第八章 NP 问题和 NP—完备性理论

8.8 NP—难问题的研究方法

8.8.1 最优化方法

一、整数规划

大多数 NP—难的组合优化问题都可以建模为整数规划。例如，背包问题、TSP 等。由于整数规划也是 NP—难的组合优化问题，它的求解也只能依靠分枝定界、割平面等指数时间算法。

二、动态规划

动态规划(dynamic programming)是求解多阶段决策优化问题的一种有效方法。它可以用于离散、连续、随机等多种类型的问题，应用领域非常广泛。

设计动态规划的一种典型思路是给出需要求解的实例最优解与另一规模较小的实例最优解之间的递推关系。由于规模最小的那些实例最优解容易求得，从而可以以此为初始条件，进而利用递推关系一步一步求得最优解。

1. 0-1 背包和动态规划

0-1 背包问题：给定一个体积为 V 的背包，然后给定 1…n 件物品，第 i 件物品的体积为 w[i]，价值为 p[i]。要求选定若干件物品放入背包，使背包内物品的价值最大。

因为每种物品只有一件，它只有放和不放入背包两种情况，对应 0 和 1 两种决策。0-1 决策，在数据量很小的情况下，考虑暴力搜索，搜索的次数是 2^n，通常这都不是一个很好的办法。如果不考虑动态规划，我们的第一反应可能是贪心策略。很明显，体积小价值高的物品与体积大价值低的物品相比，应该先放入背包。不过，这种策略对体积大价值也大或者体积小价值也小的物品就无能为力了。使用这种方法 O(n^2)先去除明显不合理的数据再暴力搜索。另外，或许你会用 p[i]/w[i]来排序，实际上这种方法是错的，因为总的价值仍然受到背包体积 V 的影响。因为很可能 p[i]/w[i]较高的物品装不满背包，使得背包的总价值比较小。

我们来考虑动态规划。能用动态规划解决的问题一定能把问题分解为一系列互相联系的单个阶段问题。并且子问题之间满足最优化定理：假设为了解决某一优化问题，需要依次作出 n 个决策 D1，D2，…，Dn，如若这个决策序列是最优的，对于任何一个整数 k，1 < k < n，不论前面 k 个决策是怎样的，以后的最优决策只取决于由前面决策所确定的当前状态，即以后的决策 Dk+1，Dk+2，…，Dn 也是最优的。就是说，无论过去的状态和决策如何，对前面的决策所形成的当前状态而言，余下的诸决策必须构成最优策略。最后还要满足无后效性。

无后效性是指：下一时刻的状态只与当前状态有关，而和当前状态之前的状态无关，当前的状态是对以往决策的总结。

以上三种要素构成了动态规划：可分解为相互关联的子问题，最优化定理，无后效性。从另一个方面看就是状态，阶段和决策。

我们这样考虑背包的状态：放入第 n 个物品时，背包的体积为 v 时，它的最大价值为 dp[v][n]。初始状态为 dp[0][0]=0。如果一件物品有放入和不放入背包两种方案，那么要决定第 n+1 件物品是不是向其中放入时：

dp[v][n+1] = max {dp[v][n], dp[v-v[n+1]][n]+w[n+1]}

这就是状态转移方程。从 1 到 n 件物品，每件物品都从 1 到 V 遍历一次背包体积，时间复杂度是 O（nV）。空间复杂度似乎也是 O(nV)，但实际上可以通

过适当的方法将空间负责度压缩为 O(V)。方法就是对每件物品，由 V 到 1 的遍历背包的体积。这样 dp[v-v[n+1]][n]肯定还没有被第 n+1 件物品影响。

至此，0-1 背包问题得到解决。

2. 完全背包

将 0-1 背包的问题进行一点变化，每件物品可以取任意多件，而非仅仅一件。就是说，我们有 n 种物品，每种物品 i 的体积为 w[i],价值为 p[i]。背包体积仍然为 V，求解让背包内物品价值最大的方案。

有了 0-1 背包的基础，我们会想把完全背包转换为 0-1 背包，最朴素的想法就是把 n 种物品变为 m 件物品，只不过其中某些物品的体积和价值完全相同。m=sum(n/w[i])件。当 w[i]比较小时 m 就会变得很大，搜索效率降低。我们换一种思路，将 n 种物品拆分时，拆成 1，2,4,8…2^k 件物品，直到 sum(2^k) * w[i] <= n。最后一件物品的体积是{n-(sum(2^k) * w[i])}*w[i]，当然价值也是相同的倍数。可以使用 2 进制编码的理论对此进行证明：0 到（2^n-1）之间任何一个数都可以用长度为 n 的 01 串表示。这样 m 的值就变为 sum(log n/w[i])，从而使得运算的复杂度大大降低。

