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

子集树中结点类型为BBnode

```
class BBnode{
        BBnode parent;      //父结点
        Boolean leftChild; //左结点节点标志
}
```

活结点优先队列用最大堆表示，堆中元素类型为HeapNode

```
class HeapNode{
        BBnode liveNode; //树结点
        int uweight; //优先级
        int level; //树中层序号
}
```

6.3 装载问题

addLiveNode将新的活结点加入到子集树中，并插入到堆中

```
void addLiveNode(int up, int lev, BBnode par,
  boolean ch){
      BBnode b = new Bbnode(par, ch);
      HeapNode node = new HeapNode(b, up, lev);
      heap.put(node);
}
```

6.3 装载问题

ew -- 当前扩展结点所相应的重量
r -- 剩余集装箱的重量

```
int maxLoading(int w[], int c){
      MaxHeap heap; BBnode e = null; int i = 1; int ew =
  0;
      int r[] = new int[n+1];
      for(int j=n-1;j>0;j--)
            r[j] = r[j+1] + w[j+1];
      while(i!=n+1)
            if(ew+w[i] <= c)
                  addLiveNode(ew+w[i]+r[i], i+1, e, true);
           addLiveNode(ew+r[i], i+1, e, false);

            //取下一扩展结点……
       return ew;
}
```

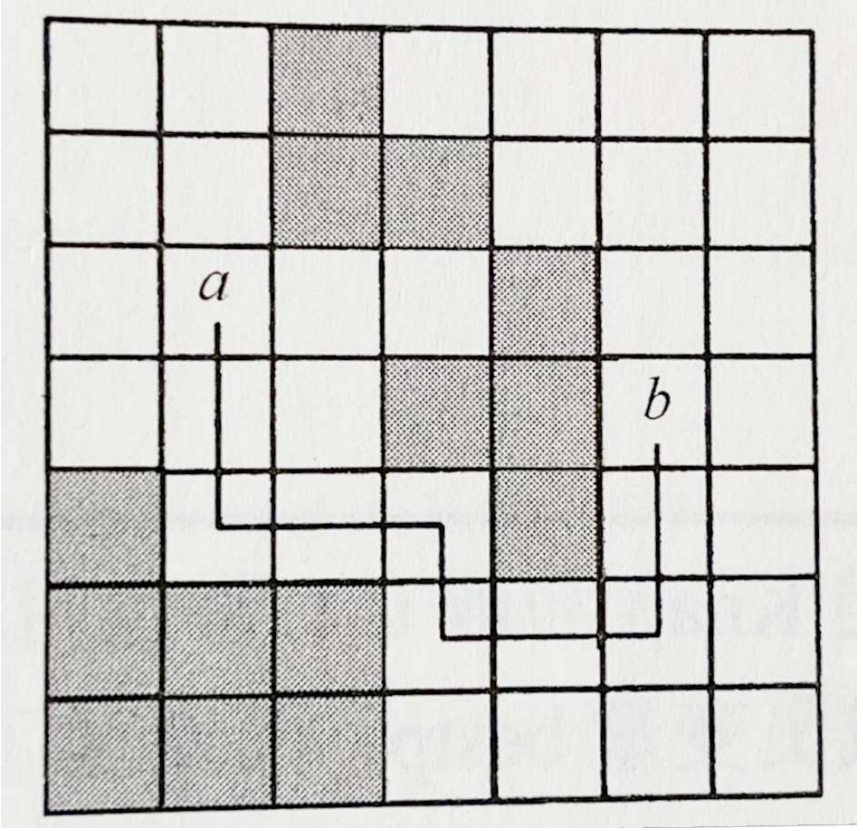
6.3 装载问题

5. 优先队列式分支限界法

改进策略：

- 在活结点插入优先队列前测试载重量上界是否大于当前最优值，如果不是，则剪枝；
- 将由于最优值的增加而产生的无效活结点从优先队列中删除。

6.4 布线问题



- 印刷电路板将布线区域划分成n*m个方格阵列；
- 电路布线问题要求确定连接方格a和方格b的中点的最短布线方案。
- 布线时，电路只能沿直线或直角布线。
- 已布了线的方格做了封锁标记，图中以灰色表示。

