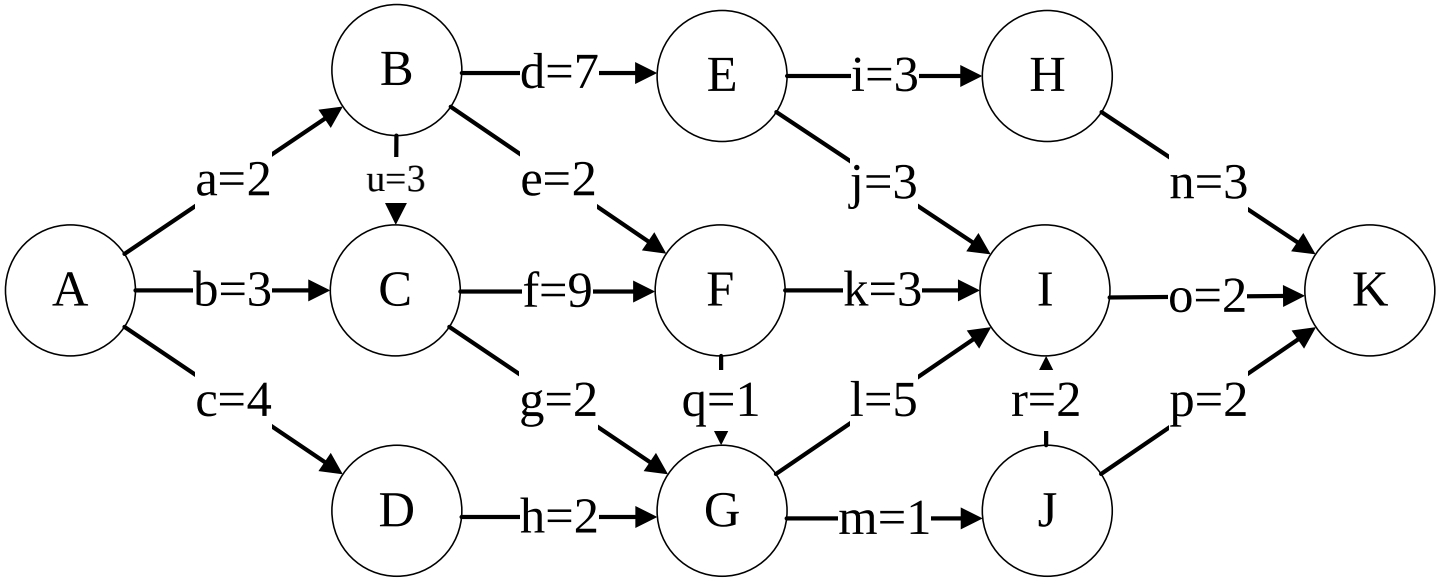
优先队列=[H,G,C]



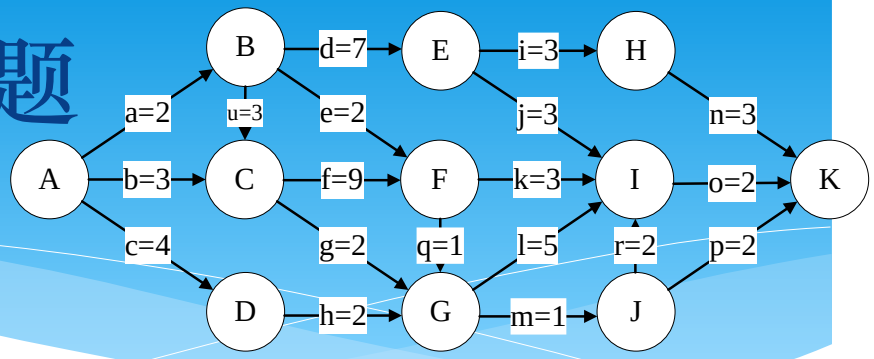
6.2 单源最短路径问题

1. 问题描述

下面以一个例子来说明单源最短路径问题：在下图所给的有向图G中，每一边都有一个非负边权。要求图G的从源顶点A到目标顶点K之间的最短路径。



6.2 单源最短路径问题

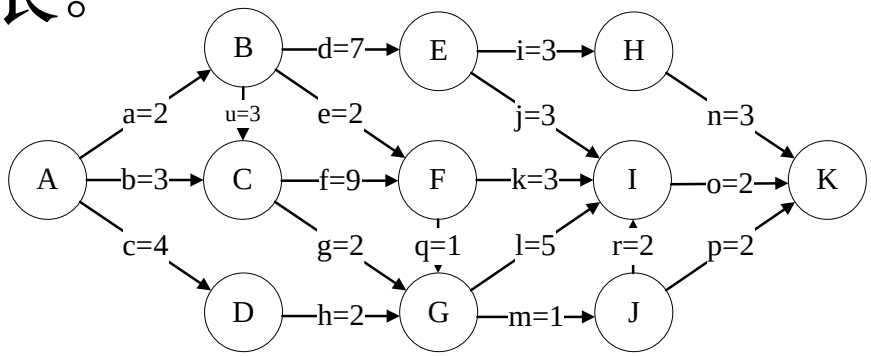


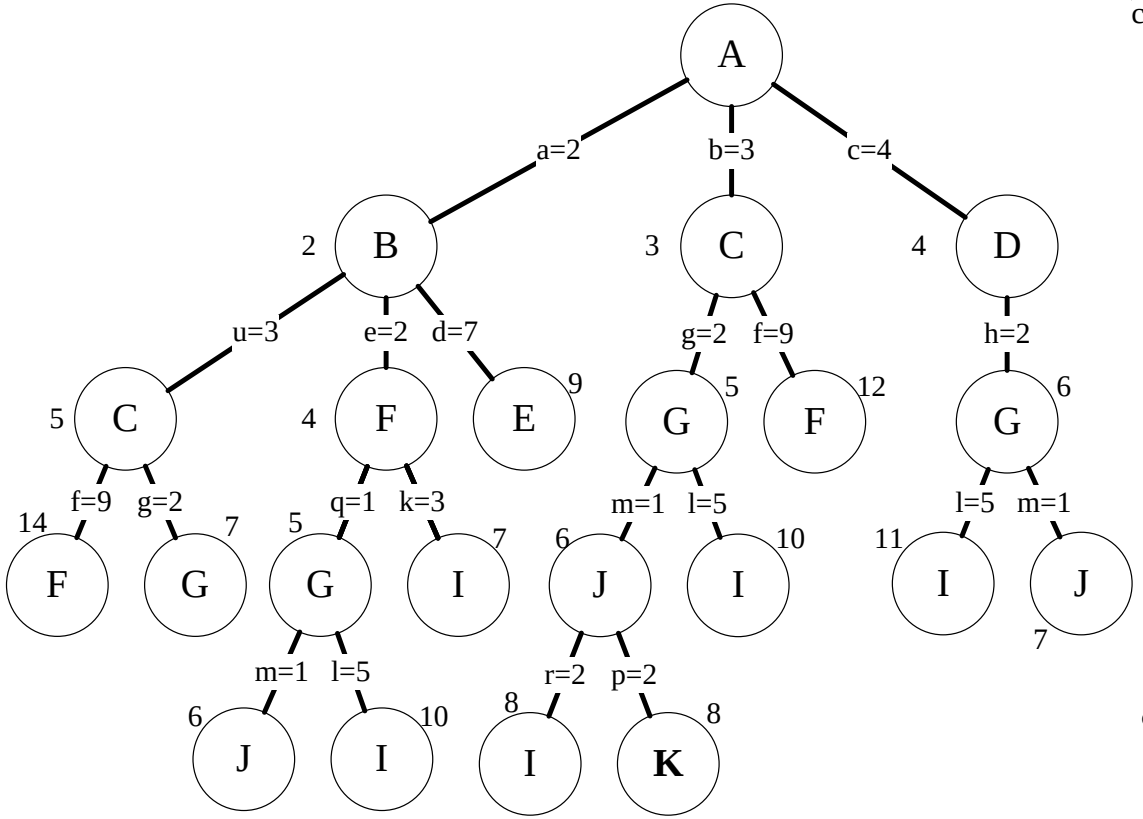
2. 算法思想

解单源最短路径问题的优先队列式分支限界法用一极小堆来存储活结点表。其优先级是结点所对应的当前路长。

算法从图G的源顶点A和空优先队列开始。结点A被扩展后，它的儿子结点被依次插入堆中。此后，算法从堆中取出具有最小当前路长的结点作为当前扩展结点，并依次检查与当前扩展结点相邻的所有顶点。如果从当前扩展结点i到顶点j有边可达，且从源出发，途经顶点i再到顶点j的所相应的路径的长度小于当前最优路径长度，则将该顶点作为活结点插入到活结点优先队列中。这个结点的扩展过程一直继续到活结点优先队列为空时为止。

下图是用优先队列式分支限界法解有向图G的单源最短路径问题产生的解空间树。其中，每一个结点旁边的数字表示该结点所对应的当前路长。





[a2,b3,c4]

[~~a2~~,b3,c4,ae4,au5,ad9]

[~~b3~~,c4,ae4,~~au5~~,bg5,ad9,bf12]

[~~c4~~,ae4, bg5,ch6,ad9,~~bf12~~]

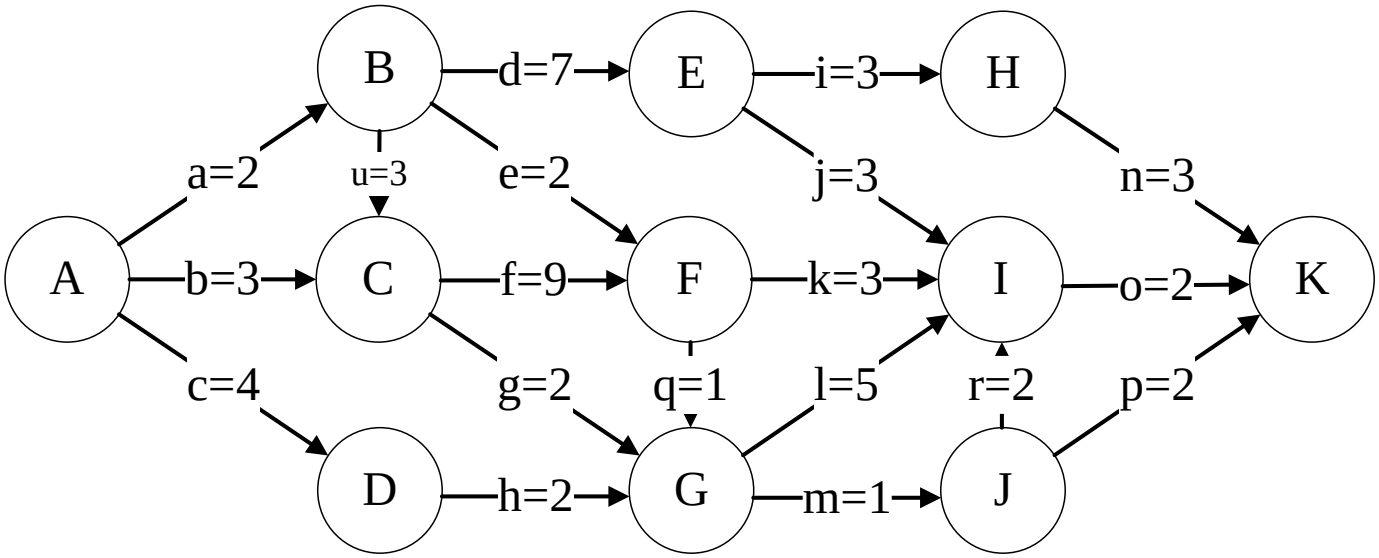
[~~ae4~~, bg5,aeq5,~~ch6~~,aek7,ad9]

[~~bg5~~,aeq5,bgm6,aek7,ad9,bgl10]

[~~aeq5~~,bgm6,aeqm6,aek7,ad9,~~bgl10~~, aeql10]

[~~bgm6~~,aeqm6,aek7,bgmp8,bgmr8, ad9, aeql10

实例：



优先队列：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
prev	0	0	0	0	0	0	0	0	0	0	0

