高松群-PB16050141-第三次实验报告

0-1背包问题介绍

问题描述：

给定n种物品和一背包。物品i的重量是Wi，其价值为Vi，背包的容量为C。问：应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

抽象描述：整数规划问题

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

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

实现算法

枚举法

1.简介

将问题的所有可能的答案一一列举，然后根据条件判断此答案是否合适，合适就保留，不合适就丢弃。

2.算法

1. 产生背包问题可能的答案（共2^n种），并将其贮存在数组numerate中，number = 2^n;
2. 对于数组numerate中的每一组答案：
3. 进行判断，看其是否满足约束条件
4. 将可行解的背包价值与max进行比较：若价值temp大于max，则max = temp
5. 遍历结束，输出问题的最优解

3.具体实现及分析

```
// main 穷举法exhaustion
int numerate[1024][10] = {0};
createExhaustion(numerate, n);  //创建numerate数组，产生背包问题可能的所有答案
```

```
int number = (int)pow(2,double(n)); //number为解空间的大小
int max,temp,index,i,j;
max = 0;
for(j = 0; j < number; j++){    //对于numerate中的每一组解
    temp = findsolution(numerate[j], n, v, w, c); //求出解的背包价值
    if(temp>max){
        max = temp;
        index = j;
    }   //若大于max价值，则贮存：temp最高价值，j最高价值对应的解
}
//printSolve(numerate[index],n);
outfile << "穷举法:" << max << endl;   //输出最高价值以及对应的解
```

```
//创建枚举数组 位置i为1，则说明将物品i装入背包；为0，则不装入背包
void createExhaustion(int numerate[][10], int n){
    int num = (int)pow(2,double(n));
    int index = num/2;
    int i,j,temp,flag;
    for(j = 0; j < n; j++){
        temp = 0;
        flag = 0;
        while(temp<num){
            for(i = temp; i<index+temp; i++){ //从 temp:index+temp 对每个元素赋值 flag(0/1)
                if(flag == 0){
                    numerate[i][j] = 0;
                }
                else
                    numerate[i][j] = 1;
            }
            if(flag == 0) flag++; //如果这次循环是0，则下次是1
            else    flag--; //如果这次循环是1，则下次是0
            temp += index;  //下次赋值的位置紧接着
        }
        index /= 2; //向下降一位，此时循环的次数*2，而循环数/2
    }
}
```

```
//验证对应解是否满足约束，若满足，返回背包价值；若不满足，返回-1
int findsolution(int* numerate, int n, int* v, int* w, int c){
    int value = 0;
    int weigh = 0;
    for(int i = 0 ; i < n ; i++){
        if(numerate[i] == 1){
            value += v[i];
            weigh += w[i];
        }//累加
    }
    if(weigh > c)
        return (-1);
    else
        return value;
}
```

分析：创建数组的思路是，总共有n个物品，则存在2^n种可能解（从 000...0 至 111...1）

从0逐一增加到(2^n-1)的二进制数数组存在有特定的规律：

数组最高位 0~2^(n-1) 都是 0 ，而2^(n-1)+1 ~ 2^n-1 都是 1。次高位的规律则是 000111000111。

我们可以采用for循环来实现遍历数组：

最高位共有两次循环，index = num/2 。第一次循环全置入0，第二次循环全置入1。

次高位共有四次循环，index = index/2。第一次循环置0，第二次循环置1，第三次循环置0，第四次循环置1。

以此类推......

*temp用于决定循环的起始位置

动态规划

1.简介

动态规划主要用于求解以时间划分阶段的动态过程的优化问题，其核心思想为将多阶段过程转化为一系列单阶段问题，逐个求解。此外，最优性原理保证了每个阶段的子问题都是最优的。

2.子问题划分与临界条件

□ 最优值的递归式如下：

$$m(i,j)=\begin{cases}\max\{m(i+1,j),m(i+1,j-w_i)+v_i\} & j\ge w_i\\ m(i+1,j) & 0\le j<w_i\end{cases}$$

说明：当$j<w_i$时，只有$x_i=0$，$\therefore m(i,j)=m(i+1,j)$;

$$\text{当}j\ge w_i\text{时，}\begin{cases}\text{取}x_i=0\text{时，} & \text{为}m(i+1,j)\\ \text{取}x_i=1\text{时，} & \text{为}m(i+1,j-w_i)+v_i\end{cases}$$