6.4 布线问题

算法的思想

- 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。

6.4 布线问题

算法的思想

定义一个表示电路板上方格位置的类Position，它有两个成员row和col分别表示方格所在的行和列。

沿右、下、左、上四个方向的移动分别标记为0、1、2、3。

```
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
```

offset[i].row和offset[i].col给出了沿四个方向相对于当前方格的相对位移。

6.4 布线问题

算法的思想

二维数组grid表示所给的方格阵列，初始时，grid[i][j]=0表示该方格允许布线，grid[i][j]=1表示该方格被封锁。移动时，grid[i][j]表示方格（i，j）距起始方格的距离，起始位置距离标记为2，以区别于标记开放封锁状态的0和1，最后的实际距离为标记距离减2。

为处理方格边界，在方格阵列四周增加标记为”1”的附加方格。

```
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
}
```

6.4 布线问题

```
for (int i = 0; i < 4; 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
              grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
              if ((nbr.row == finish.row) && (nbr.col == finish.col))
                    break;
              q.put(new Position(nbr.row, nbr.col));
}
```

找到目标位置后，可以通过回溯方法找到这条最短路径，每次向标记距离比当前方格标记距离小1的相邻方格移动，直至到达起始方格为止。

6.5 0-1背包问题

算法的思想 (优先队列分支限界法)

- 将各物品依其单位重量价值从大到小进行排列。
- 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。
- 当扩展到叶结点时为问题的最优值。

6.5 0-1背包问题

上界函数

```
double Bound(int i){
        double cleft = c - cw;   //cleft为剩余空间
        double b = cp; //b已获得价值
        while (i <= n && w[i] <= cleft)
                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
}
```

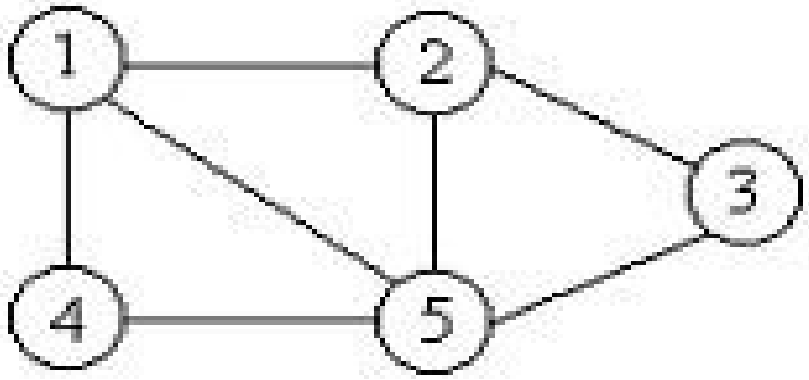
AddLiveNode(double up, double cp, double cw, int lev, BBnode par, boolean ch)

up: 结点的价值上界；cp: 结点相应的价值；cw: 结点相应的重量；par：父节点

ch: 是否为左孩子结点；lev: 活结点在子集树中所处的层序号；ch:左儿子标志

```
double MaxKnapsack{
        BBnode enode = null;
        int i=1; double cw=cp=0, bestp=0, up=Bound(1);
        while(i != n + 1)     // 非叶结点
            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);    //计算结点i+1所相应价值的上界
            if (up >= bestp)    //检查右儿子节点
               addLiveNode(up, cp, cw, i+1, enode, false);
                //    取下一个扩展节点 …
```

6.6 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。

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。

6.6 最大团问题

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，用upperSize作为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。

6.6 最大团问题

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他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时，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.
  - 接着继续考察当前扩展结点的右儿子结点。当upperSize>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。

6.6 最大团问题

3. 算法思想

- 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。
- 对于子集树中的叶结点，有upperSize = cliqueSize。此时活结点优先队列中剩余结点的upperSize值均不超过当前扩展结点的upperSize值，从而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。

