# 算法设计与分析

## 6. 分支限界法

# 本章主要知识点

6.1　分支限界法的基本思想

6.2　单源最短路径问题

6.3　装载问题：　　两种搜索队列

6.4　布线问题：　　搜索方式

6.5　0－1背包问题

6.6　最大团问题：　　优先级确定

6.7　旅行售货员问题：优先级确定

6.8　电路板排列问题

6.9　批处理作业调度

# 学习要点

- 理解分支限界法的剪枝搜索策略
- 掌握分支限界法的算法框架，会描述队列变化
  - （1）队列式(FIFO)分支限界法
  - （2）优先队列式分支限界法：会确定优先级
- 通过应用范例学习分支限界法的设计策略
- 侧重，掌握两类例子：子集树、排列树

# 分支限界法与回溯法

（1）求解目标：**基本相同**

回溯法：通过回溯找出满足约束条件的所有解或最优解

分支限界法：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：**不同**

回溯法：以深度优先DFS的方式搜索解空间树

分支限界法：以广度优先BFS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

# 分支限界法的基本思想

- 分支限界法以广度优先或以最小耗费优先的方式搜索问题的解空间树，扩展节点。
- 节点扩展方式

  1. 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，就一次性产生它的所有儿子结点。
  2. 对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

- 操作方式

从活结点表中取下一结点成为当前扩展结点。重复上述结点扩展过程，直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

（1）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

（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。

优先级：与节点有关的数值

最大（小）优先级：

最大（小）堆：

# 分支限界法问题的解空间

## 与回溯类似

1. 解空间树：子集树、排列树

2. 解向量：问题解的向量表示形式。

$(x_1, x_2, \ldots, x_k)$　$k \le n$，n:问题的规模。

3. 约束条件

1）显式约束：对分量$x_i$的取值限定。

2）隐式约束：为满足问题的解而对不同分量之间施加的约束。

4. 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

# 分支限界法问题的解空间（续1）

4、状态空间树：用于形象描述解空间的树。

子集树和排列树

5、搜索方式：构造队列、优先级队列

6、目标函数与最优解

（1）目标函数：衡量问题解的“优劣”标准。

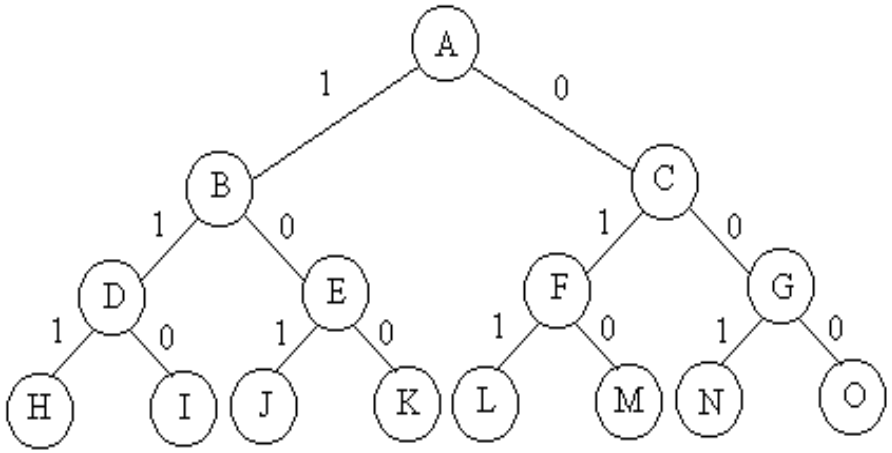
（2）最优解：使目标函数取极（大/小）值的解。

7、通过剪枝加速搜索

剪枝函数给出每一个可行节点相应的子树可能获得的最大价值的上界。若这上界不比当前最优值更大，则剪枝。

# 0-1背包问题

例：n=3时，0-1背包问题用完全二叉树表示的解空间

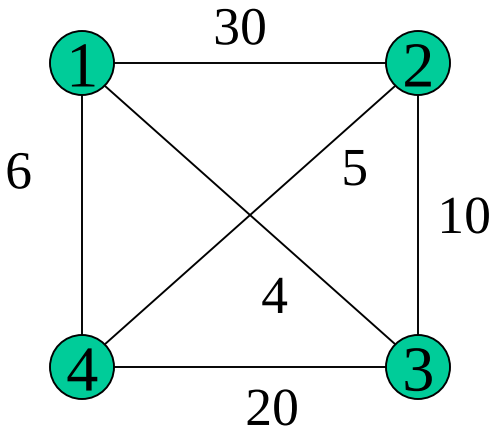


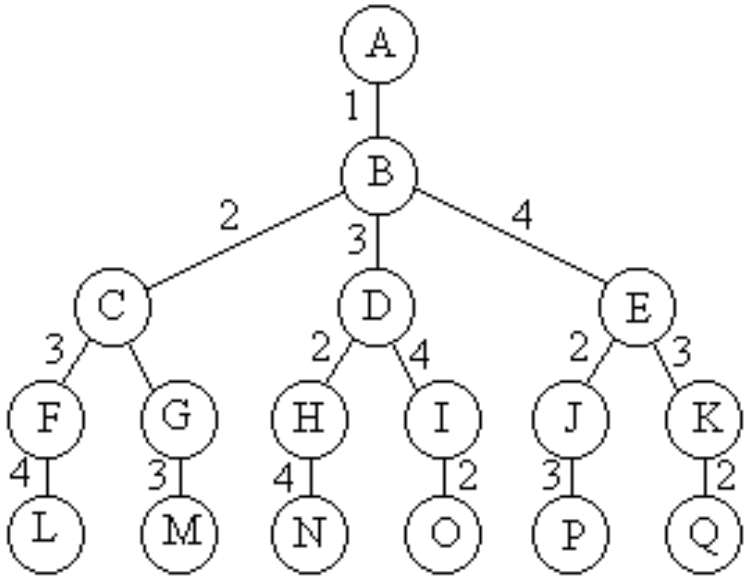

简单搜索举例：

n=3,c=30, w=[16,15,15], p=[45,25,25]

