

分支限界法

提纲

- 分支限界法的基本思想
- 装载问题
- 布线问题
- 0-1背包问题
- 小结

提纲

- 分支限界法的基本思想
- 装载问题
- 布线问题
- 0-1背包问题
- 小结

分支限界法的基本思想

■ 分支限界法与回溯法的不同

□ 求解目标

- 回溯法：所有解
- 分支限界法：找出满足约束条件的一个解，或是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

- 回溯法：深度优先搜索解空间树
- 分支限界法：广度优先或以最小耗费(最大收益)优先的方式搜索解空间树。

分支限界法的基本思想

- 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。

常见的两种分支限界法

- 从活结点表中选择下一扩展结点的不同方式导致不同的分支界限法。
 - 队列式(FIFO)分支限界法
 - 按队列先进先出（FIFO）原则选取下一节点为扩展节点。
 - 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。

实例：0-1背包问题

■ 问题陈述

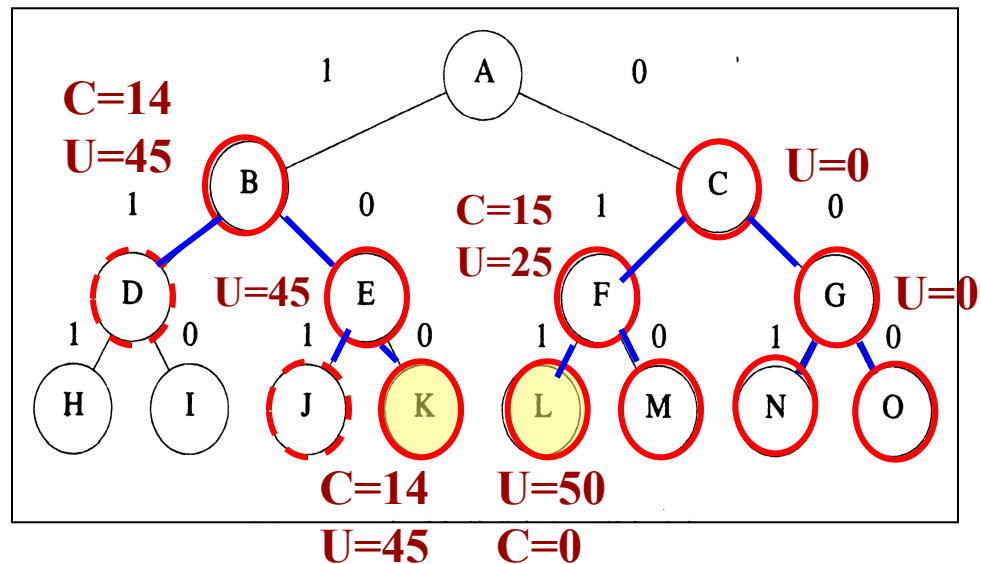
- 设有 n 个物体和一个背包，物体 i 的重量为 w_i ，价值为 p_i ，背包的容量为 C ，目标是找到一个方案，使得能放入背包的物体总价值最高。

实例：0-1背包问题

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

- ### ❑ 队列式分支限界法活动结点表:

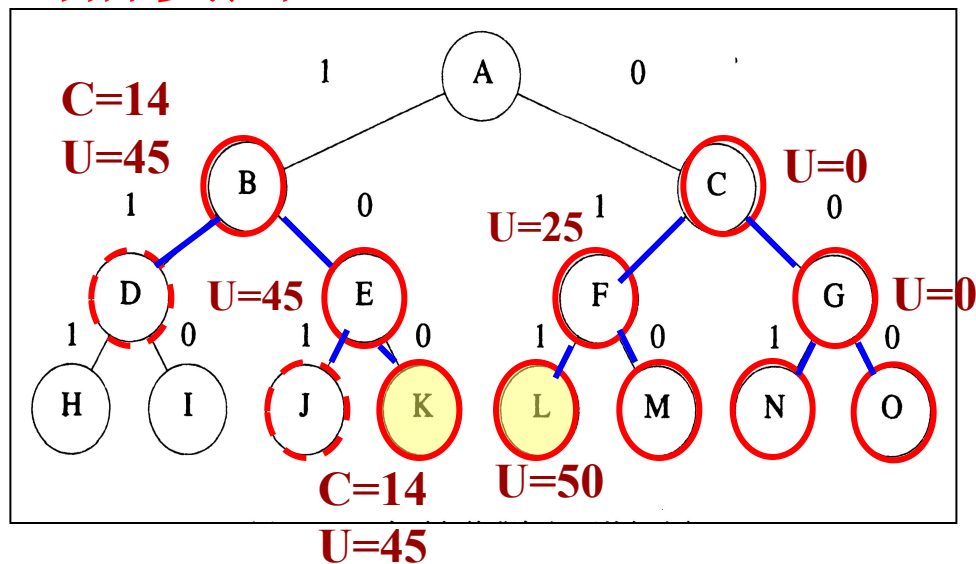
- $\{\mathbf{B}, \mathbf{C}\},$
- $\{\mathbf{C}, \mathbf{E}\},$
- $\{\mathbf{E}, \mathbf{F}, \mathbf{G}\},$
- $\{\mathbf{F}, \mathbf{G}\},$
- $\{\mathbf{G}\},$



实例：0-1背包问题

- 设 $n=3$, $W=(16,15,15)$, $P=(45,25,25)$, $C=30$
- 优先队列式分支限界法活动结点表:
- 优先级是结点的当前费用

- $\{B, C\}$,
- $\{E, C\}$,
- $\{C\}$,
- $\{F, G\}$,
- $\{G\}$,



提纲

- 分支限界法的基本思想
- 装载问题
- 布线问题
- 0-1背包问题
- 小结

装载问题

■ 问题定义

- 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
$$\sum w_i \leq c_1 + c_2$$
- 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**。
 - 当 $n=3$ ， $c_1=c_2=50$ ， $w=[10,40,40]$ ，可以将集装箱1和2装到第一艘轮船上，将集装箱3装到第二艘轮船上。如果 $w=[20,40,40]$ ，无法将3个集装箱都装上轮船。