6.7 旅行售货员问题

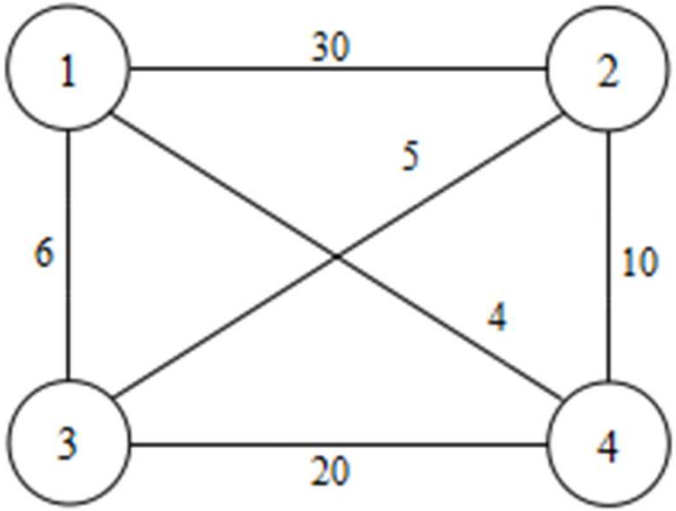
1. 问题描述

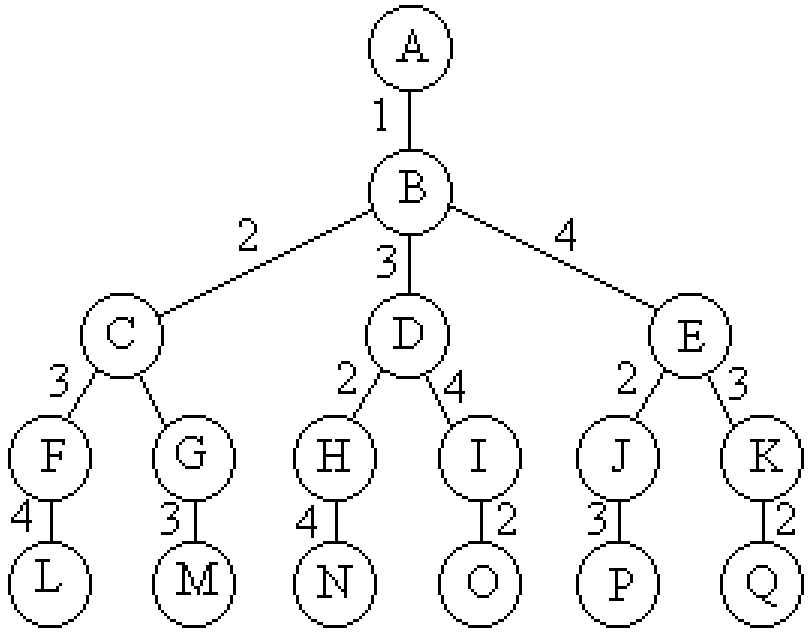
某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小。