# 例：四城市的TSP问题

**简单搜索举例：**



# 子集树与排列树算法框架

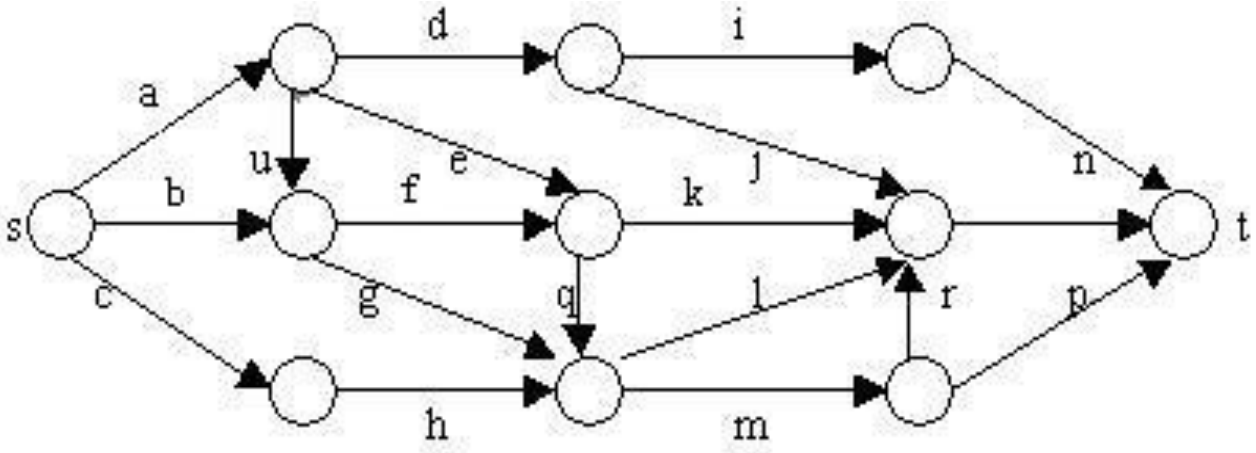
```
子集树：//r是树T的根
void branchBound (){
   Q={ };   //记Q为队列或堆
  for(r的2个儿子节点w)
     if (w是活结点) w入队列Q;
     else 舍去w;
   while (Q!=∅){
      取Q的头元素t;
      对t的2个儿子节点w {
      if (w是叶结点) 计算最优值;
      else
     { if (w是好活结点) w入队列Q;
       else 舍去w; }
    }
}
```

```
排列：//r是树T的根
void branchBound (){
   Q={ }; //记Q为队列或堆
  for(r的所有儿子节点w)
     if (w是活结点) w入队列Q;
     else 舍去w;
   while (Q!=∅){
      取Q的头元素t;
      对t的所有儿子节点w{
       if (w是叶结点) 计算最优值;
      else{ if (w是好活结点)
                 w入队列Q;
            else 舍去w; }
     }
}
```

