编号 041730117

南京航空航天大学

本科毕业设计（论文）

题　目　　混合整数规划的并行算法研究

学生姓名	周长宇
学　　号	041730117
学　　院	理学院
专　　业	理学实验班
班　　级	0817001
指导教师	王丽平　教授

二〇二一年六月

摘　要

近些年来, 多核处理器的飞速发展为并行算法的实现提供了强有力的硬件支撑, 针对整数规划的并行算法研究也迎来了新的高潮. 然而, 国内对于这方面的研究尚存在大量空白. 本文将从 0-1 背包问题入手, 研究混合整数规划的并行算法. 在已有的分支定界算法的基础上, 采用最佳节点优先的节点选择策略, 同时对多个节点进行处理, 即进行节点层面上的并行, 并编程实现.

选择 C/C++ 语言作为主要编程语言、Pthreads 作为多线程编程接口, 进行多线程并行程序的编写. 经实际测试, 并行算法大大缩短了程序运行时间: 设置最大线程数目为 8 线程时, 串行算法的执行时间约是并行算法的 5 倍, 达到了预期效果.

关键词： 0-1 背包问题，分支定界，并行算法，整数规划，运筹学

ABSTRACT

Multi-core processors have evolved tremendously in the past several years, providing parallel algorithms with strong hardware support. The study of parallel mixed integer linear programming (MILP) solving algorithms blooms as well. However, there has been little domestic research on this topic. In this article, we investigate parallel MILP solving algorithms with 0-1 knapsack problem as a starting point. Based on the existing branch-and-bound algorithm, the *best node first search* strategy is adopted, and multiple nodes are processed at the same time, that is, parallelism at the node level, and programming is implemented.

The C/C++ language is chosen to be the main programming language, and Phthreads is selected as the interface for multi-threaded programming. According to experimental observations, parallel algorithm shortens the running time significantly: the sequential algorithm takes about five times longer to run than the parallel algorithm when the maximum number of threads is set to 8, which gives a good indication that the expectation has been met.

KEY WORDS: 0-1 knapsack problem, branch-and-bound, parallel algorithm, integer programming, operations research

目　录

第一章　绪论

整数规划是指限制问题中全部或部分决策变量为整数的最优化问题. 作为一类特殊的约束最优化问题, 其具有全局建模通用性强和全局最优解可验证的优点, 因此在航空运输业[1,2]、采矿业[3]、国防[4]、供应链[5,6]、制造业[7,8]、医疗卫生[9,10]、生物学[11] 以及网络工程[12] 等行业有着广泛的应用.

混合整数规划是指限制部分变量为整数的整数规划. 通常来讲, 我们研究线性目标函数和线性约束下的混合整数规划, 也称混合整数线性规划. 若无特殊说明, 本文中提到的 (混合) 整数规划均是 (混合) 整数线性规划.

1.1　整数规划的发展历程

整数规划的历史可以追溯至上世纪五十年代. 1954 年, 单纯形法发明者 Dantzig 等人对旅行商问题 (Traveling Salesman Problem, 简称 TSP) 进行了研究[13], 构建了整数规划模型, 仅通过手动计算便求解了 49 个城市的旅行商问题, 在当时引起了巨大反响. 从此之后, 整数规划便开始了长足的发展. 整数规划的算法主要有分支定界法 (Branch and Bound) 和割平面法 (Cutting planes) 两种. 1957 年, Markowitz 和 Manne 两人合作, 在处理具体的整数规划问题时提出了分支定界的思想[14]. 在 1958 年 Eastman 的博士学位论文[15] 中也可以窥探到分支定界思想的身影. 针对一般整数规划问题的分支定界方法[16] 于 1960 年由 Land 和 Doig 正式提出. Cook 对分支定界算法的历史做了精彩的汇总阐述[17]; 1958 年, Gomory 提出了割平面方法[18], 自此之后, 整数规划逐渐形成独立分支. 两种方法的结合渐渐成为了求解整数规划问题的主流方法, 也称分支割方法 (Branch and Cut).

上世纪六七十年代, 随着整数规划理论的不断完善和计算机技术的飞速发展, 尝试用计算机来求解整数规划的想法应运而生. 首个基于分支定界方法求解 MIP 的商业代码 LP/90/94 诞生于 1963–1964 年. 经过半个多世纪的发展, 问题的求解效果已经发生了翻天覆地的变化. Bixby 于 2012 年对线性规划和整数规划的计算机求解的研究历史[19] 进行了通俗易懂的介绍. 目前主流的整数规划求解器包括 Gurobi[20], CLPEX[21], SCIP[22] 等, 采用的框架依然是分支割框架.

线性整数规划一直是整数规划研究的核心内容. 在 20 世纪 80 年代, Nemhauser, Wolsey 和 Schrijver 将线性整数规划的主要理论全面而深入地总结在整数规划领域的三本经典专著[23–25] 中, 这三部专著对整数规划的普及和发展起到了极大的推动作用.

Jünger 等人对 1958 至 2008 这 50 年间整数规划领域的研究进展[26] 做了全面而详细的

介绍. Acterberg 等人对 2001 至 2013 这 12 年间 CPLEX 在整数规划求解各个方面所取得的主要突破做了全方位的总结[27].

国内方面, 2010 年, 由孙小玲、李端两位学者编写中文版本的整数规划教材[28] 的出版对国内整数规划的研究、教学与应用起了积极的作用. 在他们 2014 年发表的文章[29] 中, 讨论了整数规划领域若干新进展. 同时, 他们还与郑小金合作, 提出了若干混合整数二次规划问题的新求解技术[30,31]. 此外, 中科院数学与系统科学研究院 CMIP 团队也在整数规划领域取得了一些新进展[32,33].

1.2　并行算法的历史发展

想象时光倒流至 2000 年, 这时你要去电脑店购置人生中的第一台电脑, 你抱回家的大概率会是一台装有 Pentium 4 或者性能差不多的处理器的电脑. 在当时, 你不会刻意地讲你的电脑是“单核”的, 因为这是默认的. 只有那些专门的并行式的大型计算机才有多核支持, 而且需要配备专门定制的主板.

2001 年, IBM 发布了世界上首个多核处理器 Power 4[34]. 2005 年, AMD 发布了首个针对 x86 架构下工作站和服务器的双核处理器 Athlon 64 X2. 同年, Intel 发布了针对 x86 架构下 PC 机的双核处理器 Pentium D. 而如今, 单核处理器已经从通用计算市场消失得无影无踪.

早在多核处理器出现之前, 就有学者对混合整数规划的并行算法开展了研究. 在 20 世纪 90 年代前后, Boehning, Butler 和 Gillett[35], Eckstein[36] 以及 Linderoth[37] 等人在这方面进行了最初的尝试. 1996 年, CPLEX 4.0.3 版本中引入了混合整数规划的并行求解机制[27].

如今, 大多数整数规划求解器, 如开源求解器 SYMPHONY[38], CBC[39], BCP[40], BLIS[41], DisCO[42] 及商业求解器 Gurobi[20] 等均具备了并行的实现机制.

除了上述具有高并行机制集成度的求解器外, 还有一种是可以与串行求解器协同工作的并行框架, 包括 CHiPPS[41,43–46]、UG[47,48] 以及 PEBBL[49] 等.

1.3　基本定义

混合整数 (线性) 规划 (Mixed Integer (Linear) Programming, 简称 MI(L)P) 问题的一般形式如下:

$$\max\{c^\top x : x \in \mathcal{F}\} \tag{1.1}$$

集合

$$\mathcal{F} = \{x \in \mathbb{R}^n_+ : Ax \le b, x_j \in \mathbb{Z}, \forall\, j \in I\} \tag{1.2}$$

被称为可行域, 其中的元素被称为问题 (1.1) 的可行解, 其中 $c \in \mathbb{Q}^n, A \in \mathbb{Q}^{m\times n}, b \in \mathbb{Q}^m, I \subseteq \{1,\ldots,n\}$.

若 $x^* \in \mathcal{F}$ 且满足 $c^\top x^* \geq c^\top x, \forall x \in \mathcal{F}$, 则称 x^* 为问题 (1.1) 的最优解.