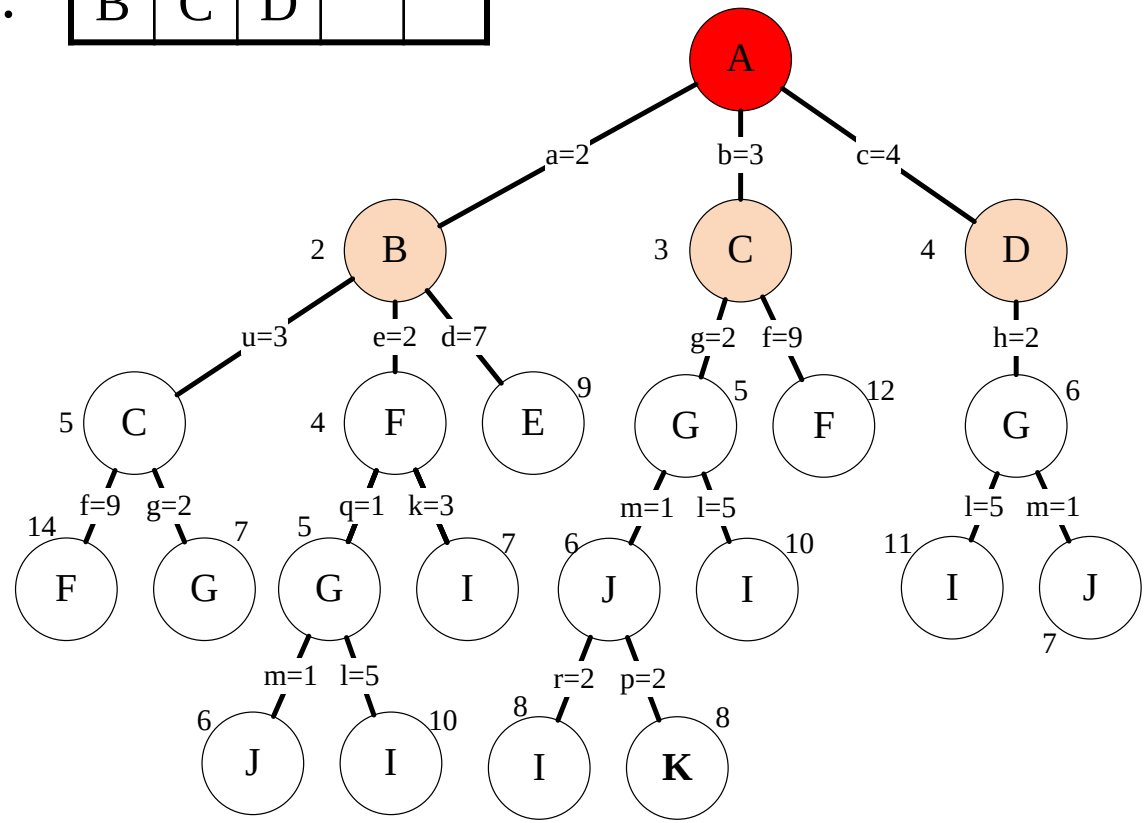
	A	B	C	D	E	F	G	H	I	J	K
A	0	2	3	4	∞	∞	∞	∞	∞	∞	∞
B	∞	0	3	∞	7	2	∞	∞	∞	∞	∞
C	∞	∞	0	∞	∞	9	2	∞	∞	∞	∞
D	∞	∞	∞	0	∞	∞	2	∞	∞	∞	∞
E	∞	∞	∞	∞	0	∞	∞	2	3	∞	∞
F	∞	∞	∞	∞	∞	0	1	∞	3	∞	∞
G	∞	∞	∞	∞	∞	∞	0	∞	5	1	∞
H	∞	∞	∞	∞	∞	∞	∞	0	∞	∞	1
I	∞	∞	∞	∞	∞	∞	∞	∞	0	∞	2
J	∞	∞	∞	∞	∞	∞	∞	∞	∞	0	2
K	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

结点A到其它结点：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	∞	∞	∞	∞	∞	∞	∞
prev		A	A	A							

优先队列：

B	C	D		

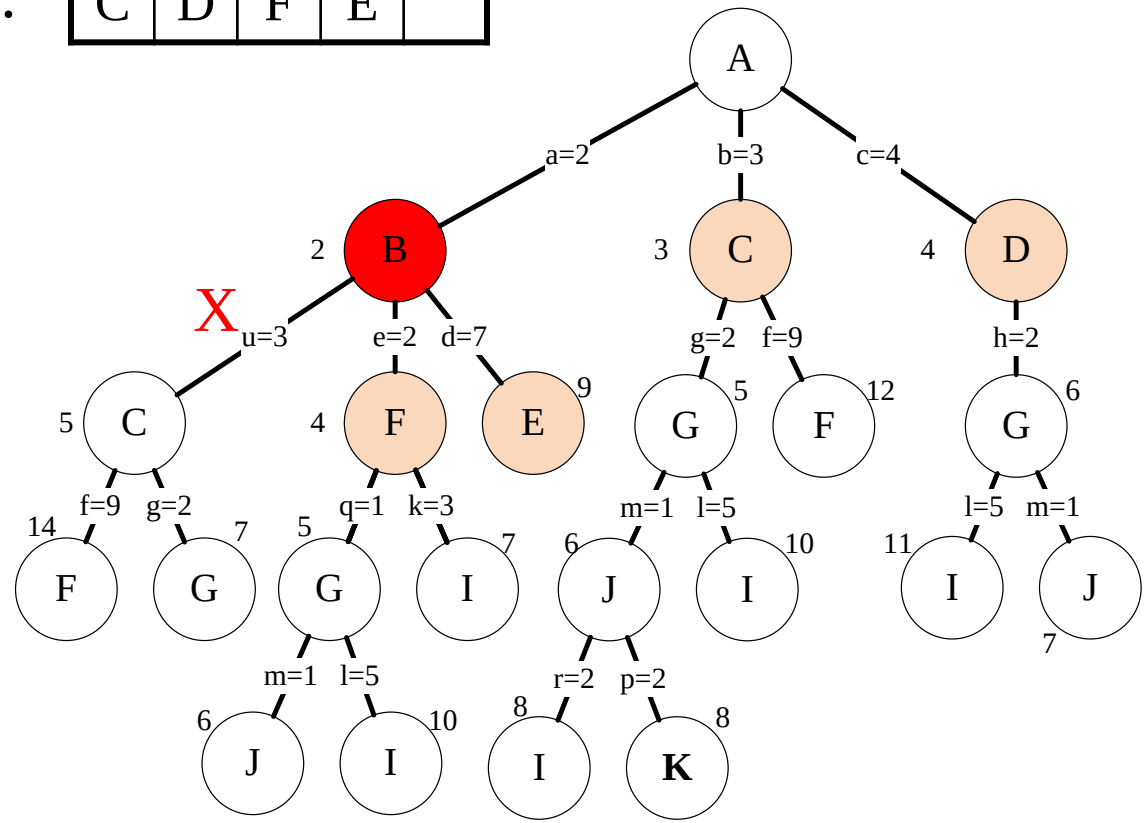


经过结点B到其它结点：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	9	4	∞	∞	∞	∞	∞
prev		A	A	A	B	B					

优先队列：

C	D	F	E	

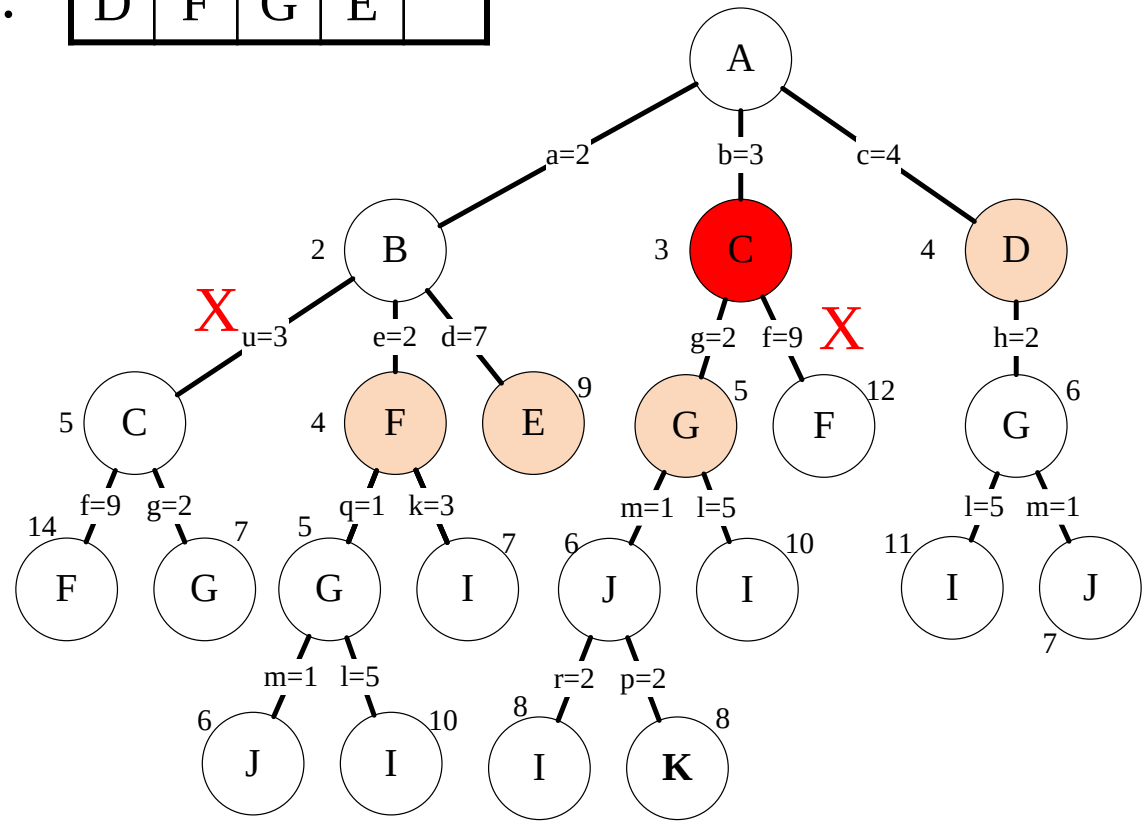


经过结点C到其它结点：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	9	4	5	∞	∞	∞	∞
prev		A	A	A	B	B	C				

优先队列：

D	F	G	E	

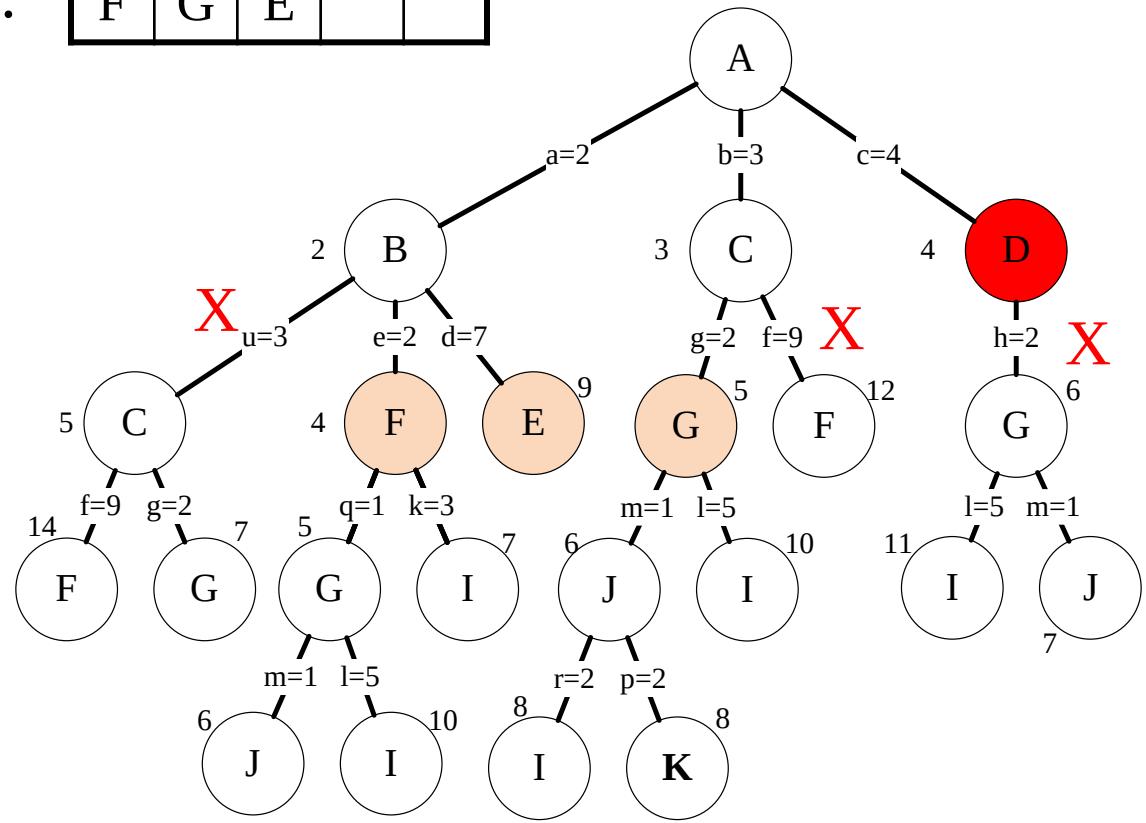


经过结点D到其它结点：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	9	4	5	∞	∞	∞	∞
prev		A	A	A	B	B	C				