装载问题

■ 问题分析

- 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。
- 将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近 c_1 。由此可知，装载问题等价于以下特殊的0-1背包问题。

装载问题

$$\max \sum_{i=1}^n w_i x_i$$

$$\text{s.t. } \sum_{i=1}^n w_i x_i \leq c_1$$

$$x_i \in \{0,1\}, 1 \leq i \leq n$$

■ 算法设计

- 这个问题是一个子集选择问题，它的解空间被组织成一个子集树。

队列式分支限界法

- 利用分支限界法设计算法MaxLoading实施对解空间的分支限界搜索。
 - 链表**队列Q**用于**保存活节点**，它的值记录着各活节点对应的权值。队列记录了**权值-1**，以标识**每一层的活节点的结尾**。
- 函数EnQueue用于把**节点对应的权值加入活节点队列**
 - 该函数首先检验**i是否等于n**，如果相等，则已到达了叶节点。叶节点不被加入队列中，因为它们不被展开。

队列式分支限界法

- 搜索中所到达的每个叶节点都对应着一个可行的解，而每个解都会与目前的最优解来比较，以确定最优解。
- 如果 $i < n$ ，则节点 i 就会被加入队列中。

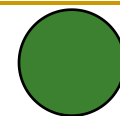
$n=3, w=[8,6,2], W=12$

$C(i)$

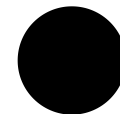
x_1

x_2

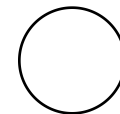
x_3



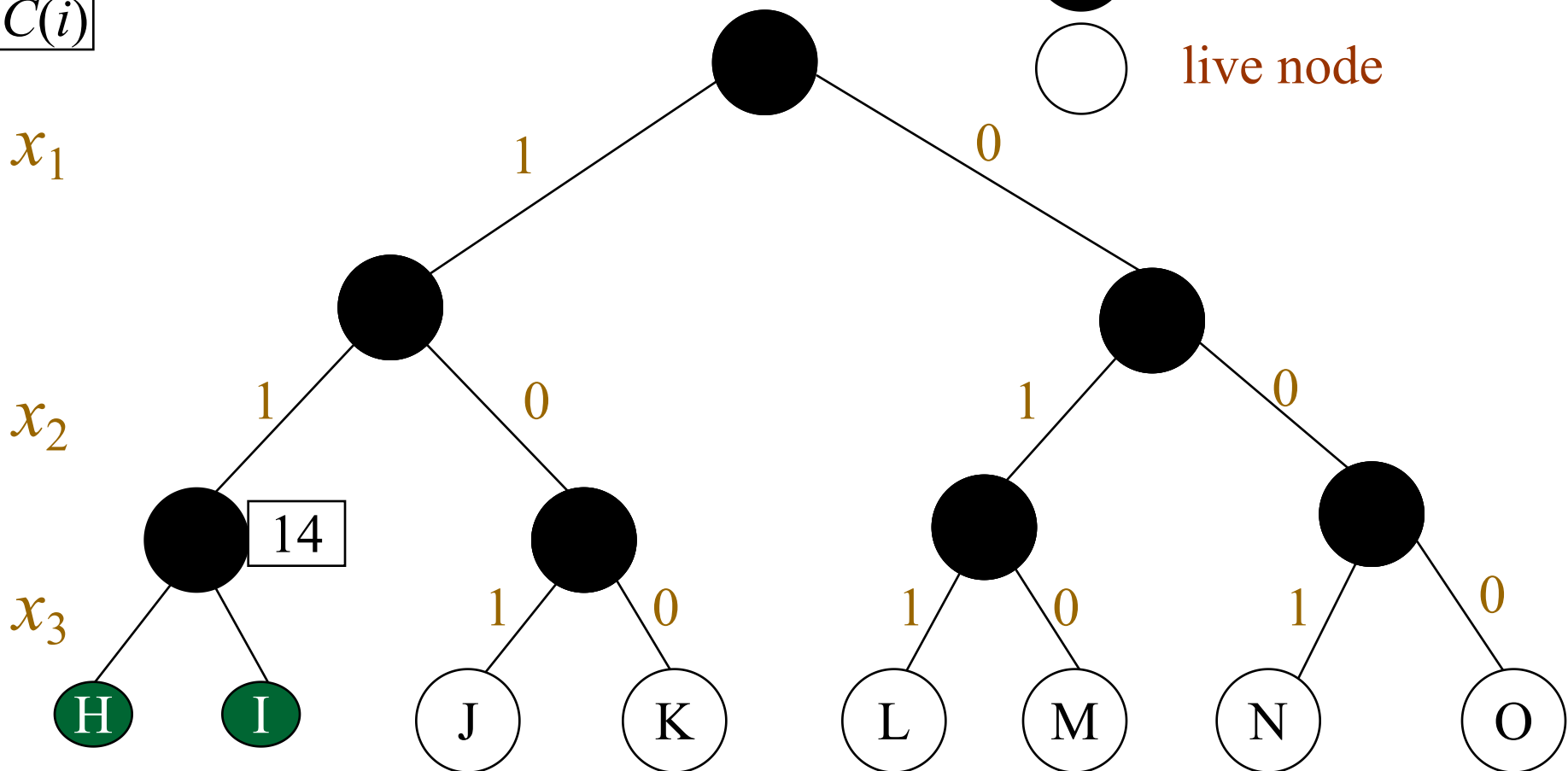
extend node



dead node



live node



Quene: -1 B C -1 E F G -1

太慢了

Bestc=10

队列式分支限界法

```
void EnQueue(Type wt, Type bestw, int i, int n)
{ // 如果不是叶节点, 则将节点权值wt加入队列Q
    if (i == n) // 叶子, 记录最优解
    {
        if (wt > bestw)
            bestw = wt;
    }
    else Q.Add(wt); // 不是叶子
}
```

队列式分支限界法

- 搜索中所到达的每个叶节点都对应着一个可行的解，而每个解都会与目前的最优解来比较，以确定最优解。
- 如果 $i < n$ ，则节点 i 就会被加入队列中。

队列式分支限界法

Type MaxLoading (Type w[], Type c, int n)

