6.1 分支限界法的基本思想

- 分支限界法类似于回溯法，也是一种在问题的解空间树T中搜索问题解的算法。
- 分支限界法求解目标：
  - 分支限界法找出满足条件的一个解，或某种意义下的最优解
- 分支限界法搜索方式：
  - 广度优先或最小耗费优先

◆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(最有利的)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。

常见的两种分支限界法

- **队列式(FIFO)分支限界法：** 按照队列**先进先出**（FIFO）原则选取下一个节点为扩展节点。
- **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。
  - **最大优先队列：** 最大堆，体现最大效益优先
  - **最小优先队列：** 最小堆，体现最小费用优先

从活结点表中**选择下一扩展结点**的不同方式

例 ：0-1背包问题

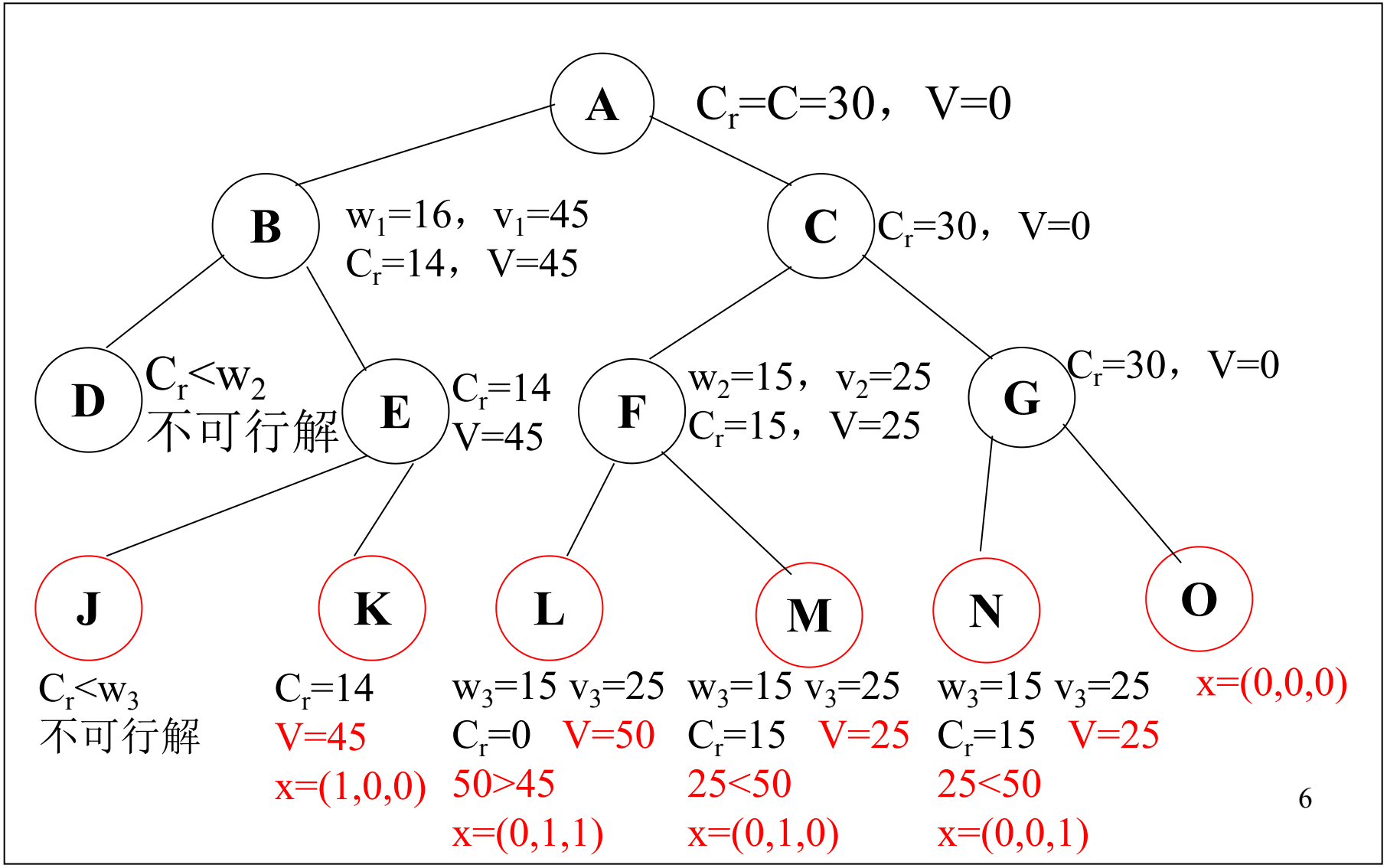
问题陈述 设有n个物体和一个背包,物体i的重量为w_i价值为p_i,背包的载荷为M, 若将物体i($1 \le i \le n$,)装入背包,则有价值为p_i .目标是找到一个方案, 使得能放入背包的物体总价值最高.

