算法分析与设计 0-1背包问题

PB20061210 赖永凡

实验要求

假设有n个物品和一个背包，每个物品重量为 $w_i(i=1,2,....,n)$，价值为 $v_i(i=1,2,....,n)$，背包最大容量为C，请问该如何选择物品才能使得装入背包中的物品总价值最大？最大价值为多少？请按照如下要求完成算法：

(1) 请利用分治法来求解该问题，给出最优解值以及求解时间；

(2) 请利用动态规划算法来求解该问题，给出得到最优解值以及求解时间；

(3) 请利用贪心算法来求解该问题，给出得到的最优解值以及求解时间；

(4) 请利用回溯法来求解该问题，给出得到的最优解值以及求解时间；

(5) 请利用分支限界法来求解该问题，给出得到的最优解值以及求解时间；

(6) 请利用蒙特卡洛算法来求解该问题，给出得到的最优解值以及求解时间；

(7) 给定相同输入，比较上述算法得到的最优解值和求解时间。当n比较大的时候，上述算法运算时间可能很长，请在算法中增加终止条件以确保在有限时间内找到最优解的值。

实验内容

分治法

对上述0-1背包问题，定义子问题 (i, c)，表示在可选物品标号为i + 1到n，可用背包容量为c时的最优解。对应的边界子问题即为 (n - 1, c)，表示只能选择最后一个物品的情形。因此，如果c大于等于最后一个物品的重量w[n - 1]时，选取此物品且价值为v[n - 1]是此边界子问题的返回值，若c小于w[n - 1]，则返回不选、价值为0。对于一般情形，即i小于n-1时，我们递归地计算子问题 (i + 1, c - w[i])（要求c >= w[i]）和 (i + 1, c)，以分别表示选和不选物品i的情形，并比较解的价值的大小，修改更大的解并返回。算法的伪代码如下：

```
DIVIDE()
    return DIVIDE_AUX(0, C)

DIVIDE_AUX(i, c)
    if(i == n - 1)
        if(c >= w[n - 1]) return ("1", v[n - 1])
        else return ("0", 0)
    aux_0 = DIVIDE_AUX(i + 1, c)
    if(c < w[i])
        return aux_0
    aux_1 = DIVIDE_AUX(i + 1, c - w[i])
    if(aux_0.value > aux_1.value)
        return ("0" + aux_0.solution , aux_0.value)
    else
        return ("1" + aux_1.solution , aux_1.value)
```

以上是分治法最基本的部分，由于其没有备忘录来记录中间子问题的结果，会导致一些子问题被重复计算，所以算法的时间复杂度较大。如果需要添加时间的限制，我们就需要保存一个全局的最优解，每个子问题在返回前都和全局最优解进行比较，如果某个子问题的解大于当前的最优解，就用此子问题的解来替代，并且记录下对应的开始物品标号i。这样，如果中途超时强制返回，我们就在目前找到的最优解的基础上，从物品i开始往前采用贪心的方法来获得一个近似的解。

固定背包容量为100，改变物品数n，实验结果如下（以下均设置时间上限为1e7）

n	20	30	40	50	100
时间	364	1479	5333	14813	TLE（最优解）

固定物品数为20，改变背包容量c，实验结果如下

c	5	50	100	200	500
时间	22	74	364	4069	102588

当n或c的值较大时，分治法所需的时间将快速增大。

动态规划

动态规划方法采用和分治法相同的子问题定义，但不同在于使用了矩阵 m[n][C + 1] 来储存每个子问题的解，并使用自底向上的顺序进行迭代。因此，对于动态规划算法，至多求解 m * (C + 1) 个子问题，时间复杂度为 $O\log(nC)$。在从m中构造最优解时，采用以下算法，自顶向下构造解，时间复杂度为 $O\log(n)$:

```
DP_CONSTRUCT_SOLUTION()
    cur_contain = c
    result.solution = ""
    for(i = 0; i < n - 1; i++)
        if(m[i][cur_contain] == m[i + 1][cur_contain])
            result.solution += '0'
        else
            result.solution += '1'
            cur_contain -= knap.weight[i]
```

虽然动态规划算法时间性能很好，我同样加入了超时停止的机制。为了方便起见，只有当每一行计算完毕时才检测是否超时，若超时，则记录下当前已经计算完毕的矩阵的行号i，表示从第i个物品开始选取的子问题我们已经研究完毕。类似于分治法中的处理，我们从矩阵m的第i行中选取对应的值最大，且列数j最小的位置，表示从i个物品开始选起，仅需要j的背包容量就可以获得最优解，对应的解向量可以用上述的方法构建。而对于i之前的物品，我们同样采用贪心的方法，来填满剩余的 C - j 的容量。

固定背包容量为100，改变物品数n，实验结果如下

n	20	30	40	50	100
时间	12	25	38	42	115

固定物品数为20，改变背包容量c，实验结果如下

c	5	50	100	200	500
时间	3	7	12	34	114

从实验结果看，动态规划算法速度远快于分治法，原因便在于其不需要重复求解子问题了。

贪心法

贪心算法比较简单，我采用的是对价值率进行贪心，从而使得背包内物品的价值尽可能的大。算法的流程就是，先计算出每个物品的价值率（价值除以重量），之后根据价值率让物品降序排列。从价值率最高的第一个物品开始，如果包内能够放下，就将其收入囊中，并修改剩余的背包容量；若不然，则直接跳过。这样依次进行，直到最后一个物品。贪心算法并不能确保找到全局的最优解，但它的时间复杂度仅为 $O(n)$，找到的解往往也能令人满意。