{// 返回**最优装载值**使用FIFO分枝定界算法

//为层次1 初始化

Queue<Type> Q; // 活节点队列

Q.Add(-1); //标记本层的尾部

int i = 1; //扩展节点的层

Type Ew = 0, // 扩展节点所相应的载重量

bestw = 0; // 目前的最优值

while (true) // 搜索子集空间树

{ if (Ew + w[i] <= c) // x[i] = 1; 检查扩展节点的左孩子

EnQueue(Q, Ew+w[i], bestw, i, n);

队列式分支限界法

```
EnQueue(Q, Ew+0*w[i], bestw, i, n); // 右孩子总是可行
Q.Delete(Ew); // 取下一个扩展节点
if (Ew == -1) // 到达层的尾部
{ if (Q.IsEmpty()) // 队列空, 结束搜索
    return bestw;
  Q.Add(-1); // 添加尾部标记
  Q.Delete(Ew); // 取下一个扩展节点
  i++; // Ew的层
}
}
```

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； ew 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 $ew+r \leq bestw$ 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。

$n=3, w=[8,6,2], W=12$

 extend node

 dead node

 live node

$C(i)$

x_1

1

0

$Bestw=8$

$B(1)=8$

x_2

1

0

14

x_3

1

0

H

I

J

K

L

M

N

O

Quene:

-1

B

-1

E

-1

10

算法的改进

// 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);

}

算法的改进

```
// 检查右儿子结点可能含最优解
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); i++;
  r -= w[i]; // E-节点中余下的重量
}
```


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。

```
private static 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
```

构造最优解

■ 算法EnQueue做如下修改:

```
void EnQueue(.....)
```

```
{// 如不是叶节点, 向Q中添加一个i 层、重量为wt的活节点
```

```
// 新节点是E的孩子。当且仅当是左孩子时, ch为true。
```

```
    if (i == n) // 叶子
```

```
    { if (wt == bestw) // 目前的最优解
```

```
        { bestE = E; bestx[n] = ch;}
```

```
        return;
```

```
    }
```

```
QNode<T> *b; // 不是叶子, 添加到队列中
```

```
b = new QNode<T>;
```

```
b->weight = wt; b->parent = E; b->LChild = ch;
```

```
Q.Add(b);
```

```
}
```

构造最优解

- 算法可以在搜索子集树的过程中保存当前已构造出的子集树中的路径，从而可在结束搜索找到最优值后，从子集树中与最优值相应的节点处根据parent向根节点回溯，找到最优解。

// 构造当前最优解

```
for (int j = n-1; j > 0; j--)  
    { bestx[j] = (e.leftChild) ? 1 : 0;  
      e = e.parent;    }
```

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。

优先队列式分支限界法

- 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。

■ 实现方法：

- 最大优先级队列中的活节点都是互相独立的，因此每个活节点内部必须记录从子集树的根到此节点的路径。
- 一旦找到了最优装载所对应的叶节点，就利用这些路径信息来计算 x 值。