- 设N=3, W=(16,15,15), P=(45,25,25), C=30

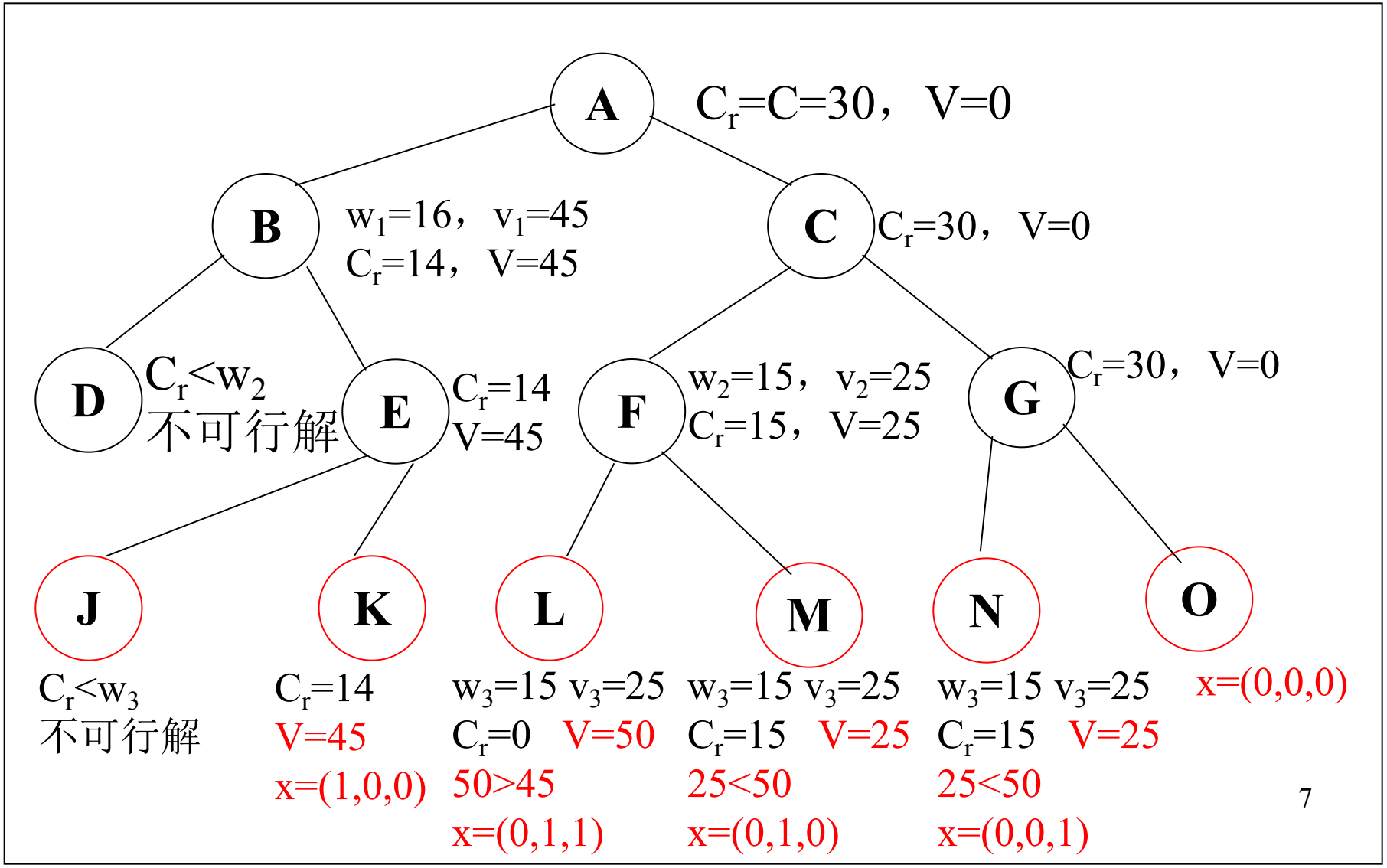
1.队列式分支限界法

2.优先队列式分支限界法

- 示例1 0-1背包问题（**队列式分支限界**）
- n=3, C=30, w={16, 15, 15}, v={45, 25, 25}



- 示例1 0-1背包问题（**最大优先**队列式分支限界）
- n=3, C=30, w={16, 15, 15}, v={45, 25, 25}



6.5 0-1背包问题

- 采用优先级队列式分支限界法
  - 使用最大堆，堆空算法中止。
  - 优先级为价值上界

```
class HeapNode   //堆结点定义
{
   public:
      double  uprofit;    //结点的价值上界
      double  profit;     //结点所相应的价值V
      double  weight;     //结点所相应的重量w
      int level;          //活结点在子集树中所处的层序号
      bbnode *ptr;        //指向活结点在子集树中相应结点
                          //的指针
};
```

```
bbnode{   //子集树结点定义
…
bbnode   *parent;  //指向父结点的指针
Bool   Lchild;       //左儿子结点标志
}
```

```
class Knap
{  private:
       bbnode *E;      //指向扩展结点的指针
       double c;       //背包容量C
       int n;          //物品总数N
       double *w;      //物品重量W数组
       double  *p;     //物品价值V数组
       double  cw;     //当前装包重量
       double  cp;     //当前装包价值
       int *bestx;     //最优解，从树根的遍历路径
      MaxHeap<Heapnode<double,double>> *H;
      void AddLiveNode(double up,double cp,double cw,bool ch,int level);
      double Bound (int i);
};
```

- 优先级的定义：（价值上界）
  - 首先将剩余物品依其单位重量价值排序
  - 然后依次装入物品，直至无法装入为止
  - 再装入该物品的一部分而装满背包

- 例如：n=4,c=7,p=[9,10,7,4],w=[3,5,2,1]这四个物品的单位价值[3,2,3.5,4]，先装入物品4，然后依次装入3，1，剩余背包容量1，装入0.2个物品2，得到最大价值22，可以证明其价值是最优值上界

➢ 活结点的优先级的定义上界Bound方法：

```
double Bound(int i)
   {   double  cleft=c-cw;     //剩余容量，初始值为当前背包剩余容量
       double  b=cp;           //价值上界，初始值为当前背包价值

      //以物品单位重量价值递减装填剩余容量
      while(i<=n&&w[i]<=cleft)
      {  cleft-=w[i];
         b+=p[i];
         i++;    }

      if(i<=n)  b+=p[i]/w[i]*cleft;     //装填剩余容量装满背包
      return b;
   }
```

注意: 上界不要求一定是可行解

背包问题的装填方式

将一个新的活结点插入到子集树和优先队列中

```
AddLiveNode(Typep  up, Typep  cp, Typew  cw, bool ch, int lev)
{
   bbnode *b = new bbnode;     //创建新结点,插入子集树
   b->parent = E;
   b->LChild = ch;

   HeapNode<Typep,Typew> N;    //建立堆结点,插入堆中
   N.uprofit = up;
   N.profit = cp;
   N.weight = cw;
   N.level  = lev;
   N.ptr = b;

   H->Insert(N);  }
```

- 分支限界搜索函数MaxKnapsack，需要使用与回溯法相似的剪枝函数加速搜索过程:
  - 左子树判断（可行性约束）：当前背包重量应当小于背包容量，
    - wt <= c
  - 右子树判断（上界约束）：当前结点最大价值的上界大于当前最优值，
    - up >= bestp

cw:当前装包重量
cp: 当前装包价值
bestp:当前最优解

```
MaxKnapsack{ …
while (i != n+1) { // 非叶结点
// 检查当前扩展结点的左儿子结点
    Typew wt = cw + w[i];       //当前重量
    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], true, i+1);}
     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, false, i+1);
//  取下一个扩展节点（略）
}
```

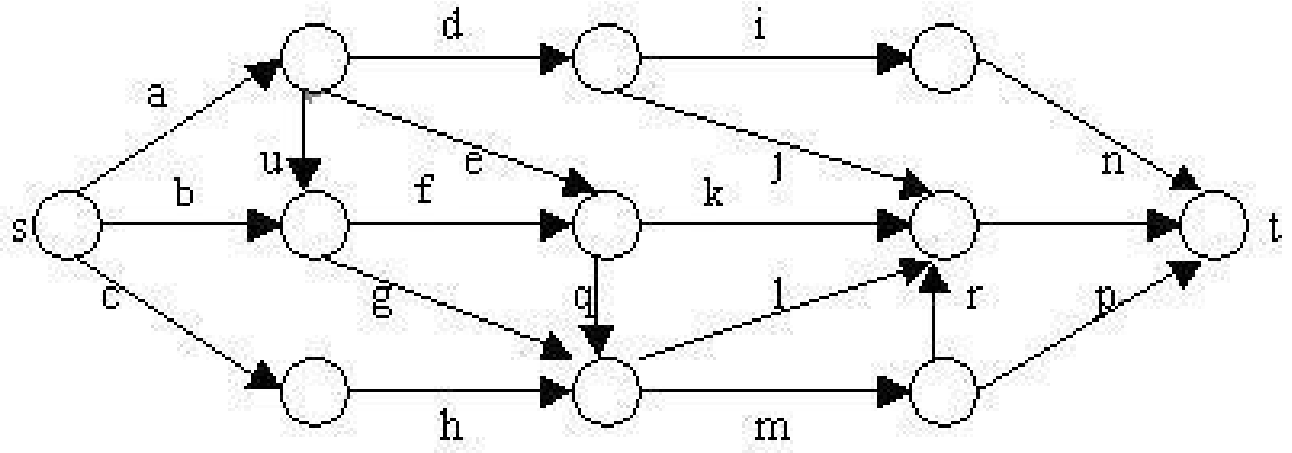
将一个新的活结点插入到子集树和优先队列中
i+1: 层数

不同点	回溯法	分支限界法
求解目标	找出树中满足约束条件的所有解	找出满足约束条件的一个解或找出使目标函数达到极大(小)的最优解
搜索方式	深度优先	广度优先或最小耗费优先
扩展结点	多次机会成为扩展结点：扩展结点变为活结点后又可成为扩展结点	每个活结点只有一次机会成为扩展结点
树结点的生成顺序	生成最近一个有希望结点的单个子女	选择其中最有希望的结点,并生成它的所有子女
行进方向	随机性	方向性：活结点表,搜索朝着解空间树上有最优解的分支推进