路线是一个带权图。图中各边的权为正数。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。

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。

6.7 旅行售货员问题



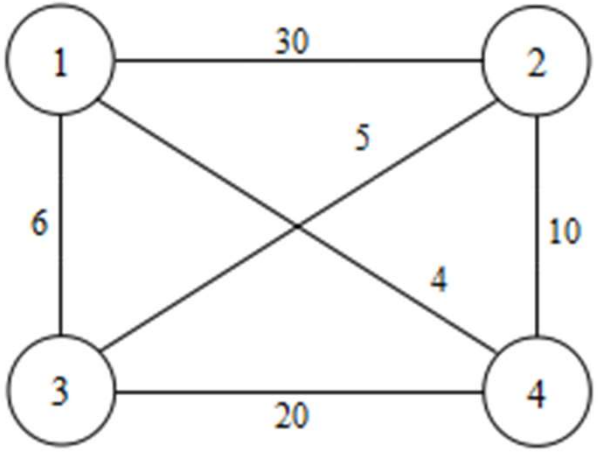


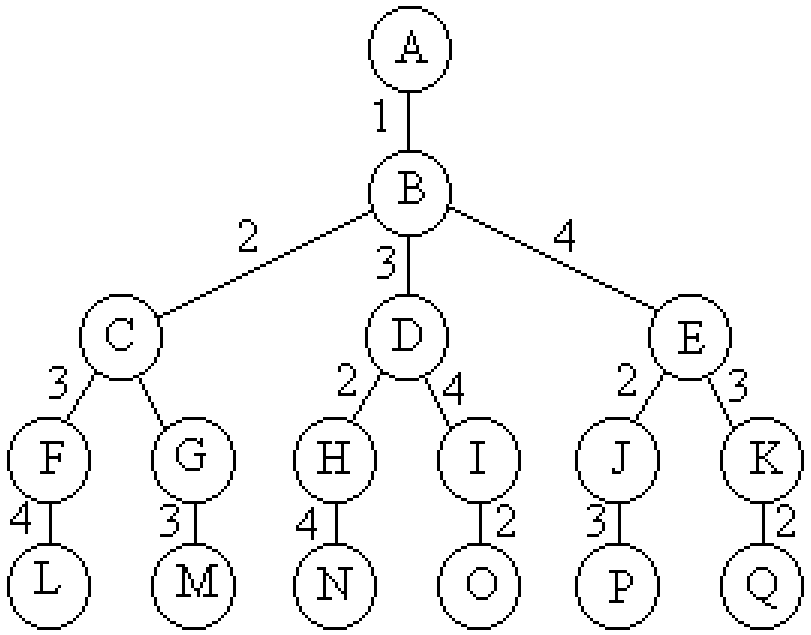
队列式：

活结点队列CDE，对于节点C、D、E，FG、HI、JK都是可行节点，于是队列为FGHIJK。

对于节点F，L为其叶节点并得到费用59。对于G扩展至M得到费用71。H扩展至N得到费用25，对于节点I，由于现有费用已经超过25，故无必要扩展。

6.7 旅行售货员问题





优先队列式：

使用优先队列来存储活结点，优先队列中的每个活结点都存储从根到该活结点的相应路径。

6.7 旅行售货员问题

2. 优先队列式算法描述

- 算法开始时创建一个最小堆，用于表示活结点优先队列。堆中每个结点的子树费用的下界lcost=cc+rcost，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，处理结点i时，rcost=$\sum_{j=i+1}^{n} minout[j]$。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。

6.7 旅行售货员问题

2. 算法描述

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。

6.7 旅行售货员问题

2. 算法描述

对于当前扩展结点，算法分2种情况进行处理：

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。

6.7 旅行售货员问题

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当前费用cc和子树费用的下届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。

6.7 旅行售货员问题

```
class HeapNode{
        float lcost,   //子树费用的下界，优先级
              cc,      //当前费用
              rcost;   //x[s:n-1]中顶点最小出边费用和
        int s;   //结点在树中的层次
        int x[];   //搜索路径
}
```

6.7 旅行售货员问题

Max表示Float.MaxValue

```
float bbTSP( ){
      MinHeap heap;
      float[ ] minOut = new float[n+1];
      float minSum = 0;
      for(int i=1;i<=n;i++){ //找出每个结点的最小出边，并计算最小出边和
          float min = Max;
          for(int j=1;j<=n;j++)
              if(a[i][j]<Max)&&a[i][j]<min)
                  min = a[i][j];
          if(min==Max)  //无回路
              return Max;
          minOut[i] = min;
          minSum += min;
}
```

