基于贪心回溯的求解完全 0-1 背包问题的局部动态规划算法

任硕 郭子杰 裘天宝 何琨*

（华中科技大学计算机科学与技术学院，武汉 430074）

（brooklet60@hust.edu.cn）

Greedy Backtracking based Local Dynamic Programming for the Complete 0-1 Knapsack Problem

Ren Shuo, Guo Zijie, Qiu Tianbao, He Kun*

(School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074)

Abstract This paper proposes a local dynamic programming algorithm for the complete 0-1 knapsack problem. The algorithm uses greedy and backtracking to find the approximate optimal solution quickly, and then backtracking the results of local dynamic programming to approach the optimal solution, which considers the correctness as well as time complexity into account. Extensive experiments show that in most cases, compared with the classical dynamic programming algorithm, the proposed algorithm can accurately find the optimal solution of this problem within shorter time, and significantly improve the efficiency.

Keywords： Complete 0-1 knapsack problem; NP-hard; dynamic programming; greedy; backtracking

摘要 对于具有 NP 难度的完全 0-1 背包问题，提出了一种基于贪心与回溯思想的局部动态规划算法。该算法借鉴贪心与回溯技术以快速找到近似最优解，再通过局部动态规划的结果回溯逼近最优解，兼顾了算法的正确性与时间复杂度。大量的实验结果表明，与经典的动态规划算法相比，所提出的新算法在绝大多数情形下均能够在更短时间内准确找到问题的最优解，显著提高了算法的执行效率。

关键词 完全 0 -1 背包问题；NP 难度；动态规划；贪心；回溯

中图法分类号 TP301.6

1. 引言

0-1 背包问题又称为子集合问题，最早由 Merkel 和 Hellman 于 1978 年提出[1]，属于经典的 NP 难问题。问题可以简单表述为：在 *n* 类有确定重量与价值的物品中选择若干个物体放入容量有限的一个背包中，使得背包装入物品的总价值最大。在背包问题中，若每个物品可以被选择多次，且装入时不可拆分，则称为完全 0-1 背包问题。该问题常见于任务调度、投资选择、密匙生成和材料切割等场景，在理论研究与实际应用方面均有着重要的价值。

求解 0-1 背包问题的算法可分为两大类：确定性算法与群智能算法。穷举法、分支界定法、回溯法和动态规划法等算法会枚举所有可行解，属于经典的确定性求解算法，这种算法能够求出精确解，但时间复杂度大多呈指数爆炸，难以求解大规模的问题。而群智能算法一般可以在多项式时间内近似地求解大规模的 0-1 背包问题。

随着科学技术的发展和新型算法策略的不断涌现，0-1 背包问题的求解算法也由传统方法逐渐发展出众多的新型智能算法，如萤火虫算法、量子狼群算法、烟花算法、混合蝙蝠算法等[2-5]。例如，混合蝙蝠算法[3]是在基本蝙蝠算法的基础上进行改进，借鉴遗传算法[6]中的交叉和反置算子进行位置转移和局部搜索，也结合了贪心策略来加速算法的收敛速度；离散正余弦算法[4]使用非线性指数递减函数来更新个体步长，避免搜索的盲目性等，虽然在一定程度上能够提高搜索能力和求解速度，但并不能保证找到问题的最优解，其算法的高效性是以牺牲算法的最优性为代价的。

纵观已有的算法，均无法同时满足最优性和高效性，本文从传统的动态规划算法出发，提出了基于贪

心回溯思想的局部动态规划算法，在保证最优的前提下，大大提升了原算法的性能。

由于动态规划（Dynamic Programming, DP）方法并不关注物品的单位平均价值，而对背包容量进行遍历，因此可能出现大量冗余计算。考虑最大单位平均价值的特殊性，基于贪心思想进行最大单位平均价值成分的选取从而避免了冗余计算，然后利用动态规划进行回溯，对贪心的结果进行小范围的修正，以确保算法的正确性。对于一般的价值数据，本文提出的基于贪心回溯思想的局部动态规划算法(Local Dynamic Programming based on Greedy and Backtracking, LDPGB)，能够大大减小需要动态规划求解的范围，从而显著提升求解速度。

2. 相关工作

20世纪50年代，Richard E. Bellman在研究决策优化问题时，将待求解的多阶段决策优化问题划分为若干个单阶段问题逐一解决，并提出了决策科学中著名的最优化理论，为动态规划算法奠定了理论基础[8]。随后，M. Sniedovich阐述了该算法的最优化原理，并表明最优化原理的有效性保证了动态规划算法的正确性[9]。