优先队列：

F	G	E		

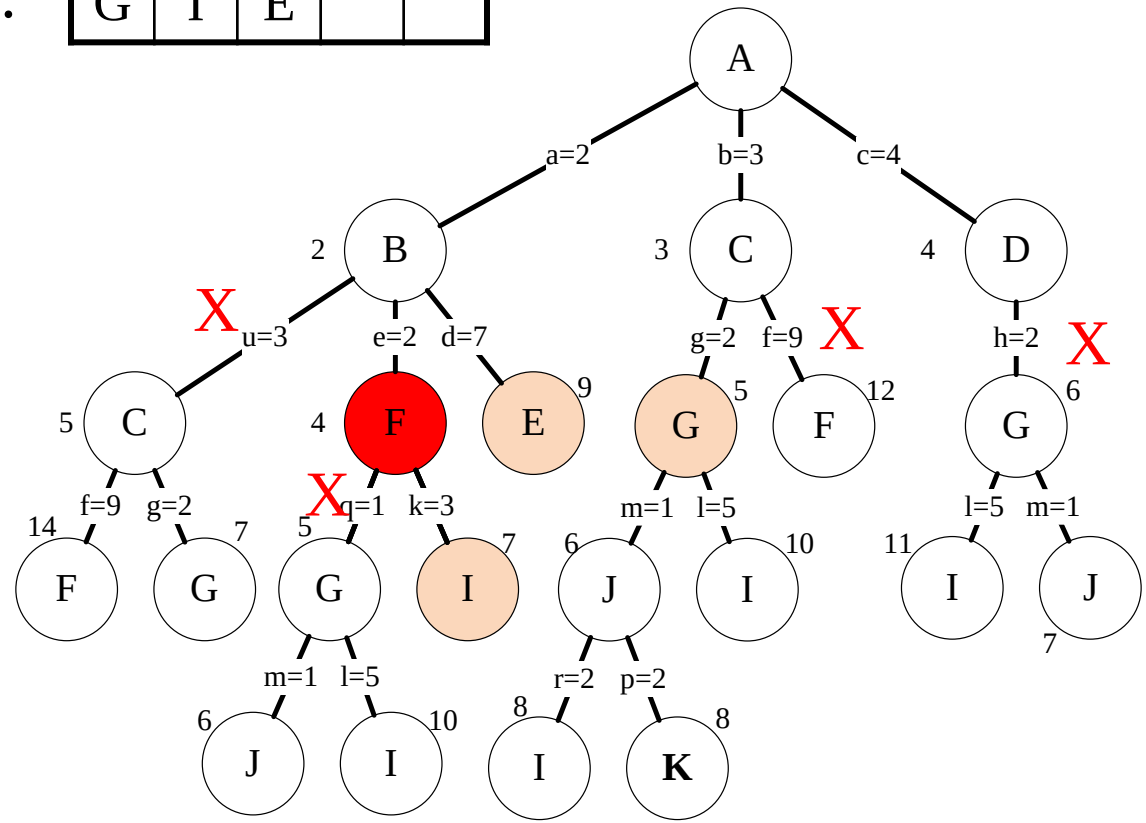


经过结点F到其它结点：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	9	4	5	∞	7	∞	∞
prev		A	A	A	B	B	C		F		

优先队列：

G	I	E		

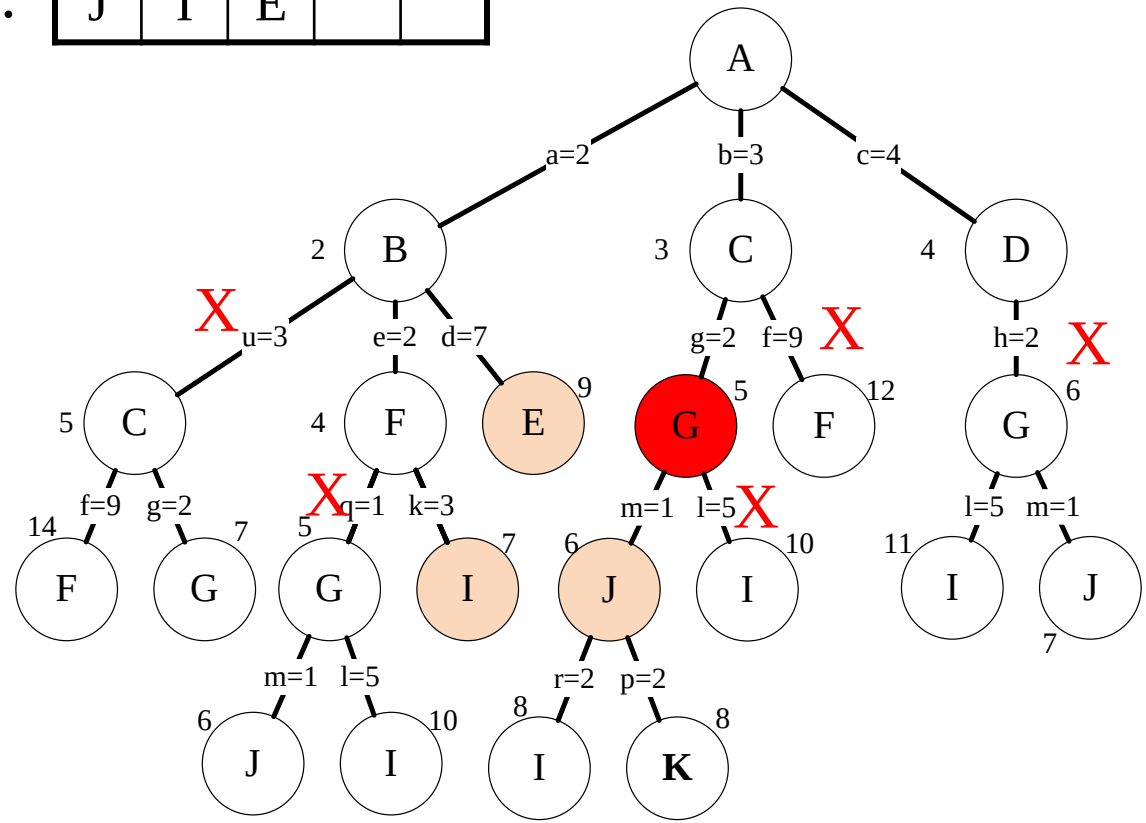


经过结点G到其它结点：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	9	4	5	∞	7	6	∞
prev		A	A	A	B	B	C		F	G	

优先队列：

J	I	E		

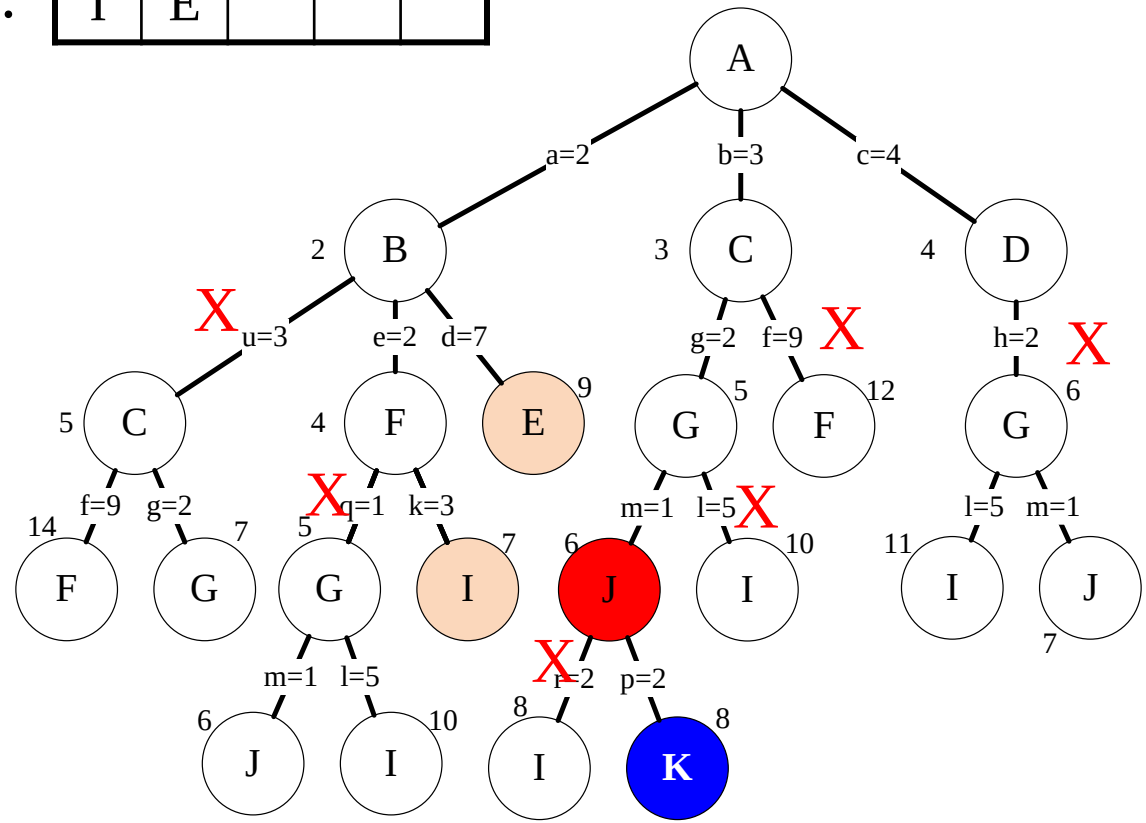


经过结点J到其它结点：

	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	9	4	5	∞	7	6	8
prev		A	A	A	B	B	C		F	G	J

优先队列：

I	E			



	A	B	C	D	E	F	G	H	I	J	K
dist	0	2	3	4	9	4	5	∞	7	6	8
prev		A	A	A	B	B	C		F	G	J

队尾反推:

K→J

J→G

G→C

C→A

因此：A→C →G →J →K

6.2 单源最短路径问题

2. 算法设计

```
static float[][]a  //图G的邻接矩阵
static float []dist //源到各顶点的距离
static int []p //源到各顶点的路径上的前驱顶点
HeapNode //最小堆元素
{  int i; //顶点编号
   float length;//当前路长
  ……
 }
```

```
while (true) {//搜索问题的解空间
    for (int j = 1; j <= n; j++)
      if ((a[enode.i][j]<Float.MAX_VALUE)&&
        (enode.length+a[enode.i][j]<dist[j])) {
        // 顶点i到顶点j可达，且满足控制约束
        dist[j]= enode.length+c[enode.i][j];
        p [j]= enode.i;
        // 加入活结点优先队列
        HeapNode node=new HeapNode(j,dist[j]);
        heap.put(node);
       }
    // 取下一扩展结点
    if ( heap.isEmpty( ) ) break;
    else enode=(HeapNode)heap.removeMin();
    }
}
```