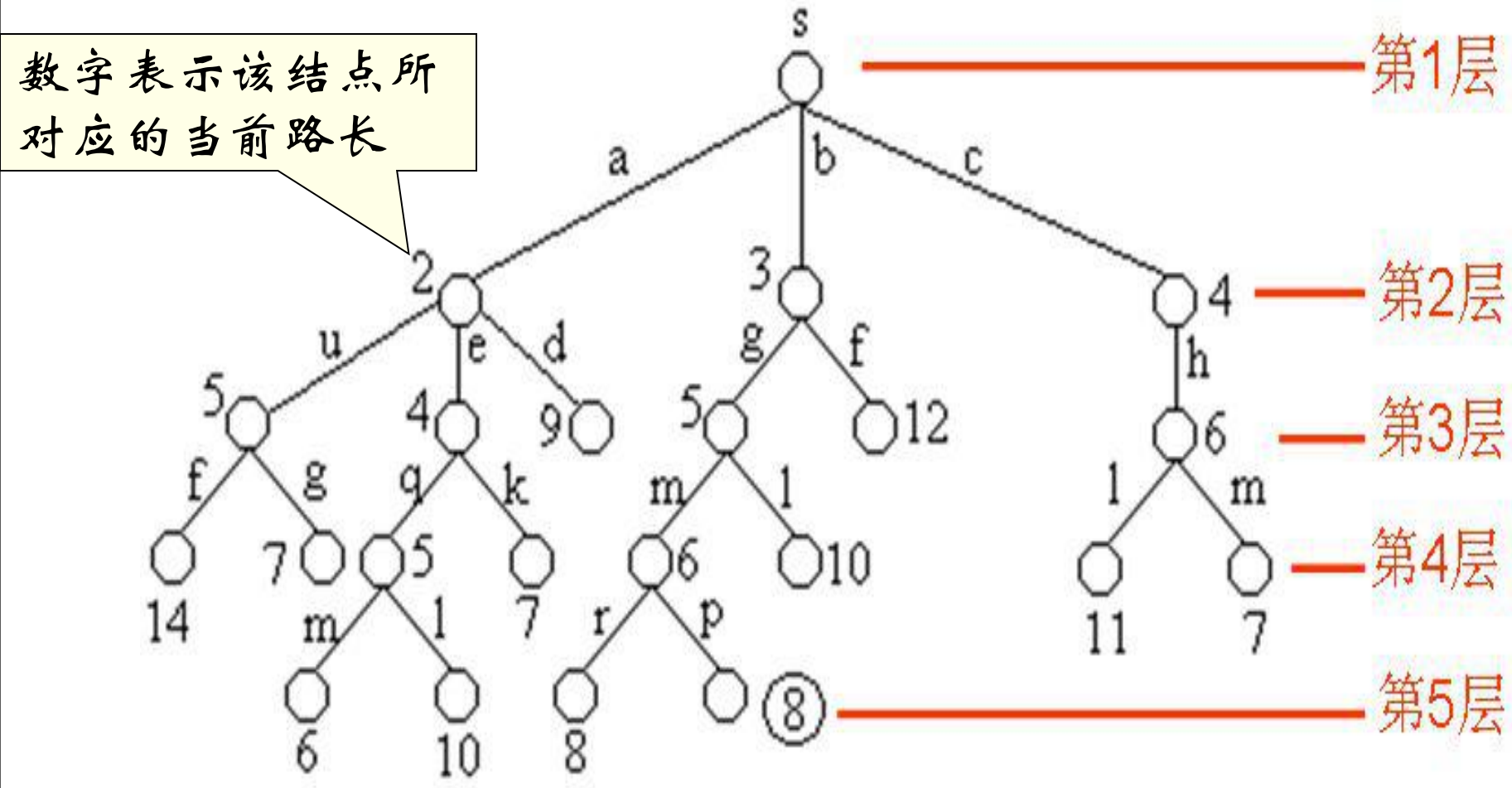
6.2 单源最短路径问题

➢问题描述

➢有向图G中，每一边都有一个非负权，要求图G的从源顶点s到目标顶点t之间的最短路径。



- 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。



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。
  - 开始：算法从图G的源顶点s和空优先队列开始。
  - 遍历:结点s被扩展后，它的儿子结点被依次插入堆中。

- 算法每次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j有边可达，且从源出发，途经i再到j的所相应路径长度，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。

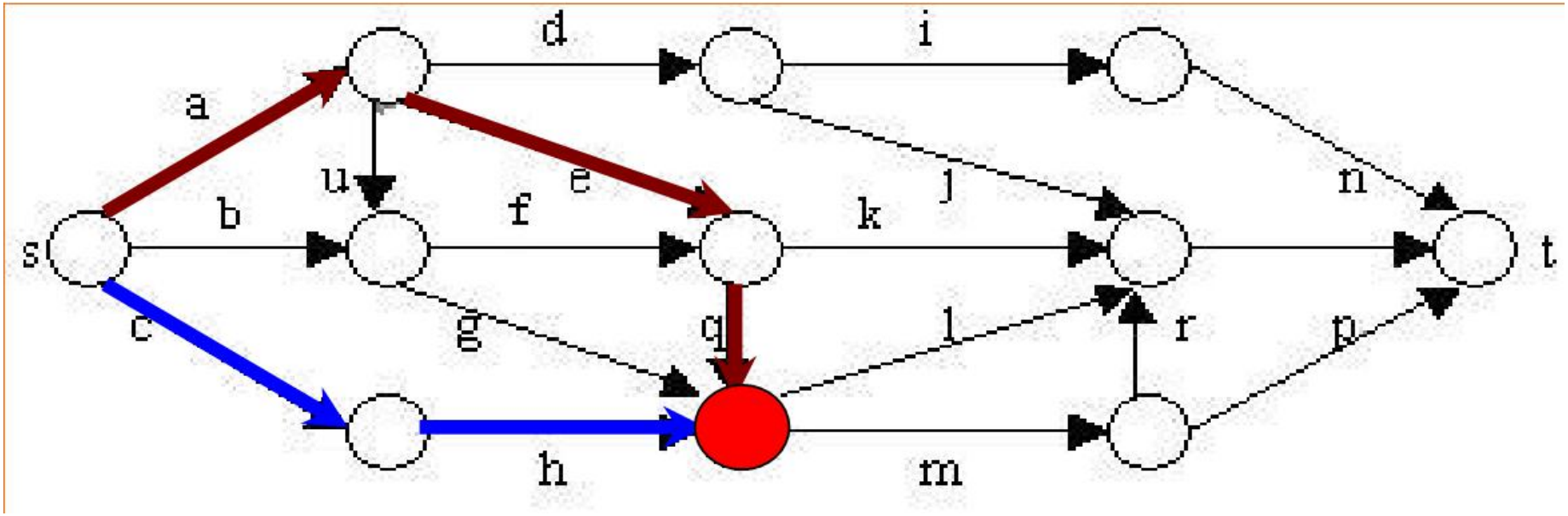
- 终止:结点扩展过程一直继续到活结点优先队列为空时为止

剪枝策略

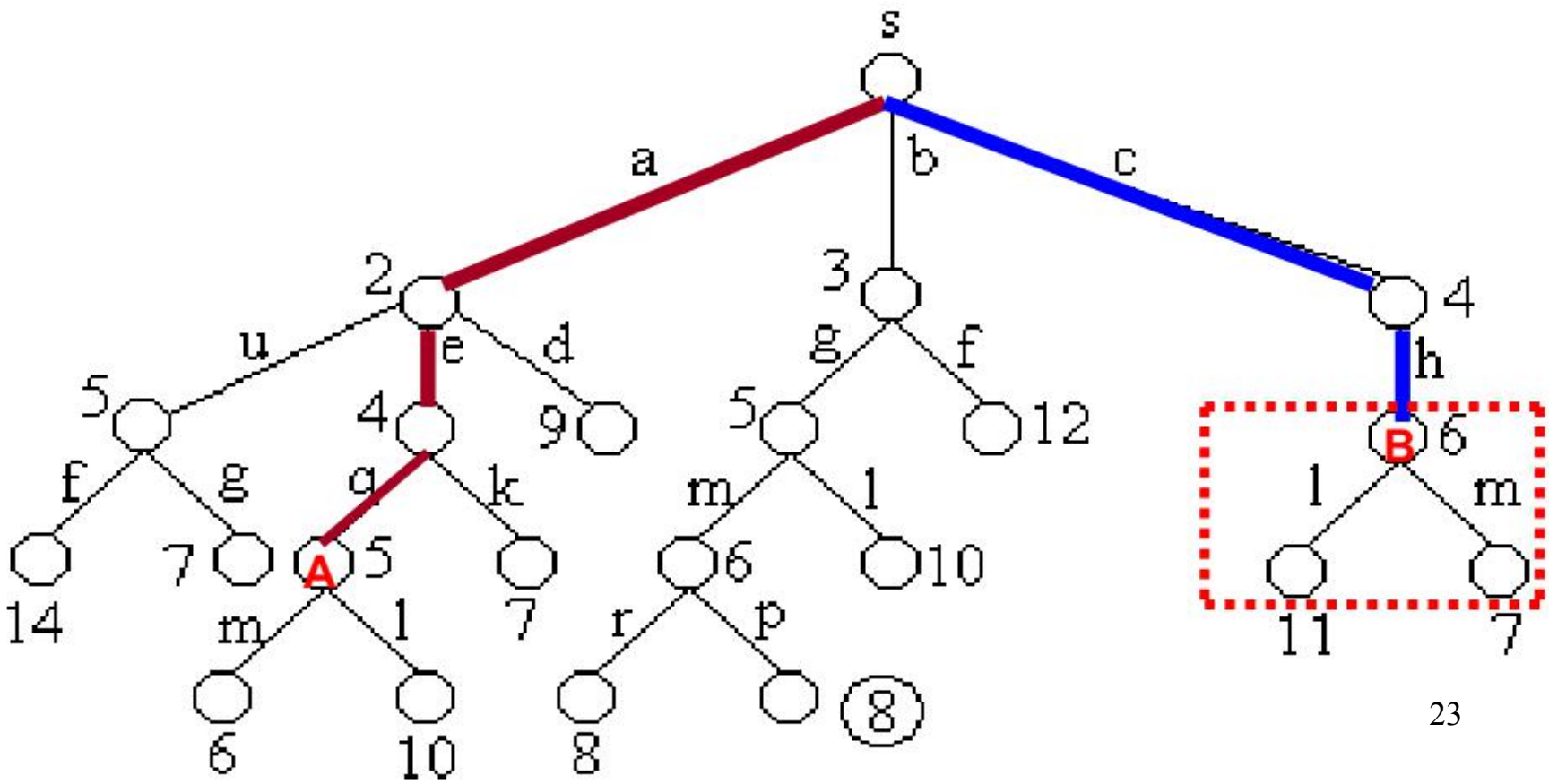
- 扩展结点过程中，一旦发现一个结点的下界不小于当前找到的最短路长，则算法剪去以该结点为根的子树。