考虑问题 (1.1) 的特殊情形:

1. 若 $I = \varnothing$, 原问题变为线性规划问题 (Linear Programming);
2. 若 $I = \{1,\ldots,n\}$, 原问题变为整数规划问题 (Integer Programming);
3. 若 $I = \{1,\ldots,n\}$ 且 $x_j \in \{0,1\}, \forall j \in I$, 原问题变为 0-1 规划问题 (0-1 or Binary Programming).

1.4 本文主要工作

本文将从特殊的整数规划问题——0-1 背包问题入手, 其数学形式如下:

$$\begin{aligned} &\max p^\top x \\ s.t.&\begin{cases} w^\top x \leq c \\ x \in \{0,1\}^n \end{cases} \end{aligned}$$

其中 $p = (p_1,\ldots,p_n)^\top, w = (w_1,\ldots,w_n)^\top, c, p_j, w_j \in \mathbb{N}^+\backslash\{0\}, j = 1,\ldots,n$, 研究混合整数规划的并行算法, 在已有的分支定界算法的基础上考虑节点层面的并行化, 并编程实现. 选择 C/C++ 语言作为主要编程语言、Pthreads 作为多线程编程接口, 进行多线程程序的编写. 之后, 选取数据集, 分别用串行程序和并行程序进行测试. 最后, 对实验结果进行分析.

1.5 本文组织结构

本文的组织结构如下:

- 第一章　介绍整数规划和并行算法的发展历史以及混合整数规划问题的基本定义;
- 第二章　介绍整数规划问题最优性条件以及详细的分支定界算法;
- 第三章　介绍针对 0-1 背包问题的分支定界算法;
- 第四章　介绍并行分支定界算法的想法来源及并行算法的难点和属性, 之后在第三章分支定界算法的基础上实现节点层面的并行, 并进行实际测试;
- 第五章　总结本文取得的成果, 并对今后的工作作出展望.

第二章　最优性条件与分支定界算法

2.1　最优性条件

对于一般的最优化问题, 寻找最优解的基本思路是:

1. 建立最优性条件;

2. 利用迭代算法寻找满足最优性条件的可行解.

很多经典的连续规划问题, 如线性规划问题、二次规划问题, 都可以利用这种思路成功求解. 而大部分的非线性规划问题, 一般是寻找满足 KKT 条件的可行解.

但是, 对于一般的线性整数规划问题:

$$Z = \max\{c^\top x : x \in \mathcal{F}\}$$

$$\mathcal{F} = \{x \in \mathbb{R}^n_+ : Ax \le b, x_j \in \mathbb{Z}, \forall\, j \in I\}$$

很难像 KKT 条件一样利用目标函数的梯度来刻画最优性条件. 那么, 该如何构建整数规划问题的最优性条件呢?

一个比较自然的想法是找出最优值 Z 的一个上界 $\overline{Z} \ge Z$ 和一个下界 $\underline{Z} \le Z$, 若二者满足 $\overline{Z} = \underline{Z}$, 则已求得最优值 $Z = \overline{Z} = \underline{Z}$. 在具体的算法中, 需要构建一个递减的上界序列 $\{\overline{Z}_n\}$:

$$\overline{Z}_1 \ge \overline{Z}_2 \ge \cdots \ge \overline{Z}_s \ge Z$$

和一个递增的下界序列 $\{\underline{Z}_n\}$:

$$\underline{Z}_1 \le \underline{Z}_2 \le \cdots \le \underline{Z}_t \le Z$$

当两者足够接近, 即

$$\overline{Z}_s - \underline{Z}_t \le \epsilon$$

时, 终止算法, 其中 ϵ 是给定的非负实数.

所以, 接下来的问题就是如何获取上界和下界?

2.1.1　下界

下界 (lower bounds), 通常也被称为原始界 (primal bounds). 显然, 对于最大化问题来说, 每个可行解 $x \in \mathcal{F}$ 都能提供一个下界 $\underline{Z} = c^\top x$. 这是我们所知道的获取下界的唯一方式. 对某些特殊的问题, 可行解的获取相对容易, 如任意两个城市均相邻的旅行商问题 (TSP), 任意一个城市序列的排列都对应一个可行解. 而对于其他类型的整数规划问题, 可行解的获取就没那么简单, 可以针对问题设计启发式方法来寻找可行解.

2.1.2 上界

上界 (upper bounds), 通常也被称为对偶界 (dual bounds). 上界的获取主要有两种方式: 松弛问题 (relaxation) 和对偶问题 (dual problem).

2.1.3 松弛问题

松弛问题的获取主要有两种方式:

1. 放松对变量的约束, 扩大可行域;
2. 对于最大 (小) 化问题, 将目标函数换为可行域上取值处处不小 (大) 于原目标函数的函数.

具体定义如下:

定义 2.1： 问题 (RP) $Z^R = \max\{f(x) : x \in T \subseteq \mathbb{R}^n\}$ 被称为是问题 (IP) $Z = \max\{c(x) : x \in X \subseteq \mathbb{R}^n\}$ 的*松弛问题*, 若其满足:

1. $X \subseteq T$;
2. $f(x) \geq c(x), \forall x \in X$.

命题 2.1： 若 RP 是 IP 的一个松弛问题, 则 $z^R \geq z$.

证明: 问题 IP 的最优解是问题 RP 的一个可行解, 故结论成立. □

松弛问题不仅提供了上界, 同时也给出了最优解的一种判定方式.

命题 2.2： 如下:

1. 若松弛问题 RP 没有可行解, 则原问题 IP 也没有可行解.
2. 假设 x^* 是松弛问题的最优解, 若 $x^* \in X$ 并且 $f(x^*) = c(x^*)$, 则 x^* 也是原问题的最优解.

证明: 如下:

1. 由松弛问题的定义, $X \subseteq T = \varnothing$, 所以 $X = \varnothing$, 原问题不可行;
2. $x^* \in X$ 是原问题的一个可行解. 又因为 $c(x^*) = f(x^*) \geq f(x) \geq c(x), \forall x \in X$, 所以 x^* 是原问题的最优解.

□

一个最常用也最容易想到的松弛问题是*线性规划松弛 (Linear Programming Relaxation)*. 顾名思义, 线性规划松弛即是将原问题中的整数约束去除后所得到的线性规划问题, 定义如下:

定义 2.2：线性规划问题

$$\begin{aligned} Z^{LP} &= \max\{c^\top x : x \in \mathcal{P}\} \\ \mathcal{P} &= \{x \in \mathbb{R}^n_+ : Ax \le b\} \end{aligned} \tag{2.1}$$

是问题 (1.1)

$$\begin{aligned} Z &= \max\{c^\top x : x \in \mathcal{F}\} \\ \mathcal{F} &= \{x \in \mathbb{R}^n_+ : Ax \le b, x_j \in \mathbb{Z}, \forall\, j \in I\} \end{aligned}$$

的线性规划松弛问题.

另一种比较常用的松弛是拉格朗日松弛 *(Lagrangian Relaxation)*. 问题 (1.1) 的拉格朗日松弛是:

$$Z(u) = \max\{c^\top x + u^\top (b - Ax) : x \in \mathcal{F}\}, \text{ with } u \in \mathbb{R}^n_+. \tag{2.2}$$

命题 2.3：$Z(u) \ge Z, \forall\, u \in \mathbb{R}^n_+$.

证明: 设 x^* 是问题 (1.1) 的最优解. 由于 x^* 是可行解, 所以 $Ax^* \le b$, 又因为 $u \in \mathbb{R}^n_+$, 所以 $Z = c^\top x^* \le c^\top x^* + u^\top (b - Ax^*) \le Z(u)$. 最后一个“$\le$”是因为 x^* 是拉格朗日松弛的一个可行解. □

2.1.4　对偶问题

定义 2.3：给定两个问题:

$$(LP) \qquad Z = \max\{c(x) : x \in X\}$$

$$(D) \qquad W = \min\{\omega(u) : u \in U\}$$

若 $c(x) \le \omega(u), \forall\, x \in X, u \in U$, 则称两个问题构成了一个 *(弱)* 对偶对 *((weak-)dual pair)*, 两者互称为对方的对偶问题. 若 $Z = W$, 则称二者构成了一个强对偶对 *(strong-dual pair)*.