# 6.2单源最短路径问题-自学

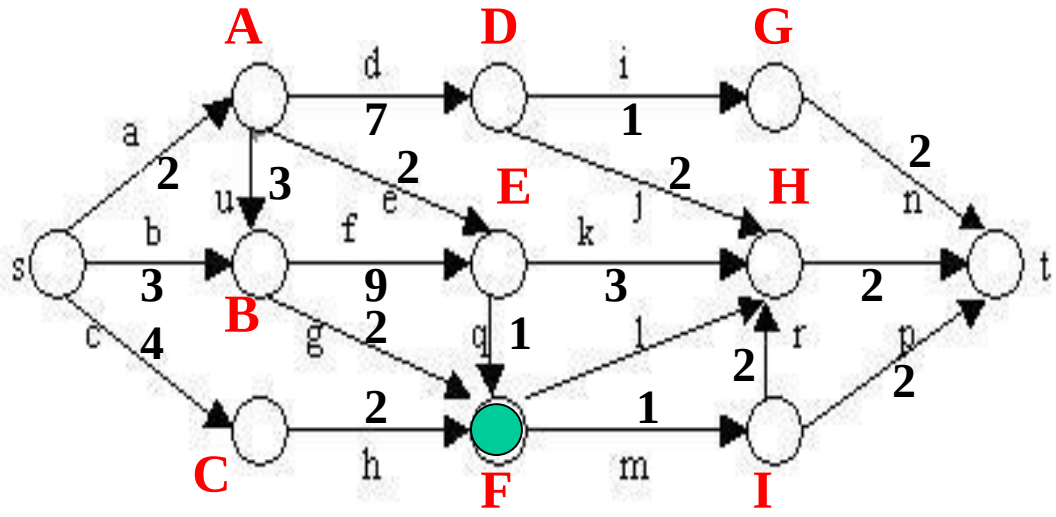
# 1. 问题描述

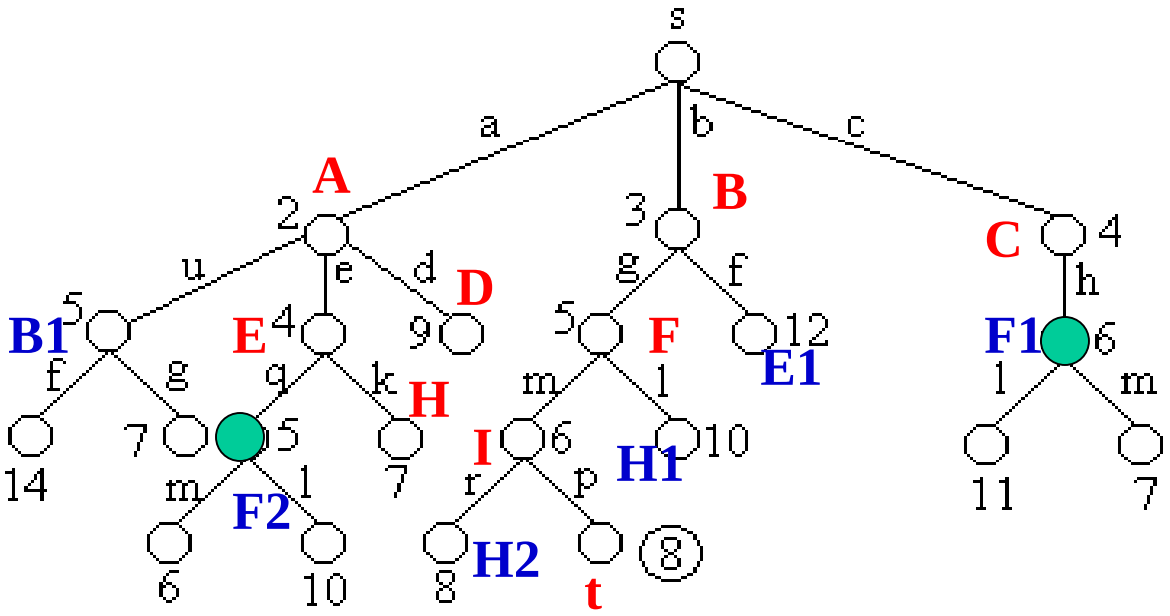
- 单源最短路径问题：在下图所给的有向图G中，每一边都有一个非负边权。要求图G的从源顶点s到目标顶点t之间的最短路径



# 算法分析

- 用极小堆Q存放结点
- 用优先队列式分支限界法
- 每一个结点旁边的数字表示该结点所对应的当前路长。见图



# 算法思想

优先队列式分支限界法：用极小堆Q来存储活结点表

**优先级:** 结点所对应的当前路长最小者优先。

算法过程：

1. 取图G的源顶点s，Q=∅。
2. 扩展结点s，它的儿子结点被依次插入堆Q中。
3. 从Q中取出当前具有最小路长的点i作为当前扩展结点，对与i相邻的j, dist[i]+c[i][j]< dist[j]，则将j作为插入到活结点优先队列Q中。
4. 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。

# 剪枝策略

1. 在算法扩展结点的过程中，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。
2. 利用结点间的控制关系进行剪枝：从源顶点s出发，2条不同路径到达图G的同一顶点，如两条路径的路长不同，可以将较长路径所对应的树中的结点为根的子树剪去。

# 关键部分算法

```
while (true){      // 搜索问题的解空间
  for (int j=1;j<=n;j++)
    if(c[E.i][j] <max &&  E.length+cE.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];  //修正数值
        prev[j]=E.i;          //确定节点j的父节点
        MinHeapNode node = new MinHeapNode(j,dist[j]);
        heap.Insert(node);     // 加入活结点优先队列
     }
     if (heap.isEmpty()) break;
     else enode = (MinHeapNode) heap.removeMin();
}
```

# 6.3 装载问题

# 1. 问题描述

- 有一批共个集装箱要装上2艘载重量分别为C1和C2的轮船，其中集装箱i的重量为$w_i$，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，则采用装载策略

(1)首先将第一艘轮船尽可能装满；

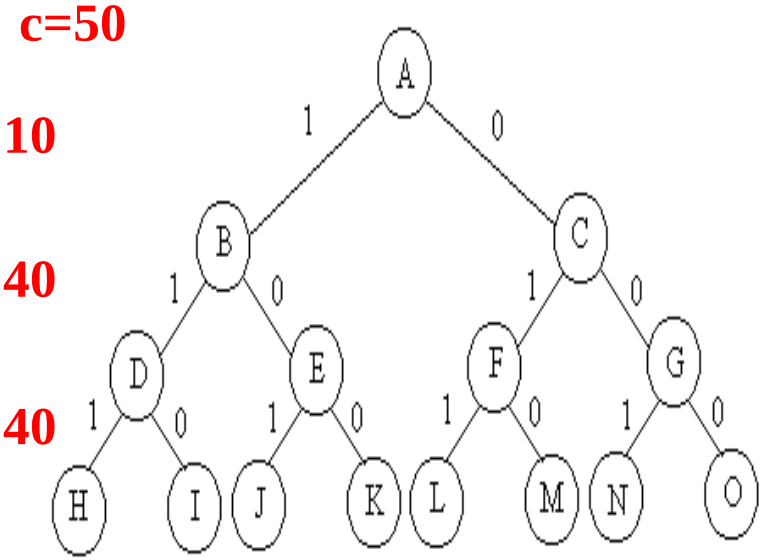
(2)将剩余的集装箱装上第二艘轮船。

# 2. 队列式分支限界法

- 循环操作。先检测当前扩展结点u的左儿子L，如可行，将L加入到Q中。右儿子R必可行，将R加入Q中。2个儿子结点都产生后，舍弃u。必要时添加一个层尾部标记-1 。