顶点i和j间有边，且此路径长小于原先从原点到j的路径长

6.2 单源最短路径问题

复杂度分析

计算时间为 $O(n!)$。

需要空间为 $O(n^2)$。

6.3 装载问题

1. 问题描述

有一批共n个集装箱要装上2艘载重量分别为C1和C2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

装载问题要求确定是否有一个合理的装载方案可将这n个集装箱装上这2艘轮船。如果有，找出一种装载方案。

如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1)首先将第一艘轮船尽可能装满；

(2)将剩余的集装箱装上第二艘轮船。

6.3 装载问题

2. 算法描述

目标：寻找第一条船的最大装载方案。

解空间：子集树。

数据结构：

queue:存放活结点队列，队列中的元素值表示活结点对应的当前载重量。元素值为-1，表示队列已到达解空间树同一层结点的尾部。

bestw:当前最优载重量。

i:表示扩展结点所处的层

cw:扩展结点所对应的载重量

6.3 装载问题

2. 队列式分支限界法

maxLoading算法的while循环中，首先检测当前扩展结点的左儿子结点是否为可行结点。如果是则将其加入到活结点队列中。然后将其右儿子结点加入到活结点队列中(右儿子结点一定是可行结点)。2个儿子结点都产生后，当前扩展结点被舍弃。

活结点队列中的队首元素被取出作为当前扩展结点，由于队列中每一层结点之后都有一个尾部标记-1，故在取队首元素时，活结点队列一定不空。当取出的元素是-1时，再判断当前队列是否为空。如果队列非空，则将尾部标记-1加入活结点队列，算法开始处理下一层的活结点。

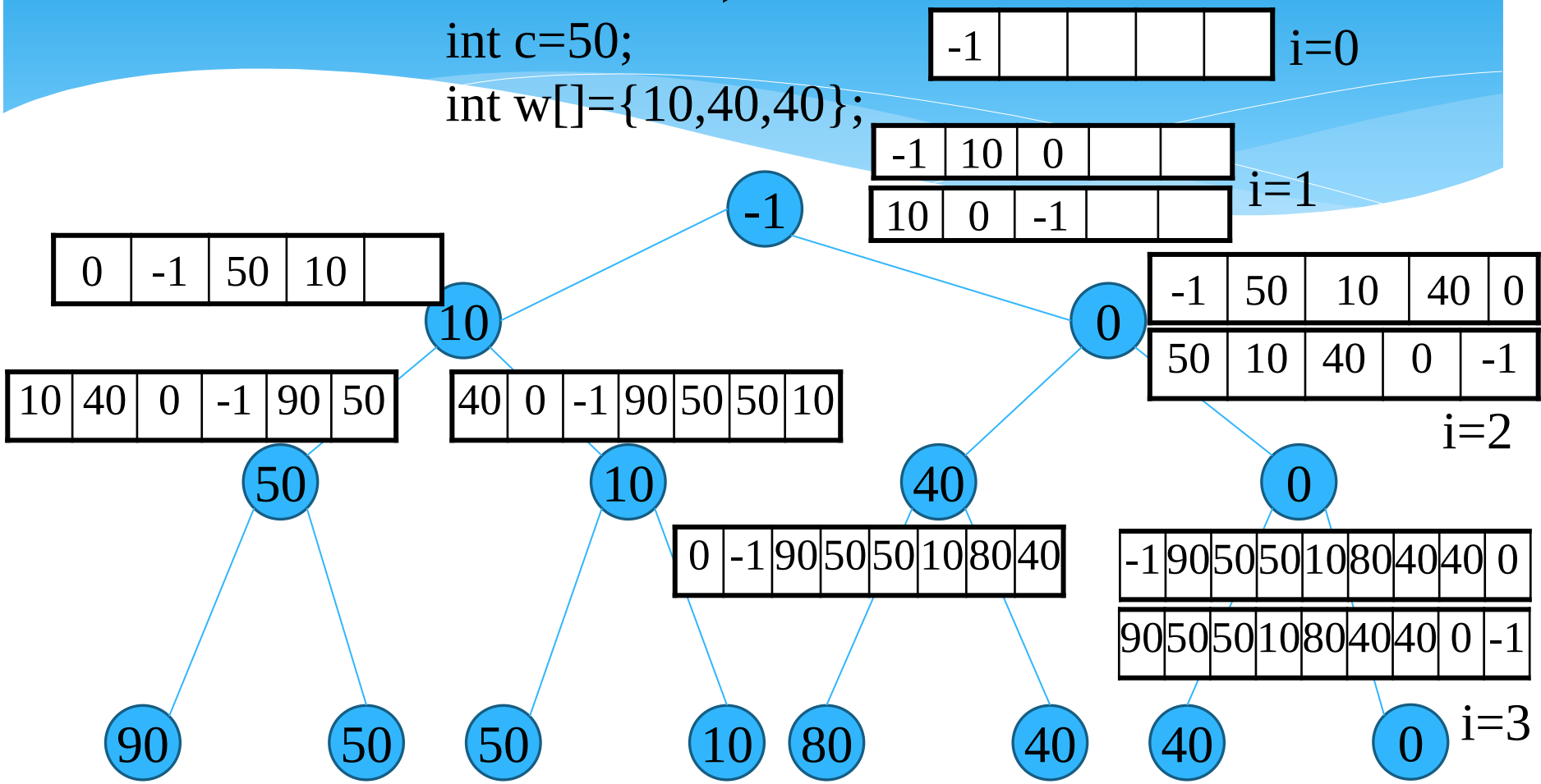
maxLoading算法的时间和空间复杂性均为：$O(2^n)$

2. 队列式分支限界法

```
while (true) { //搜索子集树
    // 检查左儿子结点
    if (cw + w[i] <= c) // x[i] = 1
       enQueue(cw + w[i], i);
    // 右儿子结点总是可行的
    enQueue(cw, i); // x[i] = 0
    cw=((Integer)queue.remove()).intValue( ); // 取下一扩展结点
    if (cw == -1) {      // 同层结点尾部
       if (queue.isEmpty()) return bestw;
       queue.put(new Integer(-1));       // 同层结点尾部标志
       queue. remove()).intValue( );  // 取下一扩展结点
       i++; // 进入下一层
     }
}
```

```
private  static void enQueue(int wt,int i)
{  //将活结点加入到活结点队列中
  if (i==n){   //可行叶结点
     if(wt > bestw){  bestw = wt ;
   }
   else  //非叶结点
     queue.put(new Integer(wt));
}
```

```
const int n=3;
int c=50;
int w[]={10,40,40};
```



装载问题

复杂度分析

计算时间为 $O(2^n)$。

需要空间为 $O(2^n)$。

6.3 装载问题

3. 算法的改进

节点的左子树表示将此集装箱装上船，右子树表示不将此集装箱装上船。设:

bestw是当前最优解;

cw是当前扩展结点所相应的重量;

r是剩余集装箱的重量。

则当cw+r≤bestw时，可将其右子树剪去，因为此时若要船装最多集装箱，就应该把此箱装上船。

另外，为了确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新bestw的值。

3. 算法的改进

```
// 检查左儿子结点
  int wt = cw + w[i];
// wt:左儿子结点的重量
      if (wt <= c) { // 可行结点
         if (wt > bestw) bestw = wt;
         // 加入活结点队列
         if (i < n)
       queue.put(new Integer(wt));
}
```

提前更新 bestw

```
// 检查左儿子结点
if (cw + w[i] <= c) // x[i] = 1
    enQueue(cw + w[i], i);
// 右儿子结点总是可行的
enQueue(cw, i); // x[i] = 0
```

3. 算法的改进

右儿子剪枝

```
// 检查右儿子结点 可能含最优解
 if (cw + r > bestw && i < n)
     queue.put(new Integer(wt));
// 取下一扩展结点
 cw=queue. remove().intValue( );
```

```
// 检查左儿子结点
if (cw + w[i] <= c) // x[i] = 1
    enQueue(cw + w[i], i);
// 右儿子结点总是可行的
    enQueue(cw, i); // x[i] = 0
```

4. 构造最优解

为了在算法结束后能方便地构造出与最优值相应的最优解，算法必须存储相应子集树中从活结点到根结点的路径。为此目的，可在每个结点处设置指向其父结点的指针，并设置左、右儿子标志。

```
static  int n;

static int bestw; // 当前最优载重量

static ArrayQueue queue;  //活结点队列

static Qnode bestE;  //当前最优扩展结点

static int [] bestx;  //当前最优解
```

```
class QNode
 {
     QNode  parent;            // 父结点
     boolean LeftChild;        // 左儿子标志
     int  weight;              // 结点所相应的载重量

static void enQueue(int wt,int I, Qnode parent, boolean leftchild)

{  if (i==n){   //可行叶结点

    if(wt==bestw){//当前最优载重量

      bestE=parent;

      bestx[n]=(leftchild)?1:0;

    }
   return;
   Qnode b=new Qnode(parent,leftchild,wt);  //非叶结点
   queue.put(b);
}
```

修改后的搜索算法

```
int wt = cw + w[i];  // 检查左儿子结点，wt:左儿子结点的重量
  if (wt <= c) { // 可行结点
        if (wt > bestw)  bestw = wt;
         enQueue(wt,i,e,ture); // 加入活结点队列
// 检查右儿子结点,可能含最优解
 if (cw + r > bestw) enQueue(cw, i,e,false);
 e=(QNode)queue.remove();
……
}
```

4. 构造最优解

找到最优值后，可以根据parent回溯到根节点，找到最优解。

```
// 构造当前最优解
for (int j = n - 1; j > 0; j--) {
    bestx[j] = (bestE.LeftChild)?1:0;
    bestE = bestE.parent;
}
```

6.3 装载问题

5. 优先队列式分支限界法

解装载问题的优先队列式分支限界法用最大优先队列存储活结点表。活结点x在优先队列中的优先级定义为从根结点到结点x的路径所相应的载重量再加上剩余集装箱的重量之和。

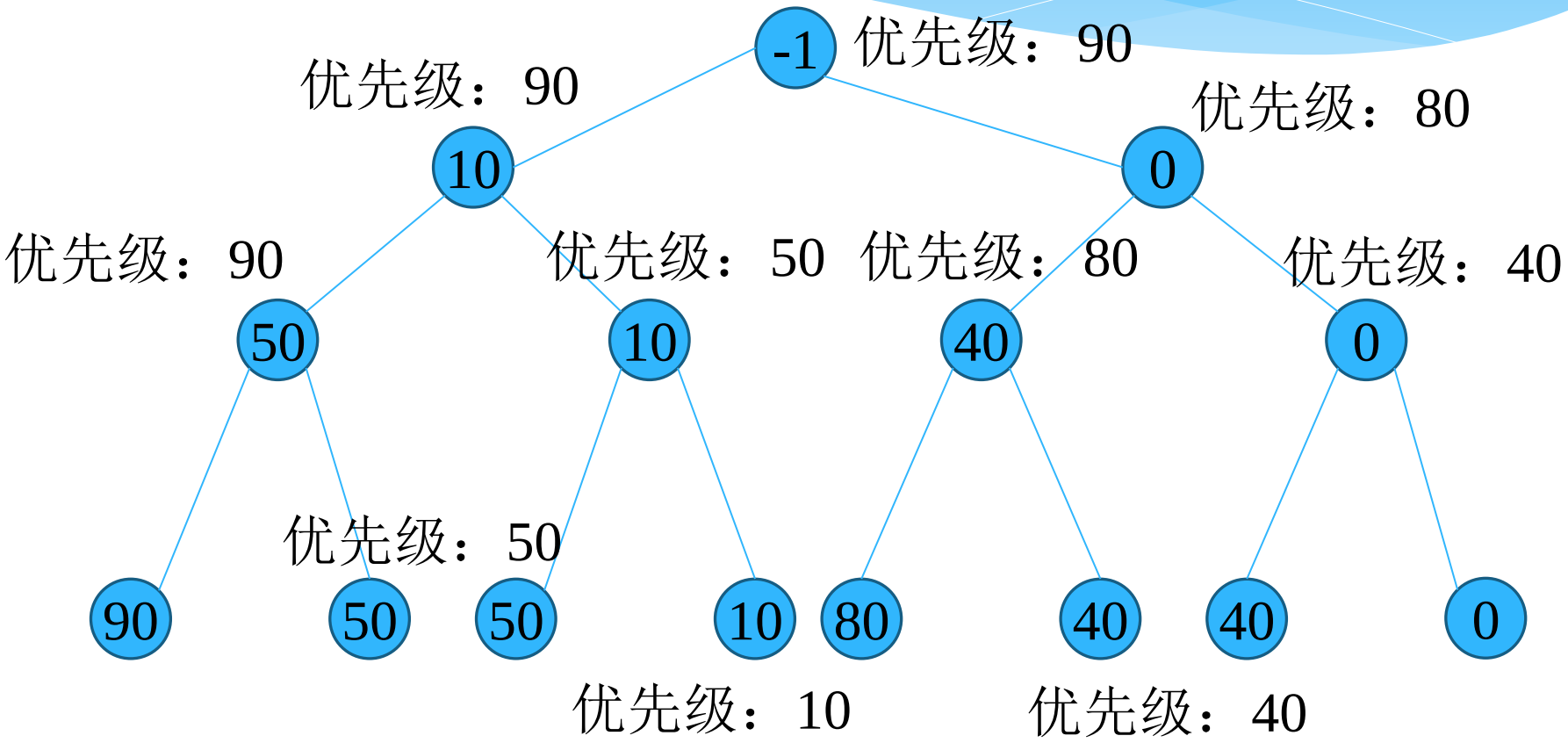
优先队列中优先级最大的活结点成为下一个扩展结点。以结点x为根的子树中所有结点相应的路径的载重量不超过它的优先级。子集树中叶结点所相应的载重量与其优先级相同。