□ 临界条件：

$$m(n,j)=\begin{cases}v_n & j\ge w_n\\ 0 & 0\le j<w_n\end{cases}$$

3.具体实现及分析

```
// main 动态规划算法 Dynamic planning
Knapsack(v,w,c,n,m);
Traceback(w,c,n,m,x);
outfile << "动态规划:" << m[0][c] << endl;
```

```
// 动态规划
```

```
void Knapsack(int* v, int* w, int c, int n, int m[][100]){
    int jMax;
    int i,j;
    jMax = (int)min(w[n-1]-1,c); //jMax为m[n][jMax]之前的所有树
    for(j = 0; j <= jMax; j++){
        m[n-1][j] = 0; //对于所有j<= jMax,m[n-1][j] = 0
    } //当背包分配给第n-1个物品的质量小于其质量时，不可能装下第n-1个物品
    for(j = w[n-1]; j <= c; j++){
        m[n-1][j] = v[n-1];
    } //当j>w[n-1]时，装下第n-1个物品可行，且为最优解
    for(i = n-2; i>0; i--){ //从后往前，对于每一个物品进行循环
        jMax = (int)min(w[i]-1,c);
        for(j = 0; j <= jMax; j++){
            m[i][j] = m[i+1][j];
        }//当分配给第i个物品质量小于其质量时，不可能装下它，因此m[i][j]=m[i+1][j]+0
        for(j = w[i]; j <= c; j++){
            m[i][j] = (int)max( m[i+1][j], m[i+1][j-w[i]] + v[i] );
        }//当分配给第i个物品质量大于其质量时，判断是否要装入物品i
    }//暂时对i=0 不作处理
    if(c>=w[0])
        m[0][c] = (int)max(m[1][c],m[1][c-w[0]]+v[0]); //判断是否要装入物品0
    else
        m[0][c] = m[1][c]; //不装入物品0
    //return m;
}
```

```
// 动态规划Traceback 将答案贮存在数组x中
void Traceback(int* w,int c,int n, int m[][100], int* x){
    int i;
    for(i=0;i<n;i++){
        if(m[i][c] == m[i+1][c])
            x[i] = 0;
        else{
            x[i] = 1;
            c -= w[i];
        }
    }
    x[n-1] = ((m[n-1][c])?1:0);
    //return x;
}
```

分析：

采用动态规划的方法，从求解先前划分的子问题的最优解开始，一步步扩大问题的规模，逼近答案，最后求出原始问题的解，即：m[0] [c]。

动态规划的思想简言之，是降低解空间的大小，在该子空间中进行遍历。

自顶向下的备忘录法

1.简介

备忘录动态规划法不仅具有通常动态规划方法的效率，同时还采取了一种自顶向下的策略。其思想是备忘原问题的自然但是低效的递归算法。像在通常的动态规划算法中一样，维护一个记录了子问题解的表，但有关填表动作的控制结构更像递归算法。