固定背包容量为100，改变物品数n，实验结果如下

n	20	30	40	50	100
时间	13	20	30	42	87

固定物品数为20，改变背包容量c，实验结果如下

c	5	50	100	200	500
时间	15	15	15	15	15

实验结果与分析一致。

回溯法

对于回溯法，我整体基于DFS来实现，伪代码如下，其中p存放目前找到的最优解。

```
BACKTRACE()
    REARANGE_WITH_VALUERATE()
    new p = {"", 0}
    BACKTRACE_AUX(0, "", 0, 0, p)
    return p

BACKTRACE_AUX(index, solution, weight, value, p)
    if(index == n - 1)
        if(weight + w[n - 1] <= c)
            solution = solution + "1"
            value = value + v[n - 1]
            if(value > p->value)
                p->value = value
                p->solution = solution
        else
            solution = solution + "0"
            if(value > p->value)
                p->value = value
                p->solution = solution

    if(value + value_sum[index] <= p->value)
        return;

    if(weight + w[index] <= c)
        BACKTRACE_AUX(index + 1, solution + "1", weight + w[index], value + v[index], p)

    BACKTRACE_AUX(index + 1, solution + "0", weight, value, p)
```

为了提高算法的效率，以及在超时之前尽量找到最优解，我采取了一些进一步的优化措施。首先为了更快地找到价值尽可能大的解，这里和贪心算法一样，都让物品根据价值率降序排列。其次是对那些会导致背包超重的节点进行剪枝，从而极大地减小了搜索空间。最后是限界函数的使用，这里我使用数组value_sum，其第i个元素表示从物品i开始，所有物品加在一起的价值。在算法中的具体含义便是，如果从当前物品开始把所有物品都放入背包中，所得的价值总量仍不如现已知的最优解，那么在当前节点的基础上继续搜索将没有意义。

由于回溯法中储存的最优解是完整的形式，故超时后无需任何处理即可作为算法的输出。

固定背包容量为100，改变物品数n，实验结果如下

n	20	30	40	50	100
时间	84	959	2714	15844	TLE（最优解）

固定物品数为20，改变背包容量c，实验结果如下

c	5	50	100	200	500
时间	10	58	84	280	107

理论上回溯法的时间复杂度为 $O(2^n)$，但由于限界函数和重量限制条件剪枝的机制，真正的搜索范围被缩小了很多。至于背包容量c的增大，虽然重量限制条件的剪枝数变少了，但由于保存的最优解的值也随之增大，限界函数的剪枝能力将提升不少，从而出现整体搜索范围减少的情形。

分支限界法

分支限界法可以看做将回溯法的DFS改成BFS，其中最显著的变化便是用队列和循环来替代递归。此外，这里我在实现的时候没对物品进行价值率的排序，因此用时增加不少。虽然我使用的是普通的FIFO队列，但使用贪心解作为当前的最优解来和后续的解进行比较，从而达到尽快获得一个解的目的。

固定背包容量为100，改变物品数n，实验结果如下

n	20	30	40	50	100
时间	414	1813	7410	23085	TLE（非最优）

固定物品数为20，改变背包容量c，实验结果如下

c	5	50	100	200	500
时间	31	139	414	2608	2193

由于没有事先将物品按照价值率排序或是采用特殊的优先队列机制，分支限界法的效率低于回溯法。

蒙特卡洛算法

作为随机算法，如果每次只是简单的生成一个长度为n的01字符串作为解，并带回检验的话，那么效率将极为低下，甚至难以找到可行的解。为了改进，我首先对物品按照价值率排序。之后从价值率最高的物品开始，如果背包放得下的话，就按照 $\epsilon-greedy$ 的准则选取物品，否则直接跳过该物品。虽然我选取 $\epsilon=0.5$ ，但由于背包容量的限制，价值率越低的物品选到的概率将越低（越后被选），在某种程度上也符合直觉。具体的伪代码如下：

```
MONTECARLO()
    REARANGE_WITH_VALUERATE()
    new result
    while(iter)
        new temp
        for(i = 0; i < n; i++)
            if(temp.weight + w[i] <= C && rand() < 0.5 )
                temp.weight += w[i]
                temp.value += v[i]
                temp.solution += "1"
            else
                temp.solution += "0"

        if temp.value > result.value
            result = temp
        iter--

    UTILIZE_SURPLUS()
    return result
```

循环重复次数由超参数iter决定以及给定的时间限制决定，在退出循环后还会对未选取的物品进行贪心，从而尽量使得背包足够满，从而使算法返回最接近最优解的结果。

固定背包容量为100，改变物品数n，实验结果如下（iter=1e6）

n	20	30	40	50	100
时间	638266	818133	919321	1064410	1870030

以上均找到了最优解。

固定物品数为20，改变背包容量c，实验结果如下

c	5	50	100	200	500
时间	328647	563387	638266	806807	1167740

对于物品数为50，背包容量500的问题，尝试不同的iter次数，观察多少次实验后可以得到最优解

iter	1e6	1e5	1e4	1e3
次数	1	2	7	34

以上结果均说明改进后的算法相较于随机生成解，不仅可以稳定地获得可行解，得到最优解的概率也大大提升了。

实验感想

第三次实验确实比第二次简单