例：n=3，c=5，w[]={10,40,40}

- 从Q中取出队首元素作为当前扩展结点。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Q是否为空。如Q非空，则将尾部标记-1加入Q，算法开始处理下一层的活结点。



# MaxLoading实现对解空间的限界操作

c：载重量　w[i]：第i个集装箱重量

Ew：扩展结点所相应的载重量　　bestw：当前最优载重量

函数EnQueue：将活结点加人到活结点队列Q中

```
while (true) {   // MaxLoading部分算法
    if (Ew + w[i] <= c)                  // 检查左儿子结点。
      EnQueue(Q, Ew + w[i], bestw,i);   //如可行，左儿子入队
    EnQueue(Q,Ew, bestw, i);          //右儿子结点总是可行的，入队
    Q.delete(Ew);          // 取下一扩展结点
    if (Ew == -1)   {
      if (Q.isEmpty())    return bestw;
      Q.Add(-1); Q.delete(Ew); // 同层结点尾部标志，取下一扩展结点
      i++;                   // 进入下一层
    }
}
```

•约束函数：Ew + w[i] <= c

# 3. 算法的改进-剪右支

- 进入结点队列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：当前最优解；r：剩余集装箱的重量。
- Ew：当前扩展结点所相应的重量；
- 约束函数：Ew + w[i] <= c，

  当 Ew + w[i]> c，可将其左子树剪去。
- 限界函数：Ew+r>bestw，

  如Ew+r≤bestw将右子树剪去。
- 为确保右子树成功剪枝，在算法每一次进入左子树的时候更新bestw的值。

# 算法的改进-部分算法

右儿子剪枝

```
// 检查左儿子结点
    int wt = Ew + w[i];
    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);
    }
```

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

提前更新bestw

bestw:当前最优解

Ew:当前扩展结点所相应的重量

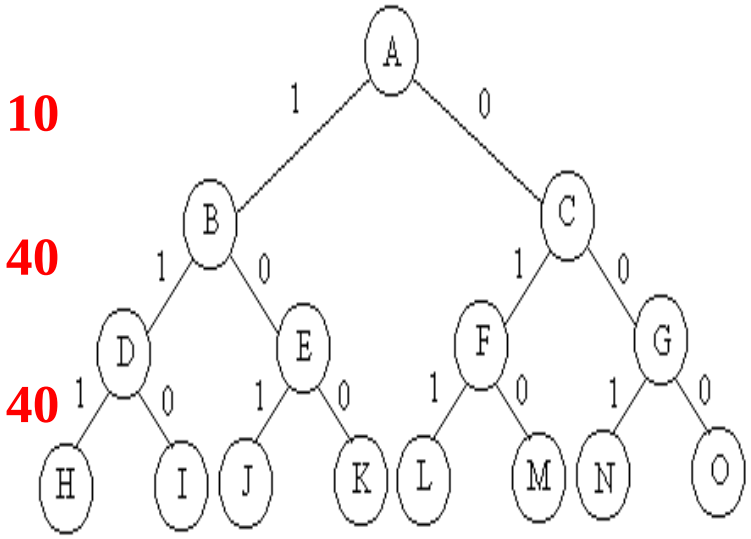
r:剩余集装箱的重量。

# MaxLoading:队列式分支限界法-完整算法

```
// 初始化
Queue Q;    //活结点队列
Q.Add(-1);     //同层结点尾部标志
int i=1;      //当前扩展结点所处的层
int bestw=0,   //当前最优载重量
        Ew= 0，//扩展结点所相应的载重量
         r=0;          //剩余集装箱重量
for(int j=2; j <=n;j++)
      r+=w[j];          //搜索子集空间树
while (true){
  int wt = Ew+w[i];//检查左儿子结点
  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);
  }
  // 检查右儿子结点
  if (Ew + r > bestw && i < n)
      Q.Add(Ew);
  Q.delete(Ew); // 取下一扩展结点
  if (Ew == -1)
  {   if (Q.isEmpty())   return bestw;
     Q.Add(-1); Q.delete(Ew);
  //同层尾部标志，取下一扩展结点
     i++; // 进入下一层
     r-=w[i];}
   }
}
```

# 实例与操作过程

- n=3，$c_1$= $c_2$= 50，w=[10,40,40]
- 初始时：bestw=0, Ew=0, r=0, i=1
- 接下来：r=40+40=80, 循环