其实针对完全背包，还有一种更好的思路：将 0-1 背包中的内层循环改为 1…v。在 v 较小时 dp[n-1]和 dp[n]实际上是相同的。仍然是 01 背包的复杂度。

3. 多重背包

对完全背包的问题进行进一步的修改：每种物品的数量不是无限的，而是有限的。这时候可以采用对于完全背包问题中，对物品进行 2 进制划分的做法。背包九讲中提到存在一种算法可以使得复杂度同样降低到 O(nV)。

4. 混合背包

混合了以上三种背包问题：有些物品是有限的，有些物品是只有一件，有些物品有若干件。解决的关键在于控制好内层循环，根据不同的物品性质采用不同的遍历顺序。

三、分枝定界法

分枝定界法也用于求解一些组合优化问题。其优劣很大程度上与采用的分枝、定界策略有关。尽管大部分组合优化问题可以建立整数规划模型，而整数规划都可以用分枝定界算法来求解。但当面临一个具体的组合优化问题时，有可能找到更好的分枝与定界策略，这是求解的关键所在。仍以背包问题为例说明。

分枝定界法，顾名思义，就是按照定好的界进行分支。这里说的分支意思是“剪枝”。剪的枝是问题解空间树的枝。所谓解空间树，即此问题所有解和中间解形成的树型结构，是有序的。常有排列树和子集树之分，举个例子，n 个物品的 0-1 背包问题的解空间树就是子集树(每个物品都可能为 0 或 1)，而最短路径问题的解空间树是一棵排列树。

由于背包问题为极大化目标问题，对某一个实例，任意一个可行解的目标函数值都可以作为其下界，希望找到的下界尽可能大。另一方面，又希望找到尽可能小得最优值的上界，这样会最大限度地减少分枝的次数，从而缩短算法的运行时间。

背包问题要求在放入物品大小之和不超过背包容量的前提下，使被放入物品的价值尽可能大。显然，对同样大小的物品，应优先放入价值大的。对价值相同的物品，应优先放入体积小的。即按价值密度由大到小的原则选择物品。

因此，不妨假设 $\frac{p[1]}{w[1]} \geq \frac{p[2]}{w[2]} \geq \cdots \geq \frac{p[n]}{w[n]}$。记物品 j 是按上述顺序将物品依次放入背包时第一个不能放入的物品，即有

$$j = \min\{k \mid \sum_{i=1}^{k} w[i] > C\}$$

显然，将前 j-1 个物品放入背包是一个可行解，因此可以得到下界

$$LB_1 = \sum_{i=1}^{j-1} p[i]。$$

上述下界可以改进。由于物品不是按体积从小到大排列，物品 j 不能再放入背包中，j 以后的物品还有可能放入，如果尝试成功，则又得到一个更好的可行解，提高了下界。

由于物品 j 是剩下的价值密度最大的物品，也可以考虑优先放入物品 j，并从前面 j-1 个物品中取出一个体积大小最合适的物品，也有可能提高下界。

总之，

$$LB_2 = \max\{LB_1,$$

$$\max_{j+1 \leq k \leq n}\{LB_1 + p[k] \sum_{i=1}^{j-1} w[i] + w[k] \leq C\}, \max_{1 \leq k \leq j-1}\{LB_1 + p[j] - p[k] \mid \sum_{i=1}^{j-1} w[i] + w[j] - w[k] \leq C\}$$

显然，$LB_1 \leq LB_2$。如果考虑调整两个及以上物品，下界还会有进一步提高，但计算量也随着增大。

再考虑最优值的上界。最直接的思路是整数规划的松弛。背包问题整数规划的松弛线性规划为：

$$\max \sum_{i=1}^{n} p[i] x_i$$

$$s.t. \quad \sum_{i=1}^{n} w[i] x_i \leq C$$

$$0 \leq x_i \leq 1, i = 1,2,\cdots,n$$

上述线性规划的最优值是一个上界。由于这是一个特殊的线性规划问题，其最优解和最优值可以直接写出来：

定理 8.6：背包问题整数规划的松弛线性规划的最优解为 x^*，

$$x_i^* = 1, i = 1,2,\cdots,j-1, x_j^* = \frac{1}{w[j]}(C - \sum_{i=1}^{j-1} w[j]), x_i^* = 0, i = j+1, j+2,\cdots,n$$

其中 $j = \min\{k \mid \sum_{i=1}^{k} w[i] > C\}$

考虑到实例中的数均为整数，可以取

$$UB_1 = \sum_{i=1}^{j-1} p[i] + \left\lfloor \frac{p[j]}{w[j]} (C - \sum_{i=1}^{j-1} w[i] \right\rfloor$$