相比于松弛问题而言, 对偶问题的优势在于任何一个可行解对应的目标函数值都是原问题 (最大化问题) 最优值的一个上界. 那我们不禁要问, 对于一般的线性整数规划问题, 是否存在对应的对偶问题?

答案是肯定的, 最小化其拉格朗日松弛所形成的最优化问题:

$$\begin{aligned} \min Z(u) = \max\{&c^\top x + u^\top (b - Ax) : x \in \mathcal{F}\} \\ &u \in \mathbb{R}^n_+ \end{aligned} \tag{2.3}$$

便是原问题 (1.1)

$$\begin{aligned} Z &= \max\{c^\top x : x \in \mathcal{F}\} \\ \mathcal{F} &= \{x \in \mathbb{R}^n_+ : Ax \le b, x_j \in \mathbb{Z}, \forall\, j \in I\} \end{aligned}$$

的一个对偶问题.

另外, 不难验证:

命题 2.4： 问题 (1.1) 的线性规划松弛的拉格朗日对偶问题: $W^{LP} = \min\{b^\top u : A^\top u \geq c, u \in \mathbb{R}^n_+\}$ 也是原问题的一个对偶问题.

与松弛问题类似, 对偶问题也能帮助证明最优性.

命题 2.5： 假设 IP 与 D 是一对弱对偶对,

1. 若 D 无界, 则 IP 不可行.

2. 若 $x^* \in X, u^* \in U$, 且满足 $c(x^*) = \omega(u^*)$, 则 x^* 是 IP 的最优解, u^* 是 D 的最优解.

2.2 分支定界

Wolsey 在他 2020 年再版的书[50] 中, 对分支定界算法进行了详细的描述. 下面对各部分进行分别介绍.

2.2.1 分而治之

仍考虑相同的问题:

$$Z = \max\{c^\top x : x \in \mathcal{F}\}$$

能否将该问题分解为一系列子问题, 逐个求解子问题, 然后将得到的结果整合, 得到原问题的可行解?

答案是肯定的, 一种实现方式是对可行域 $\mathcal{F}$ 进行划分, 得到子问题.

命题 2.6： 设 $\mathcal{F} = \mathcal{F}_1 \cup \cdots \cup \mathcal{F}_K$ 是 $\mathcal{F}$ 的一个划分, 令 $Z^k = \max\{c^\top x : x \in \mathcal{F}_k\}$, $k = 1, \ldots, K$, 则 $Z = \max_k Z^k$.

可以用枚举树来表示划分过程, 一般来说, 树是从上往下画的, 根节点在最上面. 举个例子, 若 $\mathcal{F} = \{0,1\}^2$, 可以先将 $\mathcal{F}$ 划分为 $\mathcal{F}_0 = \{x \in \mathcal{F} : x_1 = 0\}$ 和 $\mathcal{F}_1 = \{x \in \mathcal{F} : x_1 = 1\}$, 然后划分 $\mathcal{F}_0, \mathcal{F}_{00} = \{x \in \mathcal{F}_0 : x_2 = 0\}, \mathcal{F}_{01} = \{x \in \mathcal{F}_0 : x_2 = 1\}$, 以此类推, 得到如图 2.1 所示的枚举树.

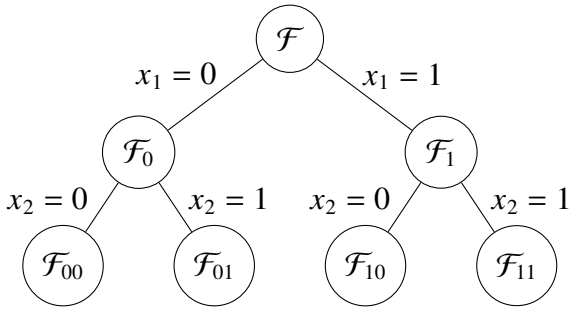

图 2.1 0-1 枚举树

2.2.2 隐式枚举

分而治之, 或者也可以称为递归, 这种方式固然可行, 但会对问题进行完全枚举, 最终的结果是, 枚举树中的每个叶子节点都会对应可行域 $\mathcal{F}$ 中的一个可行解或者空集. 一旦问题的规模扩大, 完全枚举所花费的时间将是难以接受的. 所以, 我们不禁要问: 有必要进行完全枚举吗? 是不是有些枚举是不需要的? 下面我们将回答这个问题.

延续上一节的思路, 将原问题划分为若干个子问题 $Z^k = \max\{c^\top x : x \in \mathcal{F}_k\}$, $k = 1, \ldots, K$. 由于每个子问题都是一个新的整数规划问题, 我们可以采用相同的方式, 通过求解对应的线性松弛问题来获取每个子问题的上界 $\overline{Z}^k$. 由于 $\mathcal{F}_k \subseteq \mathcal{F}$, 所以每个子问题的可行解也是原问题的可行解, 由可行解获得的子问题的下界也是原问题的下界.

借助子问题的上下界信息, 我们可以进行更加“聪明”的穷举. 首先, 考虑特殊情况, 若 $\mathcal{F}_k = \varnothing$, 显然不用对其进行再次划分; 其次, 若 $\overline{Z}^k \leq \underline{Z}$, 则表示 $\mathcal{F}_k$ 中不会含有比 $\underline{Z}$ 对应的可行解还要好的可行解, 所以也不用对 $\mathcal{F}_k$ 进行划分; 还有一种情况, 即问题 $Z^k = \max\{c^\top x : x \in \mathcal{F}_k\}$ 的最优解已被求出, 这时也不需要对 $\mathcal{F}_k$ 进行划分. 在枚举树中, “不再划分”对应于叶子节点, 即没有子节点, 我们通常将这一操作形象地称为“剪枝”, 以上三种情况分别简称为:

1. 根据不可行性剪枝 (prune by infeasibility);
2. 根据界信息剪枝 (prune by bound);
3. 根据最优性剪枝 (prune by optimality).

2.2.3 基于线性松弛的分支定界算法

对于问题 (1.1)

$$Z = \max\{c^\top x : x \in \mathcal{F}\}$$

$$\mathcal{F} = \{x \in \mathbb{R}^n_+ : Ax \leq b, x_j \in \mathbb{Z}, \forall\, j \in I\}$$

求解其线性松弛问题 (2.1):

$$\overline{Z} = \max\{c^\top x : x \in \mathcal{P}\}$$

$$\mathcal{P} = \{x \in \mathbb{R}^n_+ : Ax \leq b\}$$

将其记作 LP($\mathcal{F}$), 假设求得的最优解是 $\bar{x}$, 若 $\bar{x}$ 是整数规划问题 (1.1) 的可行解, 则它也是原问题的最优解, 算法结束; 否则, 由于当下还未找到可行解, 我们先令下界 $\underline{Z} = -\infty$. 选取一个变量 x_j, 满足 $\bar{x}_j \notin \mathbb{Z}$, $j \in I$, 对可行域进行如下划分:

$$\mathcal{F}_1 = \mathcal{F} \cap \{x : x_j \leq \lfloor \bar{x}_j \rfloor\},\ \mathcal{F}_2 = \mathcal{F} \cap \{x : x_j \geq \lceil \bar{x}_j \rceil\}$$

显然, $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ 并且 $\mathcal{F}_1 \cap \mathcal{F}_2 = \varnothing$, 得到子问题 $Z^k = \max\{c^\top x : x \in \mathcal{F}_k\}$, $k = 1, 2$. 选取

这种划分方式的好处有两个:

1. 线性松弛问题最优解 $\bar{x}$ 在两个新问题对应的线性松弛问题中均不可行. 记两个新问题对应的线性松弛问题分别是 LP($\mathcal{F}_1$), LP($\mathcal{F}_2$), 最优值分别是 $\overline{Z}_1, \overline{Z}_2$, 若原线性松弛问题 (2.1) 只有一个最优解, 则 $\max\{\overline{Z}_1, \overline{Z}_2\} < \overline{Z}$, 问题的上界将会严格减小, 有利于根据界信息剪枝.
2. 对于子问题的线性松弛问题 LP($\mathcal{F}_i$), $i = 1, 2$, 不需要利用单纯形法从头开始求解. 以 LP($\mathcal{F}_1$) 为例, 与原线性松弛问题 LP($\mathcal{F}$) 相比只是多了一个界约束 $x_j \le \lfloor \bar{x}_j \rfloor$, 将其写成等式的形式 $x_j + s = \lfloor \bar{x}_j \rfloor, s \ge 0$. 假设 LP($\mathcal{F}_1$) 可行域非空, 若 x_j 不是基变量, 将其用基变量表示并代入该等式约束中. 由于原来的基是最优基, 所以其是对偶可行的, 即是原线性松弛问题 LP($\mathcal{F}$) 的对偶问题的可行解. 可以验证, 原来的基变量与新变量 s 组成的基也是新线性松弛问题 LP($\mathcal{F}_1$) 的对偶问题的可行解. 因此, 可以在原来的线性规划最优单纯形表基础上使用对偶单纯形法进行新线性规划问题的快速求解.