| Q | -1 | 10 | 0 | -1 | 50 | 40 | 10 | 0 | -1 | … |
|---|---|---|---|---|---|---|---|---|---|---|

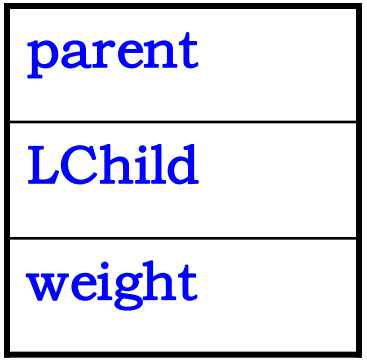
# 4. 构造最优解

做法：

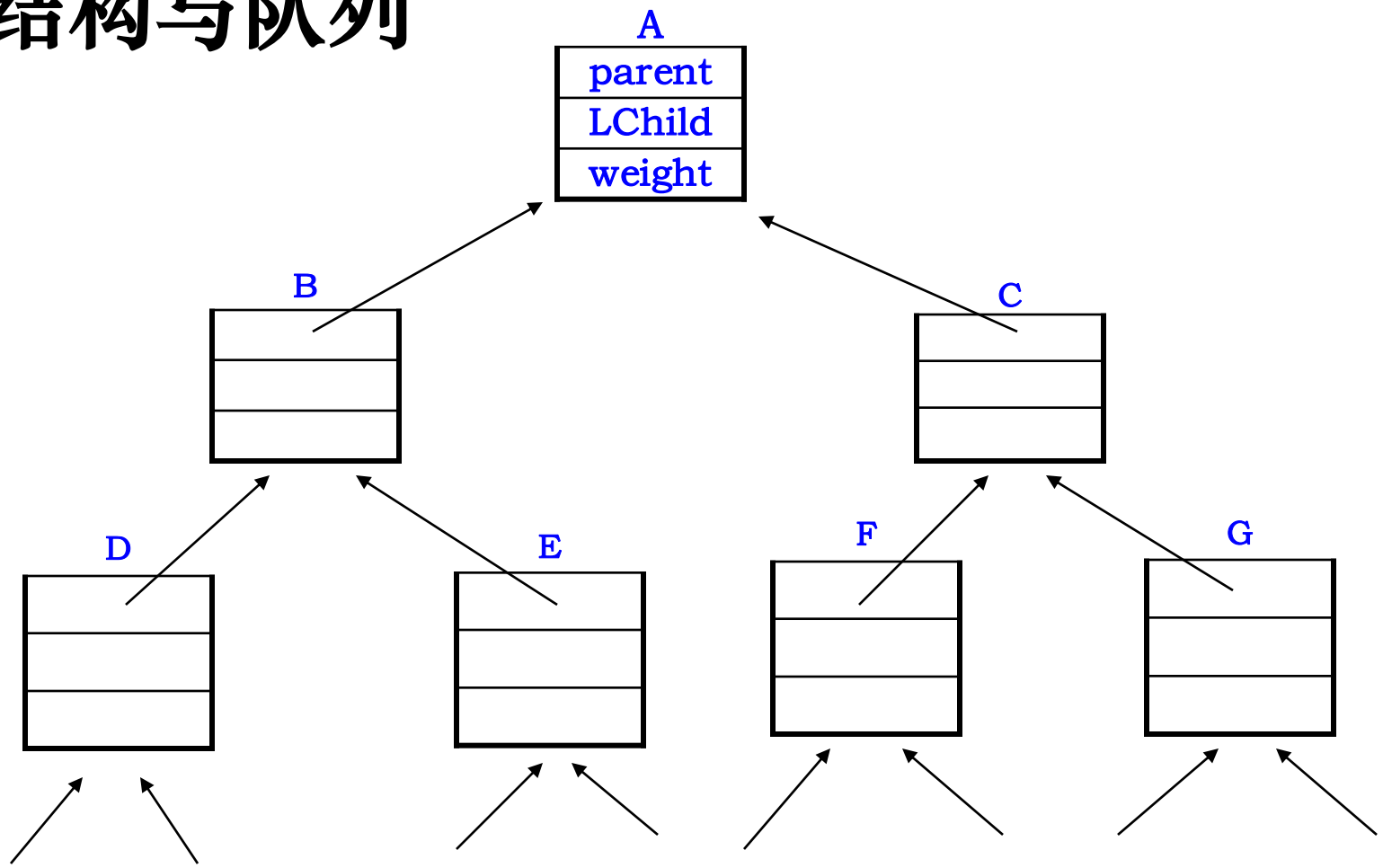
1. 保存路径：为能方便地构造出相应的最优解，算法必须存储相应子集树中从活结点到根结点的路径
2. 设置指向其父结点的指针
3. 设置左、右儿子标志

```
class QNode{
  friend void EnQueue(……);
  friend Type Maxloading(……);
  QNode parent; // 父结点
  boolean LChild;// 左儿子标志
  int weight; // 结点所相应的载重量
}
```

结点E的结构



# 结构与队列




**队列Q：如何构造？**

# 最优解构造

- 算法：修改求最优值的算法MaxLoading，添加构造队列内容
- 找到最优值后，可以根据parent回溯到根节点，找到最优解。

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

# 5. 优先队列式分支限界法

1. 活结点x优先级定义：x.uweight

- 含义：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。 x.uweight= Ew + r
- 将活结点表存储于最大优先队列Q中，优先级最大者优先。