6.7 旅行售货员问题

HeapNode(lcost, cc, rcost, s, x[])

```
float bbTSP( )续
      int [ ] x = new int[n];
      for(int i=0;i<n;i++)
            x[i] = i+1;
      HeapNode enode = new HeapNode(0, 0, minSum, 0, x);
      float bestc = Max;
      while(enode!=null && enode.s<n-1)
            x = enode.x;
            if(enode.s==n-2)
                  if(a[x[n-2]][x[n-1]]<Max && a[x[n-1]][1] <Max
                  && enode.cc+a[x[n-2]][x[n-1]]+a[x[n-1]][1]<bestc)
                        bestc = enode.cc+ a[x[n-2]][x[n-1]]+a[x[n-
1]][1];
                        enode.cc = bestc; enode.lcost  = bestc;
                        enode.s++; heap.put(enode);
}
```

6.7 旅行售货员问题

HeapNode(lcost, cc, rcost, s, x[])

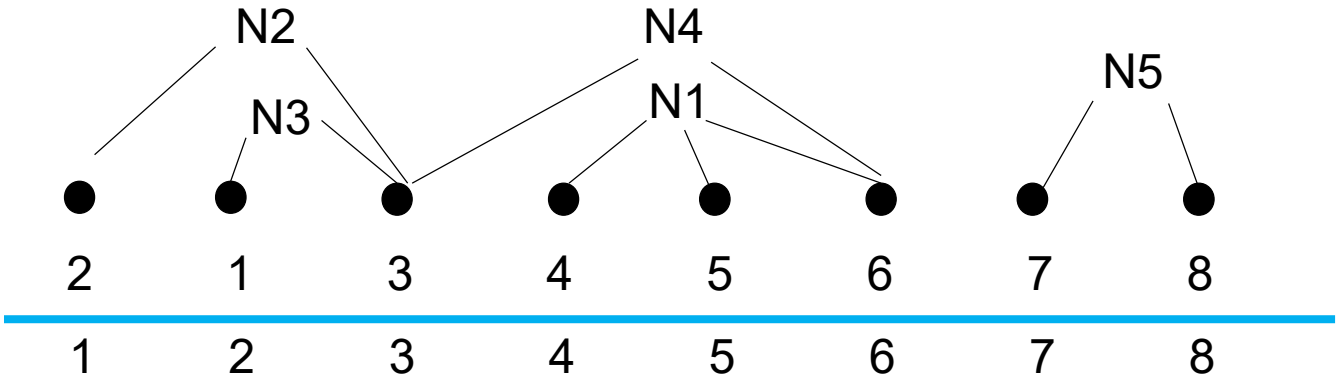
```
float bbTSP( ){ 续
        else //enode.s!=n-2
            for(int i=enode.s+1;i<n;i++)
                if(a[x[enode.s]][x[i]]<Max)
                    float cc = enode.cc + a[x[enode.s]][x[i]];
                    float rcost = enode.rcost – minOut[x[enode.s]];
                    float b = cc + rcost;
                     if(b < bestc)
                        int[] xx = new int[n];
                        for(int j=0;j<n;j++)
                            xx[j] = x[j];
                         xx[node.s+1] = x[i];
                         xx[i] = x[enode.s+1];
                         HeapNode node = new HeapNode(b, cc, rcost, enode.s+1, xx);
                         heap.out(node);
                         //取下一扩展结点继续……
}
```

6.8 电路板排列问题

- 将n块电路板以最佳排列方案插入带有n个插槽的机箱中。
- 设B={1,2,…,n}是n块电路板的集合。集合L={N1, N2, …, Nm}是n块电路板的m个连接块。其中每个连接块是B的一个子集，且Ni中的电路板用同一根导线连接在一起。
- 例如：n=8, m=5。给定n块电路板及其m个连接块如下：