在优先队列式分支限界法中，一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解。此时可终止算法。

6.3 装载问题

```
const int n=3;
int c=50;
int w[]={10,40,40};
```

5. 优先队列式分支限界法



5. 优先队列式分支限界法

解空间树：子集树

数据结构：

```
static class Bbnode{//子集树中的结点
   Bbnode  parent;
   boolean leftChild;
…}
static class HeapNode implements Comparable{
   Bbnode  liveNode;
   int uweight; //活结点优先级（上界）
   int level; //活结点在子集树中所处的层序号
 …}
```

```
while(i!=n+1){//非叶结点，检查当前扩展结点的儿子结点
   if(ew+w[i]<=c)//左儿子结点为可行结点
      addLiveNode(ew+w[i]+r[i],i+1,e,true);
   //右儿子结点总为可行结点
   addLiveNode(ew+r[i],i+1,e,false);
   //取下一扩展结点
   HeapNode node=(HeapNode)heap.removeMax();
   i=node.level;
   e=node.liveNode;
   ew=node.uweight-r[i-1];
}

//构造当前最优解
For(int j=n;j>0;j--){
   bestx[j]=(e.leftChild)?1:0;
   e=e.parent;
}
 return ew;
```

装载问题

复杂度分析

计算时间为 $O(2^n)$。

需要空间为 $O(2^n)$。

6.4 作业分配问题

* n 个操作员以 n 种不同时间完成 n 种不同作业。要求分配每位操作员完成一项工作，使完成 n 项工作的总时间最少

一 分支限界法解作业分配问题的思想方法

1. 问题描述：

n 个操作员以 n 种不同时间完成 n 种不同作业。要求分配每位操作员完成一项工作，使完成 n 项工作的总时间最少

操作员编号为 0,1,…n-1，

作业也编号为 0,1,…n-1，

矩阵 c 描述每位操作员完成每个作业时所需的时间，

元素 $c_{i,j}$ 表示第 i 位操作员完成第 j 号作业所需的时间

向量 x 描述分配给操作员的作业编号，

分量 x_i 表示分配给第 i 位操作员的作业编号。

一 分支限界法解作业分配问题的思想方法

2. 思想方法：

1）从根结点开始，每遇到一个e_结点，就对它的所有儿子结点计算其下界，把它们登记在结点表中。

2）从表中选取下界最小的结点，重复上述过程。

3）当搜索到一个叶子结点时，如果该结点的下界是结点表中最小的，那么，该结点就是问题的最优解。

4）否则，对下界最小的结点继续进行扩展

一 分支限界法解作业分配问题的思想方法

3. 下界的确定：

1）搜索深度为 0 时，把第 0 号作业分配给第 i 位操作员所需时间至少为第 i 位操作员完成第 0 号作业所需时间，加上其余 n-1个作业分别由其余 n-1 位操作员单独完成时所需最短时间之和，有：

$$t = c_{i0} + \sum_{j=1}^{n-1} \left(\min_{l \neq i} c_{l\,j} \right)$$

一 分支限界法解作业分配问题的思想方法

3. 下界的确定：

例：4个操作员完成4个作业所需的时间表如下：

操作员 \ 作业	0	1	2	3
0	3	8	4	12
1	9	12	13	5
2	8	7	9	3
3	12	7	6	8

把第 0 号作业分配给第 0 位操作员时，所需时间至少不会小于 3 + 7 + 6 + 3 = 19 ，把0号作业1 位操作员时，所需时间至少不会9+7+4+3…

一 分支限界法解作业分配问题的思想方法

3. 下界的确定：

2）搜索深度为 k 时，前面第0,1,…,k-1号作业已分别分配给编号为$i_0,i_1,\ldots,i_{k-1}$的操作员。

S={0,1,…,n-1}表示所有操作员的编号集合；

$m_{k-1}=\{i_0,i_1,\ldots,i_{k-1}\}$表示作业已分配的操作员编号集合。

当把第k号作业分配给编号为i_k的操作员时，$i_k \in S-m_{k-1}$，所需时间至少为：

$$t=\sum_{l=0}^{k} c_{i_l l}+\sum_{l=k+1}^{n-1}\left(\min_{i\in S-m_k} c_{il}\right) \qquad (1)$$

则上式为把第k号作业分配给编号为i_k的操作员时的下界

一 分支限界法解作业分配问题的思想方法

4. 算法实现步骤：

每个结点都包含已分配作业的操作员编号集合m 、

未分配作业的操作员编号集合 S、操作员的分配方案向量x 、搜索深度 k(第k个作业)、所需时间的下界 t

1）建立根结点X，令X.k = 0，X.S={0,1,…,n-1}，X.m=φ，把当前问题的可行解的最优时间下界bound 置为∞。

2）对所有编号为 i 的操作员，i∈X.S，建立儿子结点 Y_i，把结点X的数据复制到结点Y_i。

3）令$Y_i.m=Y_i.m\cup\{i\}$，$Y_i.S=Y_i.S-\{i\}$，$Y_i.x=Y_i.k$，$Y_i.k=Y_i.k+1$，按式（1）计算$Y_i.t$。

$$t=\sum_{l=0}^{k}c_{i_l l}+\sum_{l=k+1}^{n-1}\left(\min_{i\in S-m_k}c_{il}\right)$$

一 分支限界法解作业分配问题的思想方法

4. 算法实现步骤：

4）如果$Y_i.t<bound$，转步骤 5），否则剪去结点Y_i，转步骤 6）。

5）把结点 Y_i插入优先队列。如果结点 Y_i是叶结点，表明它是问题的一个可行解，用 $Y_i.t$ 更新当前可行解的最优时间下界 bound 。

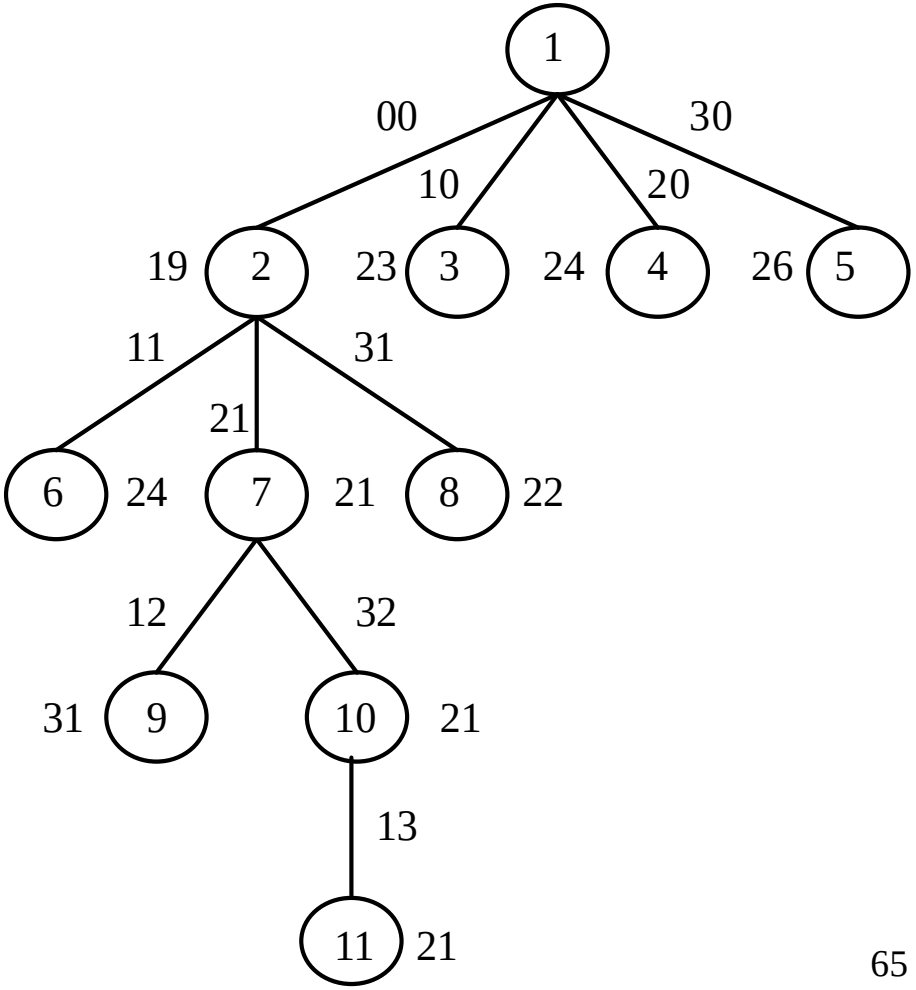
6）取下队列首元素作为子树的根结点X，若X.k=n，则该结点是叶结点，表明它是问题的最优解，算法结束，向量 X.x 便是作业最优分配分案；否则，转步骤 2）。

一 分支限界法解作业分配问题的思想方法

例：图1 所示的4个操作员的作业最优分配方案的搜索树。

作业

操作员	0	1	2	3
0	3	8	4	12
1	9	12	13	5
2	8	7	9	3
3	12	7	6	8



二 分支限界法解作业分配问题算法的实现

1. 数据结构如下：

```
struct ass_node {
    int     x[n];              /* 分配给操作员的作业 */
    int     k;                 /* 搜索深度 */
    float   t;          /* 当前搜索深度下,已分配的作业所需时间 */
    float   b;          /* 本结点所需的时间下界 */
struct ass_node *next;     /* 优先队列链指针 */
};
typedef struct ass_node ASS_NODE
float      c[n][n];  /* n个操作员分别完成n种作业所需时间 */
float      bound;  /* 当前已搜索到的可行解的最优时间 */
ASS_NODE       *qbase; /* 优先队列的首指针 */
```

二 分支限界法解作业分配问题算法的实现

2. 队列操作函数：

Q_insert(ASS_NODE *qbase，ASS_NODE *xnode);

把 所指向的结点按时间下界插入优先队列 中，下界越小，优先性越高。

ASS_NODE *Q_delete(ASS_NODE *qbase);

取下并返回优先队列 的首元素

二 分支限界法解作业分配问题算法的实现

3. 实现代码：

```
1. #define MAX_FLOAT_NUM  ∞/* 最大的浮点数 */ ∞
2. float job_assigned(float c[][],int n,int job[])
3. {
4.     int i,j,m,n_heap = 0;
5.     ASS_NODE *xnode,*ynode,*qbase = NULL;
6.     float min,bound = MAX_FLOAT_NUM;
7.     xnode = new ASS_NODE;
8.     for (i=0;i<n;i++)          /*初始化xnode所指向的根结点*/
9.         xnode->x[i] = -1;
10.    xnode->t = xnode->b = 0;
11.    xnode->k = 0;
```