- 利用结点间的控制关系进行剪枝。若从源顶点s出发，有2条不同路径到达图G的同一顶点。可将路长长的路径所对应的树中的结点为根的子树剪去。

- 例如，例中从s出发经边a、e、q（路长5）和经c、h（路长6）的两条路径到达G的同一顶点。



- 在解空间树中，这两条路径相应于解空间树的2个不同的结点A和B。
- 由于A的路长较小，故可将以结点B为根的子树剪去。此时称结点A控制了结点B。

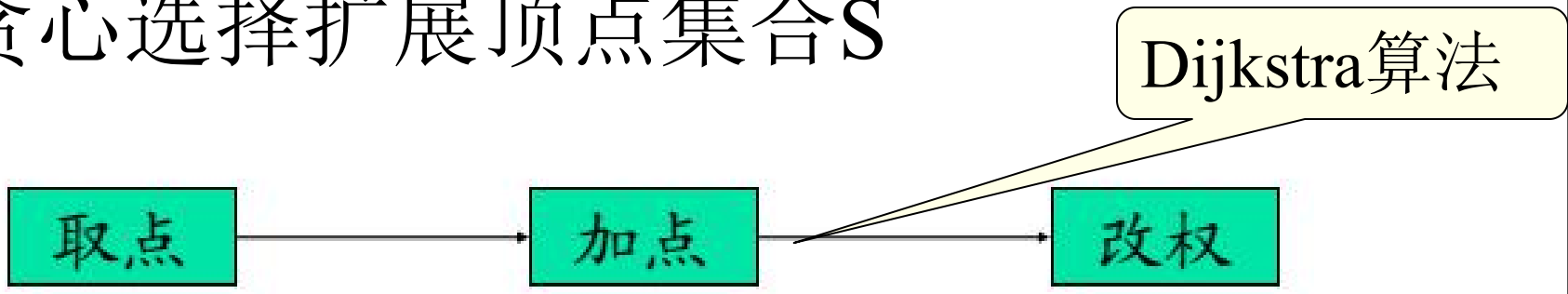


```
while (true) {
    for (int j = 1; j <= n; j++)
      if ((c[E.i][j]<inf)&&(E.length+c[E.i][j]<dist[j])) {
       // 顶点i到顶点j可达，且满足控制约束
          dist[j]=E.length+c[E.i][j];
      prev[j]=E.i;
      // 加入活结点优先队列
      MinHeapNode<Type> N;
      N.i=j;
      N.length=dist[j];
     H.Insert(N);
      }
   try {H.DeleteMin(E);} // 取下一扩展结点
   catch (OutOfBounds) {break;} // 优先队列空
}
```

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

Length: 优先队列的优先级

- 贪心选择扩展顶点集合S



- 活结点队列:最小堆H

广度优先
最短路径优先级优先
最小堆

剪枝

分支限界法

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C_1和C_2的轮船，其中集装箱i的重量为W_i，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。

✓容易证明：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。

- 首先将第一艘轮船尽可能装满；
- 将剩余的集装箱装上第二艘轮船。

2. 队列式分支限界法

- ✓解装载问题仅求出所要求的最优值的队列式分支限界法
- ✓定义：
  - ✓队列Q存放活结点表，其中元素
    - ✓Ew：表示当前载重量
    - ✓-1：表示队列已到达解空间树同一层结点的尾部
  - ✓方法EnQueue() :将活结点加入到Q队列中
  - ✓方法MaxLoading():对解空间的分支限界搜索

✓EnQueue()方法：将活结点加入到Q队列中

```
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
        Q.Add(wt);
}
}
```

它首先检查i是否等于n

- i=n：当前活结点为叶结点，则不必加入队列；
- i<n：当前结点为内部结点，则直接入队列

➢ 算法MaxLoading具体实施对解空间的分支限界搜索过程。

① 初始化的活结点队列为空，增加尾部标志

初始化

```
n=w.length-1;
bestw=0;     //当前最优载重量
Q=new ArrayQueue();    //活结点队列
Q.Add(-1);
int i=1;//当前扩展结点所处的层
int ew=0; //扩展结点所相应的载重量
```

② while循环搜索子集空间树:

- 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)

- 活结点队列中队首元素取出作为当前扩展结点，如果取出元素为-1，判断当前队列是否为空，如果非空，将尾部标记-1加入活结点队列，处理下一层活结点；否则程序结束。

```
while (true) {
    // 检查左儿子结点
    if (Ew + w[i] <= c)      // x[i] = 1
      EnQueue(Q, Ew + w[i], bestw, i, n); //将活结点加入到队列Q中
    // 右儿子结点总是可行的，将其加入到Q中
    EnQueue(Q, Ew, bestw, i, n);     // x[i] = 0

     Q.Delete(Ew);     // 取下一扩展结点
     if (Ew == -1) {     // 同层结点尾部
          if (Q.IsEmpty( )) return bestw;
          Q.Add(-1);       // 同层结点尾部标志
          Q.Delete(Ew);  // 取下一扩展结点
          i++;}                  // 进入下一层          }
}
```

3. 算法的改进

① 剪枝函数

- 定义:
  - bestw是当前最优解;
  - ew是扩展结点所相应的载重量;
  - r是剩余集装箱的重量。
- 当ew+r<=bestw时，可将其右子树减去。

② 其他改进

- 问题：
  - 算法MaxLoading设置初始时bestw=0，直到搜索到第一个叶结点才更新bestw。
  - 在搜索到第一个叶结点前，总有Ew+r>bestw，此时右子树测试不起作用。
- 改进：为确保右子树成功剪枝，应该在算法每一次进入左子树时更新bestw的值。

③ 算法

```
// 检查左儿子结点
  Type wt = Ew + w[
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) Q.Add(wt);
}
```

提前更新 bestw

```
// 检查右儿子结点
  if (Ew + r > bestw && i < n)
      Q.Add(Ew);  // 可能含最优解
Q.Delete(Ew); // 取下一扩展结点
```

右儿子剪枝

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。
- 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。

```
class QNode
{   QNode *parent;    // 指向父结点的指针
    bool  LChild;     // 左儿子标志
    Type weight;      // 结点所相应的载重量
}
```