P. Steffen指出，动态规划通过将问题递推分解并保留问题的中间结果来解决决策优化问题[8]，设计动态规划算法首先需要确保算法执行过程中保存了子问题的最优解。由于需要保存子问题的解，因此算法的空间复杂度依赖于问题需要计算的子问题数。文献中讨论动态规划的问题求解时，通过缩小问题规模的模式和缓存表的设计，给出相应的实例进行说明。

对于一个给定的问题，当该问题可以由其子问题的最优解获得时，则该问题具有“最优子结构”性质[1]。0-1背包问题具有最优子结构性。设$r_i(1\le i\le C)$为容量为i的背包的最优装包收益，如果$r_C = r_i + r_{C-i}$是最优装包收益，则r_i、r_{C-i}是相应子问题的最优装包收益。对于最优装包方案，将容量为C的背包看作大小为i与大小为$C-i$的两部分，则有：

$$r_C = r_i + r_{C-i} \tag{2.1}$$

对装包过程做如下简化：将一重量为i物体放入背包中，然后只对剩余容量为$C-i$的背包继续进行装包。此时有：

$$r_C = \max_{1\le i\le C}\{p_i + r_{C-i}\} \tag{2.2}$$

即原问题的最优解只包含一个相关子问题的解。根据上式递推求出r_C即可。

经典动态规划算法伪码如算法 1 所示。动态规划时间复杂度是 O(nC)。

算法1: DP算法

输入: $value, weight, C$

输出: maximum profit

```
let r[0..C] be a new array
For j = 1 to C do
    q = −∞
    For i = 1 to n do
        q = max(q, value[i] + r[max(0, j − weight[i])])
    End
    r[j] = q
End
Return r[C]
```

3．基于贪心回溯的局部动态规划算法

3.1 0-1 背包问题概述

经典的 0-1 背包问题可以简单描述为：给定 n 种物品集合 S = {1, 2, …, n}和一个背包容量为 C 的背包。每一种物品 i 都有它的重量w_i和价值 v_i（1, 2, 3, …, n）v_i。

求最优装包方案($x_1,x_2,...,x_n$), $x_i \in C$(x_i 表示背包中装载物品 i 的数量)，使得背包的总重量 W 不超过 C 的同时获得的价值 V 达到最大。在选择物品装入背包时，物品只有两种状态：整体装入或不装入。问题可以表述为：

$$\begin{aligned} &V = \max\left\{\sum_{i=1}^{n} v_i \cdot x_i\right\}, \\ &s.t. \ \sum_{i=1}^{n} w_i \cdot x_i \le C,\ x_i \in \{0,\ 1\} \end{aligned} \tag{3.1}$$

在式(3.1)中，如果x_i的取值范围由 0 或 1 拓展为非负整数，则问题转变为完全 0-1 背包问题。相应地，问题可以表述为：

$$\begin{aligned} &V = \max\left\{\sum_{i=1}^{n} v_i \cdot x_i\right\} \\ &s.t. \ \sum_{i=1}^{n} w_i \cdot x_i \le C,\ x_i \in \{0,\ 1,\ 2,\ ...\} \end{aligned} \tag{3.2}$$

下面以表 1 中数据为例进行说明。

表 1 物品参数表

w_i	1	2	3	4	5	6	7	8	9	10
v_i	1	5	8	9	10	17	17	20	24	3

3.2 完全 0-1 背包问题的求解思路

首先求解各个物品所对应的单位平均价值 p_i：

$$p_i = v_i / w_i \tag{3.3}$$

然后比较各个物品的单位平均价值，得到最大的单位平均价值p_α:

$$p_\alpha = \max_{1 \le i \le n} \{p_i\} \tag{3.4}$$

以及次大单位平均价值p_β：

$$p_\beta = \max_{\substack{1 \le i \le n \\ p_i \ne p_\alpha}} \{p_i\} \tag{3.5}$$

记p_α和p_β的对应物品的重量、价值、序号分别为w_α、v_α、α和w_β、v_β、β。

定理 1. 当容量 C 恰好是w_α的整数倍时，直接用重量为w_α的物品装满背包即可得到最大价值。

证明：

针对任意的装包方案$X = \{x_1, x_2, ..., x_n\}$，其应该满足如下限界条件：

$$\begin{cases} x_i\text{为非负整数 且 } x_i \le \lfloor C / w_i \rfloor \\ \sum_{i=1}^{n} x_i \cdot w_i \le C \end{cases} \tag{3.6}$$