假设算法进行至某一步时, 子问题有 $Z^k = \max\{c^\top x : x \in \mathcal{F}_k\}$, $k = 1, \ldots, K$. 选取一个子问题 $Z^k = \max\{c^\top x : x \in \mathcal{F}_k\}$, 记作 Q^k. 首先, 求解其线性松弛问题 LP(Q^k), 若可行域为空集, 则直接*根据不可行性剪枝*; 否则, 设求得的最优解为 $\bar{x}^k$, 对应的目标函数值为 $\overline{Z}^k$. 若 $\overline{Z}^k \le \underline{Z}$, 则*根据界信息剪枝*; 否则, 若 $\bar{x}^k$ 满足整数约束, 是整数规划问题的可行解, 更新下界和对应的可行解 $\underline{x} := \bar{x}^k$, $\underline{Z} := \overline{Z}^k$, *根据最优性剪枝*; 若不满足整数约束, 则采用相同的方式分支, 产生两个新的子问题 Q_1^k, Q_2^k.

不断重复此过程, 直至没有子问题, 此时最好的可行解 $\underline{x}$ 和下界 $\underline{Z}$ 即是原问题的最优解 x^* 和最优值 Z. 具体算法流程如 Algorithm 1 所示.

可以看到, 在实际算法中, 只存储了仍需进一步探索的子问题 (节点), 或称为*活跃子问题* (节点), 将其放置在一个节点列表 $\mathcal{L}$ 中, 并没有对整棵分支定界树进行存储, 事实上也没有必要这样做.

2.2.4 分支策略

首先讨论分支策略. 在上述算法中, 只提到在线性松弛问题最优解中选取一个值为分数的整数变量进行分支. 但是如果有多个值为分数的整数变量, 那该如何选择? 这就是分支策略所要回答的问题. 选择分支变量的主要依据是使得分支后产生的两个子问题所对应的上界与当前问题的上界相差尽可能地大, 即子问题的上界会有一个明显的下降量, 这样就有利于尽可能早地进行剪枝.

Algorithm 1 LP-Based Branch and Bound

Input: 最大化问题 (1.1): $Z = \max\{c^\top x : x \in \mathcal{F}\}$, 记作 Q^0.
Output: 最优解 x^* 及对应的最优值 Z.
初始化 $\mathcal{L} := \{Q^0\}, \underline{Z} := -\infty$;
调用启发式方法寻找可行解;
if 找到一个可行解 x^H, 对应目标函数值为 Z^H **then**
    令 $\underline{x} := x^H$, $\underline{Z} := Z^H$.
end if
while $\mathcal{L} \neq \emptyset$ **do**
    选取 $Q^k \in \mathcal{L}$, 令 $\mathcal{L} := \mathcal{L} \backslash \{Q^k\}$, 求解 Q^k 的线性松弛 LP(Q^k) ;
    if LP(Q^k) 不可行 **then**
        根据不可行性剪枝.
    else
        令 $\overline{x}_i^k$ 是 LP(Q_i^k) 的一个最优解, $\overline{Z}_i^k$ 为对应目标函数值;
        if $\overline{Z}^k \leq \underline{Z}$ **then**
            根据界信息剪枝.
        else
            if $\overline{x}^k$ 满足整数约束 **then**
                更新可行解及对应的目标函数值 $\underline{x} := \overline{x}^k$, $\underline{Z} := \overline{Z}^k$;
                根据最优性剪枝.
            else
                将 Q^k 划分为两个子问题 Q_1^k, Q_2^k, 令 $\mathcal{L} := \mathcal{L} \cup \{Q_1^k, Q_2^k\}$.
            end if
        end if
    end if
end while
停止算法, 返回 $x^* := \underline{x}, Z := \underline{Z}$.

下面介绍两种分支策略, 不妨设线性松弛问题最优解中选取一个值为分数的整数变量下标组成的集合为 C.

1. **最不可行分支 (Most Infeasible Branching)**

   这种分支策略的思想是选取小数部分最接近 0.5 的整数变量, 即选取下标

   $$j^* = \arg\max_{j \in C} \min\{\overline{x}_j - \lfloor \overline{x}_j \rfloor, \lceil \overline{x}_j \rceil - \overline{x}_j\}$$

   对应的变量作为分支变量. 其背后的原因是, 如果变量的值足够接近 0.5, 那么向上分支和向下分支所产生问题的线性规划松弛界应该不会相差太大, 生成的分支定界树应该会比较均衡, 不会出现一边节点数明显多于另一边的情形.

   但令人遗憾的是, 这种策略的效果并没有想象中的那么好. 在实际测试中, 其总体效果要弱于随机选取策略.

2. **强分支 (Strong Branching)**

   强分支的思想最早是在求解旅行商问题 (TSP) 时提出的, 见 Applegate 等人 1995 年

的报告[51]. 不久之后, 这种方法便就成了整数规划求解器的标准组成部分. 其主要思想是: 在困难的问题上, 值得做更多的工作来选择一个更好的分支变量. 具体做法是:

分别对集合 C 中的每个变量进行向上和向下分支, 产生新的线性规划松弛, 然后求得最优解或进行指定次数的对偶单纯形法迭代. 之后, 得到变量 $x_j(j \in C)$ 的向下分支的上界为 Z_j^D, 向上分支的上界为 Z_j^U. 选取上界下降量最大, 即上界最小的变量作为分支变量, 即选择变量

$$j^* = \arg\min_{j \in C} \max\{Z_j^D, Z_j^U\}.$$

在实际计算中, 这种方法能够非常有效地减少分支定界树中的节点数目[52]. 但是为了达到这么好的效果, 所付出的代价是昂贵的, 因为需要对 C 中的每个变量都进行两个线性规划松弛的求解.

2.2.5 节点选择策略

其次, 我们来讨论节点选择策略. 节点选择策略所要回答的问题是如何从剩余节点列表 $\mathcal{L}$ 中选取一个节点进行处理. 与分支策略一样, 不同的节点选择策略将会直接影响到分支定界算法的求解效率[53], 如节点数目和求解时间等等. 同样的, 我们介绍两种节点选择策略.

1. **深度优先策略 (Depth First Search)**.

   在整数规划领域, 这种策略最早是由 Little, Murty, Sweeney 等人在 1963 年求解旅行商问题 (TSP) 时提出[54]. 两年后, Dakin 提出了针对一般混合整数规划问题的深度优先策略[55].

   这种策略选取当前节点的子节点作为下一个要处理的节点, 其背后的原因是: 如果找到的可行解对应的目标函数值较大, 那么它能充当一个很好的下界. 在这种情况下有可能对分支定界树进行大规模的剪枝, 从而减少实际枚举的数量. 因此, 应当尽快地向下搜索. 使用这种策略的另一个原因是: 无需更换现有的线性规划最优单纯形表, 直接在其基础上使用对偶单纯形法迭代即可.

   这种策略的缺点是: 可能会进行大量的冗余计算. 如果换用其他策略, 可能会更早地得到一个好的下界, 这样就有可能对深度优先策略中处理过的节点进行剪枝操作.