- 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->LChild;    //bestx存储最优解路径
    bestE = bestE->parent;       //回溯构造最优解
}
```

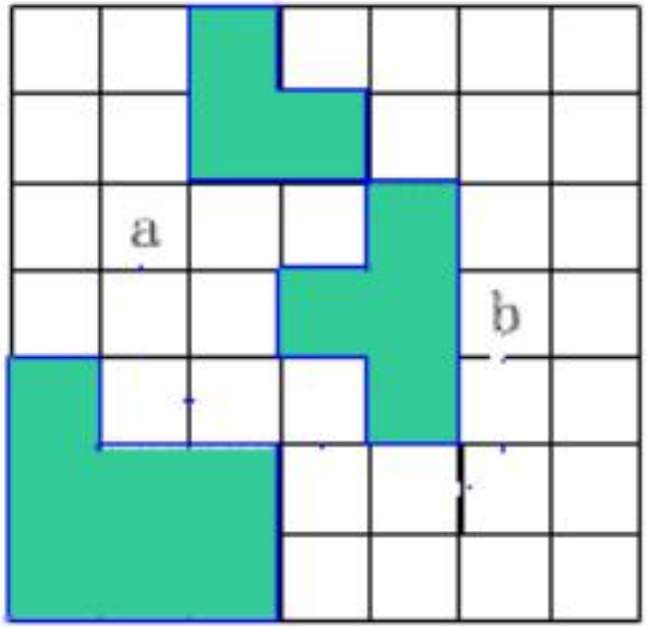
5. 优先队列式分支限界法

- 优先队列表示：最大优先队列
- 优先级：从根结点到当前结点x的路径所对应的载重量加上剩余集装箱的重量之和记为x.uweight
- 特性：
  - 结点x为根的子树中所有结点的载重量不超过x.uweight
  - 叶结点的载重量即为叶结点的优先级
- 所以只要有一个叶结点成为当前扩展结点，则该结点所对应的解必为最优解。此时可终止算法。

6.4 布线问题

1、问题描述

- 印刷电路板将布线区域划分为n×m个方格阵列。布线问题要求确定连接方格a的中点到方格b的中点的最短布线方案。



原图

- 要求：
  - 布线时电路只能沿直线或直角布线。
  - 已布线方格做上封闭标记，其他线路布线不允许穿过封闭区域。

算法思想——队列式分支限界法

① 初始情况从起始位置a开始，作为第一个扩展结点。

② 与该扩展结点相邻并可达的方格，作为可行结点加入到活结点队列中，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。

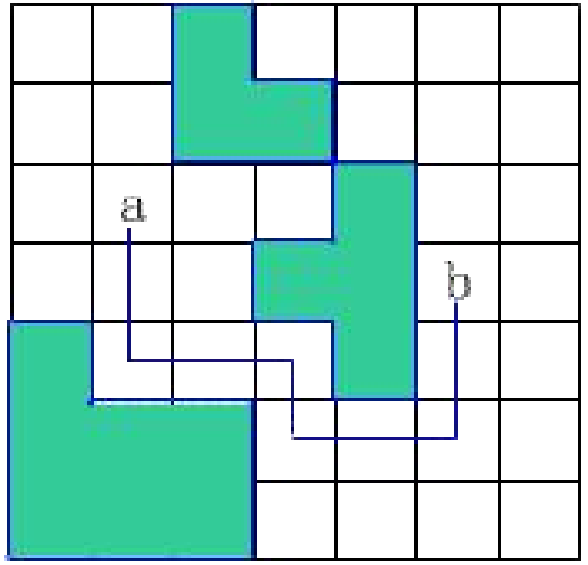
③ 算法从活结点队列中,取出队首结点作为下一个扩展结点，将与当前扩展结点相邻且未标记过的方格标记为2，并存入活结点队列。

④ 重复步骤②③，直到算法搜索到目标方格b，或活结点队列为空时为止。

- 一旦方格b成为活节点，表示找到了最优方案。

- **为什么这条路径一定就是最短的呢?**

- 由搜索过程的特点决定。假设存在一条由a至b的更短的路径，b一定会更早地被加入到活结点队列中，并得到处理。



(b) 最短布线路径

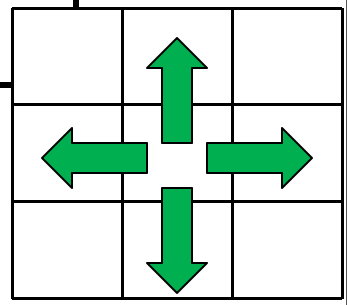
数据表示

① 方格位置类：Position

- 成员row和col：表示行和列

- 定义offset[4]，表示移动方向

移动i	方向	offset[i].row	offset[i].col
0	右	0	1
1	下	1	0
2	左	0	-1
3	上	-1	0



② 方格阵列数组：grid[][]记录距离

- 0表示允许布线，1表示封锁。
- 边界通过四周的封锁方格表示，增设方阵的围墙如下：

```
for(int i=0;i<=size+1;i++){
    //上下围墙
    grid[0][i]=grid[size+1][i]=1;
    //左右围墙
    grid[i][0]=grid[i][size+1]=1;
}
```

- **注意：起始位置的距离标记为2**

③ 其他成员变量的定义：

```
private  static int [][]grid;          //方格阵列
private  static int size;              //方格阵列大小
private  static ArrayQueue q;          //扩展结点队列
private  static Position start,        //起点
                        finish;        //终点
private  static Position []path;       //最短路经
private  static int pathLen;           //最短线路长度
```

算法实现

- 将当前扩展结点按照相邻方格顺序遍历，标记可达相邻方格
  - 如果该方格未被标记，更改标记距离；
  - 判断该方格是否为目标位置，是则退出循环
  - 将方格插入队列
- 判断是否到达目标位置，布线完成
- 判断队列是否为空，如果为空表示无解，算法终止。
- 依次循环取出下一扩展结点

```
Do{
    for (int i=0;i<numOfNbrs;i++){
        //以方向i进行移动，移动到nbr
        nbr.row=here.row+offset[i].row;
        nbr.col=here.col+offset[i].col;

        //若该位置未被布线
        if (grid[nbr.row][nbr.col]==0){
           grid[nbr.row][nbr.col]=grid[here.row][here.col]+1;
           if ((nbr.row==finish.row)&&(nbr.col==finish.col)) break;
           Q.add(nbr); }}

    if ((nbr.row==finish.row) && (nbr.col==finish.col)) break;
    if (q.isEmpty()) return false;     //无解
    Q.Delete(here);      //取下一个扩展结点
}while(true);
```

构造最优解

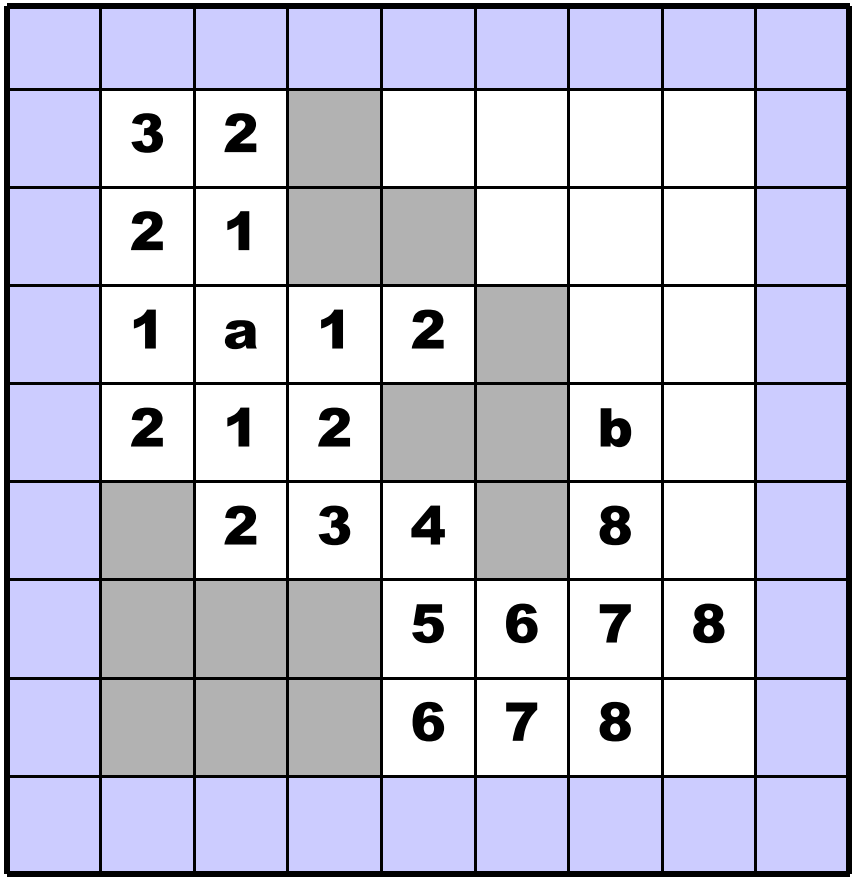
- 为了构造最短布线路径，需要从目标方格开始向起始方格回溯，逐步构造出最优解。
- 每次向标记距离比当前方格距离少1的相邻方格移动，直至到达起始方格时为止。
- 初始化：

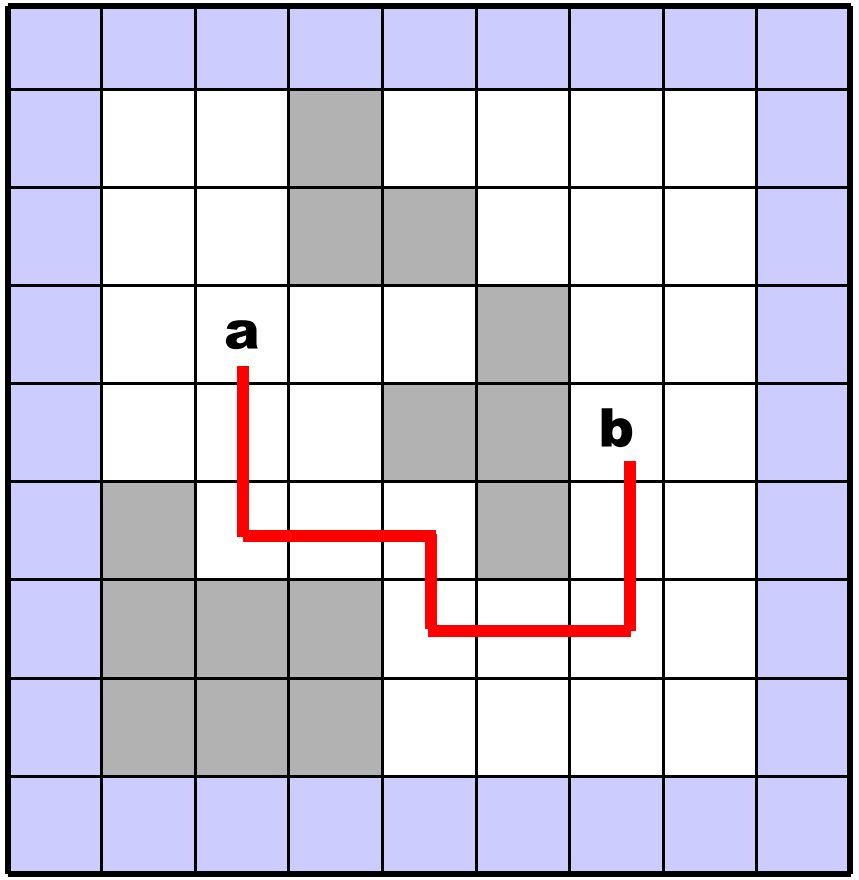
```
pathLen=grid[finish.row][finish.col]-2;
path = new Position[pathLen];
//从目标位置开始向起始位置回溯
here = finish;
```