二 分支限界法解作业分配问题算法的实现

```
12.    while (xnode->k!=n) {          /* 非叶子结点,继续向下搜索 */
13.       for (i=0;i<n;i++) {                                  /* 对n个操作员分别判断处理 */
14.          if (xnode->x[i]==-1) {                  /* 操作员i尚未分配作业 */
15.             ynode = new ASS_NODE;       /* 为操作员i建立结点 */
16.             *ynode = *xnode;      /*把父亲结点的数据复制给它 */
17.             ynode->x[i] = ynode->k;          /* 作业k分配给操作员i */
18.             ynode->t += c[i][ynode->k];    /*已分配作业时间累计*/
19.             ynode->b = ynode->t;
20.             ynode->k++;
```

二 分支限界法解作业分配问题算法的实现

```
21.             for (j=ynode->k;j<n;j++) {/* 未分配作业最小时间估计 */
22.                 min = MAX_FLOAT_NUM;
23.                 for (m=0;m<n;m++) {
24.                     if ((ynode->x[m]==-1)&&(c[m][j]<min))
25.                         min = c[m][j];
26.                 }
27.                 ynode->b += min;
28.             }
29.             if (ynode->b<bound) {/* 小于可行解的最优下界 */
30.                 Q_insert(qbase,ynode);      /* 把结点按下界插入堆中 */
31.                 if (ynode->k==n)                /* 已得到一个可行解 */
32.                     bound = ynode->b;  /* 更新可行解的最优下界 */
33.             }
34.             else delete ynode;       /* 大于可行解的最优下界,剪除*/
35.         }
36.     }
```

二 分支限界法解作业分配问题算法的实现

```
37.         delete xnode;/* 释放结点xnode的缓冲区 */
38.         xnode = Q_delete(qbase);/* 取下队列首元素 */
39.     }
40.     min = xnode->b;/* 保存下界,以便作为返回值返回 */
41.     for (i=0;i<n;i++)     /* 把分配方案保存于数组x中返回*/
42.         job[i] = xnode->x[i];
43.     while (qbase) {       /* 释放结点缓冲区 */
44.         xnode = Q_delete(qbase);
45.         delete xnode;
46.     }
47.     return min;
48. }
```

二 分支限界法解作业分配问题算法的实现

4.复杂性分析

在第11行之前的初始化部分：第8～9行的 for 循环，初始化根结点的向量 x 需O(n)时间；其余需O(1)时间。

第13~39行的while循环：

假定需进行 c 个结点的处理。每处理一个结点，需要：

第16行把父亲结点的数据复制给儿子结点，需O(n)时间，

第21~28行的二重循环，需要$O(n^2)$时间，

第30行把结点插入优先队列，第38行取下队列首元素，需O(c)时间，其余需O(1)时间。

因此，第13~39行的while循环，在最坏情况下需$O(cn^2)$时间。

算法的结束部分：第41行及43行的for循环分别需O(n)时间和O(c)时间

因此，算法在最坏情况下，需$O(cn^2)$时间。

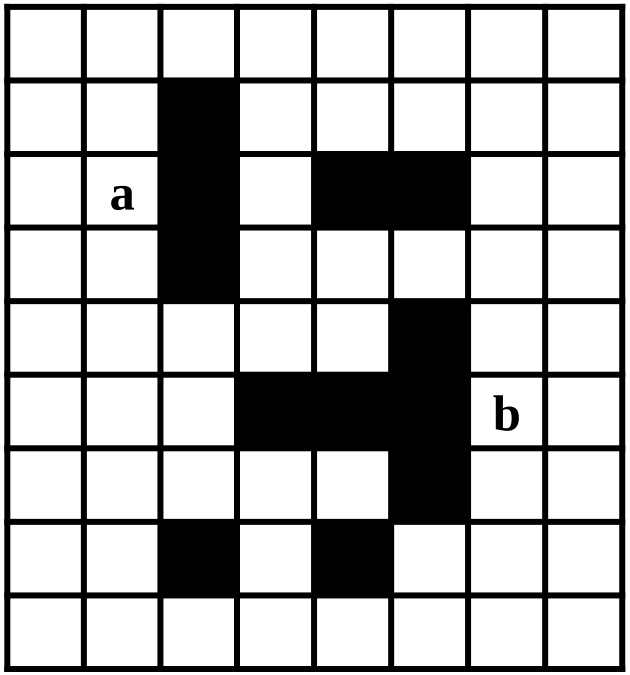
共处理c个结点。每个结点需O(n)空间。

因此，在最坏情况下，算法的空间复杂性是O(cn)。

示例4： 布线问题

问题描述

* 在N×M的方格阵列中，指定一个方格为a，另一个方格为b，问题要求找出a到b的最短布线方案（即最短路径）。布线时只能沿直线或直角，不能走斜线。黑色的单元格代表不可以通过的封锁方格。

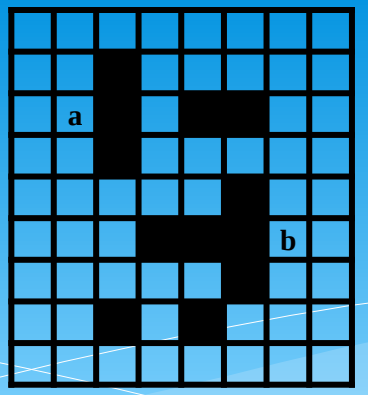


9×8阵列

问题分析

* 将方格抽象为顶点，中心方格和相邻四个方向（上、下、左、右）能通过的方格用一条边连起来。这样，可以把问题的解空间定义为一个图。
* 该问题是特殊的最短路径问题，特殊之处在于用布线走过的方格数代表布线的长度，布线时每布一个方格，布线长度累加1。
* 只能朝上、下、左、右四个方向进行布线

布线问题

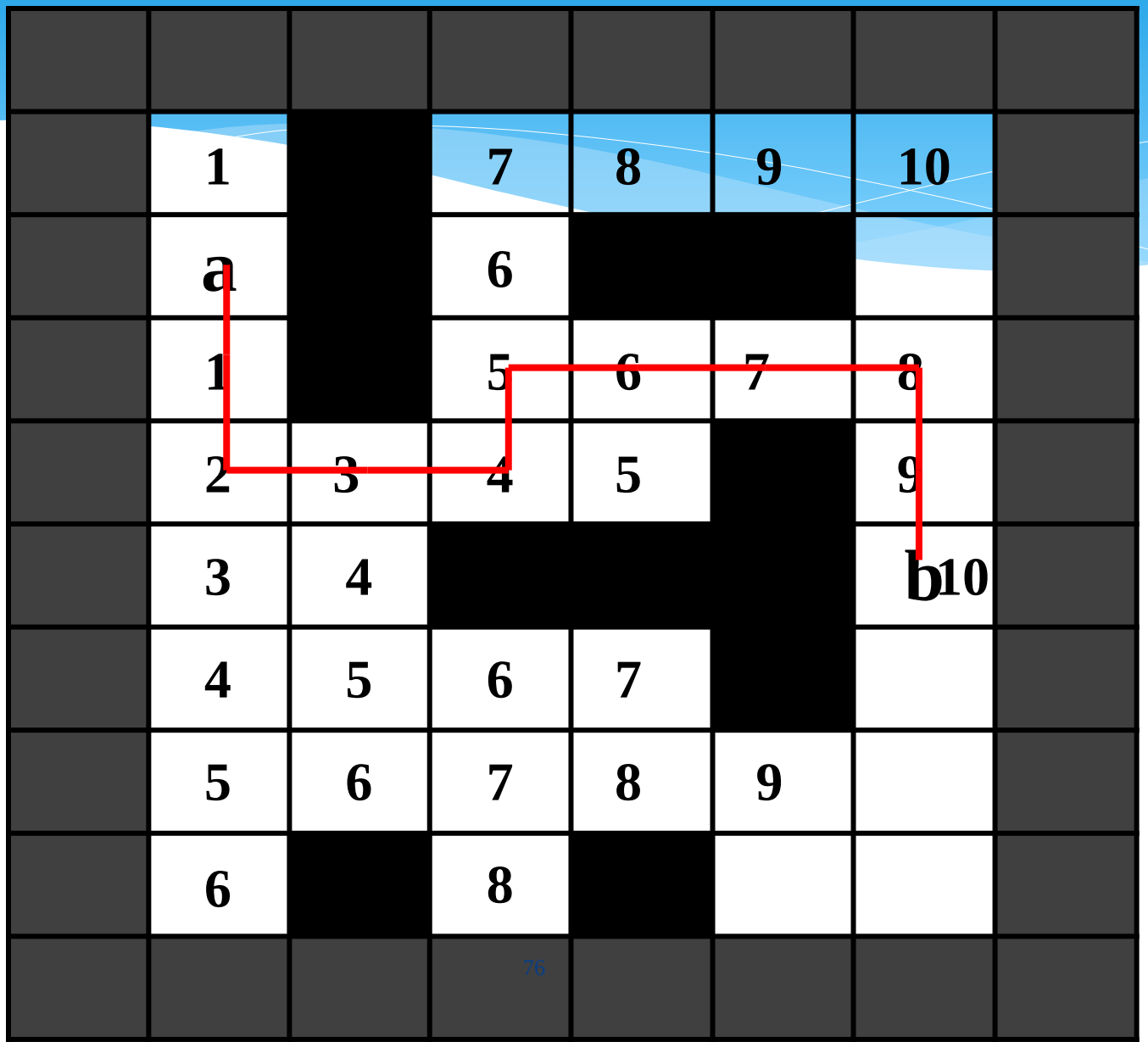


1. 算法思想

解此问题的队列式分支限界法从起始位置a开始将它作为第一个扩展结点。与该扩展结点相邻并且可达的方格成为可行结点被加入到活结点队列中，并且将这些方格标记为1，即从起始方格a到这些方格的距离为1。

接着，算法从活结点队列中取出队首结点作为下一个扩展结点，并将与当前扩展结点相邻且未标记过的方格标记为2，并存入活结点队列。这个过程一直继续到算法搜索到目标方格b或活结点队列为空时为止。即加入剪枝的广度优先搜索。

实例



左

上

右

下

算法描述

思考1：如何表示左、上、右、下四个方向？

```
Position offset[4];
  offset[0].row = 0; offset[0].col = 1; // 右
  offset[1].row = 1; offset[1].col = 0; // 下
  offset[2].row = 0; offset[2].col = -1; // 左
  offset[3].row = -1; offset[3].col = 0; // 上
```

定义移动方向的相对位移

思考2：如何表示阵列的边界？

```
for (int i = 0; i <= m+1; i++)
    grid[0][i] = grid[n+1][i] = -2; // 顶部和底部
for (int i = 0; i <= n+1; i++)
    grid[i][0] = grid[i][m+1] = -2; // 左翼和右翼
```

设置边界的围墙

思考3：如何表示可以布线的方格?

```
do {
    for(int i=0; i<Numofnbrs;i++)
     {
        nbr.row=here.row+offset[i].row;
        nbr.col=here.col+offset[i].col;
        if(grid[nbr.row][nbr.col]==-1)//不等于-2(边界、障碍)，不等于正数(已有值)
         {
                grid[nbr.row][nbr.col]=grid[here.row][here.col]+1;
                if((nbr.row==finish.row)&&(nbr.col==finish.col))
                        break;
                Q.Add(nbr);  //此邻结点放入队列
         }
    if((nbr.row==finish.row)&&(nbr.col==finish.col))
        break;  //完成布线
    if(Q.Isempty())
        return;
    Q.Delete(here);//从队列中取下一个扩展结点
  } while(true)
```

思考4：找到目标位置后，如何找到布线方案?

```
PathLen=grid[finish.row][finish.col];
path=new Position[Pathlen];
here=finish;
for(int j=PathLen-1;j>=0;j--)
{
    path[j]=here;
    for(int i=0;i<Numofnbrs;i++)
    {
       nbr.row=here.row+offset[i].row;
       nbr.col=here.col+offset[i].col;
       if (grid[nbr.row][nbr.col]==j)
           break;
    }
    here=nbr;//往回推进
}
```

6.4 布线问题

复杂度分析

计算时间为O(mn)+O(L)

需要空间为 O(mn)。

6.5　0-1背包问题

算法的思想

首先，要对输入数据进行预处理，将各物品依其单位重量价值从大到小进行排列。

在优先队列分支限界法中，节点的优先级由已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和。

算法首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点优先队列中。当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。当扩展到叶节点时为问题的最优值。