定义对于装包方案$\boldsymbol{X}$，其对应的价值和为：

$$s_X = \sum_{i=1}^{n} v_i \cdot x_i \tag{3.7}$$

当 C 可以被w_α整除时，若全部用w_α对应的物品装满背包，记价值和为：

$$s_\alpha = v_\alpha \cdot \frac{C}{w_\alpha} = p_\alpha \cdot C \tag{3.8}$$

根据式(3.4)和式(3.7)，可以证明如下：

$$s_X = \sum_{i=1}^{n} (p_i \cdot w_i \cdot x_i) \le \sum_{i=1}^{n} (p_\alpha \cdot w_i \cdot x_i)$$

$$= p_\alpha \cdot \sum_{i=1}^{n} w_i \cdot x_i \le p_\alpha \cdot C = s_\alpha$$

从而得到结论：对于任意装包方案$\boldsymbol{X}$，只要w_α能够整除背包容量 C，则$\boldsymbol{X}$对应的装包物品价值总和小于s_α恒成立。

$$\forall w_\alpha, \boldsymbol{X}, C, w_\alpha \mid C \Rightarrow s_{\boldsymbol{X}} < s_\alpha \tag{3.9}$$

但是在一般情况下，即 C 不能够被w_α整除时，无法保证只用w_α对应的物品装入背包得到的总价值最大。例如，当 $C = 27$ 时，可以得到最优装包方案$\boldsymbol{X_1} = \{1, 0, 0, 0, 0, 1, 0, 0, 0, 2\}$及其总价值$s_{\boldsymbol{X_1}} = 78$。即背包中物品重量为{1，6，10，10}。而只用重量为w_α填装背包的装包方案$\boldsymbol{X_2}$总价值$s_{\boldsymbol{X_2}} = 77$，其中：

$$\begin{cases} \boldsymbol{X}_2 = \{0,0,0,0,0,0,1,0,0,2\} \\ \boldsymbol{S}_{X_2} = 77 \end{cases} \tag{3.10}$$

显然$s_{\boldsymbol{X_1}} > s_{\boldsymbol{X_2}}$，发生这种情况的原因是：在$p_1$中将$\boldsymbol{X}_2$中的容量 7 拆解成了容量 1 + 6，然后分别填装。因此可知用平均价值最大的物品贪心地装入背包不一定得到最优解，需要用动态规划进行部分回溯来保证算法的正确性。

但是，注意到如果 $C > n$，在动态规划得到的方案$\boldsymbol{X_1}$中也存在两个w_α(按照表 1 值为 10)，不难想到在剩余容量大于w_α时，装包方案中包含的w_α对应物品越多越好。

3.3 最大价值成分定理

本文采用以下的子问题划分方式：任何一种装包方案都可以拆分为两个部分，即只用第 α 个物体和不用第 α 个物体装包。

也就是对于一个方案$\boldsymbol{X} = (x_1, x_2, \ldots x_\alpha \ldots, x_n)$，可将其拆分为$\boldsymbol{X_1}$、$\boldsymbol{X_2}$，使其满足：

$$\begin{cases} \boldsymbol{X} = \boldsymbol{X}_1 + \boldsymbol{X}_2 \\ \boldsymbol{X}_1 \cdot \boldsymbol{X}_2 = 0 \end{cases} \tag{3.11}$$

假设$\boldsymbol{X_1}$、$\boldsymbol{X_2}$的形式如下：

$$\begin{cases} \boldsymbol{X}_1 = \{x_1, x_2, .., x_{\alpha-1}, 0, x_{\alpha+1}, .., x_n\} \\ \boldsymbol{X}_2 = \{0, 0, .., x_\alpha ..., 0\} \end{cases}$$

同样的，可以将背包分割为两部分，其中一部分的容量为 $x_\alpha \cdot w_\alpha$，用于$\boldsymbol{X_2}$方案；另一部分容量为$C - x_\alpha \cdot w_\alpha$，用于$\boldsymbol{X_1}$方案。而由式(3.9)可知，$\boldsymbol{X_2}$方案即为其对应部分背包的最优解。根据最优子结构性，$\boldsymbol{X_1}$方案也应当为对应部分背包的最优解。于是，要求解最优方案，只需求解该分割点。

定理 2. 最大价值成分定理. 设i为某一分割点，定义价值成分 $\Omega_i = i \cdot w_\alpha / C$，剩余容量$m = C \bmod w_\alpha$ 则：

$$\forall \Omega_i \le \Omega_t, s_i \le s_t \tag{3.12}$$

其中：

$$t = \left\lfloor \frac{C}{w_\alpha} \right\rfloor - \left\lceil \frac{p_\beta \cdot m - r[m]}{(p_\alpha - p_\beta) \cdot l_\alpha} \right\rceil - 1 \tag{3.13}$$