```
for (int j=pathLen-1;j>=0;j--){
        path[j] = here;
        //找前驱结点，从当前位置向4个方向移动
        for (int i=0;i<numOfNbrs;i++){
            nbr.row=here.row+offset[i].row;
            nbr.col=here.col+offset[i].col;
           //若移动后方格距离为路线距离，则为路线上的方格
            if(grid[nbr.row][nbr.col]==j+2) break;}

       //把找到的路线上的新方格作为当前方格
        here =nbr;
    }
    return true;
}
```

实例：





■这个问题用回溯法来处理如何？

- 回溯法的搜索是依据深度优先的原则进行的。
- 如果把上下左右四个方向规定一个固定的优先顺序去进行搜索，搜索会沿着某个路径一直进行下去，直到碰壁才换到另一个子路径。
- 开始时，根本无法判断正确的路径方向，这就造成了搜索的盲目和浪费。

- 即使搜索到了一条由a至b的路径，根本无法保证它就是所有路径中最短的。
- 必须把整个区域的所有路径逐一搜索后，才能得到最优解。
- 因此，布线问题不适合用回溯法解决。

6.6 最大团问题

1. 问题描述

- 完全子图：
  - 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
- 团：
  - 当且仅当G的完全子图U不包含在G的更大的完全子图中则U是G的团。
- 最大团：
  - G的最大团是指G中所含顶点数最多的团。

2.算法设计

- **解空间：子集树。**最大团可看作G的顶点V的子集选取问题。
  - **可行性约束函数：**顶点i与已选入的顶点集中每一个顶点都有边相连。
  - **上界函数：**有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。
- **解决方案：优先级队列式分支限界法**

优先级定义

- 变量定义:
  - cn: 当前团的顶点数;
  - level: 结点在子集空间树中所处的层次;
- 优先级定义:
  - un=cn+n-level+1
  - 表示该结点为根的子树中最大顶点数的上界。

3. 算法思想

- 初始化：根结点是初始扩展结点，cn=0
- 算法while循环扩展内部结点，首先考察其左儿子结点。
  - 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。

```
//搜索子集空间树,非叶子结点
while (i!=n+1)
{
    //检查顶点i与当前团中其他顶点之间是否有边相连
    boolean ok = true;
    BBnode  *B =E;
    for (int j=i-1;j>0;B=B->parent,j--)
        if (B->LChild &&a[i][j]==0)
        {
            ok = false;
            break;
        }
    //左儿子结点为可行结点
    if (ok){
        if (cn+1>bestn) bestn=cn+1;
          addLiveNode(H,cn+n-i+1,cn+1,i+1,E,true);
    }
```

◆继续考察当前扩展结点的右儿子结点。当un>bestn时，右子树中可能含有最优解，此时将右儿子结点加入到子集树中,并插入到活结点优先队列中。

◆while循环的终止条件是: 遇到子集树中的一个叶结点(即n+1层结点)成为当前扩展结点。

◆对于子集树中的叶结点，有un＝cn。（un是当前团最大顶点数的上界，cn表示当前团的顶点数。）此时活结点优先队列中，剩余结点的un值均不超过当前扩展结点的un值。

```
        if (cn+n-i>=bestn)        //右子树可能含最优解
            addLiveNode(H,cn+n-i,cn,i+1,E,false);