6.5　0-1背包问题

上界函数

```
while (i <= n && w[i] <= cleft)   // n表示物品总数，cleft为剩余空间
{
        cleft -= w[i];                    //w[i]表示i所占空间
        b += p[i];                        //p[i]表示i的价值
        i++;
    }
if (i <= n) b += p[i] / w[i] * cleft;  // 装填剩余容量装满背包
return b;                                 //b为上界函数值
```

6.5 0-1背包问题

```
static class  Bbnode{
    BBnode  parent;  //父结点
    boolean  leftChild; //左儿子结点标志
…}
static class HeapNode implements Comparable{
    BBnode  liveNode;  //活结点
    double  upperProfit; //结点的价值上界
    double  profit;        //结点所相应的价值
    double  weight;       //结点所相应的重量
    int   level;              //活结点在子集树中所处的层序号
```

```
while (i != n+1) {// 非叶结点
    // 检查当前扩展结点的左儿子结点
    double  wt = cw + w[i];
    if (wt <= c) {// 左儿子结点为可行结点
       if (cp+p[i] > bestp) bestp = cp+p[i];
       addLiveNode(up, cp+p[i], cw+w[i], i+1,enode, true );}
       up = bound(i+1);
       // 检查当前扩展结点的右儿子结点
       if (up >= bestp) // 右子树可能含最优解
          addLiveNode(up, cp, cw, i+1, enode, false);
       // 取下一个扩展节点
       HeapNode  node=(HeapNode)heap.removeMax();
       enode=node.liveNode;
       cw=node.weight;
       cp=node.profit;
       up=node.upperProfit;
       i=node.level;
}
```

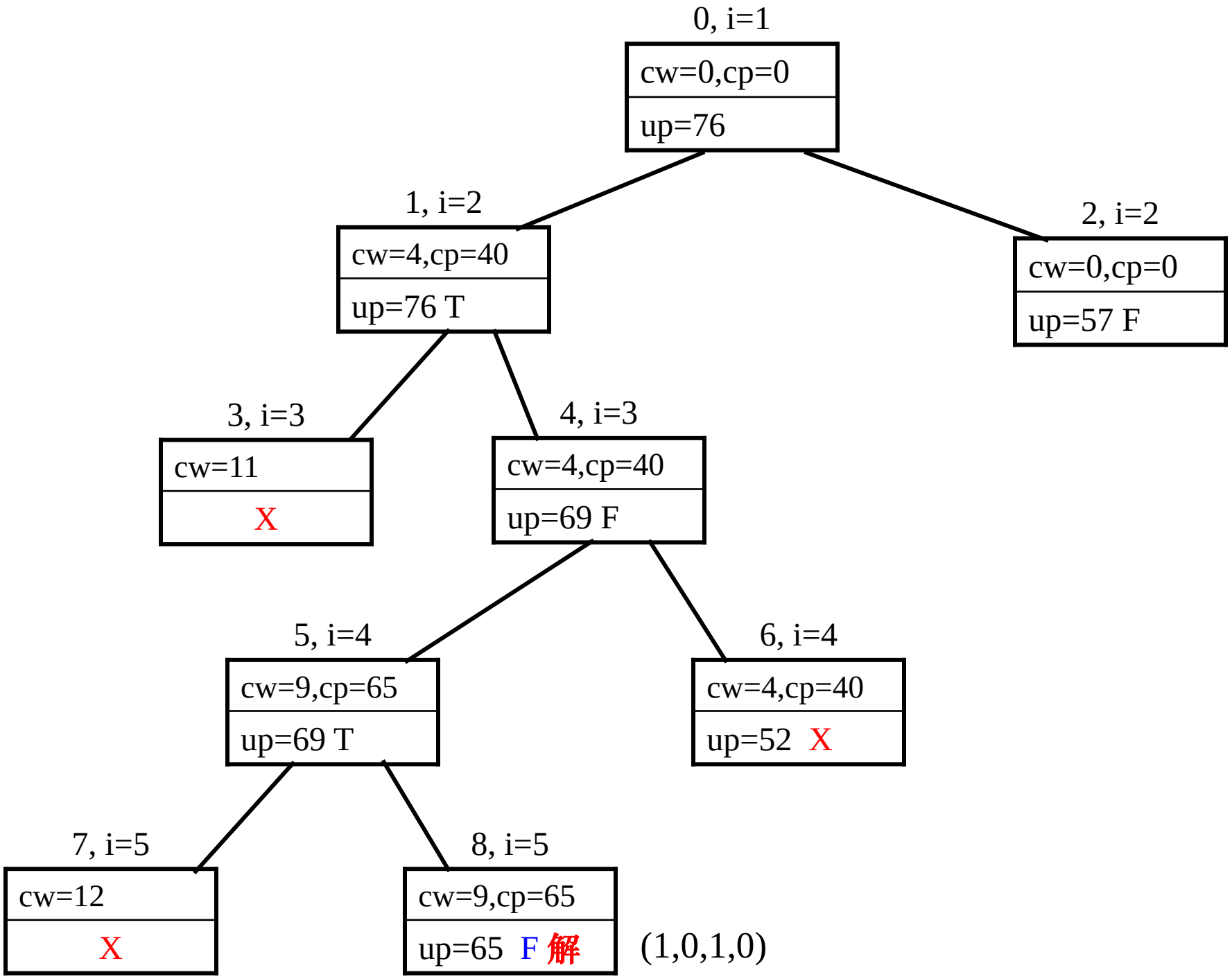
分支限界搜索过程

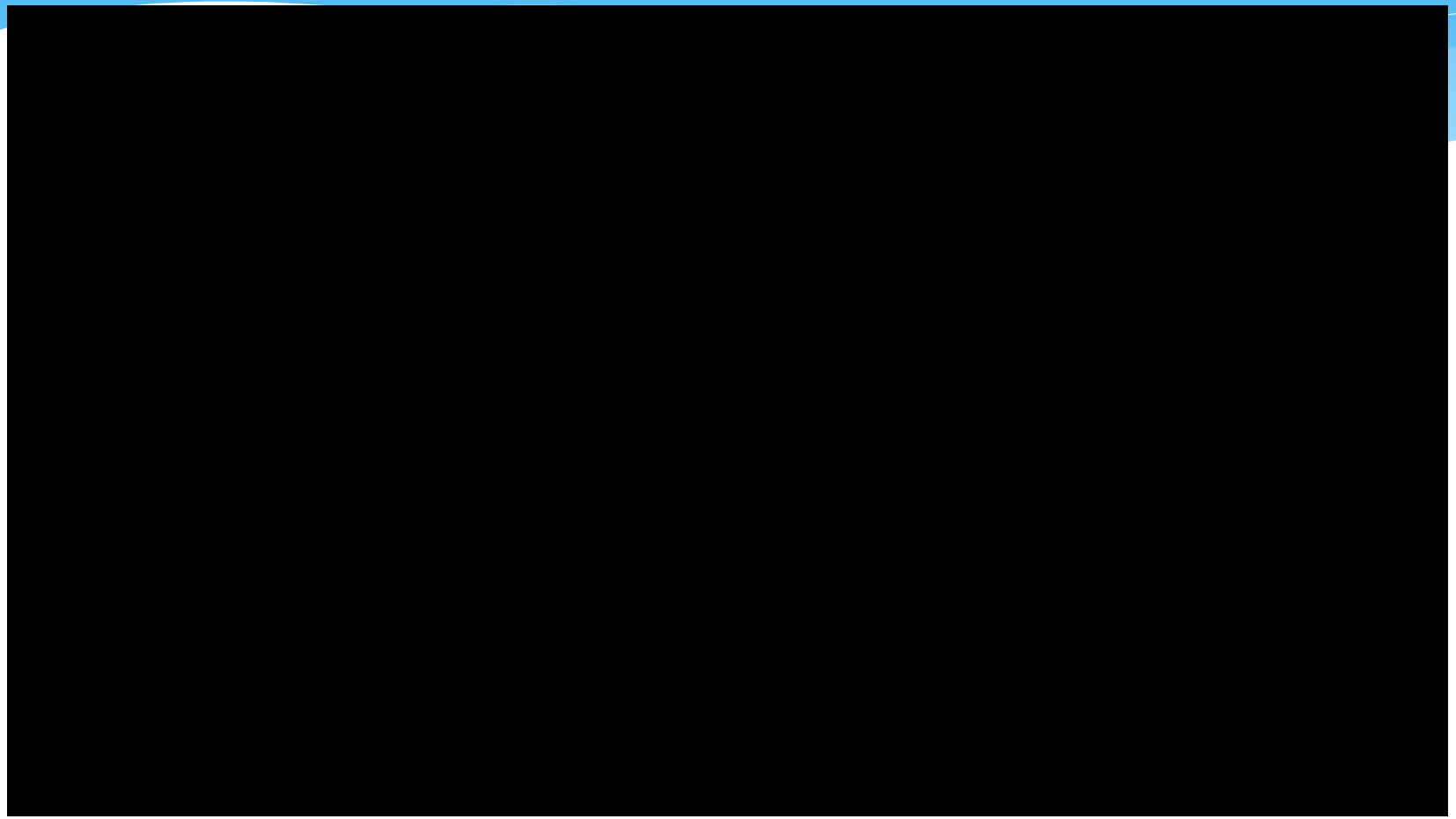
计算上界

```
//构造当前最优解
for(int j=n;j>0;j--){
    bestx[j]=(e.leftChild)?1:0;
    e=e.parent;
}
 return cp;
```

实例：背包的承重量W等于10。

物品	重量w	价值p	价值/重量
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4


0, i=1
cw=0,cp=0
up=76
1, i=2
cw=4,cp=40
up=76 T
2, i=2
cw=0,cp=0
up=57 F
3, i=3
cw=11
X
4, i=3
cw=4,cp=40
up=69 F
5, i=4
cw=9,cp=65
up=69 T
6, i=4
cw=4,cp=40
up=52 X
7, i=5
cw=12
X
8, i=5
cw=9,cp=65
up=65 F 解
(1,0,1,0)



视频来源：Bilibili　　Up：WAY_zhong

6.7 旅行售货员问题

* 一、问题描述
* 二、问题分析
* 三、算法描述

一、问题描述

- 某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小。
- 路线是一个带权图。图中各边的费用（权）为正数。图的一条周游路线是包括V中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。
- 旅行售货员问题的解空间可以组织成一棵树，从树的根结点到任一叶结点的路径定义了图的一条周游路线。旅行售货员问题要在图G中找出费用最小的周游路线。

二、问题分析

- 算法开始时创建一个最小堆，用于表示活结点优先队列。堆中每个结点的子树费用的下界lcost值是优先队列的优先级。接着算法计算出图中每个顶点的最小费用出边并用minout记录。如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法即告结束。如果每个顶点都有出边，则根据计算出的minout作算法初始化。
- 算法的while循环体完成对排列树内部结点的扩展。对于当前扩展结点，算法分2种情况进行处理：
  - 1、首先考虑s=n-2的情形，此时当前扩展结点是排列树中某个叶结点的父结点。如果该叶结点相应一条可行回路且费用小于当前最小费用，则将该叶结点插入到优先队列中，否则舍去该叶结点。
  - 2、当s<n-2时，算法依次产生当前扩展结点的所有儿子结点。由于当前扩展结点所相应的路径是x[0:s]，其可行儿子结点是从剩余顶点x[s+1:n-1]中选取的顶点x[i]，且(x[s]，x[i])是所给有向图G中的一条边。对于当前扩展结点的每一个可行儿子结点，计算出其前缀(x[0:s]，x[i])的费用cc和相应的下界lcost。当lcost<bestc时，将这个可行儿子结点插入到活结点优先队列中。

（续）

- 算法中while循环的终止条件是排列树的一个叶结点成为当前扩展结点。当s=n-1时，已找到的回路前缀是x[0:n-1]，它已包含图G的所有n个顶点。因此，当s=n-1时，相应的扩展结点表示一个叶结点。此时该叶结点所相应的回路的费用等于cc和lcost的值。剩余的活结点的lcost值不小于已找到的回路的费用。它们都不可能导致费用更小的回路。因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。
- 算法结束时返回找到的最小费用，相应的最优解由数组v给出。