作为最优解的一个上界。

分枝定界法一般有两种实现形式：1.优先队列法 2.FIFO 队列法。这与分枝定界的思想无太多本质联系，只是前者在一般情况下能更快的求得问题解。分枝定界法要对问题的解空间树进行“剪枝”操作以减少对解空间树的搜索。那么问题是，如何“剪枝”？这就要回答如何定界的问题。在分枝定界法中，“界”的作用就是用来阻止对不可行分枝的搜索的。当解空间树很深时（叶子节点为解），如果能在前面几层就预先的知道了“此路不通”或者“此路不是最优”而停止此路的继续，这样能大幅度的提高算法效率。如何定界要放入具体问题中考虑，一般可以以“理论最大最小”这个概念来求界。以 0-1 背包问题为例，设所有物品预先已经按照单位价值量递减排列。在解空间树的第 i 层（此时正在考虑第 i 个物品是否应该被放入的时刻），设左子树为放入 i 物品，右子树为不放 i 物品。那么在确定左子树的上界的时候有：界=当前价值+i 的价值+MaxValue(背包剩余重量-i 物品重量);其中的 MaxValue 为放 i 后剩余背包容量能获得的最大价值，应该注意的是此最大价值为理论意义上的最大价值，比如在继续放入 p 个后(按单位价值量递减),放不下第 p+1 个，此时应该按(Value[p+1]/Weight[p+1])*(WeightLeft)来计 p+1 物品的价值,(实际中不可能放入零点几个某物品)；右子树的情形类似。

8.8.2 近似算法

对于 NP—难的组合优化问题，如果实例规模比较大，求得最优解往往需要相当长的时间。因此，如果时间上的限制比对精度上的要求更严格，可以考虑“用精度换时间”，即在多项式时间内得到一个与最优解较为接近的可行解，称为近似解。近似解有优劣之分，要有衡量的标准。

设Π是一个极小化目标函数的优化问题，A是它的一个算法。对Π的任何一个实例I，算法A能够在多项式时间内给出实例I的可行解，记为$C^A(I)$是相应的目标函数值，$C^*(I)$为实例为I的最优值。显然$\frac{C^A(I)}{C^*(I)} \geq 1$，它表示近似解和最优解之间的近似程度，越接近 1 越好。由于$\frac{C^A(I)}{C^*(I)}$既与算法有关，又与实例I有关，为了得到关于算法A性能的总体评价，引入下面最坏情况界的概念。

定义：称

$$r_A = \inf\{r \geq 1 | C^A(I) \leq rC^*(I), \forall I\}$$

为算法A的最坏情况界(worst-case ratio)

若算法A的最坏情况界为r_A，则对任意实例I，均有$\frac{C^A(I)}{C^*(I)} \leq r_A$，或者说$C^A(I) \leq r_A C^*(I)$，从而对算法的性能有了保障，我们把这样的算法称为近似所近

似算法(approximation algorithm)。

对于极大化目标的优化问题，定义

$$r_A = \inf\{r \geq 1 \mid C^*(I) \leq rC^A(I), \forall I\}$$

近似算法的设计和最坏情况界的证明是组合优化研究的难点。

在讨论背包问题的分枝定界法时，给出最优值的下界 LB_1。由于求得该下界对应的可行解显然只需要多项式时间，这一过程是一个近似算法。

算法 Greedy Knapsack(GK)

1. 将物品按价值密度非增顺序排列，即有

$$\frac{p[1]}{w[1]} \geq \frac{p[2]}{w[2]} \geq \cdots \geq \frac{p[n]}{w[n]}$$

2. 令 $j = \min\{k \mid \sum_{i=1}^{k} w[i] > C\}$，将物品 1，2，…，j-1 放入背包中。

算法遵循的贪婪策略。构造一个最坏下界的实例，估计算法的界。

考虑实例 I_1：N=2,p[1]=2,w[1]=1,p[2]=2M,w[2]=2M,C=2M。

显然，物品已经按价值密度非增的顺序排列，算法首先将物品 1 放入，因而不能放入物品 2，$C^{GK}(I_1) = 2$，而在最优解中，物品 2 被放入背包中，$C^*(I_1) = 2M$，从而 $\frac{C^*(I_1)}{C^{GK}(I_1)} = M$。由于 M 可以为任意大的整数，说明 GK 算法的近似性能非常差。改进算法，使得能够较好地求解实例 I_1。记 $p_{\max} = \max\{p_j : 1 \leq j \leq n\}$。

算法 Combined Knapsack(CK)