2. **最佳节点优先策略 (Best First Search)**

   选择具有最佳 (最大) 上界的节点, 即选择节点 s, 其中 $\overline{Z}_s = \max_t \overline{Z}_t$, 这里的上界可以是线性规划松弛界, 也可以是从父节点继承的上界. 在这种策略下, 我们永远不会

对上界 $\overline{Z}_t$ 小于最优值 Z 的节点进行分支.

命题 2.7：在最佳节点优先策略下, 分支定界树中的节点数目是最小的.

详细证明过程可以参考 Achterberg 的博士学位论文[52] 中的 Proposition 6.1.

尽管有着多种多样的策略, 但不存在一个能够完美处理所有情况的最佳策略, 每个策略都有着自身的优缺点. 通常情况下, 往往会同时使用多种策略或者不同策略混合而成的策略. Achterberg 的博士学位论文[52] 中介绍了更多的分支策略和节点选择策略以及对应的实验结果.

第三章　针对 0-1 背包问题的分支定界算法

3.1　问题介绍

一般来说, 整数规划问题都是 $\mathcal{NP}$- 难的[24]. 所以除非 $\mathcal{P} = \mathcal{NP}$, 否则是不存在针对一般整数规划问题的多项式时间算法的. 所以, 我们从最简单的整数规划问题——0-1 背包问题 (0-1 knapsack problems) 入手, 进行并行分支定界算法的设计实现.

0-1 背包问题可以这样定义: 给定一个物品集合 $N = \{1, \ldots, n\}$, 包含 n 个物品. 每个物品 $j \in N$ 都有两个属性: 价值 p_j 和重量 w_j, 通常情况下取值为正整数, 即 $p_j, w_j \in \mathbb{N}^+ \backslash \{0\}, j \in N$. 目标是设计一种背包方案, 即选取物品集合 N 的一个子集, 在子集中物品总重量不超过 c 的条件下, 使得物品的总价值最大.

定义决策变量 $x = (x_1, \ldots, x_n)^\top, j \in N$, 其中分量 x_j 定义如下:

$$x_j = \begin{cases} 1, \text{ 物品 } j \text{ 被装进包内} \\ 0, \text{ 物品 } j \text{ 不被装进包内} \end{cases}$$

之后可以将问题建模为整数规划问题:

$$\begin{aligned} &\max p^\top x \\ s.t. &\begin{cases} w^\top x \le c \\ x \in \{0, 1\}^n \end{cases} \end{aligned} \tag{3.1}$$

其中 $p = (p_1, \ldots, p_n)^\top, w = (w_1, \ldots, w_n)^\top$.

为什么说 0-1 背包问题是最简单的整数规划问题呢? 首先, 变量均为 0-1 变量, 只有两个取值; 其次, 只有一条约束 $w^\top x \le c$ 且约束系数均为正数. 但即使这样, 整数约束的添加使得最简单的 0-1 背包问题也是 $\mathcal{NP}$- 难的[56].

但是在问题规模 (n 和 c) 固定的情况下, 从动态规划的角度出发, 存在伪多项式时间的求解算法, 对应的时间复杂度为 $O(nc)$, 在 Kellerer, Pferschy 等人编写的背包问题专著中, 给出了时间复杂度的详细证明过程[57].

3.2　0-1 背包问题的分支定界算法

在进行并行之前, 首先考虑 0-1 背包问题的串行分支定界算法, 从以下几个方面展开.

3.2.1　线性松弛

0-1 背包问题的线性规划松弛定义如下:

$$\begin{aligned} &\max p^\top x \\ s.t. &\begin{cases} w^\top x \le c \\ 0 \le x_j \le 1, \ j \in N \end{cases} \end{aligned} \tag{3.2}$$

p, w 的定义与前面相同.

定义物品 j 的价值密度 *(profit per weight ratio)* e_j 如下:

$$e_j := \frac{p_j}{w_j}, \tag{3.3}$$

不妨假设物品已经按价值密度递减的顺序进行了排序, 即:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}. \tag{3.4}$$

可以用贪心算法 *(greedy algorithm)* 对线性规划松弛进行求解, 具体做法是依次将价值密度最高的物品放入背包, 直至出现以下三种情况之一 (假设已经放入了 $s-1$ 个物品):

1. 物品全部装完, 即 $s-1=n$;
2. 装入物品的总重量恰好达到重量要求, 即 $\sum_{j=1}^{s-1} w_j = c$;
3. 尚未装满且有物品剩余, 但是不能完整装下下一个物品, 即 $\sum_{j=1}^{s-1} w_j < c$ 并且 $\sum_{j=1}^{s} w_j > c$.

若是最后一种情况, 则放入下一个物品的一部分 $(c - \sum_{j=1}^{s-1} w_j)/w_s$, 使背包重量达到要求, 并称物品 s 为关键物品.

定理 3.1：若问题 (3.2) 的下标满足式 (3.4), 则其最优解[58] 为 $\overline{x} = (\overline{x}_1, \ldots, \overline{x}_n)$, 其中

$$\overline{x}_j = \begin{cases} 1, & j = 1, \ldots, s-1 \\ (c - \sum_{j=1}^{s-1} w_j)/w_s, & j = s \\ 0, & j = s+1, \ldots, n \end{cases} \tag{3.5}$$

其中 s 满足:

$$\sum_{j=1}^{s-1} w_j \leq c \text{ 并且 } \sum_{j=1}^{s} w_j > c. \tag{3.6}$$

若 $\sum_{j=1}^{n} w_j \leq c$, 则令 $s = n+1$.

证明: 不妨假设物品的价值密度各不相同, 即下标满足:

$$\frac{p_1}{w_1} > \frac{p_2}{w_2} > \cdots > \frac{p_n}{w_n}. \tag{3.7}$$

因为若有价值密度相同的物品, 可以将其“合并”为单个物品.

利用反证法. 假设问题 (3.2) 存在与 $\overline{x}$ 不同的最优解, 设其中一个为 $\hat{x}$. 那么 $\hat{x}$ 一定满足 $w^\top \hat{x} = c$, 否则, 可以提升 $\hat{x}$ 的任一分量, 使目标函数值严格增加.

若 $\hat{x}_u \neq \overline{x}_u, u < s$, 则 $\hat{x}_u < \overline{x}_u = 1$. 由于 $w^\top \hat{x} = w^\top \overline{x} = c$, 所以必然存在 $v \geq s$, 使得 $\hat{x}_v > \overline{x}_v \geq 0$.

取

$$d = \min\{w_u(1-\hat{x}_u), w_v\hat{x}_v\},$$

由 $\hat{x}_u < 1, \hat{x}_v > 0$ 得 $d > 0$, 构造 x' 满足:

$$\begin{aligned} x'_u &= \hat{x}_u + \frac{d}{w_u}, \\ x'_v &= \hat{x}_v - \frac{d}{w_v}, \\ x'_j &= \hat{x}_j, j \in N\backslash\{u, v\}. \end{aligned}$$

由 $d > 0$, 则 $x'_u > \hat{x}_u \geq 0, x'_v < \hat{x}_v \leq 1$.

由 d 的取法,

$$\begin{aligned} x'_u &= \hat{x}_u + \frac{d}{w_u} \leq \hat{x}_u + \frac{w_u(1-\hat{x}_u)}{w_u} = 1, \\ x'_v &= \hat{x}_v - \frac{d}{w_v} \geq \hat{x}_v - \frac{w_v\hat{x}_v}{w_v} = 0. \end{aligned}$$

所以 $0 \leq x'_j \leq 1, \forall\, j \in N$.

另一方面,

$$w^\top x' = w^\top \hat{x} + w_u\frac{d}{w_u} - w_v\frac{d}{w_v} = w^\top\hat{x} = c.$$

所以 x' 是线性规划松弛 (3.2) 的可行解.

但是

$$p^\top x' = p^\top\hat{x} + p_u\frac{d}{w_u} - p_v\frac{d}{w_v} = p^\top\hat{x} + d\Big(\frac{p_u}{w_u} - \frac{p_v}{w_v}\Big)$$

由 $u < v$ 和式 (3.7) 得 $p_u/w_u - p_v/w_v > 0$, 所以

$$p^\top x' = p^\top\hat{x} + d\Big(\frac{p_u}{w_u} - \frac{p_v}{w_v}\Big) > p^\top\hat{x}$$

与 $\hat{x}$ 为最优解矛盾.