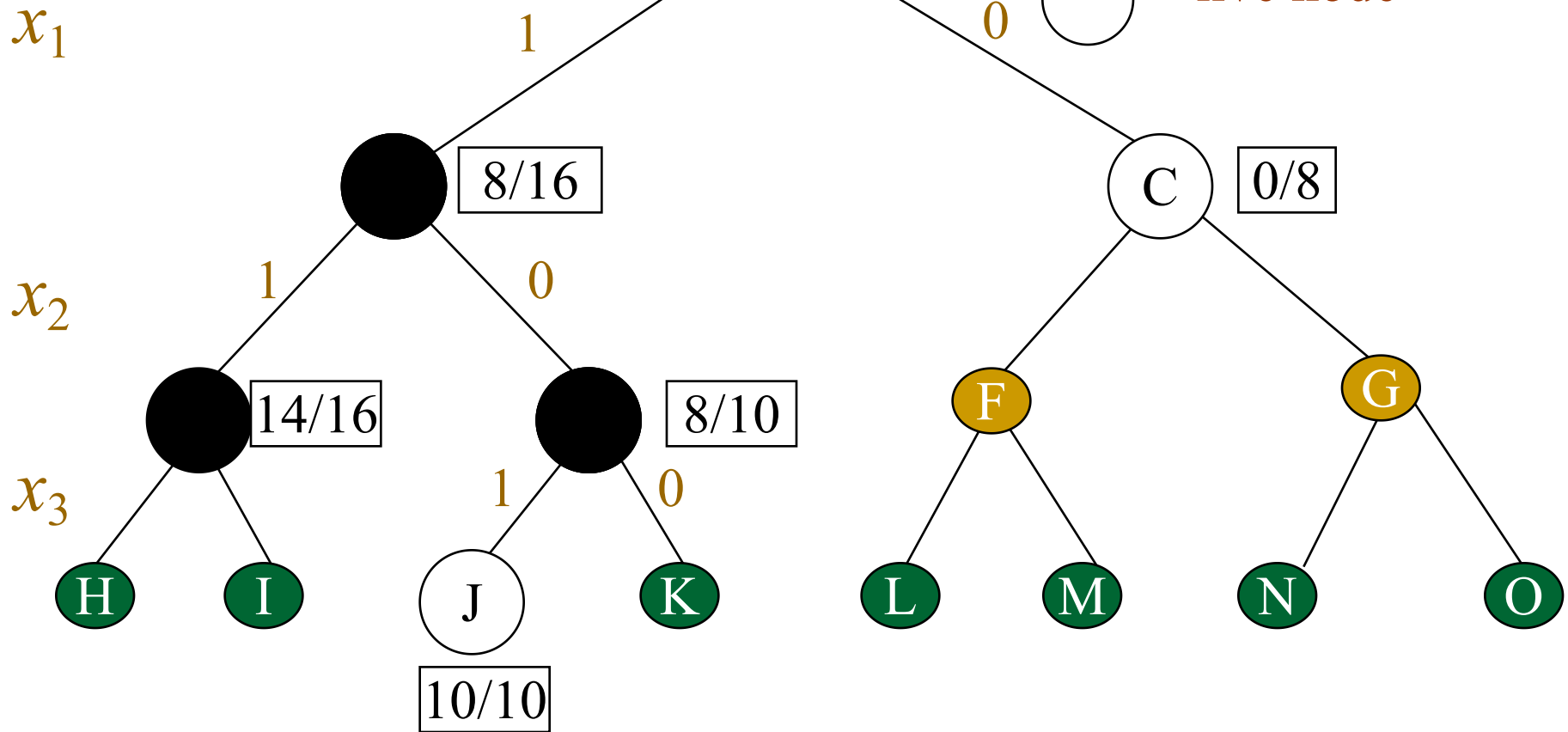
$n=3, w=[8,6,2], W=12$

$C(i)/B(i)$

 extend node

 dead node

 live node



优先队列式分支限界法

```
Void AddliveNode(MaxHeap<HeapNode<Type>> &H,  
    bbnode *E, Type wt, bool ch, int lev)  
{ //将活结点加入到最大堆H中  
    bbnode *b=new bbnode;  
    b->parent=E; b-Lchild=ch;  
    HeapNode<Type> N;  
    N.uweight =wt;  
    N.level=lev;  
    N.ptr=b;  
    H.Insert(N);  
}
```

优先队列式分支限界法

```
Type MaxLoading(Type w[],Type c,int n,int bestx[])  
{ //优先队列式分支限界法,返回最优载重量,bestx返回最优解  
Type *r=new Type[n+1] ;  
  r[n]=0;  
  for (int j=n-1;j>0;j--)  
    r[j]=r[j+1]+w[j+1];  
//初始化  
  int i=1;//当前扩展结点所处的层  
  bbnode *E=0;//当前扩展结点  
  Type Ew=0;//当前扩展结点相应的载重量
```


优先队列式分支限界法

//搜索子集空间树

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(H,E,Ew+w[i]+r[i],true,i+1);}

//右儿子结点

AddLiveNode(H,E,Ew+r[i],false,i+1); //取下一扩展结点

HeapNode N;

H.DeleteMax(N);

i=N.level;

E=N.ptr;

Ew=N.uweight-r[i-1];

}

优先队列式分支限界法

```
for(int j=n;j>0;j--)  
{  
    bestx[j]=E->Lchild;  
    E=E->parent;  
}  
return Ew;  
}
```

提纲

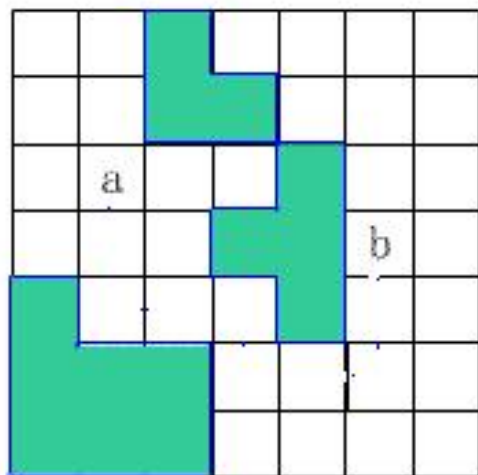
- 分支限界法的基本思想
- 装载问题
- 布线问题
- 0-1背包问题
- 小结

布线问题

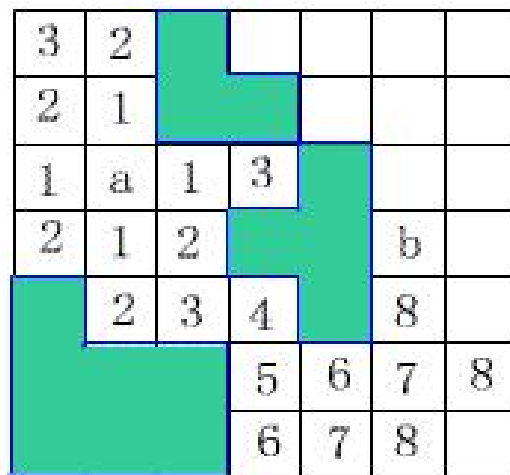
■ 问题描述

- 印刷电路板将布线区域划分成 $n*m$ 个方格阵列。精确的电路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。
- 在布线时，电路只能沿直线或直角布线。
- 为了避免线路相交，已布了线的方格做了封锁标记，其他线路不允许穿过被封锁的方格。

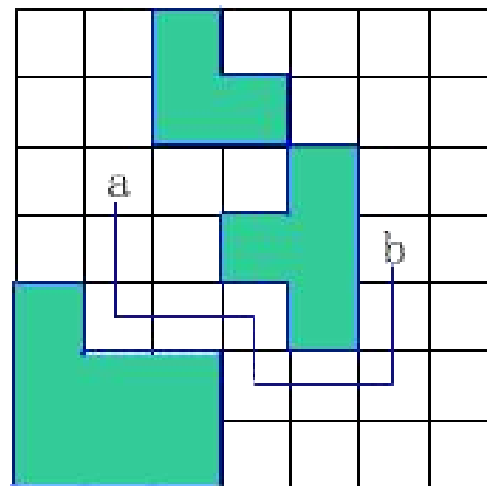
布线问题



原图

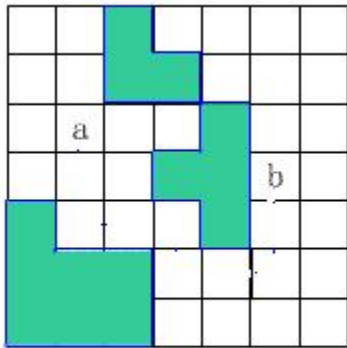


(a) 标记距离



(b) 最短布线路径

布线问题



原图

3	2					
2	1					
1	a	1	2			
2	1	2			b	
	2	3	4		8	
			5	6	7	8
			6	7	8	

布线问题

- 布线问题的解空间是一个图，适合采用**队列式分支限界法**来解决。
 - 从起始位置a开始将它作为第一个扩展结点。与该结点相邻并且可达的方格被加入到**活结点**队列中，并且将这些方格**标记为1**。
 - 接着从活结点队列中取出**队首**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**，并存入活节点队列。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（表示没有通路）。