自顶向下的备忘录法避免求解不需要的子问题，同时避免相同问题的重复求解，因此能够提高求解时的效率。

2.具体实现及分析

```
//main 自顶向下的备忘录法 Memo method
Memorized(m,n,v,w,c);
Traceback(w,c,n,m,x);
outfile << "自顶向下的备忘录法:" << m[0][c] << endl;
```

```
int Lookup(int i, int j, int m[][100],int n, int* v, int* w, int c){
    int jMax;
    if(i<0 || j<0) return -1;
    if(m[i][j]>0 ) return m[i][j];
    //如果查到，返回该值 如果没查到，则创造这个值。
    if(i == n-1){
        jMax = (int)min(w[n-1]-1,c);
        if( j <= jMax)  m[n-1][j] = 0; //当背包分配给第n-1个物品质量小于其质量时，装不下
        else            m[n-1][j] = v[n-1];//当j>w[n-1]时，装下第n-1个物品可行，且为最优解
        return m[n-1][j];
    }//创建初始条件
    else if(i == 0){
        if(c>=w[0])
            m[0][c] = (int)max( (double)Lookup(1,c,m,n,v,w,c) , (double)Lookup(1,c-
w[0],m,n,v,w,c) + v[0]); //查询 m[1][c] 和 m[1][c-w[0]]哪个更好，要物品0还是不要物品0
        else
            m[0][c] = Lookup(1,c,m,n,v,w,c); //查询m[1][c]
        return m[0][c];
    }//最后一层（终止条件）
    else{
        jMax = (int)min(w[i]-1,c);
        if (j <= jMax ) {
             m[i][j] = Lookup(i+1,j,m,n,v,w,c); //不装入i,m[i][j] = m[i+1][j]
        }
        else
             m[i][j] = (int)max( (double)Lookup(i+1,j,m,n,v,w,c) , (double)(Lookup(i+1,j-
w[i],m,n,v,w,c)+v[i]) );//当分配给第i个物品质量大于其质量时，判断是否要装入物品i
        return m[i][j];
    }//正常情况
}
```

```
int Memorized(int m[][100], int n,int *v,int *w,int c){
    for (int i = 0; i <= n - 1; i++)
        for (int j = 0; j <= c; j++)
            m[i][j] = -1;
    // 查询m[0][c]
    return Lookup(0, c, m,n,v,w,c);
}
```

回溯法(含剪枝)

1.简介

回溯法式一个既带有系统性又带有跳跃性的搜索算法：

系统性：它依据深度优先的策略，从根节点出发搜索解空间树

跳跃性：算法搜索至解空间树的任意节点时，判断该节点为根的子树是否包含问题解，如果不包含，就进行剪枝，跳过该子树的搜索。

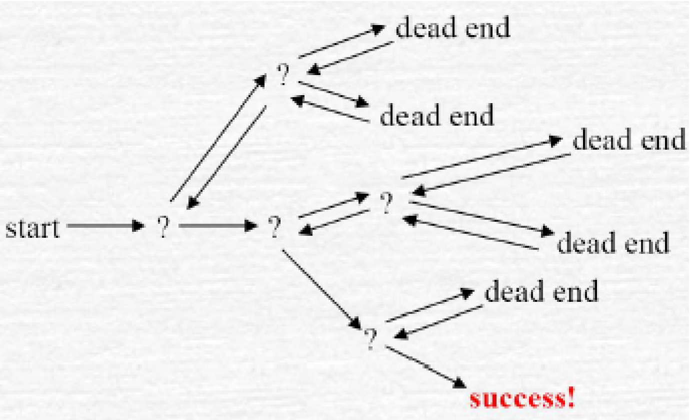
2.基本思想

搜索从根节点出发，以深度优先搜索整个解空间

开始节点变为活节点，同时也成为当前的拓展节点。从当前拓建节点，搜索向纵深方向移至一个新节点。该新节点成为新的活节点，并成为当前拓展节点。

如果当前节点不能向纵深方向进行拓展，则成为死节点

此时回溯到最近的一个或节点处，并使之成为当前的拓展节点，知道找到一个解或者全部解。



3.具体实现及分析

```
// main 回溯法 要求包含限界函数 recall
BackTrack_Packet( n, w, v, x, c);
outfile << "回溯法:" << m[0][c] << endl;
```

```
int KnapBacktrack(int i, int cw,int cv, int bestv, int* temp, int n, int* w,int* v, int*
x,int c)
{
    if (i > n-1) {
        if (bestv < cv) {
            bestv = cv;
            for (int j = 0; j < n; x[j] = temp[j++]);
        }
        return bestv;
    }// 搜索到可行解，返回
    else {
        if (cw + w[i] <= c) { //走左子树
            int sum = 0;
```

```
            for (int t = i; t <= n - 1; sum += v[t++]); //限界
            if (cv + sum > bestv) { //若cv+sum<=bestv，该解不为最优解，对子树剪枝
                temp[i] = 1;
                cw = cw + w[i];
                cv = cv + v[i];
                bestv = KnapBacktrack(i + 1, cw, cv, bestv, temp,n,w,v,x,c);
                cw = cw - w[i];
                cv = cv - v[i];
            }
        }
        //以下走右子树
        int sum = 0;
        for (int t = i + 1; t <= n - 1; sum += v[t++]); //限界
        if (cv + sum > bestv) {    //若cv+sum<=bestv，该解不为最优解，对子树剪枝
            temp[i] = 0;
            bestv = KnapBacktrack(i + 1, cw, cv, bestv, temp,n,w,v,x,c);
        }
    }
    return bestv;
}
```

```
void BackTrack_Packet(int n, int* w,int* v, int* x,int c)
{
    //主程序，进入
    int* temp = (int*)malloc(n * sizeof(int));
    KnapBacktrack(0,0,0,0,temp,n,w,v,x,c);
}
```