最大价值成分定理表明，在寻找最大价值的装包方案时，只需要关注分割点限界 t 以及 t 之后的情况即可，无需再考虑 t 之前的分割情况。

证明：

由$0 \le i \le \lfloor C/w_\alpha \rfloor$可知，在这种划分方式下，装包方案只有$\lfloor C/w_\alpha \rfloor + 1$种情况，只要计算这$\lfloor C/w_\alpha \rfloor + 1$种情况中总收益的最大值即可。最大收益可表示为：

$$ans = \max_{0 \le i \le \lfloor C/w_\alpha \rfloor} \{p_\alpha \cdot i \cdot w_\alpha + r[C - i \cdot w_\alpha]\} \quad (3.14)$$

当$i \le t$时，即$i \le \left\lfloor \frac{C}{w_\alpha} \right\rfloor - \left\lceil \frac{p_\beta \cdot m - r[m]}{(p_\alpha - p_\beta) \cdot w_\alpha} \right\rceil - 1$时，

移项可得：$\left\lfloor \frac{C}{w_\alpha} \right\rfloor - i \ge \left\lceil \frac{p_\beta \cdot m - r[m]}{(p_\alpha - p_\beta) \cdot w_\alpha} \right\rceil + 1$，从而有：

$$\frac{C}{w_\alpha} - i \ge \frac{p_\beta \cdot m - r[m]}{(p_\alpha - p_\beta) \cdot w_\alpha} + 1 \quad (3.15)$$

从而得到：

$$\begin{aligned} p_\beta \cdot (C - i \cdot w_\alpha) \le (C - (i+1) \cdot w_\alpha) \cdot p_\alpha \\ + r[m] - p_\beta \cdot m + p_\beta \cdot w_\alpha \end{aligned} \quad (3.16)$$

因为在对$\boldsymbol{X_1}$进行动态规划时不用α号物品装包，所以$\boldsymbol{X_1}$的所有装包方式的平均价值必然小于等于p_β，则：

$$r^{new}[C - i \cdot w_\alpha] \le p_\beta \cdot (C - i \cdot w_\alpha) \quad (3.17)$$

用$r^{new}[\cdot]$表示将第α号物品去掉后按照传统动态规划算法求得的最大值，因此可做以下推导：

第 i 种分割方法的价值为：

$$s_i = p_\alpha \cdot i \cdot w_\alpha + r^{new}[C - i \cdot w_\alpha] \quad (3.18)$$

在动态规划部分，最大单位平均价值为p_β，所以这一部分的价值一定小于等于全部按照p_β售出的价值，从而可以放缩如下：

$$s_i \le p_\alpha \cdot i \cdot w_\alpha + p_\beta \cdot (C - i \cdot w_\alpha) \quad (3.19)$$

又由式(3.16)可以将式(3.19)继续放缩：

$$\begin{aligned} s_i &\le p_\alpha \cdot i \cdot w_\alpha + (C - (i+1) \cdot w_\alpha) \cdot p_\alpha \\ &\quad + r[m] - p_\beta \cdot (m - w_\alpha) \\ &= p_\alpha \cdot (C - m) + r[m] - (w_\alpha - m) \cdot (p_\alpha - p_\beta) \\ &\le p_\alpha \cdot (C - m) + r[m] \end{aligned}$$

即得到：$s_i \le p_\alpha \cdot (C - m) + r[m]$ (3.20)

也就是说，$i \le t$时的价值小于$i = \lfloor C/w_\alpha \rfloor$时的价值，则最优分割点 i 必然大于等于 t。值得注意的是，当p_α和p_b大小接近时，t为负数，在这种特殊情况下，算法退化为传统动态规划算法。

4 局部动态规划算法

本节在最大价值成分定理的支持下，提出局部动态规划算法 LDPGB。算法的思路是采用贪心缩小问题规模，再进行回溯保证结果正确性的动态规划算法。

4.1 算法描述

首先为了保证最终的价值最大，应尽量多地装入α号物品。因此，运用贪心思想，先填装C/w_α个α号物品，再对剩余容量$m = C \bmod w_\alpha$进行回溯，每次回溯使剩余容量增加w_α，直至找到某一个分割点，使得价值最大。由式（3.10）可知，此时背包被分为两部分，一部分全部装入α号物品，而另一部分的装包方案可以使用动态规划算法在去除 α 号物品的剩余可选物品中进行求解。由最大价值成分定理可知，回溯次数上界 k_m 可提前计算：

$$k_m = \left\lfloor \frac{C}{w_\alpha} \right\rfloor - t = \left\lceil \frac{p_\beta \cdot m - r[m]}{(p_\alpha - p_\beta) \cdot l_\alpha} \right\rceil + 1 \quad (3.21)$$

于是需要动态规划的容量范围也可以事先确定。因此，在事先计算好回溯上界以及需要用动态规划求解的一定容量范围内的最大价值后，可以求出总体的最大价值。

LDPGB 算法首先进行小规模的动态规划求出$r[w_\alpha]$，之后根据式(3.12)求出分割点的限界 t，对k_m大小的背包使用动态规划，最后计算每个分割点的对应总价值并选出其中最大值。算法伪码如算法 2 所示。