对于 $\hat{x}_u \neq \bar{x}_u, u \geq s$ 的情况, 可用相同的思路证明, 不再赘述. □

所以 0-1 背包问题的线性规划松弛有显式最优解, 无需利用单纯形法进行求解, 易于编程实现.

3.2.2 分支

从上一部分看出, 0-1 背包问题的线性松弛解 $\bar{x}$ 中至多只有一个分量取值为分数, 若存在, 不妨设其为 $\bar{x}_j$, 直接对该分量进行分支:

$$\begin{aligned} 0 \leq x_j \leq \lfloor\bar{x}_j\rfloor = 0 \Rightarrow x_j = 0 \\ 1 \geq x_j \geq \lceil\bar{x}_j\rceil = 1 \Rightarrow x_j = 1 \end{aligned} \tag{3.8}$$

则变量 $\bar{x}_j$ 直接被固定为 0 或 1.

若 $\bar{x}_j$ 被固定为 1, 则原问题变为

$$\begin{aligned} &\max \sum_{j=1}^{s-1} p_j x_j + p_s + \sum_{j=s+1}^{n} p_j x_j \\ &s.t. \begin{cases} \sum_{j=1}^{s-1} w_j x_j + w_s + \sum_{j=s+1}^{n} w_j x_j \le c \\ x_j \in \{0,1\},\ j \in N \backslash \{s\} \end{cases} \end{aligned} \tag{3.9}$$

令

$$\begin{aligned} y &= (x_1, \ldots, x_{s-1}, x_{s+1}, \ldots, x_n)^\top, \\ p' &= (p_1, \ldots, p_{s-1}, p_{s+1}, \ldots, p_n)^\top, \\ w' &= (w_1, \ldots, w_{s-1}, w_{s+1}, \ldots, w_n)^\top, \\ c' &= c - w_s, \end{aligned}$$

则可以将问题变为:

$$\begin{aligned} &p_s + \max p'^\top y \\ &s.t. \begin{cases} w'^\top y \le c' \\ y \in \{0,1\}^{n-1} \end{cases} \end{aligned} \tag{3.10}$$

即 $n-1$ 维的 0-1 背包问题.

若 $\bar{x}_j$ 被固定为 0, 则可以采用相同的方式将原问题等价变形为 $n-1$ 维的 0-1 背包问题.

所以分支产生的新问题仍是 0-1 背包问题, 可以利用相同的方法求线性松弛解.

若线性松弛解 $\bar{x}$ 中所有分量均为整数, 由命题 2.2 知, $\bar{x}$ 是原问题的最优解.

3.2.3　节点选择

为了控制分支定界树中的节点数量, 在节点选择上我们选用最佳节点优先策略. 为此, 需要对原有的分支定界框架进行一些调整.

由于最佳节点优先策略需要事先知道节点的上界, 而对于新产生的节点, 其上界的获取方式有两种:

1. 直接继承父节点的上界;
2. 根据松弛问题或对偶问题获得.

这里我们选取后一种方式, 节点新产生后, 立即求解其 LP 松弛, 获取最优值作为上界.

通常来讲, 在实现算法时, 我们将节点信息存储在链表 *(list)* 或者数组 *(array)* 中. 但是为了更好地适配最佳节点优先策略, 我们采用堆 *(heap)* 作为存储结构. 原因是若是采用链表或数组作为存储结构, 假设长度为 n, 那么插入一个节点的时间复杂度为 $O(n)$, 若是利用堆来存储, 则时间复杂度会降至 $O(\log n)$[59].

3.2.4 算法框架

将串行分支定界算法框架按照最佳节点优先的节点选择策略进行修改, 修改后的算法框架如 Algorithm 2 所示.

为了让程序能够读懂 0-1 背包问题, 我们需要将数学形式的例子以一定的格式转化为输入文件, 作为算法运行的前提.

3.2.5 正确性测试

正确性验证算例来源: 2005 年 Pisinger 发表的一篇关于背包问题求解的难易程度的文章[60], 算例 (及其最优解) 可以在 Pisinger 的个人网站[1]进行下载.

[1]http://hjemmesider.diku.dk/~pisinger/codes.html

Algorithm 2 Revised LP-Based Branch and Bound (Best First Search): Part 1

Input: 最大化问题 (1.1): $Z = \max\{c^\top x : x \in \mathcal{F}\}$, 记作 Q^0.
Output: 最优解 x^* 及对应的最优值 Z.
初始化 $\mathcal{L} := \varnothing, \underline{Z} := -\infty$
调用启发式方法寻找可行解;
if 找到一个可行解 x^H, 对应目标函数值为 Z^H **then**
    令 $\underline{x} := x^H$, $\underline{Z} := Z^H$.
end if
求解 Q^0 的线性松弛 LP(Q^0) ;
if LP(Q^0) 不可行 **then**
    输出“原问题不可行”.
else
    令 $\overline{x}^0$ 是 LP(Q^0) 的一个最优解, $\overline{Z}^0$ 为对应目标函数值;
    if $\overline{Z}^0 \le \underline{Z}$ **then**
        根据界信息剪枝.
    else
        if $\overline{x}^0$ 满足整数约束 **then**
            更新可行解及对应的目标函数值 $\underline{x} := \overline{x}^0$, $\underline{Z} := \overline{Z}^0$;
            根据最优性剪枝.
        else
            令 $\mathcal{L} := \mathcal{L} \cup \{Q^0\}$.
        end if
    end if
end if
while $\mathcal{L} \neq \varnothing$ **do**
    从列表 $\mathcal{L}$ 中选取上界最大节点 Q^k, 令 $\mathcal{L} := \mathcal{L} \backslash \{Q^k\}$;
    令 $\overline{x}^k$ 是 LP(Q^k) 的一个最优解, $\overline{Z}^k$ 为对应目标函数值;
    if $\overline{Z}^k \le \underline{Z}$ **then**
        根据界信息剪枝.
    else
        将 Q^k 划分为两个子问题 Q_1^k, Q_2^k, 因为 $\overline{x}^k$ 一定不是满足整数约束的解;
        对于 $Q_i^k, i = 1, 2$, 求解 Q_i^k 的线性松弛 LP(Q_i^k) ;
        if LP(Q_i^k) 不可行 **then**
            根据不可行性剪枝.
        else
            令 $\overline{x}_i^k$ 是 LP(Q_i^k) 的一个最优解, $\overline{Z}_i^k$ 为对应目标函数值;
            if $\overline{Z}_i^k \le \underline{Z}$ **then**
                根据界信息剪枝.
            else
                if $\overline{x}_i^k$ 满足整数约束 **then**
                    更新可行解及对应的目标函数值 $\underline{x} := \overline{x}_i^k$, $\underline{Z} := \overline{Z}_i^k$;
                    根据最优性剪枝.
                else
                    令 $\mathcal{L} := \mathcal{L} \cup \{Q_i^k\}$.
                end if
            end if
        end if
    end if

Algorithm 3 Revised LP-Based Branch and Bound (Best First Search): Part 2

48: **end while**
49: 算法停止, 返回 $x^* := \underline{x}, Z := \underline{Z}$.

第四章　0-1 背包问题的并行分支定界求解

4.1　一般整数规划问题的并行算法

章节 2.2.3 介绍了基于线性松弛的分支定界算法 Algorithm 1, 这种算法每次只从节点列表 $\mathcal{L}$ 中选取一个节点进行处理. 我们不禁会想: 既然现在处理器都是多核配置, 能否选取多个节点同时进行处理? 从而更有效地利用计算资源. 这便是并行算法的想法来源, 表示在分支定界树中如图 4.1[1] 所示.