布线问题

■ 算法实现：

□ 定义小方格位置

- 定义一个表示电路板上小方格位置的类**Position**，它的两个私有成员**row**和**col**分别表示小方格所在的行和列。
- 在电路板的任何一个方格处，布线可沿**右、下、左、上**4个方向进行。沿这4个方向的移动分别记为0, 1, 2, 3。沿着这4个方向**前进一步**相对于当前方格的位移如下表所示。

布线问题

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

□ 实现方格阵列

- 用二维数组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，而 $grid[i][j]=1$ 表示该方格被封锁，不允许布线。
- 为了便于处理方格边界的情况，算法在所给方格阵列四周设置一圈标记为“1”的附加方格，即可识别方阵边界。

布线问题

□ 初始化

- 算法开始时，测试初始方格与目标方格是否相同。如果相同，则不必计算，直接放回最短距离0，否则算法设置方格阵列的边界，初始化位移矩阵offset。

□ 算法搜索步骤

- 算法从start开始，标记所有标记距离为1的方格并存入活结点队列，然后依次标记所有标记距离为2，3.....的方格，直到到达目标方格finish或活结点队列为空时为止

布线问题

- 为什么这个问题用回溯法来处理就是相当低效的？
 - 回溯法的搜索是依据深度优先的原则进行的，如果我们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，但是我们最开始根本无法判断正确的路径方向是什么，这就造成了搜索的盲目和浪费。

布线问题

- 更为致命的是，即使我们搜索到了一条由a至b的路径，我们根本无法保证它就是所有路径中最短的，这要求我们必须把整个区域的所有路径逐一搜索后才能得到最优解。正因为如此，布线问题不适合用回溯法解决。

布线问题

```
#include <iostream>
#include <queue>
using namespace std;
int m=8;
int n=8;
int grid[10][10];
int indexcount=0;
struct Position
{
    int row;
    int col;
};
```

布线问题

```
void showPath()
{
    for(int i=0; i<10; i++)
    {
        for(int j=0; j<10; j++)
            cout<<grid[i][j]<<" ";
        cout<<endl;
    }
    cout<<"-----"<<endl;
}
```

布线问题

```
bool FindPath(Position start, Position finish, int &PathLen, Position
    *&path)
{ //计算从起点位置start到目标位置finish的最短布线路径，找到最
  短布线路径则返回true，否则返回false
  if((start.row==finish.row) && (start.col==finish.col))
  { PathLen=0;
    cout<<"start=finish"<<endl;
    return true;
  } //start=finish
  //设置方格阵列“围墙”；初始化图，-1为未访问
  for(int i=1; i<9; i++)
  { for(int j=1; j<9; j++)  grid[i][j]=-1; }
```

布线问题

//添加阻挡点

grid[2][3]=-2;

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

//初始化相对位移

cout<<"完整图"<<endl; showPath();

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

int NumOfNbrs=4; //相邻方格数

布线问题

```
Position here, nbr;
here.row=start.row;
here.col=start.col;
grid[start.row][start.col]=0;
//标记可达方格位置
cout<<"布线前图"<<endl;    showPath();
queue<Position> Q;
do //标记相邻可达方格
{ 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) //该方格未被标记
```

布线问题

```
{  grid[nbr.row][nbr.col]=grid[here.row][here.col]+1;
    //cout<<nbr.col<<" " <<nbr.row<<endl;//显示坐标
    if((nbr.row==finish.row) &&(nbr.col==finish.col)) break; //完成
    Q.push(nbr);
}
}
```

//是否到达目标位置finish?

```
    if((nbr.row==finish.row)&&(nbr.col==finish.col)) break;//完成
//活结点队列是否非空?
    if(Q.empty()) return false;//无解
    here = Q.front();    //cout<<here.col<<" " <<here.row<<endl;
    Q.pop();//取下一个扩展结点
```

布线问题

```
    indexcount++;  
    // cout<<"下一节点"<<indexcount<<endl;  
} while(true);  
//构造最短布线路径  
    PathLen=grid[finish.row][finish.col];  
    path=new Position[PathLen];  
//从目标位置finish开始向起始位置回溯  
    here=finish;  
    for(int j=PathLen-1; j>=0; j--)  
    {  
        path[j]=here;
```

布线问题

//找前驱位置

```
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;//向前移动

}

return **PathLen**;

}

int main()

{

布线问题

```
int main()
{  Position start;  start.col=1;  start.row=1;
   cout<<"布线起点"<<endl;
   cout<<start.col<<" "<<start.row<<endl;
   Position finish;  finish.row=3;  finish.col=4;
   cout<<"布线结束点"<<endl;
   cout<<finish.col<<" "<<finish.row<<endl;
   int PathLen=0;
   Position *path;
   FindPath(start,finish,PathLen,path);
   cout<<"布线后路径图"<<endl;
   showPath();
```

布线问题

```
cout<<"路径"<<endl;
for(int i=0; i<PathLen; i++)
{
    cout<<path[i].col<<" "<<path[i].row<<endl;
}
cout << "布线问题完毕!" << endl;
system("pause");
return 0;
}
```

布线起点

1 1

布线结束点

4 3

布线前图

```
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2
-2 0 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -2 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2
```

布线后路径图