算法2: LDPGB算法

输入: $value, weight, n, C$

输出: maximum profit

1: let $r[0..w_\alpha]$ be a new array

2: use DP to figure out the array r

3: $value[\alpha] = 0$

4: $k_m = \frac{p_\beta * m - r[m]}{(p_\alpha - p_\beta) * w_\alpha} + 1$

5: let $r^{new}[0..k_m * w_\alpha + m]$ *be a new array*

6: use DP to figure out the array r^{new}

7: $ans = 0$

8: **For** $i = 0$ to k_m **do**

9: $ans = \max(ans, p_\alpha w_\alpha (C - i) + r^{new}[m + i * w_\alpha])$

10: **End**

11: **Return** ans

4.2 算法复杂度分析

本算法中使用了两个动态规划数组，第一个动态规划数组 r 的大小正比于w_α，其空间复杂度为$O(w_\alpha)$；第二个动态规划数组 r^{new} 的空间复杂度为$O(k_m w_\alpha)$。

对于时间复杂度，初始化时找到w_α和w_β只需要遍历一遍 w 数组故而时间复杂度为$O(n)$。LDPGB 算法第一步动态规划的时间复杂度为$O(nw_\alpha)$，第二步动态规划时间复杂度为$O(nk_m w_\alpha)$。最后求 ans 的循环的时间复杂度为$O(k_m)$。

值得注意的是，当 p_α与 p_β非常接近时，k_m非常大，这时本算法相当于用动态规划对问题进行求解，即算法退化为传统动态规划算法。

5 实验结果与分析

目前，除了 DP 与本文提出的 LDPGB 算法之外，

并没有能够很好解决完全背包问题的完备精确算法或者近似优化算法，所以接下来均以 DP 作为比较对象测试 LDPGB 的性能。LDPGB 算法的正确性可以由后文表 2 和表 3 中 DP 与 LDPGB 的运行结果总是相同得到验证，因此实验分析时会着重讨论 LDPGB 算法求解时间的变化规律。

5.1 LDPGB 与 DP 在常规数据下的求解性能对比

本部分主要考虑在常规数据下，LDPGB 与 DP 的效率对比。对正常规模的设定如下：n=1,000，p_i 的差异在 0.01~1 之间，C 的规模从 2,000 到 20,000 变化，得到的结果如图 1（a）所示。

表 2 记录的是部分特定规模 C 对应的 DP 和 LDPGB 的运行时间与结果的具体数据。

橙色曲线表示随着 C 的规模从 2,000 到 200,000 时 LDPGB 算法需要花费的时间，蓝色曲线表示 DP 在同样的 C 的变化下的时间花费情况。可以得到，DP 的时间随 C 的增加呈线性增长趋势，而 LDPGB 算法对 C 的增加不敏感。同时，DP 的时间随 C 的增加呈线性增长趋势，而 LDPGB 对 C 的增加不敏感。

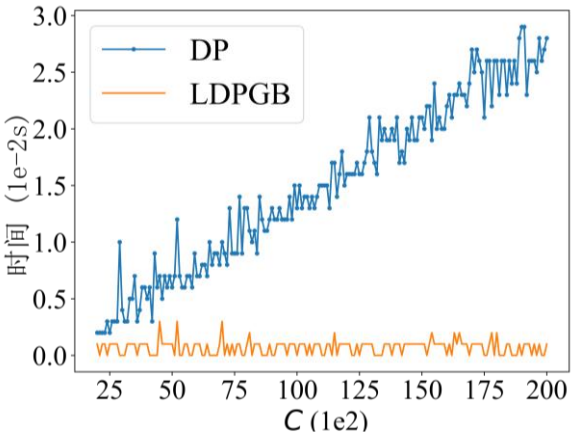
虽然能够比较直观得看出 DP 和 LDPGB 得变化趋势，但是也能够从图 1（a）以及表 2 前两行数据中看出由于此时的求解规模还比较小，两种算法的时间都比较小，所以由于测试环境的微小变化会在同一时刻产生同向的波动，为了减少这种波动对实验结果造成的影响，本文采用增加求解规模的方式得到了图 1(b)。求解规模较大时，能够明显得到 DP 和 LDPGB 对应的曲线都平滑了许多，但是曲线的变化趋势依旧分别是线性增加和接近常数。

另外，从图中可以明显看出，在一般情况下，LDPGB 在求解此类问题时的时间性能是非常优越的，原因在于 LDPGB 的时间复杂度只依赖于数据成分，而不依赖于数据规模，这点从表 2 中各行 LDPGB 的求解时间数据几乎相等可以进一步得到验证。