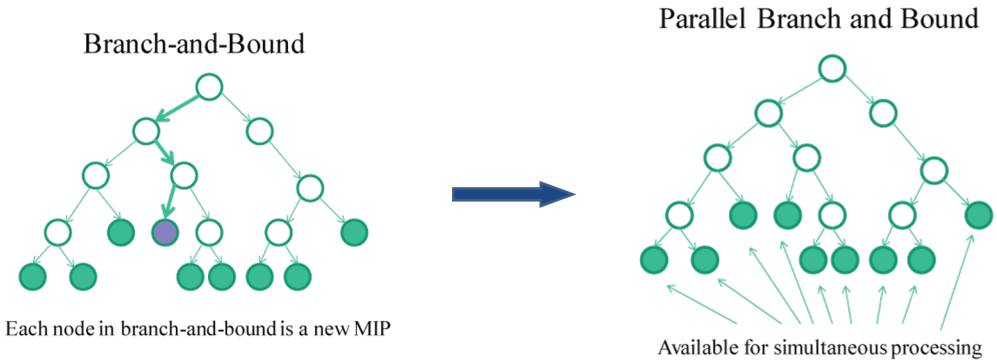

图 4.1　并行算法的最初想法

其实早在多核处理器出现之前, 便有学者对混合整数规划的并行算法开展了研究[35–37].

从分支定界树的角度考虑, 并行处理节点对应于并行的单层树搜索, 而针对树状结构的并行搜索效果应相当不错. 在二十世纪六七十年代之交, 伴随着网络计算 (Network Computing) 浪潮的兴起, 研究人员开始对一般的分支定界方法进行并行实验, Gendron 和 Crainic 两人在文章[61] 中对这方面早期的研究做了记录. 在首个大规模系统 (large-scale system) 问世后不久, 人们便对并行算法进行了编程实验, 却发现难以达到好的效果.

4.1.1　并行算法难点

除并行树搜索本身的困难外, 并行分支定界算法有着自身特殊的难点:

1. 问题的求解效率并不随计算资源的增加呈线性或者近似线性增长, 主要原因在于全局信息的共享和各部分之间消息的传递.
2. 并行算法难以重现串行算法中的节点处理顺序, 而处理顺序对求解效果影响甚大, 直接影响到分支定界树中的节点数目和求解时间.
3. 串行算法可以很容易地处理当前节点的子节点, 而这一点在并行时难以实现, 因为要考虑父节点线性规划最优单纯形表的读取和装载所花费的时间.

[1] 图片来源: https://www.gurobi.com/resource/mip-basics/

4. 大部分对并行算法的研究只是对简单串行算法的并行实现, 难以达到好的并行效果且对不同问题的适应性较弱.

4.1.2 并行算法属性

Ralphs 等人在 2016 年的文章[62] 中给出了混合整数规划并行求解算法的一些基本属性:

1. 资源利用率 (Scalability): 对增加的计算资源 (主要是处理器资源) 进行合理利用的能力. 可以通过并行成本进行评估, 并行成本指的是并行算法必需的但在串行算法中不会涉及的工作, 可以分为以下几类:
   (a) 通信成本: 数据传递所需的时间, 包括将数据存入缓冲区和从缓冲区读取数据所需的时间;
   (b) 闲置时间: 等待任务分配或算法结束的时间;
   (c) 等待时间: 数据从存储位置迁移到使用位置所花费的时间;
   (d) 冗余工作: 比如不必要的节点处理. 产生冗余工作的主要原因是分支定界树局部产生的信息, 如全局下界和对应的可行解, 不能及时地被全局共享.
2. 抽象性与耦合性
   - 抽象性 (Abstract) 可以分为两类: 算法抽象和实现抽象. 算法抽象指的是只列出算法框架, 不限定具体的实现细节; 实现抽象指的是在实际实现中上层并行框架与下层串行算法的关联程度.
   - 耦合性 (Integration) 是指并行算法与其调用的串行算法之间的耦合程度.
3. 并行粒度 (Granularity): 并行算法中原子工作的大小. 根据粒度, 可以将并行策略分为以下四类:
   (a) 树并行 (Tree parallelism), 即同时搜索几棵树, 可使用不同的搜索策略;
   (b) 子树并行 (Subree parallelism), 即同时搜索几棵子树;
   (c) 节点并行 (Node parallelism), 即同时对多个节点进行处理, 这也是最为常见的并行策略;
   (d) 次节点并行 (Subnode parallelism), 即同时进行若干个节点处理过程中的子模块, 如求解对应的线性规划松弛、生成割平面等.
4. 负载均衡 (Load Balancing): 对任务进行调整或分配, 使得多个处理器承担的任务尽量均衡, 可以分为静态均衡和动态均衡两类:
   - 静态均衡 (Static load balancing) 是指仅在算法开始时进行一次任务分配, 之后不调

整;

- 动态均衡 (Dynamic load balancing) 是指在算法运行过程中会随时根据具体情况进行任务调整.

5. 确定性 (Determinism): 指的是对于相同的输入, 经过相同的操作后, 所产生输出结果是相同的. 对并行分支定界算法而言, 确定性的最直接体现是: 对于同一个问题, 不论算法执行多少次, 分支定界树中节点的产生顺序和处理顺序都是相同的. 2007 年, CPLEX 11.0 版本中首次引入了确定性并行机制, 实现思路可参阅相关文献[27].

4.2　并行求解 0-1 背包问题

在讲述具体的并行实现之前, 先来介绍一下进程和线程[63].

4.2.1　进程与线程

进程 (process) 就是程序 (如 Windows 系统中的 `.exe` 文件、Linux 系统中的 `.out` 文件) 的一次执行. 同一程序可以同时执行多次, 分别对应多个进程, 比如可以同时打开多个 MATLAB 应用程序. 进程之间相互独立, 在内存中分别占据不同的空间.

从操作系统的角度来讲, *线程 (thread)* 是可以供调度器 (scheduler) 调度的最小单位. 在大多数情况下, 进程是线程的组成部分, 一个进程里可以包含多个线程, 它们共享同属进程的内存, 包括代码段、数据段等, 如图 4.2 所示.

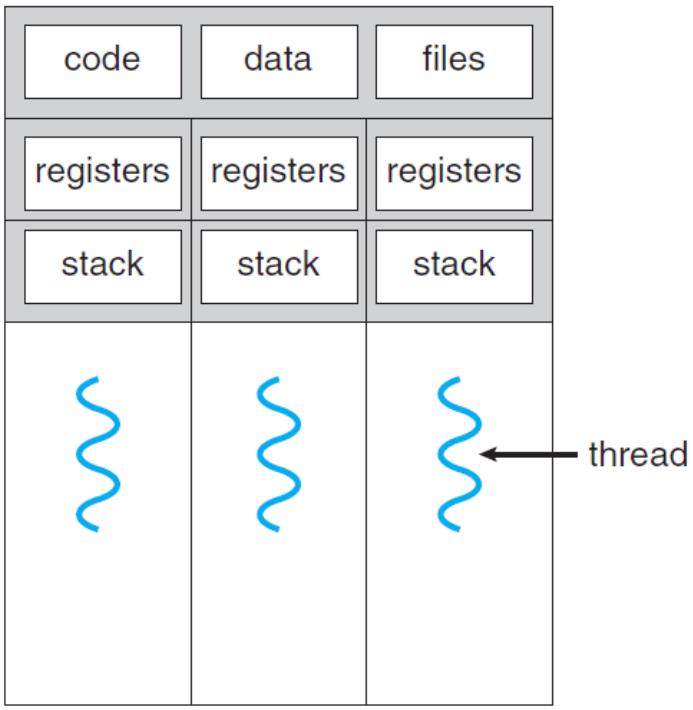

图 4.2　多线程进程

多线程程序相比于多进程程序的优势:

- 同一进程内的线程共享同一块内存, 所以线程之间的通信成本较低.
- 由于是共享式内存机制, 线程创建以及线程切换所需成本较低.

并行的实现方式主要有两种: 多线程并行和多进程并行, 分别对应共享式内存架构 (shared memory architectures) 和分布式内存架构 (distributed memory architectures). 对于多线程并行, 当下主流的接口标准 (interface standard) 有两种: OpenMP[64] 和 POSIX Threads[65,66](简称 Pthreads); 而对于多进程并行主要采用 Message Passing Interface(简称 MPI)[67] 通信机制.