分支限界法

1.简介

有两种常见的分支限界法，

一是队列式（FIFO）分支限界法：从活节点表中取出结点的顺序与加入结点的顺序相同，因此活结点表的性质与队列相同

二是优先队列（代价最小或增益最大）分支限界法：每个结点都有一个对应的耗费或收益，以此决定结点的优先级。

0-1背包问题的求解采取的是FIFO队列分支限界法

2.具体实现及分析

```
    // main 分支限界法 branch bounding method
    max = BranchBound(n,v,w,c);
    outfile << "分支限界法" << max << endl;
```

```
int Bound(int i, int cw, int cv,int* w,int* v,int c, int n){
    //计算结点所对应的上界
    int cleft = c - cw; //剩余背包容量
    int bound = cv;     //价值上界
    while (i <= n && w[i - 1] <= cleft) { //以物品单位重量价值递减顺序装填剩余容量
        cleft -= w[i - 1];
        bound += v[i - 1];
        i++;
    }
    //此时i物品还没装但是已经不能完全装下了，所以就按比例把剩下的装下
    if (i <= n) bound += v[i - 1] / w[i - 1] * cleft;
    return bound;
}
```

```
int BranchBound(int n,int* v,int* w, int c){
    int temp, tempv, tempw;
    int j;
    int* kn = (int*)malloc( n * sizeof(int));
    for (int i = 0; i < n; i++) kn[i] = v[i] / w[i];
    for (int i = 1; i <= n - 1; i++) {
        if (kn[i] <= kn[i - 1])
            continue;
        else {
            temp = kn[i];
            tempw = w[i];
            tempv = v[i];
            for (j = i - 1; j >= 0; j--) {
                if (kn[j] < temp) {
                    kn[j + 1] = kn[j];
                    w[j + 1] = w[j];
                    v[j + 1] = v[j];
                }
                else break;
            }
            kn[j + 1] = temp;
            w[j + 1] = tempw;
            v[j + 1] = tempv;
        }
    }
    //cw为当前装包重量，cv为当前装包价值，bestv为当前最优值
    int cw = 0;
    int cv = 0;
    int bestv = 0;
    int i = 1;
    int up = 0; //up为结点的价值上界;
    priority_queue<PriorNode> q;        //定义优先队列
    up = Bound(i,cw,cv,w,v,c,n); // i是第i个stuff
    while (i != n+1) {
        // 左孩子即为将i装入背包，右孩子为不将i装入背包
        if (cw + w[i-1] <= c) {         //左孩子是可行结点，则加入背包
            if (cv + v[i-1] > bestv) bestv = cv + v[i-1];
            PriorNode plnode = { up,cv + v[i-1], cw + w[i-1], true, i + 1 };
```

```
            q.push(plnode);
        }
        up = Bound(i+1, cw,cv,w,v,c,n);
        if (up >= bestv) {//如果右子树可能包含最优解
            PriorNode prnode = { up,cv , cw , false, i + 1 };
            q.push(prnode);
        }
        up = q.top().up;
        cv = q.top().cv;
        cw = q.top().cw;
        i = q.top().number;
        q.pop();
    }
    return cv;
}
```

4.分支限界法解01背包问题时为什么要对物品按价值率排序?

bound函数计算的是结点对应价值的上界，这是一个估计值。为了保证估计的准确性，需要将物品按照价值率排序，这样可以证明，该估计值是严格大于等于该节点拓展后的最大价值。此时可以通过bound函数检查每个拓展结点的右结点。如果不按照价值率排序，每次对bound进行估计需要花费更多的时间进行排序。