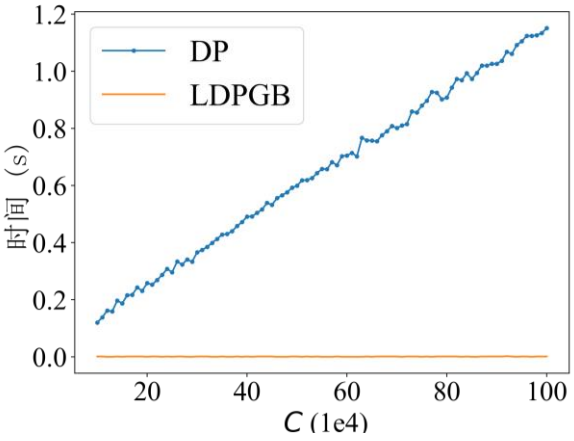
由 4.2 节可知，LDPGB 求解的时间花费主要取决于限界 t 的大小；t 的计算如式(3.13)所示，显然，$(p_\alpha - p_\beta)$越小，LDPGB 回溯路径就越长。

接下来，主要通过控制$(p_\alpha - p_\beta)$的大小来控制变量，进行多组实验。部分典型数据实例如表 2 所示，可以看到 LDPGB 表现出的时间性能非常优越且稳定。

图 1 中，T_{DP}曲线的变化趋势验证了在第 1.2 节分析的 DP 的时间复杂度$T_{DP}(n,C) = O(nC)$，在 n 一定的情况下$T_{DP} \propto C$；而在 LDPGB 中，时间耗费完全取决于回溯的过程，当回溯完成后问题也就求解完成了，如图 1，n 与 w、v 数组都给定的情况下，T_{LDPGB}的曲线近似为一条直线，也验证了这一点。



(a)



(b)

Fig. 1 time-capacity curve of LDPGB and DP

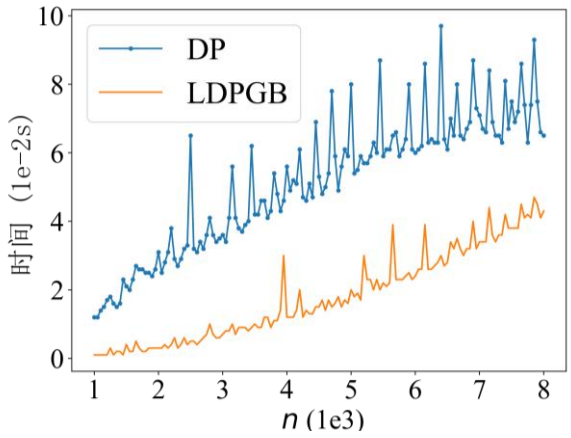
图 1 LDPGB 和 DP 的求解时间与背包容量关系曲线

表 2 DP 与 LDPGB 的运行时间与结果随 C 变化情况

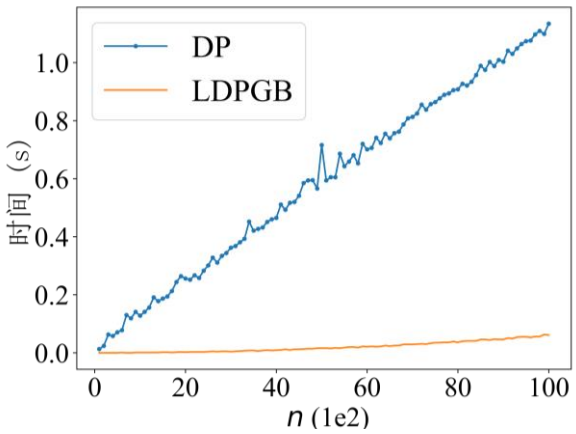
物品数 n	背包容量 C	运行时间（s）		结果	
		DP	LDPGB	DP	LDPGB
1000	2,000	0.002	0.001	811.111	811.111
1000	20,000	0.028	0.001	18000	18000
1000	100,000	0.120	0.001	40555.6	40555.6
1000	200,000	0.258	0.001	133333	133333
1000	500,000	0.599	0.001	450000	450000
1000	1000,000	1.151	0.001	453333	453333

5.2 LDPGB 与 DP 在非常规数据下求解性能对比

在经过第 5.1 节部分的探讨之后，本节尝试实验寻找 LDPGB 比传统 DP 优越的边界条件，因此有意缩小最大单位平均价值和次大单位平均价值，意在增加 LDPGB 的回溯路径长度以提高 LDPGB 的求解时间，然后对于更大跨度的 n 进行随机生成与求解。具体的操作是控制$p_\alpha - p_\beta$由原来的 1e-1 量级进一步缩小到 1e-3 乃至 1e-4 量级，得到的结果如图 2（a）所示，与本文 5.1 节同理，为了增加实验结果的准确性