        //取一下扩展结点
        H.DeleteMax(N);
        E=N.ptr;
        cn=N.cn;
        i=node.level;
    }
    //构造当前最优解
    for (int j=n;j>0;j--){
        bestx[j]=E.->Lchild
        E=E->parent;
    }
  return bestn;
}
```

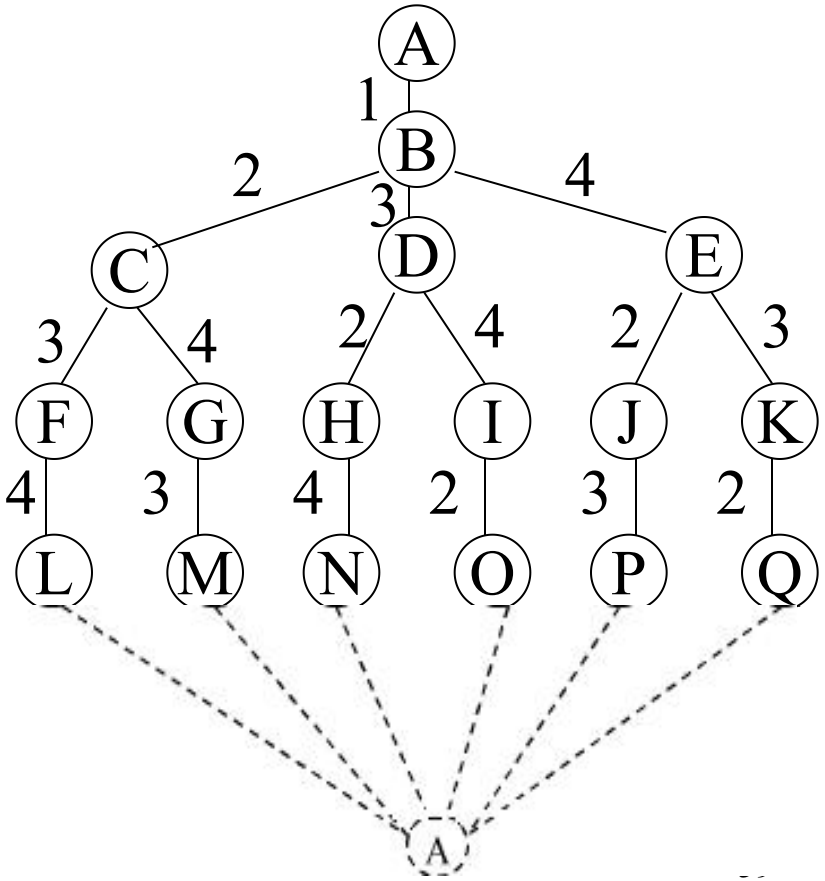
6.7 旅行售货员问题 1. 问题描述

- 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)。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为一棵排列树，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。**

- 旅行售货员问题是在图G中找出费用最小的周游路线。



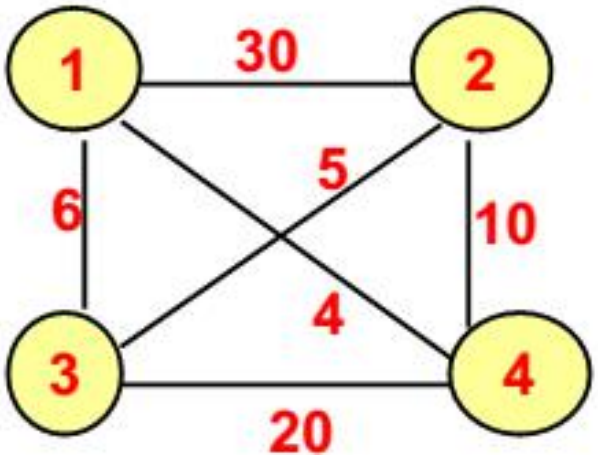
- 实现对排列树搜索的优先队列式分支限界法可以有两种不同的实现方式：

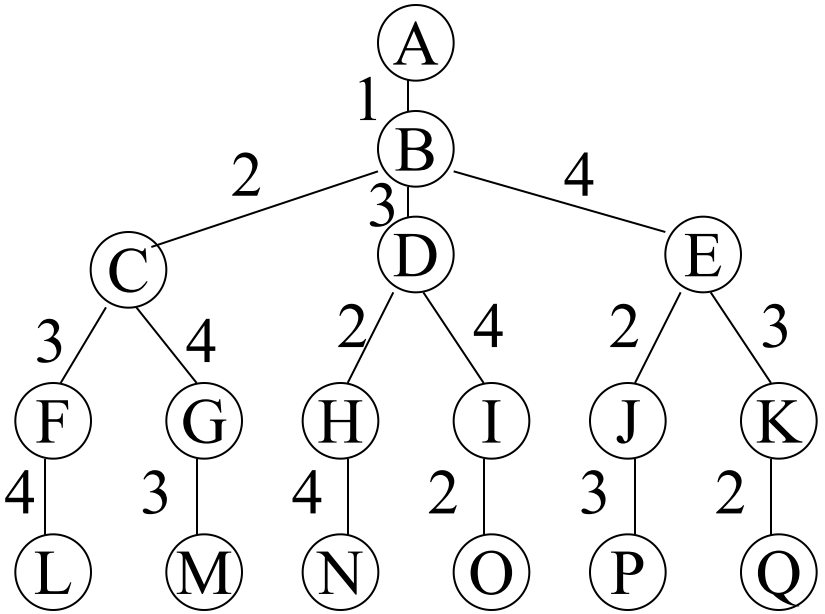
（1）用优先队列来存储活结点。优先队列中每个活结点都存储从根到该活结点的相应路径。

（2）用优先队列来存储活结点+当前已构造出的部分排列树。优先队列中的活结点不必存储从根到该活结点的相应路径，该路径必要时从存储的部分排列树中获得。

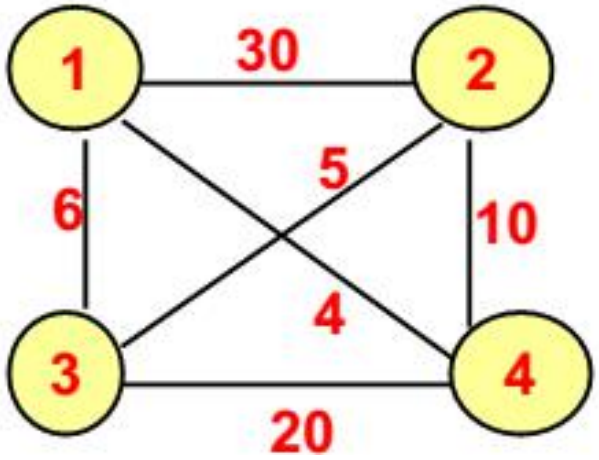
例 4城市旅行售货员问题

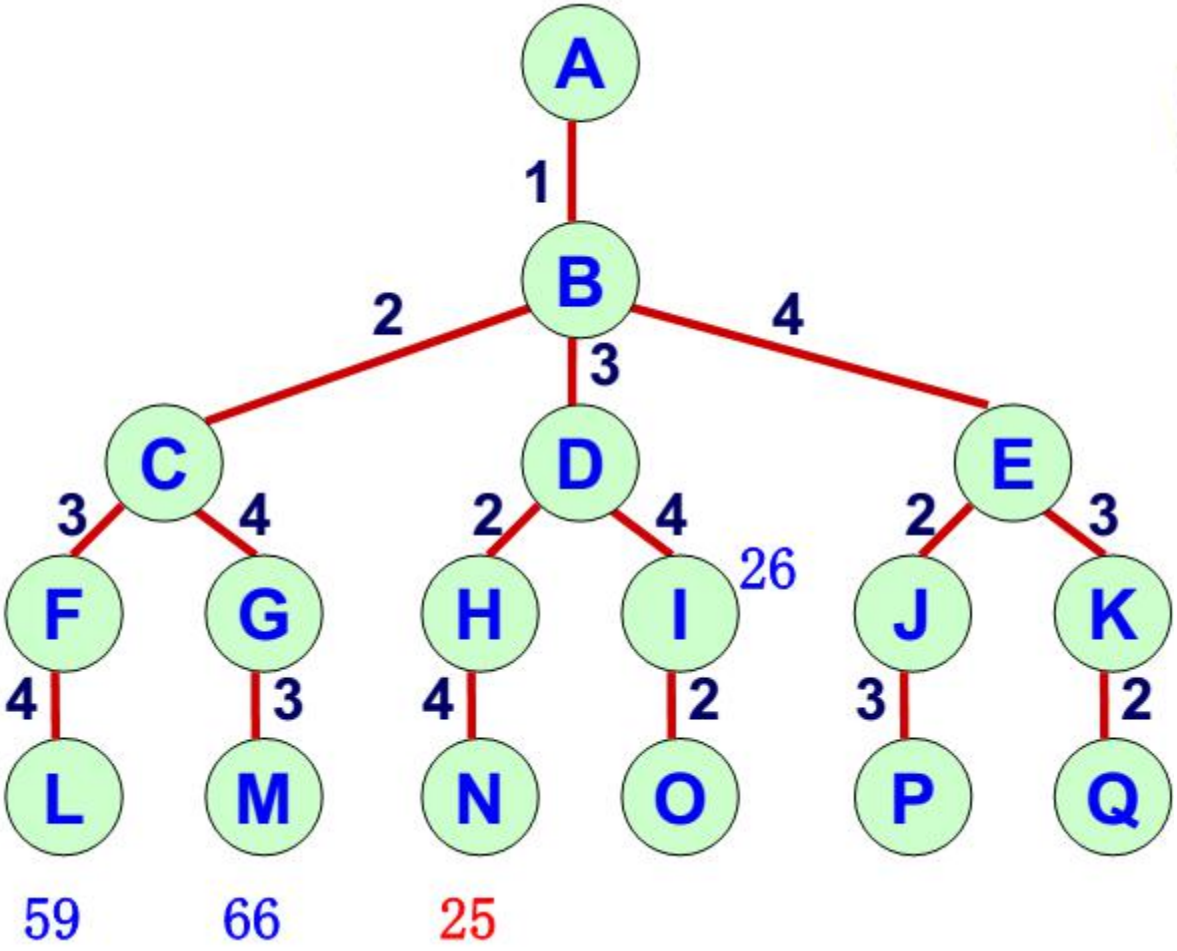
- 当n=4时的带权无向图，售货员从节点1开始经过每一个节点回到节点1的周游路线可以表示为右图的解空间树。





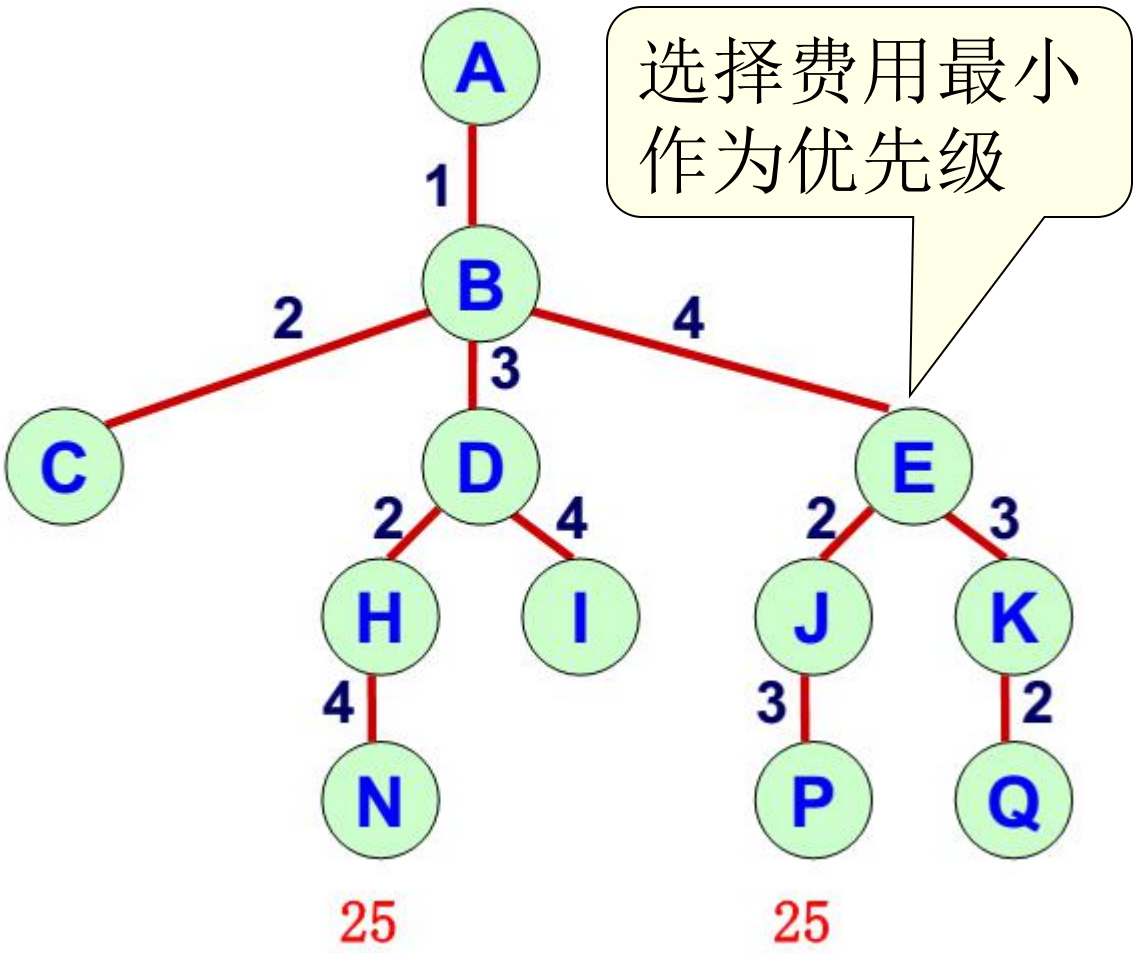
FIFO队列式分支限界法

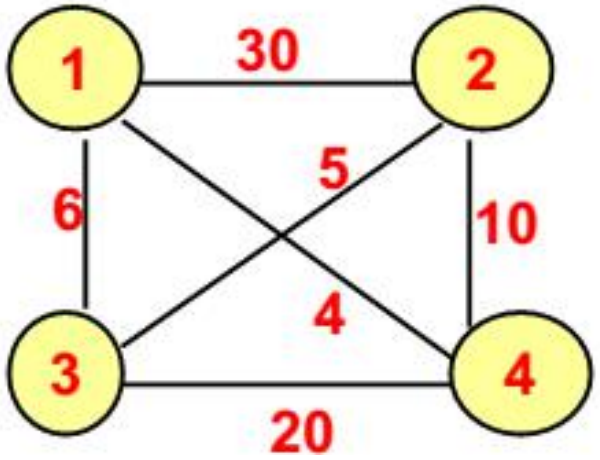




- ✓ B
- ✓ C, D, E
- ✓ F, G, H, I, J, K
- ✓ L, M, N, P, Q

优先级队列式分支限界法





- B
- E, D, C
- H, J, K, I, C
- N, P, I, Q, C

2. 算法描述

- 算法开始时创建一个最小堆，用于表示活结点优先队列。
- 定义：
  - x:当前解
  - s：结点在排列树中的层次，根结点到当前结点的路径为x[0:s];进一步搜索的顶点x[s+1:n-1]
  - cc:当前费用
  - lcost:子树费用的下界（优先级）
  - rcost: 是x[s:n-1]中顶点最小出边费用和

- 优先级定义(子树费用的下界)
  - 当前费用+剩余结点的最小出边费用和
    - lcost=cc+rcost
    - cc=0
    - Rcost=MinSum
  - 初始情况循环找出每个顶点的最小出边费用，遍历过程中依次减去当前扩展结点的最小出边费用