5.分支限界法为什么可以保证在拓展到叶节点时即可获得最优解

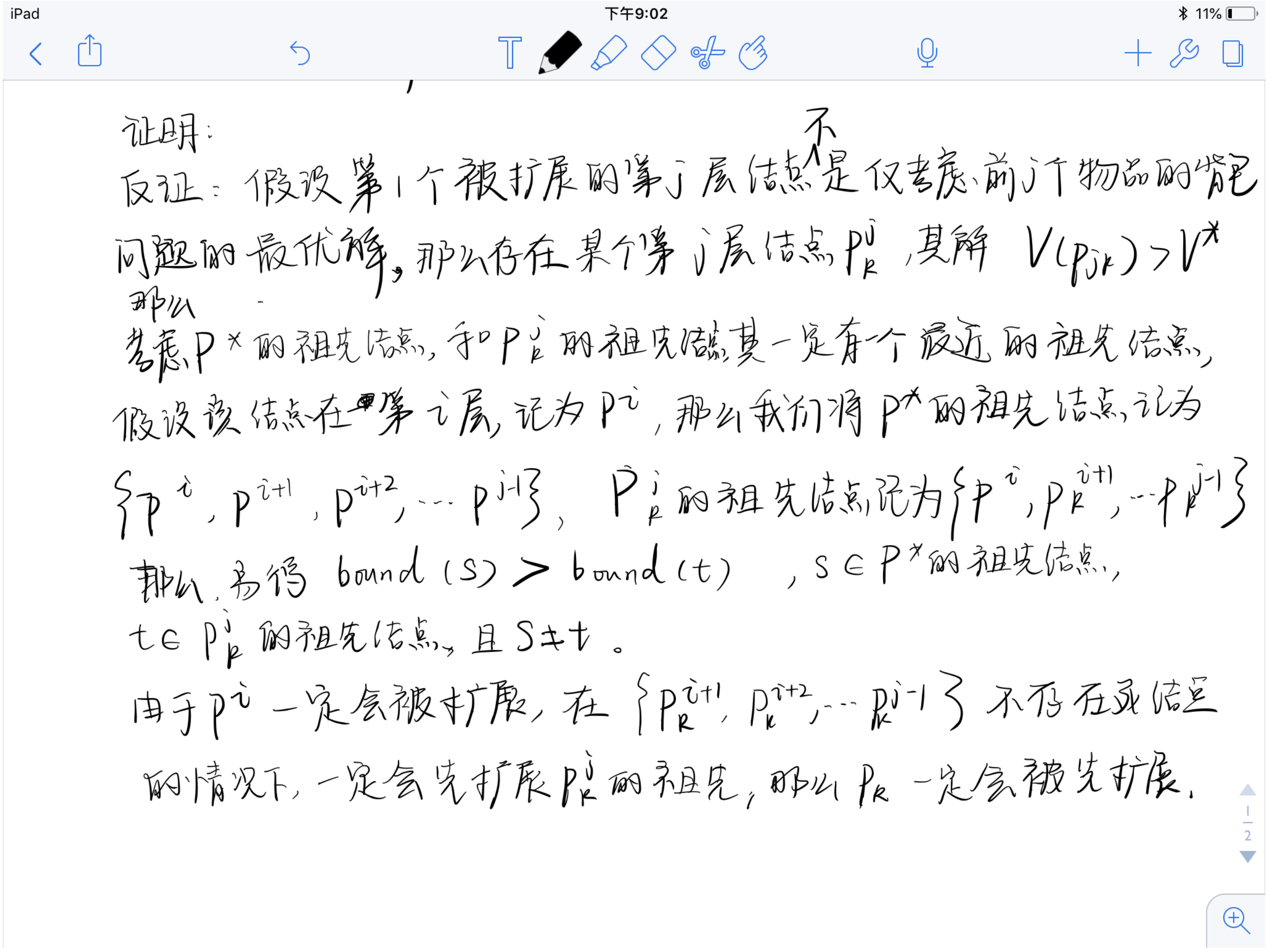
（反证法）：

假设拓展到叶结点时不是最优解，则存在另一个叶结点B，可行且价值大于该A结点，同时，**B结点而未被拓展。**

可以证明B结点在A结点的左边。根据队列和层序遍历的性质，在左边的结点最早被遍历。所以**B结点比A结点先被拓展**。此时与条件发生矛盾！

因此假设不成立。

第一个拓展到叶结点就是最优解！



证明：
反证：假设第1个被扩展的第j层结点不是仅考虑前j个物品的背包问题的最优解，那么存在某个第j层结点 p_k^j，其解 $V(p_{jk}) > V^*$
那么考虑 P^* 的祖先结点，和 P_k^j 的祖先结点，其一定有一个最近的祖先结点，假设该结点在第 i 层，记为 P^i，那么我们将 P^* 的祖先结点，记为 $\{P^i, P^{i+1}, P^{i+2}, \cdots P^{j-1}\}$，$P_k^j$ 的祖先结点记为 $\{P^i, P_k^{i+1}, \cdots P_k^{j-1}\}$
那么，易得 $bound(s) > bound(t)$，$s \in P^*$ 的祖先结点，$t \in P_k^j$ 的祖先结点，且 $s \neq t$。
由于 P^i 一定会被扩展，在 $\{P_k^{i+1}, P_k^{i+2}, \cdots P_k^{j-1}\}$ 不存在该结点的情况下，一定会先扩展 P_k^j 的祖先，那么 P_k 一定会被先扩展。

蒙特卡洛法

1.简介

当所求解问题是某种随机事件出现的概率，或者是某个随机变量的期望值时，通过某种“实验”的方法，以这种事件出现的频率估计这一随机事件的概率，或者得到这个随机变量的某些数字特征，并将其作为问题的解。（大数定理）

2.具体实现及分析

```
// main 蒙特卡洛法 Monte Carlo
max = MC_Packet(w,v,x,c,n);
outfile << "蒙特卡洛法:" << max << endl;
```

```
int MC_Packet(int* w, int* v, int* x, int c, int n)
{
    int test[100000] = { 0 }; //投点100000次
    int value = 0;
    srand((unsigned)time(NULL));
    int max = 0;
    for (int j = 0; j <= 100000 - 1; j++) {