(a)



(b)

Fig. 2 time-*n* curve of LDPGB and DP

图 2 LDPGB 和 DP 求解时间与 *n* 的关系曲线

表 3 DP 与 LDPGB 的运行时间与结果变化情况

物品数 n	背包容量 *C*	运行时间（s）		结果	
		DP	LDPGB	DP	LDPGB
1000	10,000	0.012	0.001	0.0405	0.0405
3000	10,000	0.036	0.007	0.0326	0.0326
8000	10,000	0.065	0.043	0.0300	0.0300
1000	100,000	0.128	0.001	1.0222	1.0222
5000	100,000	0.716	0.016	0.5889	0.5889
10000	100,000	1.134	0.061	0.8556	0.8556

与可信度，增加求解规模可以得到图 2（b），并且记录一些具体的数据结果得到表 3。

从图 2(a)中可以看出，对于$p_\alpha - p_\beta$的量级缩小，LDPGB 的求解性能仍然远优于 DP。接下来，就需要进一步增加量级的缩小规模去寻找 LDPGB 的最优执行范围边界。同时由图 2（a）（b）对比以及表 3 中数据可以得到，随着规模 C 的增长 DP 求解时间呈线性增长趋势而 LDPGB 基本不变，两曲线在图 2（a）比图 2（b）更接近，所以在下一部分 5.3 节中为了得到最佳适用范围的边界（需要探索 DP 和 LDPGB 对应

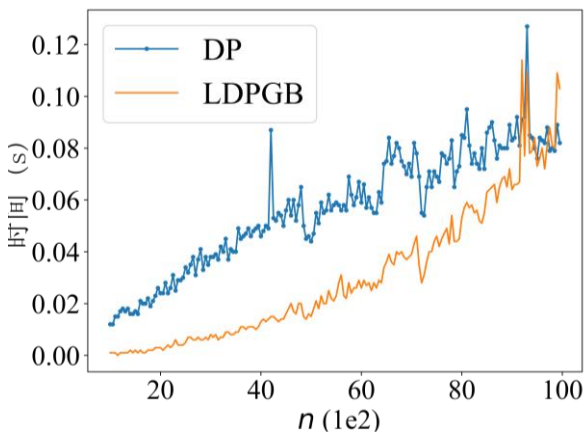


Fig. 3 time-*n* curve of LDPGB and DP

图 3 LDPGB 和 DP 的求解时间与 *n* 的关系曲线

表 4 DP 与 LDPGB 运行时间对比情况

物品数 n	背包容量 *C*	运行时间（s）		结果	
		DP	LDPGB	DP	LDPGB
1000	10,000	0.012	0.001	4.06e-7	4.07e-7
3000	10,000	0.038	0.008	6.11e-7	6.11e-7
6000	10,000	0.059	0.026	9.11e-7	9.11e-7
9000	10,000	0.083	0.064	7.00e-7	7.00e-7
10000	10,000	0.095	0.134	7.22e-7	7.22e-7

的曲线何时有相交的趋势），是在图 2（a）的小求解规模上 C 进行操作的。

值得一提的是，在图 2（a）（b）两图中，LDPGB 求解算法的时间并不是像图 1 中一样一成不变的，因为从第 4.2 节的算法复杂度分析可知，LDPGB 有一个时间复杂度为$O(nw_\alpha)$的求解 t 的过程和时间复杂度为$O(n)$的寻找w_α和w_β的过程，所以总体时间在w_α一定时$O(nw_\alpha + n)$正比于 n。在图 1 中，*n* 是保持不变的，而在图 2 中，*n* 是线性增加的，所以 LDPGB 在前者中时间保持近乎恒定，在后者中线性增加。

5.3 LDPGB 与 DP 在极端数据下的求解性能对比

在第 5.2 节中图 2（a），可以得到$p_\alpha - p_\beta$减少 3 个量级时，LDPGB 曲线并不能明显地逼近 DP，所以为了寻求 LDPGB 最佳适用范围的边界，下面直接将$p_\alpha - p_\beta$的量级在之前的基础上进一步缩小 9 个量级，然后在同样的变量控制条件下分别测试 LDPGB 和 DP 的求解性能，得到的结果如图 3 所示。

从图 3 中得到，在极端数据下随 *n* 的增长，T_{LDPGB}曲线逐渐逼近 T_{DP} 曲线。

下面将解释为什么在极端数据下，LDPGB 的执行时间会趋近甚至超过 DP。*t* 的定义可知，当$p_\alpha - p_\beta$非常小时，k_m非常大，而本算法回溯时动态规划的规模是 $O(k_m w_\alpha)$，此时 LDPGB 趋近于 DP；并且