2. 优先级最大的活结点成为下一个扩展结点

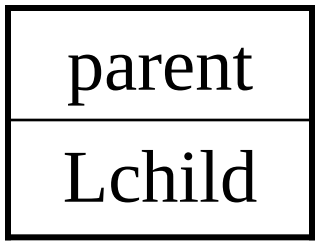
3. 子集树中叶结点所相应的载重量与其优先级相同。

# 优先队列式分支限界法求解策略

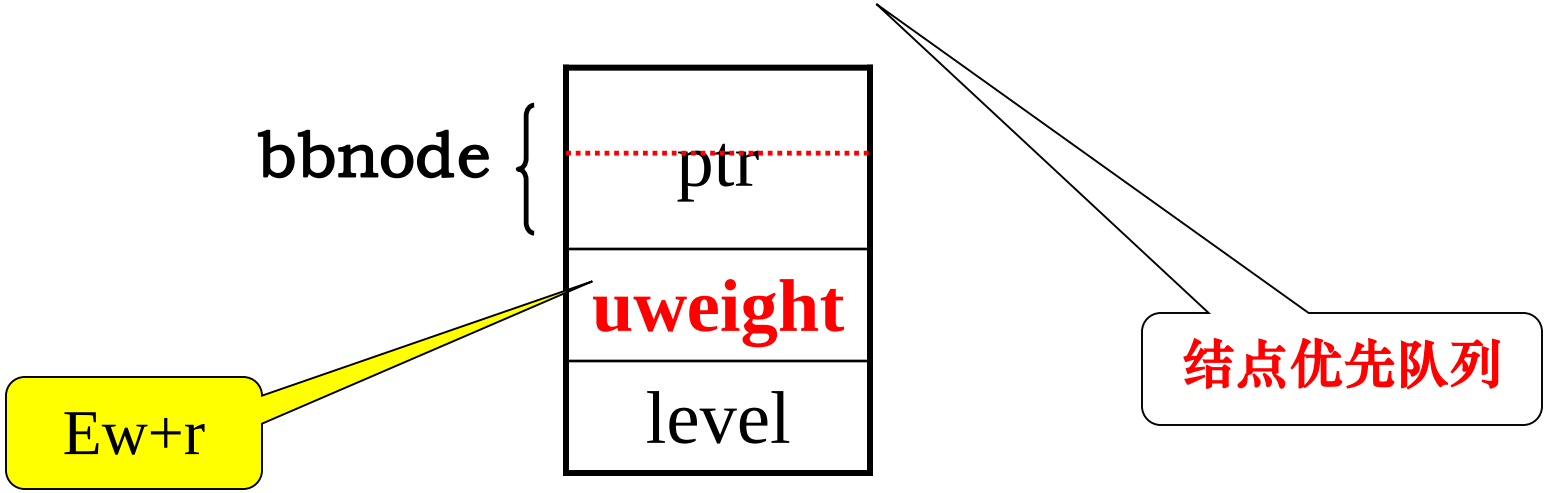
- 搜索时保存当前已构造出的部分解空间树，到达最优值的叶结点，然后在解空间树中从该叶结点开始向根结点回溯，构造最优解。

# 两种结点类型

- 子集树结点类型bbnode




- 堆结点类型Heapnode:最大堆H



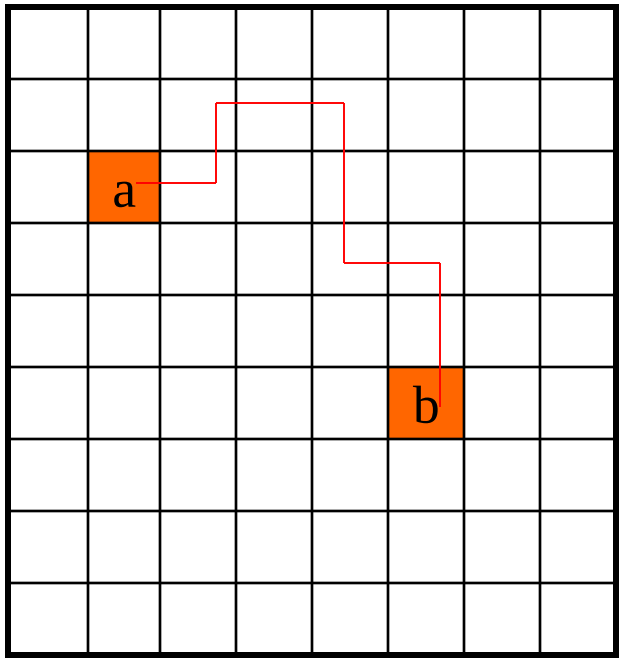
# 函数

- 函数AddLiveNode：将活结点加入到子集树中，并产生一个堆类型Heapnode结点加入到表示活结点优先队列的最大堆H中，
- 涉及的有关数据：
  - 优先级（上界）wt,子集树当前结点的父结点E，是否左儿子标志ch,层次lev，
  - 更新堆H
- 函数MaxLoading：具体实施对解空间的优先队列式分支限界搜索。

# 6.4 布线问题

# 布线问题描述

- 印刷电路板：布线区域划分成n×m方格阵列
- 内有2个方格a,b。现须确定方格a的中点到方格b的中点的最短布线方案
- 布线要求：
  1. 沿直线或直角进行
  2. 已布线的方格做封锁标记
  3. 其他线路不允许穿过被封锁的方格



# 布线问题求解方法

1. 队列式分支限界法来解
2. 解空间是一个图
3. 求解方法
   1) 起始位置a第一个扩展
   2) 依次考虑距a距离为1、2、3、…、的方格，并作标记，并存入活结点队列
   3) 从活结点队列取结点扩展
   4) 一直搜索到目标方格b或活结点队列为空时为止

# 设定

- 二维数组grid[i][j]:表示方格阵列
- 初始时，
  - grid[i][j]= 0：该方格允许布线
  - grid[i][j]= 1：该方格被封锁，不允许布线
- 边界处理：四周用方格阵列围起来
- 方格的方位offset:

  （1，0）（-1，0）（0，1）（0，-1）
- 位置position:

# 方向和边界

```
Position [] offset = new Position [4];
offset[0] = new Position(0, 1);      // 右
offset[1] = new Position(1, 0);      // 下
offset[2] = new Position(0, -1);     // 左
offset[3] = new Position(-1, 0);     // 上
```