```
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2
-2 0 1 2 3 4 5 -1 -1 -2
-2 1 2 -2 4 5 -1 -1 -1 -2
-2 2 3 4 5 -1 -1 -1 -1 -2
-2 3 4 -1 -1 -1 -1 -1 -1 -2
-2 4 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -1 -1 -1 -1 -1 -1 -1 -1 -2
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2
```

路径

1 2

1 3

2 3

3 3

4 3

布线问题完毕!

提纲

- 分支限界法的基本思想
- 装载问题
- 布线问题
- 0-1背包问题
- 小结

0-1背包问题

- 给定 n 种物品和一个背包，物品 i 的重量是 w_i ，价值 p_i ，背包容量为 C ，问如何选择装入背包的物品，使装入背包中的物品的总价值最大？
- 对于每种物品只能选择完全装入或不装入，一个物品至多装入一次。

0-1背包问题

■ 优先级如何确定？

- 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。

■ 如何实现？

- 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。

0-1背包问题算法思想

- 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。

0-1背包问题举例

- 设 $N=3$, $W=(16,15,15)$,
 $P=(45,25,25)$, $C=30$
- 优先队列式分支限界法

活动结点表:

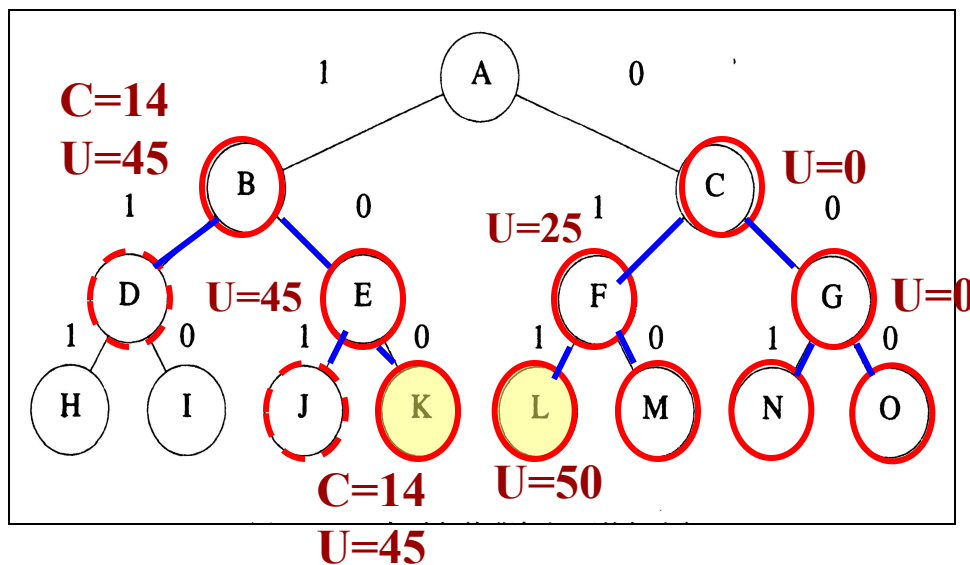
{B, C},

{E, C},,

{C},

{F, G},,

{G}



0-1背包问题上界函数

Typep Bound(int i)

{ //计算结点所相应价值的上界

Typew cleft=c-cw; //剩余容量; cw, 当前装包重量

Typep b=cp; //价值上界; cp, 当前装包价值

//以物品单位重量价值递减序装填剩余容量

while(i<=n&&w[i]<=cleft) //n表示物品总数

{ 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; //b为上界函数

}

将新活结点插入优先队列中

```
void AddLiveNode(Typep up, Typep cp, Typew cw, bool ch,
    int lev)
{ //将一个新的活结点插入到子集树最大堆H中
  bbnode *b=new bbnode;
  b.parent=E;   b.LChild=ch; //E, 指向扩展结点的指针
  HeapNode N;
  N.upprofit=up;   N.profit=cp;
  N.weight=cw;      N.level=lev;
  N.ptr=b;
  H.Insert(N);
}
```

对子集树优先队列分支限界搜索

```
Typep MaxKnapsack( )  
{ //返回最大值, bestx返回最优解  
    bestx=new int[n+1];  
    int i=1; E=0; cw=cp=0; //初始化  
    Typep bestp=0; //当前最优值  
    Typep up=Bound(1); //价值上界  
    //搜索子集树  
    while (i != n+1) { // 非叶结点  
        Typep 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], 1, i+1);}
```

对子集树优先队列分支限界搜索

```
    up = Bound(i+1);
    if (up>=bestp) // 右子树可能含最优解
        AddLiveNode(up, cp, cw, 0, i+1);
    HeapNode N;    DeleteMax(N); //取下一个扩展节点
    E=N.ptr;        cw=N.weight;    cp=N.profit;
    up=N.uprofit;    i=N.level;
}
//构造当前最优解
for(int j=n;j>0;j--)
    { bestx[j]=E.LChild; E=E.parent; }
return cp;
}
```


0-1背包问题源代码

```
#include "test.h"
#include <iostream>
using namespace std;
class Object
{ template<class Typew, class Typep>
  friend Typep Knapsack(Typep p[], Typew w[], Typew c,
    int n, int bestx[]);
  public:
    int operator <= (Object a) const
    { return d>=a.d; }
  private: int ID; float d;//单位重量价值
};
```

0-1背包问题源代码

```
template<class Typew,class Typep> class Knap;  
class bbnode  
{  
    friend Knap<int,int>;  
    template<class Typew,class Typep>  
    friend Typep Knapsack(Typep p[],Typew w[],Typew c,int  
        n, int bestx[]);  
    private:  
        bbnode * parent;    //指向父节点的指针  
        bool LChild;        //左儿子节点标识  
};
```

0-1背包问题源代码

```
template<class Typew,class Typep>
class HeapNode
{ friend Knap<Typew,Typep>;
  public:
      operator Typep() const
      { return uprofit; }
  private:
      Typep uprofit, profit; //节点的价值上界, 相应的价值
      Typew weight;         //节点所相应的重量
      int level;            //活节点在子集树中所处的层序号
      bbnode *ptr; //指向活节点在子集中相应节的指针
};
```

0-1背包问题源代码

```
template<class Typew,class Typep>
class Knap
{
    template<class Typew,class Typep>
    friend Typep Knapsack(Typep p[],Typew w[],Typew c,int
        n, int bestx[]);
    public: Typep MaxKnapsack();
    private: MaxHeap<HeapNode<Typep,Typew>> *H;
    Typep Bound(int i);
    void AddLiveNode(Typep up,Typep cp,Typew cw,bool
        ch,int lev);
};
```

0-1背包问题源代码

bbnode *E; //指向扩展节点的指针

Typew c; //背包容量

int n; //物品数

Typew *w; //物品重量数组

Typep *p; //物品价值数组

Typew cw; //当前重量

Typep cp; //当前价值

int *bestx; //最优解

};

0-1背包问题源代码

