经典算法设计技术研讨

分支界限法

目录/CONTENTS

01

概述

概　述

求解目标：分支界限法的求解目标则是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。
搜索方式：以广度优先或以最小耗费优先的方式搜索解空间树。分支界限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。
在分支界限法中，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。分支界限一般也分为两种，一种是队列式分支界限法，另一种是优先队列式分支界限法。
说人话，就是把所有结果变成一棵解空间树，从根结点到每个叶子结点都是一种结果，从中取最优解。
具体内容会在后面的算法思想部分具体阐述。

02

应用实例

应　　用　　实　　例

作为一种经典算法，分支界限法可以用于用于许多NP难题，例如

1）旅行商问题（TSP）

假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

2）01背包问题

给定n个重量为w1,w2,w3...wn的物品和容量为C的背包，求这个物品中一个最有价值的子集，使得在满足背包的容量的前提下，包内的总价值最大。

3）二次分配问题(QAP)

许多问题像集成电路布线、工厂位置布局、打字机键盘设计、作业调度问题等等，都可形式化为二次分配问题，如已知有n个位置和n家工厂，对于相对应的位置，距离是确定的，而对应的工厂之间，运输量是确定的，现在要将n家工厂建设在n个位置上，以最小化距离之和乘以相应的运输量，也可认为是距离最小运输量最大。

4）流水车间调度

有n台机器和m个作业，每个作业正好包含n个操作。作业的第i个操作必须在第i个机器上执行，没有机器可以同时执行一项以上的操作。每个作业的每个操作有确定的执行时间。一项作业中的操作必须按照指定的顺序执行，第一个操作在第一台机器上执行，然后（当第一个操作完成时）在第二台机器上执行第二个操作，依此类推，直到第n个操作。作业可以按任何顺序执行。问题定义意味着作业顺序对于每台机器都是完全相同的。问题在于确定最佳布置，实现最短全部作业完成的时间。