```
static class MinHeapNode  implements Comparable{
      float lcost,          //子树费用的下界
         cc,                //当前费用
         rcost;             //x[s:n-1]中顶点最小出边费用和
      int s;                //根结点到当前结点的路径为x[0:s]
      int  *x;              //需要进一步搜索的顶点是x[s+1,n-1]

class Traveling  {
  private:
     int n;
    float **a;
    float NoEdge;
   float cc;
   float bestc;
public :   float BBTSP(int v[]);
```

- 初始化：
  - 判断如果图中每个顶点有出边，计算出图中顶点的最小出边费用，用minout记录
  - 否则，如果图中某个顶点没有出边，图无回路算法结束。

```
//计算各顶点的最小出边费用和所有顶点最小出边费用和
for(int i=1;i<=n;i++)
{  float min=NoEdge;    //NoEdge=Float.MAX_VALUE
     for(int j=1;j<=n;j++) //计算顶点i的最小出边费用
          if (a[i][j]!=Nodge&&(a[i][j]<min||min==NoEdge)) min=a[i][j];
     if(min==NoEdge)return NoEdge; //若某一顶点无出边则无回路
     minOut[i]=min;
     minSum+=min;}
```

- 然后，初始化各个变量

```
MinHeapNode <Type> E;
E.x=new int[n];
For(int i=0;i<n;i++)
    E.x[i]=i;
E.s=0;
E.cc=0;
E.rcost=Minsum;
Float bestc=NoEdge;
```

S=n-1时，找到回路前缀是x[0:n-1],包含所有顶点，找到最优解，**叶结点所相应的回路的费用等于cc和lcost的值**。

- 搜索排列空间树，利用**while循环完成对排列树内部结点的扩展**。<
  - 当前扩展结点是叶结点的父结点，树层次s=n-2的情形
    - 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。
  - 其他情况，即当排列树层次s<n-2时，算法依次产生当前扩展结点的**所有儿子结点**。

```
While(E.s<n-1){//若扩展结点是叶子结点的父结点
    If(E.s==n-2)
    {//若叶子结点和结点和起始结点有边相连，且回路小于当前解
        if(a[x[n-2]][x[n-1]]!=NoEdge&& a[x[n-1]][1]!=NoEdge &&
        (E.cc+a[E.x[n-2]][E.x[n-1]]+a[E.x[n-1]][1]<bestc||best==NoEdge))
        {    //记录当前回路
            bestc=E.cc+a[E.x[n-2]][E.x[n-1]]+a[E.x[n-1]][1];
            E.cc=bestc;
            E.lcost=bestc;
            E.s++;
            heap.put(E);
        }}
```

- 其他情况，即当排列树层次s<n-2时，算法依次产生当前扩展结点的**所有儿子结点**。
  - 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。

```
else
{   //产生当前扩展结点的儿子结点
    for(int i=E.s+1;i<n;i++)
        if(a[E.x[E.s]][x[i]]!=NoEdge)   //若有边相连
        {   //cc为当前路径长度
            float cc=E.cc+a[E.x[E.s]][x[i]];
            //rcost为剩余结点的最小出边和
            float rcost=E.rcost-minOut[E.x[E.s]];
            //b为当前路径的长度下界
            float b=cc+rcost;
```

```
if(b<bestc){//若下界小于当前解，则可能含有更优解
//根据新扩展的结点创建堆结点
    MinHeapNode <Type> N;
  N.x=new int[n];
  for(int j=0;j<n;j++)
    N.x[j]=E.x[j];
  N.x[E.s+1]=E.x[i];
  N.x[i]=E.x[E.s+1];
  N.cc=cc;
  N.s=E.s+1;
  N.lcost=b;
  N.rcost=rcost;
  //将堆结点加入到最小堆
  heap.put(node);}}}
```