```
template <class Type>
inline void Swap(Type &a,Type &b);
template<class Type>
void BubbleSort(Type a[],int n);
int main()
{ int n = 3;//物品数
  int c = 30;//背包容量
  int p[] = {0,45,25,25};//物品价值 下标从1开始
  int w[] = {0,16,15,15};//物品重量 下标从1开始
  int bestx[4];//最优解
  cout<<"背包容量为: "<<c<<endl;
  cout<<"物品重量和价值分别为: "<<endl;
```

0-1背包问题源代码

```
for(int i=1; i<=n; i++)
{ cout<<"("<<w[i]<<","<<p[i]<<") "; }    cout<<endl;
    cout<<"背包能装下的最大价值为:
"<<Knapsack(p,w,c,n,bestx)<<endl;
cout<<"此背包问题最优解为:"<<endl;
for(int i=1; i<=n; i++)
{
    cout<<bestx[i]<<" ";
}
cout<<endl;
return 0;
}
```

0-1背包问题源代码

```
template<class Typew,class Typep>
Typep Knap<Typew,Typep>::Bound(int i)//计算上界
{  Typew cleft = c-cw;           //剩余容量高
   Typep b = cp;                 //价值上界
   //以物品单位重量价值递减序装填剩余容量
   while(i<=n && w[i]<=cleft)
   {   cleft -= w[i]; b += p[i]; i++;}
   //装填剩余容量装满背包
   if(i<=n)
   {b += p[i]/w[i]*cleft;}
   return b;
}
```


0-1背包问题源代码

//将一个活节点插入到子集树和优先队列中

```
template<class Typew,class Typep>
void Knap<Typew,Typep>::AddLiveNode(Typep up, Typep
    cp, Typew cw, bool ch, int lev)
{  bbnode *b = new bbnode;
   b->parent = E;
   b->LChild = ch;
   HeapNode<Typep,Typew> N;
   N.upprofit = up;    N.profit = cp;    N.weight = cw;
   N.level = lev;      N.ptr = b;      H->Insert(N);
}
```

0-1背包问题源代码

//优先队列式分支限界法，返回**最大价值**，**bestx**返回**最优解**

```
template<class Typew,class Typep>
```

```
Typep Knap<Typew,Typep>::MaxKnapsack()
```

```
{ H = new MaxHeap<HeapNode<Typep,Typew>>(1000);
```

```
    //为bestx分配存储空间
```

```
    bestx = new int[n+1];
```

```
    //初始化
```

```
    int i = 1; E = 0; cw = cp = 0;
```

```
    Typep bestp = 0;//当前最优值
```

```
    Typep up = Bound(1);    //价值上界
```

0-1背包问题源代码

//搜索子集空间树

while(i!=n+1)

{ //检查当前扩展节点的左儿子节点

Typew wt = cw + w[i];

if(wt <= c)//左儿子节点为可行节点

{ if(cp+p[i]>bestp)

{ bestp = cp + p[i]; }

AddLiveNode(up,cp+p[i],cw+w[i],true,i+1);

}

up = Bound(i+1);

0-1背包问题源代码

```
//检查当前扩展节点的右儿子节点
if(up>=bestp)//右子树可能含有最优解
{AddLiveNode(up,cp,cw,false,i+1);}
//取下一扩展节点
HeapNode<Typep,Typew> N;
H->DeleteMax(N);
E = N.ptr;
cw = N.weight;
cp = N.profit;
up = N.uprofit;
i = N.level;
}
```

0-1背包问题源代码

```
//构造当前最优解
for(int j=n; j>0; j--)
{
    bestx[j] = E->LChild;
    E = E->parent;
}
return cp;
}
```

0-1背包问题源代码

//返回最大价值，bestx返回最优解

```
template<class Typew,class Typep>
```

```
Typep Knapsack(Typep p[], Typew w[], Typew c, int n, int  
    bestx[])
```

```
{ //初始化
```

```
    Typew W = 0;    //装包物品重量
```

```
    Typep P = 0;    //装包物品价值
```

```
//定义依单位重量价值排序的物品数组
```

```
    Object *Q = new Object[n]; //单位重量价值数组
```

```
    for(int i=1; i<=n; i++)
```

```
    {    Q[i-1].ID = I; Q[i-1].d = 1.0*p[i]/w[i];
```

```
        P += p[i]; W += w[i];    }
```

0-1背包问题源代码

```
if(W<=c)
{ return P;//所有物品装包
}
//依单位价值重量排序
BubbleSort(Q,n);
//创建类Knap的数据成员
Knap<Typew,Typep> K;
K.p = new Typep[n+1];
K.w = new Typew[n+1];
for(int i=1; i<=n; i++)
{   K.p[i] = p[Q[i-1].ID]; K.w[i] = w[Q[i-1].ID]; }
```

0-1背包问题源代码