5）最大可满足性问题（MAX-SAT）

6）最近邻居搜索（NNS）

03

应用举例

应 用 举 例

我们以最简单的01背包，来详细阐述分支界限法。

P1048[NOIP2005 普及组] 采药

题目描述

辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是辰辰，你能完成这个任务吗？

输入格式

第一行有2个整数T（1≤T≤1000）和M（1≤M≤100），用一个空格隔开，T代表总共能够用来采药的时间，M代表山洞里的草药的数目。

接下来的M行每行包括两个在1到100之间（包括1和100）的整数，分别表示采摘某株草药的时间和这株草药的价值。

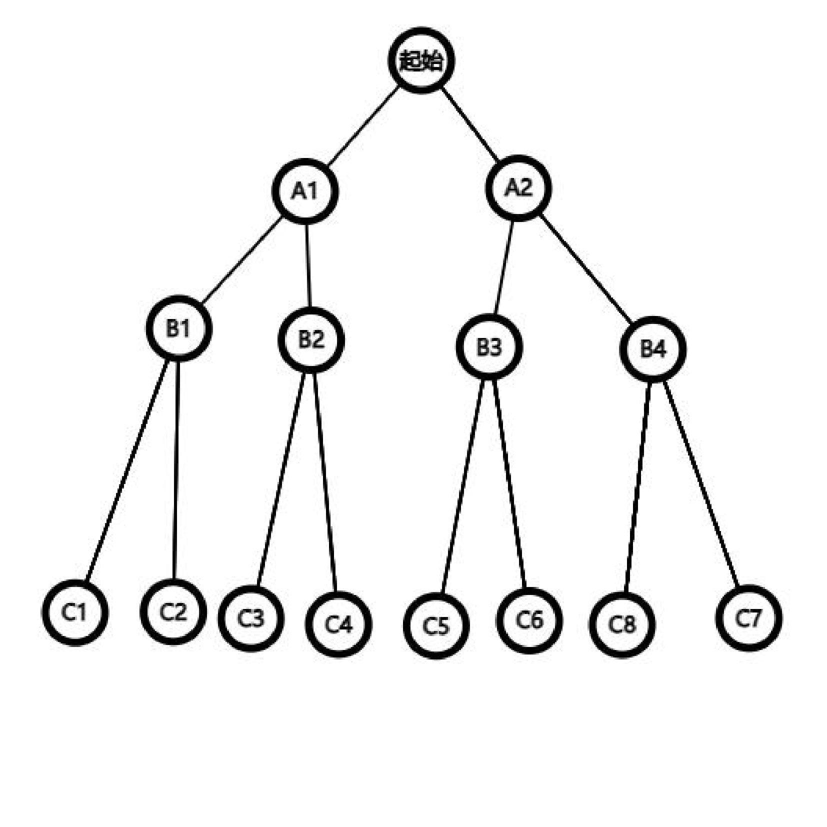
输出格式

输出在规定的时间内可以采到的草药的最大总价值。

输入：70 3　　输出：3
71 100
69 1
1 2

应 用 举 例

算法思想： 按照《算法设计与分析导论》一书，最初的01背包问题是求最大值问题，即获得最大价值，不能使用分支界限策略解决该问题，原因是书上定义构建一颗树后，分支界限是通过找最短路径到叶结点获得最优解的，因此我们需要将其修改为最小值问题，在其前面最大价值前添加负号，就能转换为最小值。
任何分支界限策略都需要一个分支机制，这个机制如右图，第一个分支将所有的解划分为两组，左边A1是选择该物品塞入背包，右边A2则是选择不把该物品塞入背包，如此重复，当列举完n个物品就能找到可行解，即最大值。
那么我们如何知道自己找到了可行解？并且以较小的代价获得？这里就需要用到所谓的扩展结点，我们希望找到一个较小的上界，如果知道找出的这个上界不能再小了时（已等于下界），就不再扩展这个结点，即不再遍历它的子结点。不扩展的条件为：
①这个结点本身表示一个不可行解。（如重量超过背包容量）
②这个结点的下界大于或等于当前找出的最小上界。
③这个结点的下界等于上界。



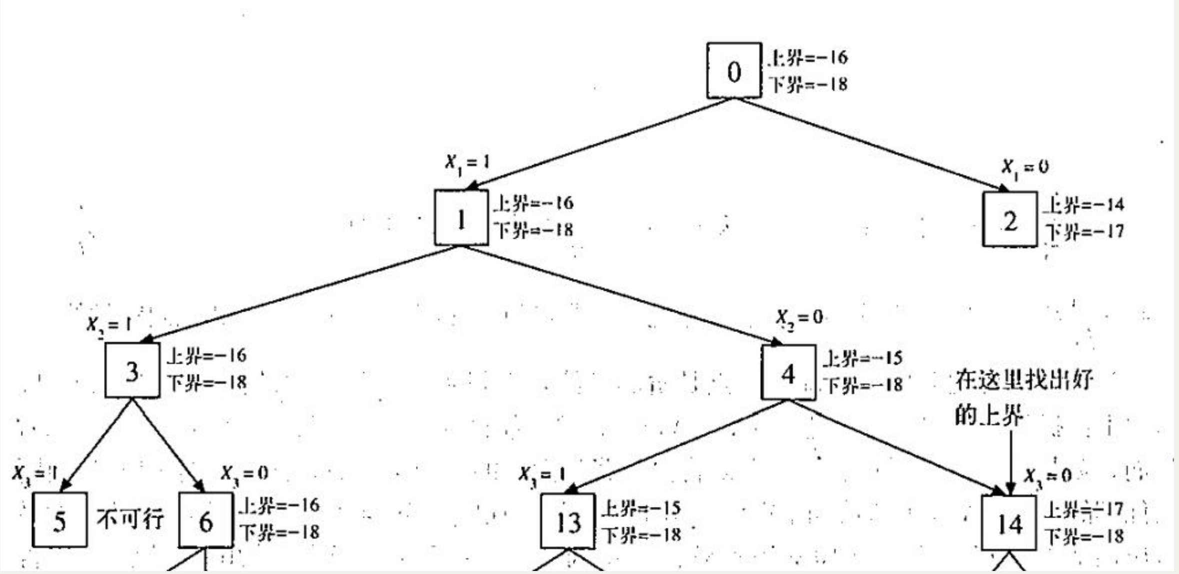
应　用　举　例

这里简单介绍一下上下界的计算，取下面这么一组数据，已经按照单位价值排好了序。

i	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$M = 34$

若我们选取1，2物品：W=29，P=-16(记得取负号)
若选取3物品，则W=37>34(×)，若刚好能满足34，则切割3物品，则29+8*5/8=34
P=-16-4*5/8=-18.5，取更大的下界，则P=-18
而若不切割3物品，背包留有空位，则上界就为P=-16，具体如下图