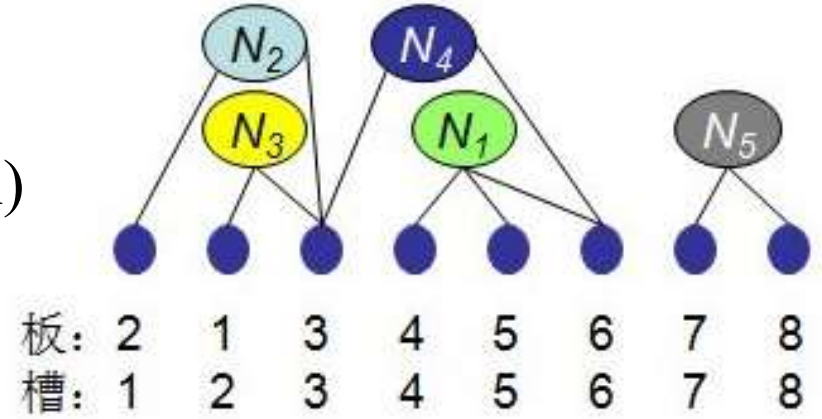
  B ={1,2,3,4,5,6,7,8}，L={N1,N2,N3,N4,N5}

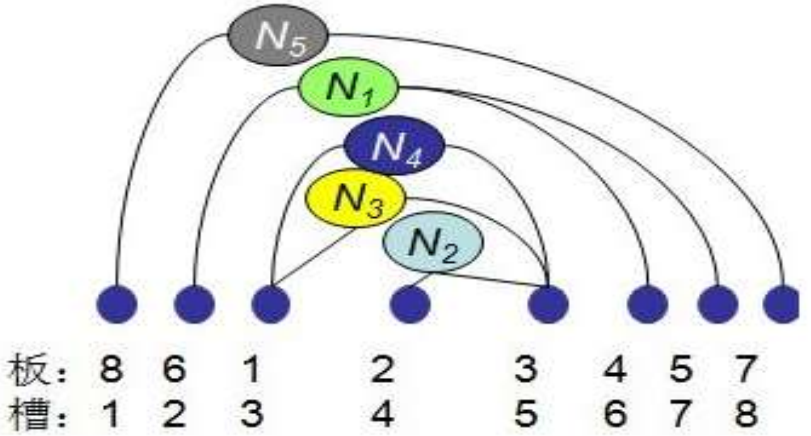
  N1={4,5,6}　N2={2,3}　N3={1,3}　N4={3,6}　N5={7,8}



电路板排列问题

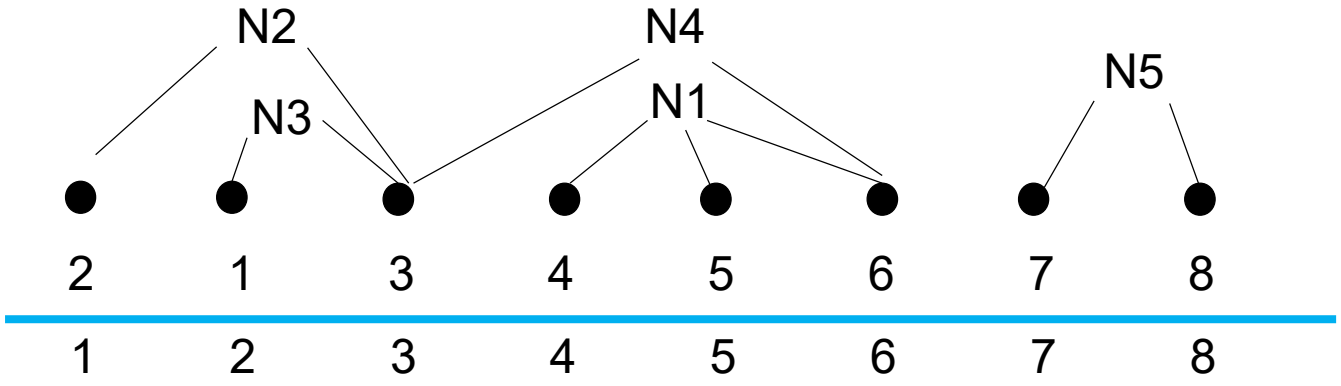
- 设x表示n块电路板的排列，即在机箱的第i个插槽插入x[i]。
- x[i]确定的电路板排列密度density(x)定义为跨越相邻电路板插槽的最大连接数。图中电路板排列的密度为2，跨越插槽2和3、4和5、5和6的连接数均为2。插槽6和7之间无跨越连接线，其余相邻插槽之间有一条跨越连线。
- 电路板排列问题要求对于给定连接块，确定电路板的最佳排列，使其具有最小密度。