```
K.cp = 0;   K.cw = 0;   K.c = c; K.n = n;  
//调用MaxKnapsack求问题的最优解  
Typep bestp = K.MaxKnapsack();  
for(int j=1; j<=n; j++)  
{ bestx[Q[j-1].ID] = K.bestx[j];  
}  
delete Q;  
delete []K.w;  
delete []K.p;  
delete []K.bestx;  
return bestp;  
}
```

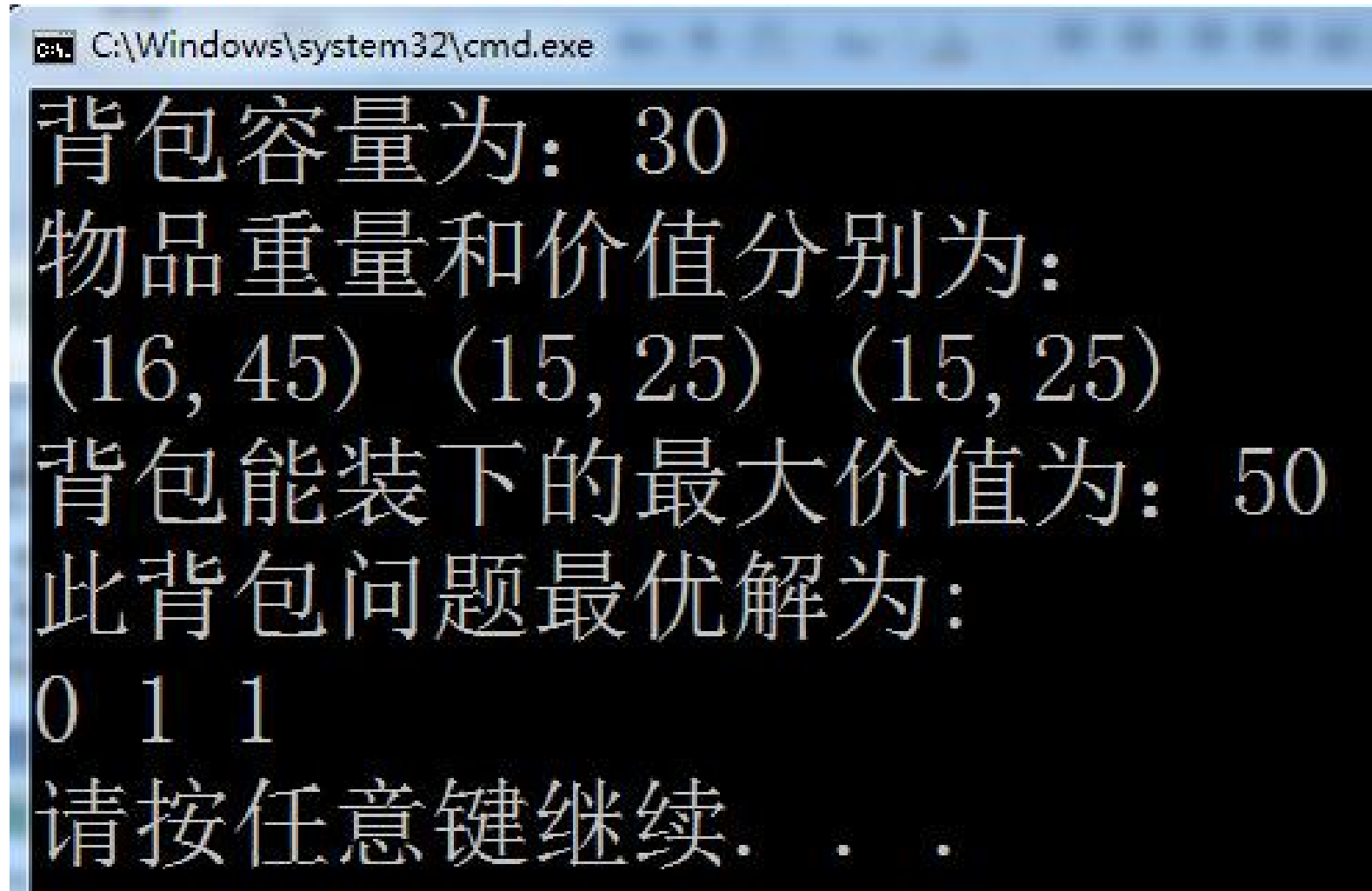

0-1背包问题源代码

```
template<class Type>
void BubbleSort(Type a[],int n)
{  bool exchange;
   for(int i=0; i<n-1;i++)
   { exchange = false ;
     for(int j=i+1; j<=n-1; j++)
     { if(a[j]<=a[j-1]){ Swap(a[j],a[j-1]);  exchange = true;}}
     //如果这次遍历没有元素的交换,那么排序结束
     if(false == exchange)
     {  break ;  }
   }
}
```

0-1背包问题源代码

```
template <class Type>
inline void Swap(Type &a,Type &b)
{
    Type temp = a;
    a = b;
    b = temp;
}
```

0-1背包问题运行结果



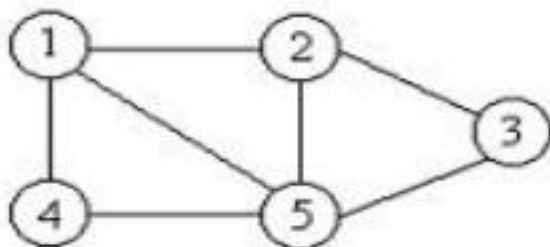
```
C:\Windows\system32\cmd.exe
背包容量为: 30
物品重量和价值分别为:
(16, 45) (15, 25) (15, 25)
背包能装下的最大价值为: 50
此背包问题最优解为:
0 1 1
请按任意键继续. . .
```

6.6 最大团问题

1. 问题描述

- 给定无向图 $G=(V, E)$ 。如果 $U \subseteq V$ ，且对任意 $u, v \in U$ 有 $(u, v) \in E$ ，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。
- 最大团问题的解空间树是一棵子集树
- 解最大团问题的优先队列式分支限界法, 与解装载问题的优先队列式分支限界法相似。

➤ 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

- 活结点优先队列中元素类型为CliqueNode
- 对每个活结点，用 cn 表示与该结点相应团的顶点数； $level$ 表示结点在子集空间树中所处的层次；用 $cn+n-level+1$ 作为顶点数上界 un 的值，其中 un 是该结点为根的子树中相应团的最大顶点数的上界。
- un 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 un 值的元素作为下一个扩展结点。

un 是当前团最大顶点数的上界， cn 表示当前团的顶点数。

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 cn 的值为0。 cn 表示当前团的顶点数。
 - ◆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。

◆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 $un = cn$ 。

◆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。

un 是当前团最大顶点数的上界， cn 表示当前团的顶点数。

提纲

- 分支限界法的基本思想
- 装载问题
- 布线问题
- 0-1背包问题
- 小结

小结

■ 分支限界法的基本思想

- 找出满足约束条件的一个解，或是满足约束条件的解中找出在某种意义下的最优解。

- 广度优先或以最小耗费(最大收益)优先的方式搜索解空间树。

■ 从活结点表中选择下一扩展结点的不同方式导致不同的分支界限法。

- 队列式(FIFO)分支限界法

- 优先队列式分支限界法

小结

- 应用举例
 - 装载问题
 - 布线问题
 - 0-1背包问题
 - 最大团问题

谢谢大家