由于 LDPGB 的预处理等步骤需要一定运行时间，因此在最大单位平均价值与次大单位平均价值的差距非常小时，LDPGB 的运行时间可能超过 DP。

而这也给出提示，在实际应用 LDPGB 求解问题时，可以加入一个特判：当$p_\alpha - p_\beta$成分的量级超出最优执行边界时，LDPGB 退化为 DP 算法，从而保证在任何情况下，LDPGB 的时间性能总是不劣于并且绝大多数时候是优于 DP 算法的。

表 4 给出两组在人为构造的极端特殊用例下 LDPGB 和 DP 的性能对比，此时的 LDPGB 运行时间比 DP 长，不过仍相差不大。

6 结束语

本文根据平均价值提出了一种基于贪心回溯的局部动态规划算法，显著缩减了经典动态规划算法针对常见的完全背包问题的计算复杂度，在时间复杂度以及空间复杂度方面均有明显的优势。同时，“局部贪心带回溯”的思想也为需要保证精确性的算法优化问题提供了新思路：并不是所有的优化方法都需要像群智能等优化算法一样需要以牺牲结果精确性为代价来提升算法的执行效率的。

当然，关于 LDPGB 算法的其他问题还有待进一步研究，比如说当求解规模 C 稍微大于 n 的时候，是否可以给出更高效的判断策略，判断此时是否不适合直接使用 LDPGB 算法，而非等待不适合时 LDPGB 算法因回溯路径过长而被动退化为 DP 算法等。

参 考 文 献

[1] T. H. Cormen，C. E. Leiserson and R. L. Rivest， Introduction to Algorithms，The MIT Press， 1990.

[2] 严雅榕，项华春，聂飞，等. 求解 0-1 背包问题的量子狼群算法[J]. 微电子学与计算机，2018，35(7): 1-5，12.

[3] 程魁，马良. 0-1 背包问题的萤火虫群优化算法[J]. 计算机应用研究，2013,30(4): 993-994，998.

[4] 郑健. 离散正弦余弦算法求解大规模 0-1 背包问题[J]. 山东大学学报（理学版），2020,55(11): 87-95.

[5] 徐小平，庞润娟，王峰等. 求解 0-1 背包问题的烟花算法[J]. 计算机系统应用，2019,28(2): 164-170.

[6] 万晓琼，张惠珍. 求解 0-1 背包问题的混合蝙蝠算法[J]. 计算机应用研究，2019，36(9): 2579-2583.

[7] Hristakeva M, Shrestha D. Solving the 0-1 knapsack problem with genetic algorithms[C]//Midwest instruction and computing symposium. 2004: 16-17.

[8] Bellman R. Dynamic Programming[M]. Princeton University Press, 1957.

[9] Sniedovich M. Dynamic Programming and Principles of Optimality[J]. Journal of Mathematical Analysis and Applications, 1978(65): 586-606.

[10] Steffen P， Giegerich R. Table design in dynamic programming[J]. Information and Computation, 2006(204):1325-1345.

[11] 李强. 动态规划算法时间效率优化策略研究[D]. 中南民族大学，2015.

[12] 孙佳宁，马海龙，张立臣，李鹏. 求解 0-1 背包问题的融合贪心策略的回溯算法[J]. 计算机技术与发展，2022,32(02):190-195.

[13] 王熙照，贺毅朝. 求解背包问题的演化算法[J]. 软件学报，2017，27 (1) : 1-16.

[14] Dexuan Zou,Liqun Gao,Steven Li,Jianhua Wua. Solving 0-1 knapsack problem by a novel global harmony search algorithm[J]. Applied Soft Computing,2011,11(2).

[15] Chen G L，Wang X F，Zhuang Z Q，et al.Genetic algorithm and its applications[M]. Beijing: The posts and Telecommunications Press，2003.1-192.

[16] Poli R，Kennedy J，Blackwell T.Particle swarm optimization[J]. Swarm Intelligence，2007，1 (1) :33-57.

作者介绍

Ren Shuo, born in 2001. Undergraduate in School of Computer Science and Technology, Huazhong University of Science and Technology.

Guo Zijie, born in 2002. Undergraduate in School of Computer Science and Technology, Huazhong University of Science and Technology.

Qiu Tianbao, born in 2003. Undergraduate in School of Computer Science and Technology, Huazhong University of Science and Technology.

He Kun, Professor, Doctoral supervisor, School of Computer Science, Huazhong University of Science and Technology. Senior member of ACM, Senior member of IEEE, CCF Distinguished member. Her research interests include machine learning, deep learning, optimization algorithm, and social networks