        for (int i = 0; i < n; i++) {
            x[i] = rand() % 2;
            test[j] += x[i] * w[i]; //计算能否装入背包
```

```
            value += x[i] * v[i];//计算其价值
        }
        if (test[j] > c){
            test[j] = -1;
        }
        else
            if (value[j] > max) max = value[j];
    }
    return max;
}
```

算法间复杂度比较

枚举法

空间复杂度：O(2 ^ n) 时间复杂度：O(2^n) n为物品的数量

动态规划算法

空间复杂度：O(c * n) 时间复杂度：O(c*n)

自顶向下的备忘录法

空间复杂度：O(c * n) 时间复杂度：最坏情况O(c*n)

回溯法

空间复杂度：O(n) 时间复杂度：最坏情况:O(2^n)

回溯法效率主要依赖于：1.产生X[t]的时间 2.满足显约束的x[t]值的个数 3.计算约束函数constraint的时间 4.bound的时间 5.满足约束函数和上界函数约束的所有x[k]个数

分支限界法

空间复杂度：O(2^n) 时间复杂度：最坏情况O(2^n)

蒙特卡洛法

空间复杂度：Θ(n) 时间复杂度：Θ(N) N为实验的次数

算法间对比

枚举法与蒙特卡洛法：

枚举法利用计算机运算速度快、精确度高的特点，对要解决问题的所有可能情况，一个不漏地进行检验，从中找出符合要求的答案，因此枚举法是通过牺牲时间来换取答案的全面性。而蒙特卡洛法则是通过某种"实验"的方法，得到这个随机的结果，并将其作为问题的解，该解不一定是最优的。

动态规划与备忘录动态规划法：

动态规划算法通过最优子结构，将问题转换为子问题的求解。而备忘录动态规划法避免求取重复的子问题，此外只求取需要的子问题，从而减少时间复杂度。

分支限界法与回溯法：

回溯法根据数学表达式，搜索解向量(x1, x2, ..., xn)的整个解空间。搜索的时候利用贪心性质（按照单位重量价值递减排序，估算可能的最高上界）、以及已经计算出的可行解作为界限进行剪枝。

分支限界法的剪枝方法同回溯法是一样的。其不同点在于搜索解空间的遍历方式不同。**回溯法是深度优先，要穷尽解空间的所有可能，找到最优解。 分支限界法是广度优先，本质上也是穷尽了解空间的所有可能，找到最优解。**