三、算法描述

* 旅行售货员问题分支限界法算法描述
* 略

6.8 电路板排列问题

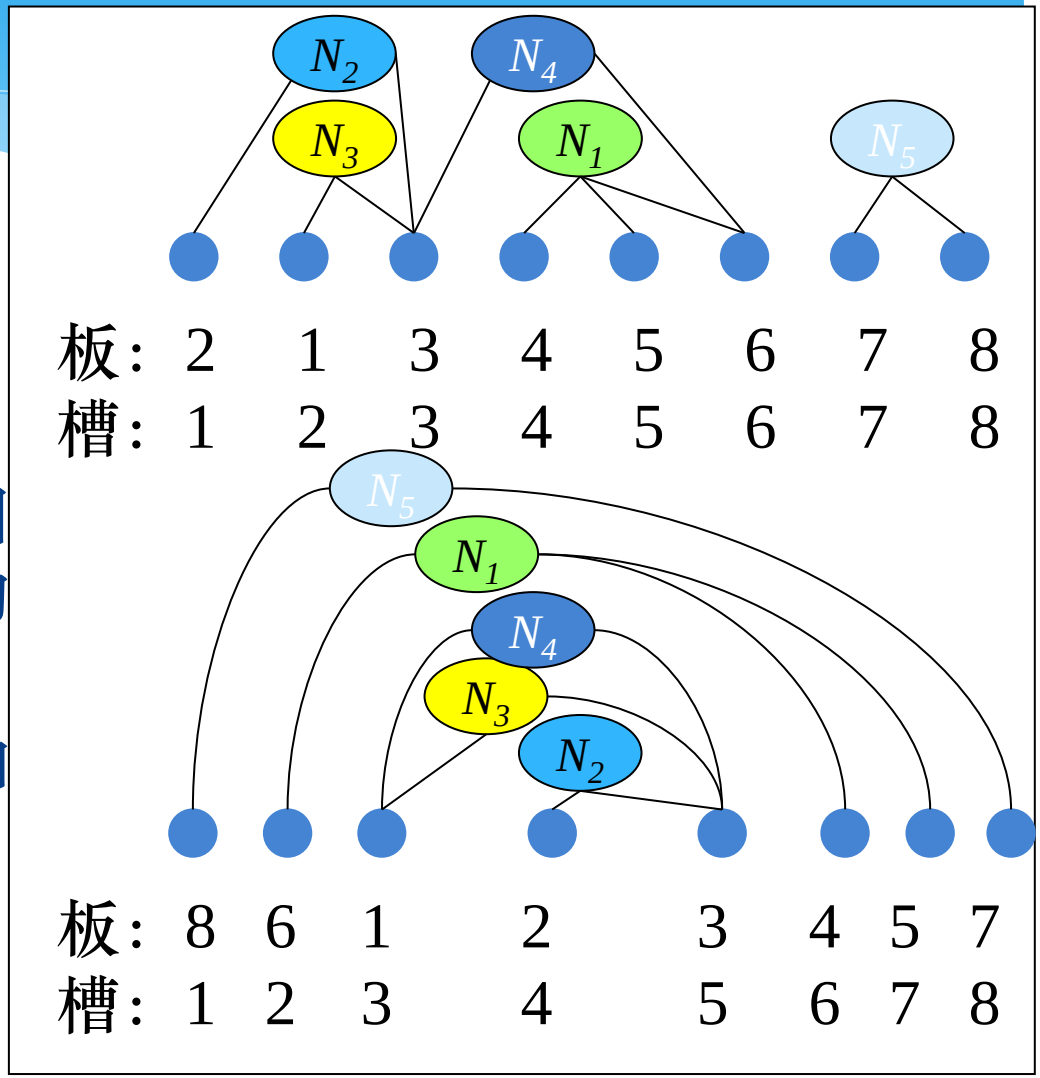
* 一、问题描述
* 二、问题实例
* 三、问题分析
* 四、算法描述

一、问题描述

- 电路板排列问题是大规模电子系统设计中提出的一个实际问题。
- 该问题的经典提法是：将n块电路板以最佳排列方式插入带有n个插槽的机箱中。n块电路板的不同排列方式对应于不同的电路板插入方案。
- 设B={1, 2, …, n}是n块电路板的集合，L={N1, N2, …, Nm}是连接这n块电路板中若干电路板的m个连接块。Ni是B的一个子集，且Ni中的电路板用同一条导线连接在一起。
- 设x表示n块电路板的一个排列，即在机箱的第i个插槽中插入的电路板编号是x[i]。x所确定的电路板排列Density (x)密度定义为跨越相邻电路板插槽的最大连线数。
- 在设计机箱时，插槽一侧的布线间隙由电路板排列的密度确定。因此，电路板排列问题要求对于给定的电路板连接条件，确定电路板的最佳排列，使其具有最小密度。

二、问题实例

- n=8, m=5
- B={1, 2, 3, 4, 5, 6, 7, 8}
- N1={4, 5, 6};
- N2={2, 3}; N3={1, 3};
- N4={3, 6}; N5={7, 8}
- 其中一个可能的排列如图所示，则该电路板排列的密度是2
- 另一种可能的组合如下如所示，密度?



三、问题分析

- 算法开始时，将排列树的根结点置为当前扩展结点。在do-while循环体内算法依次从活结点优先队列中取出具有最小cd值的结点作为当前扩展结点，并加以扩展。
- 首先考虑s=n-1的情形，当前扩展结点是排列树中的一个叶结点的父结点。x表示相应于该叶结点的电路板排列。计算出与x相应的密度并在必要时更新当前最优值和相应的当前最优解。
- 当s<n-1时，算法依次产生当前扩展结点的所有儿子结点。对于当前扩展结点的每一个儿子结点node，计算出其相应的密度node.cd。当node.cd<bestd时，将该儿子结点N插入到活结点优先队列中。

四、算法描述

```
do {// 结点扩展
 if (E.s == n - 1) {// 仅一个儿子结点
  int ld = 0; // 最后一块电路板的密度
  for (int j = 1; j <= m; j++)
   ld += B[E.x[n]][j];
  if (ld < bestd) {// 密度更小的电路板排列
   delete [] bestx;
   bestx = E.x;
   bestd = max(ld, E.cd);
  }
  else{// 产生当前扩展结点的所有儿子结点
   for (int i = E.s + 1; i <= n; i++) {
    BoardNode N;
    N.now = new int [m+1];
    for (int j = 1; j <= m; j++)
     // 新插入的电路板
     N.now[j] = E.now[j] + B[E.x[i]][j];
     int ld = 0; // 新插入电路板的密度
     for (int j = 1; j <= m; j++)
      if (N.now[j] > 0 && total[j] != N.now[j]) ld++;
     N.cd = max(ld, E.cd);
     if (N.cd < bestd) {// 可能产生更好的叶结点
      N.x = new int [n+1]; N.s = E.s + 1;
      for (int j = 1; j <= n; j++) N.x[j] = E.x[j];
      N.x[N.s] = E.x[i]; N.x[i] = E.x[N.s]; H.Insert(N);}
     else delete [] N.now;}
    delete [] E.x;
}
```

6.9 批处理作业调度问题

* 一、问题描述
* 二、限界函数
* 三、问题分析
* 四、算法描述

一、问题描述

* 给定n个作业的集合J={J1,J2,…,Jn}。每一个作业Ji都有2项任务要分别在2台机器上完成。每一个作业必须先由机器1处理，然后再由机器2处理。作业Ji需要机器j的处理时间为tji，i=1,2,…,n；j=1,2。对于一个确定的作业调度，设是Fji是作业i在机器j上完成处理的时间。则所有作业在机器2上完成处理的时间和f=$\sum F_{2i}$称为该作业调度的完成时间和。批处理作业调度问题要求对于给定的n个作业，制定最佳作业调度方案，使其完成时间和达到最小。

二、限界函数

* 在结点E处相应子树中叶结点完成时间和的下界是：

$$f \geq \sum_{i \in M} F_{2i} + \max\{S_1, S_2\}$$

* 注意到如果选择Pk，使t1pk在k>=r+1时依非减序排列，S1则取得极小值。同理如果选择Pk使t2pk依非减序排列，则S2取得极小值。

* 这可以作为优先队列式分支限界法中的限界函数。

三、问题分析

- 算法的while循环完成对排列树内部结点的有序扩展。在while循环体内算法依次从活结点优先队列中取出具有最小bb值（完成时间和下界）的结点作为当前扩展结点，并加以扩展。
- 首先考虑E.s=n的情形，当前扩展结点E是排列树中的叶结点。E.sf2是相应于该叶结点的完成时间和。当E.sf2 < bestc时更新当前最优值bestc和相应的当前最优解bestx。
- 当E.s<n时，算法依次产生当前扩展结点E的所有儿子结点。对于当前扩展结点的每一个儿子结点node，计算出其相应的完成时间和的下界bb。当bb < bestc时，将该儿子结点插入到活结点优先队列中。而当bb　bestc时，可将结点node舍去。

四、算法描述

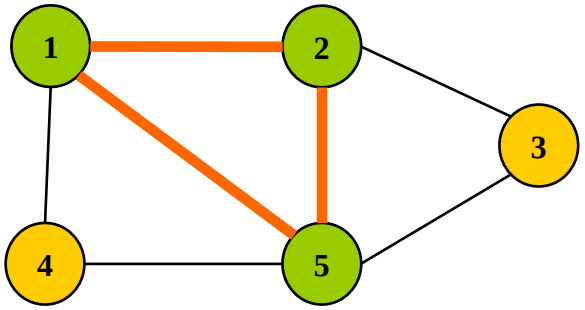
```
while (E.s <= n )
{  if (E.s == n ) // 叶结点
   { if (E.sf2 < bestc)
     {  bestc = E.sf2;
        for (int i = 0; i < n; i++)
        bestx[i] = E.x[i];}
     delete [] E.x;}
   else
   // 产生当前扩展结点的儿子
{  for (int i = E.s; i < n; i++)
   {  Swap(E.x[E.s],E.x[i]);
      int f1,f2;
      int bb=Bound(E,f1,f2,y);
      if (bb < bestc )
      {  MinHeapNode N;
         N.NewNode(E,f1,f2,bb,n);
         H.Insert(N);}
      Swap(E.x[E.s],E.x[i]);
   }
delete [] E.x;}  // 完成结点扩展
```

6.8 最大团问题

1. 问题描述

给定无向图G=(V，E)。如果U　V，且对任意u，v　U有(u，v)　E，则称U是G的完全子图。G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中。G的最大团是指G中所含顶点数最多的团。

下图G中，子集{1，2}是G的大小为2的完全子图。这个完全子图不是团，因为它被G的更大的完全子图{1，2，5}包含。{1，2，5}是G的最大团。{1，4，5}和{2，3，5}也是G的最大团。



2. 上界函数

用变量cliqueSize表示与该结点相应的团的顶点数；level表示结点在子集空间树中所处的层次；用cliqueSize +n-level+1作为顶点数上界upperSize的值。

在此优先队列式分支限界法中，upperSize实际上也是优先队列中元素的优先级。算法总是从活结点优先队列中抽取具有最大upperSize值的元素作为下一个扩展元素。

3. 算法思想

子集树的根结点是初始扩展结点，对于这个特殊的扩展结点，其cliqueSize的值为0。

算法在扩展内部结点时，首先考察其左儿子结点。在左儿子结点处，将顶点i加入到当前团中，并检查该顶点与当前团中其它顶点之间是否有边相连。当顶点i与当前团中所有顶点之间都有边相连，则相应的左儿子结点是可行结点，将它加入到子集树中并插入活结点优先队列，否则就不是可行结点。

分支限界法与回溯法的比较

* 相同点
  * 均需要先定义问题的解空间，确定的解空间组织结构一般都是树或图。
  * 在问题的解空间树上搜索问题解。
  * 搜索前均需确定判断条件，该判断条件用于判断扩展生成的结点是否为可行结点。
  * 搜索过程中必须判断扩展生成的结点是否满足判断条件，如果满足，则保留该扩展生成的结点，否则舍弃。

* 不同点
  * 搜索目标：回溯法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出在某种意义下的最优解。
  * 搜索方式不同：回溯法以深度优先的方式搜索解空间树，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树
  * 扩展方式不同：在回溯法搜索中，扩展结点一次生成一个孩子结点，而在分支限界法搜索中，扩展结点一次生成它所有的孩子结点。