1. 运行 GK 算法。
2. 若 $C^{GK} \leq P_{\max}$，将 $P_{\max}$ 的单个物品放入背包中；$C^{GK} > P_{\max}$，按 GK 算法给出的解将物品放入背包。

定理 8.7：CK 算法的最坏情况界为 $r_{CK} = 2$。

证明：先证明 $\frac{C^*(I)}{C^{CK}(I)} \leq 2$，再构造实例 I_0，使得 $\frac{C^*(I_0)}{C^{CK}(I_0)} = 2$，即 $\frac{C^*(I)}{C^{CK}(I)} \geq 2$，从而 $\frac{C^*(I)}{C^{CK}(I)} = 2$。

（1） 先证明 $\frac{C^*(I)}{C^{CK}(I)} \leq 2$。

由物品 j 的定义可知 $C^{GK}(I)=\sum_{i=1}^{j-1}p_j$。因此，

$$C^{CK}(I)\geq \max\{C^{GK}(I),p_{\max}\}=\max\{\sum_{i=1}^{j-1}p_j,p_{\max}\}$$

另一方面，由于 $UB_1=\sum_{i=1}^{j-1}p[i]+\left\lfloor\frac{p[j]}{w[j]}(C-\sum_{i=1}^{j-1}w[i]\right\rfloor$，所以

$$C^*(I)\leq\sum_{i=1}^{j-1}p_i+\frac{p_j}{w_j}(C-\sum_{i=1}^{j-1}w_i)\leq\sum_{i=1}^{j-1}p_i+\frac{p_j}{w_j}(C-\sum_{i=1}^{j-1}w_i)$$

$$=\sum_{i=1}^{j-1}p_i+\frac{p_j}{w_j}C-\frac{p_j}{w_j}(\sum_{i=1}^{j}w_i-w_j)$$

由于 $\sum_{i=1}^{j}w_i>C$，所以

$$C^*(I)\leq\sum_{i=1}^{j-1}p_i+\frac{p_j}{w_j}C-\frac{p_j}{w_j}(\sum_{i=1}^{j}w_i-w_j)\leq\sum_{i=1}^{j-1}p_i+p_j=\sum_{i=1}^{j-1}p_i+p_{\max}$$

所以，$\frac{C^*(I)}{C^{CK}(I)}\leq\frac{\sum_{i=1}^{j-1}p_i+p_{\max}}{\max\{\sum_{i=1}^{j-1}p_j,p_{\max}\}}\leq 2$。

（2）为了证明算法最坏情况的界恰好为 2，构造下面实例 I_0：$n=3,p_1=w_1=1,p_2=p_3=w_2=w_3=M,C=2M$。物品已经按价值密度非增顺序排列。GK 算法将物品 1,2 放入背包，$C^{CK}(I_0)=M+1$。而最优解为将物品 2，3 放入背包，$C^*(I_0)=2M$，所以 $\frac{C^*(I_0)}{C^{CK}(I_0)}=\frac{2M}{M+1}\to 2(M\to\infty)$。

注：

（1） 算法 A 的最坏情况界的证明需要对 $\frac{C^*(I)}{C^A(I)}$ 的值作出估计，但是 $C^*(I)$ 的值往往是未知的，因此需要给出 $C^*(I)$ 的上界 $C^*_{UB}(I)$（对极小化问题是下界），通过证明 $\frac{C^*_{UB}(I)}{C^A(I)}\leq r_A$，得到 $\frac{C^*(I)}{C^A(I)}\leq r_A$。$C^*_{UB}(I)$ 和 $C^*(I)$ 应该尽量接近，否则肯出现证得的最坏情况界大于实际的最坏情况

界的情况。

（2） 近似算法的设计和最坏情况界的证明应该抓住主要矛盾。如 GK 算法，第一个未放入背包的物品 j 时一个关键物品。如果修改算法，将关键物品 j 之后的物品 $j+1, j+2, \cdots, n$ 能放入背包的就放入背包，也不会使得最坏情况界得到改进。因为在算法表现最坏的实例 I_1 中，根本不存在这样的物品。当然修改后的算法的平均性能肯定不比 GK 差，但在证明最坏情况界时仍然可以忽略这些物品，而不会改进最坏情况界。

（3） 上述定理的证明分成了两个部分，首先证明 $\frac{C^*(I)}{C^{CK}(I)} \le 2$ 对任何一个实例 I 都成立，说明最坏情况界不会超过 2.随后构造实例 I_0（或者一族实例 $\{I_k\}$）使得 $\frac{C^*(I_0)}{C^{CK}(I_0)} = 2$（或者 $\frac{C^*(I_k)}{C^{CK}(I_k)} \to 2 (k \to \infty)$，说明 $\frac{C^*(I)}{C^{CK}(I)} \ge 2$。