应 用 举 例

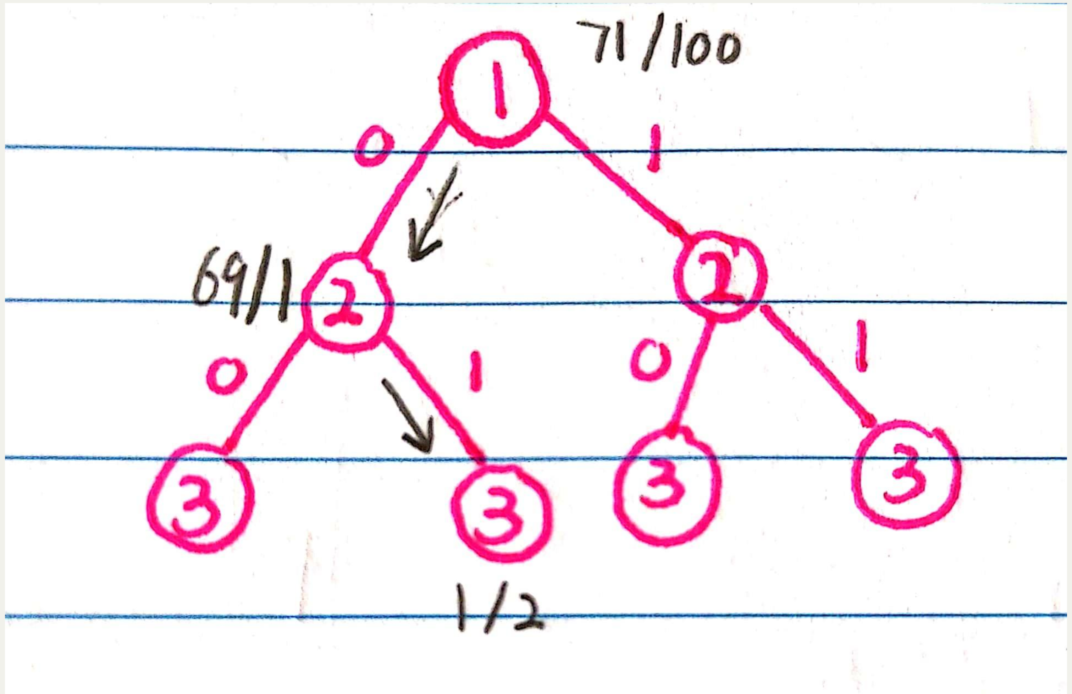
算法步骤： （采用队列式分支界限法）

① 用一个队列存储活结点，初始为空

② 1为当前扩展结点，其子结点左2为可行结点，右2结点不可行，舍弃，左2入队，并舍弃1。

③ 按FIFO原则，下一扩展结点为左2，其子结点左3右3均为可行叶结点，获得两个可行解1和3。

④ 左2为最后一个扩展结点，出队后，队列为空，算法结束，最优解为3。



应 用 举 例

代码实现：让我们用代码来理解一下。

```
//这里的代码采取计算上界，作为最大值问题简单处理，没有按照前文介绍的上下界做法
//来完成。
#include <bits/stdc++.h>
using namespace std;

struct Item
{
    float weight;//物品质量
    int value;//物品价值
};

//存储信息的结点
struct Node
{
    //level：层级，用来指示装入哪个物品
    //profit：目前装入的价值
    //bound：以该节点为根的子树能达到的价值上界
    int level, profit, bound;
    //目前装入的总重量
    float weight;
};
```