定义移动方向的相对位移

```
for (int i = 0; i <= size + 1; i++) {
    grid[0][i] = grid[size + 1][i] = 1; // 顶部和底部
    grid[i][0] = grid[i][size + 1] = 1; // 左翼和右翼
}
```

设置边界的围墙

# BFS算法

```
here. row = start.row; here.col = start. col;
grid[start.row][start.col] = 2;   /*循环*/
while（队列Q不空）从Q中取一节点，做下面工作
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] == 0)
       { // 该方格未标记，尝试布线
            grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
            if ((nbr.row == finish.row) && (nbr.col == finish.col)) break;
            Q.Add(nbr);
       }
}
```

# 路径查找

找到目标位置后，可以通过回溯方法找到这条最短路径。

构造最短路径：

1. 从终点finish出发，开始向起始方格方向回溯。先把它作为当前结点。
2. 每次向标记距离比当前方格标记距离少1的相邻方格移动，直至到达起始方格时为止。搜索时，比较相邻4个方格的标号。

# 深度优先搜索DFS解布线问题-补充

- 二维数组grid[i][j]:表示方格阵列
- 函数search(pos,count)：判断是否出界；该次移动次数是否大于前次搜索的移动次数

```
void search(position pos,int count) {
  if  ((pos.x<1)||(pos.y<1)||(pos.x>m)||(pos.y>n))  return;
   else if (grid[pos.x][pos.y]<=count)return;
   else if ((pos.x==b.x)&&(pos.y==b.y))输出解;
   else{
      grid[pos.x,pos.y]=count;
      search(pos.x,pos.y+1,count+1); search(pos.x+1,pos.y,count+1);
     search(pos.x,pos.y-1,count+1); search(pos.x-1,pos.y-1,count+1);}
}
```

右旋

# 主程序

```
main(){
   for(int i=1;i<=m;i++)
      for(int j=1;j<=n;j++)
         grid[i][j]=30000;
      cin>>x1>>y1>>x2>>y2;  //取起点和终点坐标
      search(x1,y1,0)
      …//输出路径长度等
}
```

# 思考题

- 象棋棋盘上位置a(x1,y1)处有一只马，现要跳到位置b(x2,y2)。求最少的步数和路径。
- 上机实验题6

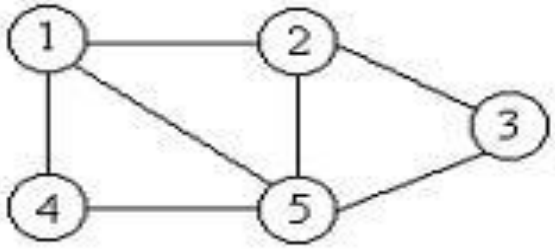
# 6.6 最大团问题

# 1、问题描述

- 概念回顾：给定无向图G=(V，E)。

1. 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。
2. 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。
3. G的最大团是指G中所含顶点数最多的团。

- 例：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，{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-限界

求解方法：优先队列式分支限界法

优先级：以团可能的顶点数上界 cn+ n-i作为优先队列的优先级，最大者优先。

cn：表示与该结点相应的团的顶点数

n-i：表示当前节点后未处理过的节点总数

level：表示结点在子集空间树中所处的层次，

level=1、2、…、n, 记 i=level-1

un：当前团最大顶点数的上界，即

un=cn+（n-i)

# 3. 算法思想

- 总是从优先队列Q中取具有最大un值的元素作为下一个扩展结点。
- 子集树的根结点是初始扩展结点，其cn的值为0。

在扩展内部结点时，考察其左、右儿子结点i

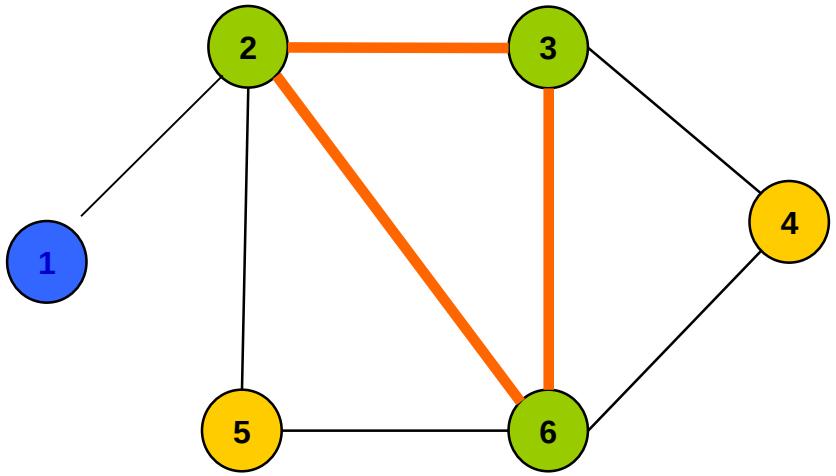
1. 左儿子结点i：尝试将顶点i加入到当前团中。当i与当前团中所有顶点都有边相连，则i是可行结点，将它加入到子集树中并插入Q，否则就不是可行结点。
2. 右儿子结点：当un>bestn时，右子树中可能含有最优解，此时将右儿子结点加入到子集树中并插入到活结点优先队列中。

# 搜索的终止条件

1. 算法的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。
2. 对于子集树中的叶结点，有un＝cn。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，进一步搜索不可能得到更大的团，此时算法已找到一个最优解。