电路板排列问题

- 用整形数组B表示输入，B[i][j]的值为1当且仅当电路板i在连接块N[j]中。
- 设total[j]是连接块Nj中的电路板数。对于电路板的部分排列x[1:i]，now[j]是x[1:i]中包含的Nj中的电路板数。
- 连接块Nj的连线跨越插槽i和i+1当且仅当now[j]>0且now[j]≠total[j]。据此可计算插槽i和i+1间的连接密度。



6.8 电路板排列问题

算法描述

解空间树是一棵排列树，采用优先队列式分支限界法找出最小密度布局。用当前密度作为结点的优先级。

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当前密度cd值的结点作为当前扩展结点，并加以扩展。

6.8 电路板排列问题

算法描述

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。

6.9 批处理作业问题

1. 问题的描述

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t_{ji}，i=1,2,…,n；j=1,2。

对于一个确定的作业调度，设F_{ji}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$f = \sum_{i=1}^{n} F_{2i}$，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。

6.9 批处理作业问题

2. 限界函数

解空间是一棵排列树，用优先队列式分支限界法。

排列树中每个结点E对应一个已安排的作业集M⊆{1,2，。。。。，n}。以该结点为根的子树所含叶结点的完成时间和可表示为：

$$f = \sum_{i \in M} F_{2i} + \sum_{i \notin M} F_{2i}$$

设|M| = r，且L是以E为根的子树中的叶结点，相应的作业调度为$\{p_k\}(k=1,2,\ldots,n)$。

6.9 批处理作业问题

2. 限界函数

如果从E开始到叶结点L的路上，每个作业pk在机器1上处理完都能立即在机器2上开始处理，则对于L有：

$$\sum_{i\notin M} F_{2i} = \sum_{k=r+1}^{n} [F_{1Pr} + (n-k+1)t_{1Pk} + t_{2Pk}] = S1$$

如果不能做到这点，S1只会增加，从而有$\sum_{i\notin M} F_{2i} \geq S1$。

如果从E到L的路上，从作业p_{r+1}开始机器2没有空闲时间，则有：

$$\sum_{i\notin M} F_{2i} \geq$$

$$\sum_{k=r+1}^{n} [\max(F_{2Pr}, F_{1Pr} + \min_{i\notin m} t_{1i}) + (n-k+1)t_{2Pk}] = S2$$

从而有S2也是$\sum_{i\notin M} F_{2i}$的下界。

6.9 批处理作业问题

2. 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P_k，使t_{1p_k}在k>=r+1时依非减序排列，S1则取得极小值$\hat{S1}$。同理如果选择Pk使t_{2p_k}依非减序排列，则S2取得极小值$\hat{S2}$。

$$f \geq \sum_{i \in M} F_{2i} + \max\{\hat{S}_1, \hat{S}_2\}$$

这可以作为优先队列式分支限界法中的限界函数，
并用该下界作为结点的优先级。

6.9 批处理作业问题

3. 算法描述

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。

首先考虑已安排作业数enode.s=n的情形，当前扩展结点enode是排列树中的叶结点。enode.sf2是相应于该叶结点的完成时间和。当enode.sf2 < bestc时更新当前最优值bestc和相应的当前最优解bestx。

6.9 批处理作业问题

3. 算法描述

当enode.s<n时，算法依次产生当前扩展结点enode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。

总结

回溯法

- 深度优先
- 约束函数和限界函数剪枝

分支限界法

- 队列式：广度优先
- 优先队列式：优先级
- 约束函数和限界函数剪枝

许多典型问题可以用两种不同的算法策略求解，对比体会算法的精髓和各自的优点。