应　用　举　例

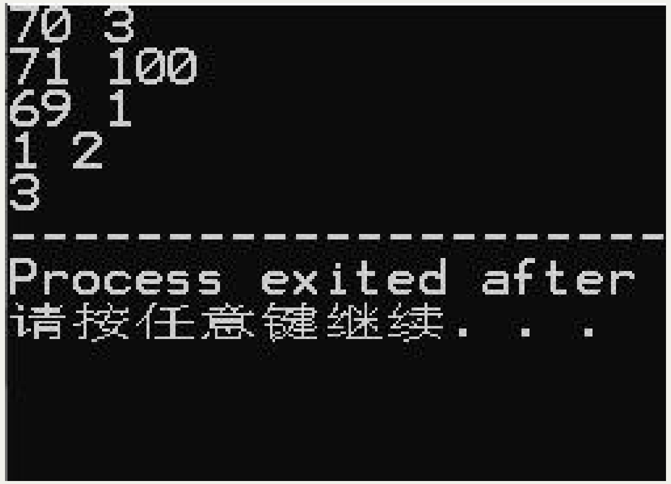
```
bool cmp(Item a, Item b)//单位价值排序
{
    double r1 = (double)a.value / a.weight;//单位价值
    double r2 = (double)b.value / b.weight;
    return r1 > r2;//降序排列
}

//返回以u为根的子树中的利润边界，主要使用贪心算法来寻找最大利润的上限值。
int bound(Node u, int n, int W, Item arr[])
{
    if (u.weight >= W)//如果该物品重量大于背包重量，返回0
        return 0;
    int profit_bound = u.profit;//通过当前利润初始化利润的界限
    int j = u.level + 1;//j总比当前层级多1
    int totweight = u.weight;
    // while循环，当物品索引小于n且累计重量小于总重量
    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight    += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }

    if (j < n)//如果j不是n，则包括最后一项的部分内容为利润的上限值
    profit_bound += (W - totweight) * arr[j].value/arr[j].weight;
    return profit_bound;
}
```

应　用　举　例

```
int knapsack(int W, Item arr[], int n)//返回值是背包容量为W时最大利润的函数
{
    sort(arr, arr + n, cmp);//按照单位价值排序
    queue<Node> Q;//建队来遍历结点
    Node u, v;
    u.level = -1; //开始时的空节点
    u.profit = u.weight = 0;
    Q.push(u);
    int maxProfit = 0;//逐一从决策树中提取一个物品，计算被提取物品的所有子物品的利润，并继续保存最大利润
    while (!Q.empty())
    {   u = Q.front();//出队
        Q.pop();
        if (u.level == -1)//如果是初始结点，层级设为0
            v.level = 0;
        if (u.level == n-1) //如果下一级为空
            break;
        v.level = u.level + 1;//如果不是最后一级，则增加层级，并计算子节点的利润。
        v.weight = u.weight + arr[v.level].weight;//将当前物品的重量和价值加到当前层级的树结点上
        v.profit = u.profit + arr[v.level].value;
        if (v.weight <= W && v.profit > maxProfit)//如果累计重量小于W，且利润大于之前的利润，则更新maxprofit
            maxProfit = v.profit;
        v.bound = bound(v, n, W, arr);//获取利润的上限，以决定是否将v添加到Q中
        if (v.bound > maxProfit)//如果边界值大于最大利润，则入队
            Q.push(v);
        v.weight = u.weight;//重复上述操作，但不添加该物品到背包中，即另一种可能性
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
    }
    return maxProfit;}
```

应 用 举 例

```
int main()
{
    int W;   //背包重量
    cin>>W;
    int n;   //物品数目
    cin>>n;
    Item* arr=new Item[n];
    for(int i=0;i<n;i++){
            cin>>arr[i].weight>>arr[i].value;
            }
    cout << "最大利润是: "<< knapsack(W, arr, n);
    return 0;
}
```



性能分析：单看前文代码，由于在knapsack函数的while中还调用了bound函数，而bound中也含有一个while函数，所以它的时间复杂度为O(n^2)。

参　　考　　文　　献

百度百科
《算法设计与分析导论》—— 李家同

THANKS FOR WHATCHING