# 实例与操作过程

- 对下列无向图求最大团




- 结点类型

| cn |
| --- |
| un |
| level |
| ptr |

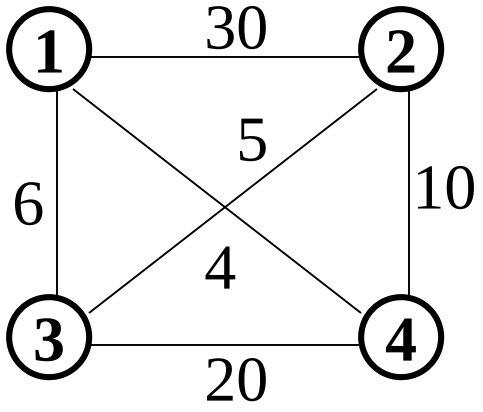
# 6.7 旅行售货员问题

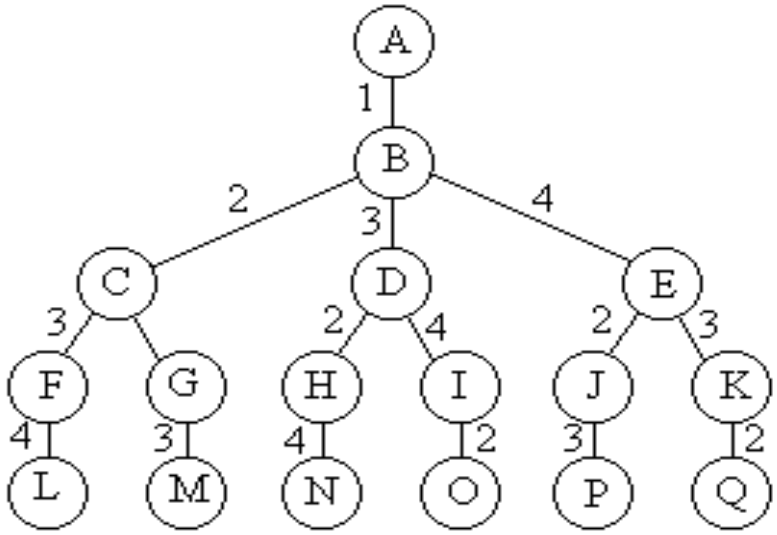
# 1. 问题描述

- 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。
- 周游路线是包括图中的每个顶点的一条回路。费用是这条路线上所有边的费用之和。
- 旅行售货员问题：在图G中找出费用最小的周游路线。

# 2. 算法

**解空间：可组织成一棵树，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。**



# 算法描述

- 采用优先队列分支限界法
- 用一个最小堆表示活结点优先队列。
- 优先级：堆中每个结点的子树费用的下界lcost值

- 算法计算出图中每个顶点的最小出边费用并用minout记录：

1. 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
2. 如果每个顶点都有出边，则根据计算出的minout作算法初始化

# 一些约定

**n：图G的顶点数**

**a：图G的邻接矩阵**

**NoEdge：图的无边输出，一般取最大数，如65537**

**cc：当前费用**

**bestc：当前最小费用**

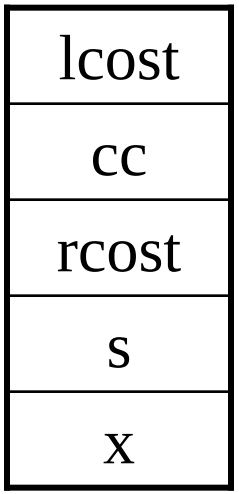
**lcost：子树费用的下界**

**rcost：顶点最小出边费用和**

**s：根结点到当前结点的路径为x[0:s]**

**x：需搜索的顶点x[s+1:n-1]**

**结点结构**



# 3、程序描述

- 最小堆H：容量为1000
- minout[i]：计算图中每个顶点的最小出边费用
- MinSum：计算所有顶点的最小出边费用和
- 初始化：s=0; x[0:n-1]={1,2,…,n}
  rcost=MinSum, cc=0,bestc=inf
- 搜索排列树：扩展第一个结点B
- 循环进行：删除B，取下一个扩展结点，直到堆空。
- 判别是否有回路：如无，返回；如有，记下路径，最优解由数组v给出

关键代码

# 4、关键代码-看书

```
While(E.s<n-1){
 if (E.s==n-2){
  if (a[E.x[n-2]][E.x[n-1]]!= NoEdge
   &&
   a[E.x[n-1]][1]]!=NoEdge &&
   (E.cc+a[E.x[n-2]][E.x[n-1]]+
   a[E.x[n-1]][1]<bestc
   ||bestc=NoEdge)){
    bestc= a[E.x[n-2]][E.x[n-1]]+
   a[E.x[n-1]][1] + E.cc;
   E.cc=bestc; E.s++; H.Insert(E);}
 else delete[] E.x;}
else{
for (int i=E.s+1;i<n;i++)
  if (a[E.x[E.s]][E.x[i]]!=NoEdge){
```

```
Type cc=E.cc+a[E.x[E.s]][E.x[i]]
Type rcost=E.rcost-MinOut[E.x[E.s]];
Type b=cc+rcost;
if (b<bestc||bestc=NoEdge}{
  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;
    N.Insert(N);}
}
```

# 约束函数和限界函数

- 约束函数

  a[i][j]!=NoEdge

- 限界函数

  **cc+rcost>bestc**

# 5、实例与操作过程