在本毕设中, 我们选择基于 Pthreads 的多线程编程进行 0-1 背包问题的并行分支定界算法实现.

4.2.2 具体实现

主要思路: 在串行算法 Algorithm 2 的基础上进行更改, 当从 $\mathcal{L}$ 中获取节点时, 取出多个节点, 交给多个线程同时进行节点处理.

多线程程序与单线程程序的最大不同在于: 会有多个线程对同一块内存区域进行操作, 比如全局变量对应的内存区域, 这时就会产生冲突, 如读-写冲突、写-写冲突等. 为了解决这一问题, 人们设计了多种方案, 其中一种是*互斥锁 (**mut**ual **ex**clusion, 简写为 mutex)*. 这种方案的主要思想是: 当某个线程想要对某块公共内存进行编辑时, 需要先获取 (acquire) 互斥锁, 然后才能执行相应操作, 编辑完成后需要释放 (release) 互斥锁. 如果互斥锁已被其他线程获取, 则该线程必须等待互斥锁被释放.

在本毕设中, 需要互斥锁保护的全局变量有: 全局下界及对应的可行解、节点队列以及分支定界树中的节点数目. 在访问或编辑对应的全局变量都需要取锁、放锁来避免同步错误.

并行分支定界算法与串行算法的另一个不同在于**终止条件的判断**. 在串行算法中, 当节点队列为空时, 便可终止算法. 但是在并行算法中, 不能直接套用. 考虑一种可能的情况: 在主线程获知节点队列为空的同时, 可能有其他线程刚生成了新的节点, 但尚未放入队列. 若此时终止算法, 则显然是不正确的. 为了解决这一问题, 我们提出如下解决方法:

1. 每次获取节点时, 取队列中节点数目和最大线程数目 `NUM_THREADS` 的较小值作为取出节点的数量, 然后依次创建同等数量的线程对这些节点进行处理.
2. 在判断队列是否为空之前, 主线程需要等待其余线程**全部**执行结束.

这样便可保证算法不会提前终止, 保证最终得到的解是原问题的最优解.

但是这样实现也是存在一定问题的, 比如原有的最佳节点优先的选择顺序将不再成立. 考虑一种可能出现的情况: 在为下一进程分配节点时, 实际最佳节点是前面进程刚产生的节点, 但尚未放入队列. 这时, 下一进程所拿到的节点就不是最佳节点. 而这一顺序一旦被打破, 由命题 2.7, 分支定界树中的节点数目就有可能增加, 相应的计算量也会增加.

4.2.3　实验结果

1. 程序运行环境
   - 运行平台: 中科院“科学与工程计算国家重点实验室”LSSC-IV 高性能计算机集群系统
   - 操作系统: Red Hat Enterprise Linux Server 7.3
   - 作业调度系统: Platform Load Sharing Facility, 简称 LSF
   - 编译器: gcc-10.2.0 和 g++-10.2.0
   - 处理器: Intel Xeon Gold 6140 18 核 Purley 处理器, 主频 2.3 GHz
2. 测试用例

   选择小系数 (small coefficients) 类别中的 `knapPI_3_200_1000.csv` 文件中的前 50 个例子作为测试算例, 分别用串行算法和并行算法求解, 比较实验结果.
3. 测试结果

   全局设置: 运行时间上限为 2 小时, 串行程序所用核数为 1 核, 并行程序所用核数为 36 核.

   表头含义:
   - Instance: 算例名称.
   - sRealTime(sec): **串行**分支定界算法在**节点处理**上所花费的时间 (包含从队列中获取节点和将节点放入队列), 单位为秒.
   - sNodeCount: **串行**算法分支定界树中的节点总数 (含根节点).
   - pRealTime(sec): **并行**分支定界算法在**节点处理**上所花费的时间 (包含从队列中获取节点和将节点放入队列), 单位为秒.
   - pNodeCount: **并行**算法分支定界树中的节点总数 (含根节点).
   - tRatio: 时间比, sRealTime/pRealTime.
   - tRatio: 节点数目比, sNodeCount/pNodeCount.

   (a) 设置最大线程数目为 8 线程时的运行结果如表 4.1 所示:

表 4.1　最大线程数目为 8 线程时的测试结果

Instance	sRealTime(sec)	sNodeCount	pRealTime(sec)	pNodeCount	tRatio	nRatio
inputFile1	19.00	219433	5.00	219433	3.80	1.00
inputFile2	700.00	6386731	164.00	6386731	4.27	1.00
inputFile3	30.00	260491	7.00	260491	4.29	1.00
inputFile4	23.00	195823	5.00	195823	4.60	1.00
inputFile5	142.00	1111741	31.00	1111741	4.58	1.00
inputFile6	1650.00	11856665	354.00	11856665	4.66	1.00
inputFile7	timeout	–	4758.00	154886985	–	–
inputFile8	2117.00	15054457	454.00	15054457	4.66	1.00
inputFile9	44.00	329041	10.00	329041	4.40	1.00
inputFile10	timeout	–	timeout	–	–	–
inputFile11	timeout	–	timeout	–	–	–
inputFile12	timeout	–	timeout	–	–	–
inputFile13	timeout	–	timeout	–	–	–
inputFile14	timeout	–	timeout	–	–	–
inputFile15	timeout	–	timeout	–	–	–
inputFile16	6.00	41899	1.00	41899	6.00	1.00
inputFile17	timeout	–	timeout	–	–	–
inputFile18	timeout	–	timeout	–	–	–
inputFile19	589.00	3918381	125.00	3918381	4.71	1.00
inputFile20	timeout	–	timeout	–	–	–
inputFile21	3.00	19931	1.00	19931	3.00	1.00
inputFile22	6.00	44781	1.00	44781	6.00	1.00
inputFile23	timeout	–	timeout	–	–	–
inputFile24	timeout	–	timeout	–	–	–
inputFile25	4.00	31473	1.00	31473	4.00	1.00
inputFile26	457.00	2925201	96.00	2925201	4.76	1.00
inputFile27	timeout	–	timeout	–	–	–
inputFile28	20.00	141263	4.00	141263	5.00	1.00
inputFile29	timeout	–	timeout	–	–	–
inputFile30	3.00	19943	0.00	19943	–	1.00
inputFile31	99.00	693071	21.00	693071	4.71	1.00
inputFile32	13.00	93047	3.00	93047	4.33	1.00
inputFile33	timeout	–	timeout	–	–	–
inputFile34	timeout	–	1720.00	51792541	–	–
inputFile35	181.00	1287593	39.00	1287593	4.64	1.00
inputFile36	0.00	2689	0.00	2689	–	1.00
inputFile37	1755.00	11881239	372.00	11881239	4.72	1.00
inputFile38	1.00	7517	0.00	7517	–	1.00
inputFile39	1.00	6589	0.00	6591	–	1.00
inputFile40	timeout	–	timeout	–	–	–
inputFile41	timeout	–	timeout	–	–	–

续下页

续表 4.1　最大线程数目为 8 线程的测试结果

Instance	sRealTime(sec)	sNodeCount	pRealTime(sec)	pNodeCount	tRatio	nRatio
inputFile42	timeout	–	timeout	–	–	–
inputFile43	3102.00	20518643	654.00	20518643	4.74	1.00
inputFile44	5751.00	38011971	1206.00	38011971	4.77	1.00
inputFile45	timeout	–	timeout	–	–	–
inputFile46	timeout	–	timeout	–	–	–
inputFile47	timeout	–	timeout	–	–	–
inputFile48	timeout	–	timeout	–	–	–
inputFile49	timeout	–	timeout	–	–	–
inputFile50	timeout	–	5682.00	176321643	–	–

统计结果：

- 在规定时间内, 串行算法能够求解的例子为 25 个, 并行算法能够求解的例子为 28 个. 串行算法在规定时间内能求解的例子并行算法也能求解, 除此之外, 例子 inputFile7、inputFile34、inputFile50 也能被并行算法求解.
- 对于两种算法均能求解的例子, 串行算法在节点处理上所花费的平均时间为 668.64 秒, 并行算法为 142.16 秒, 平均时间比 (串行时间/并行时间) 为 4.70.

(b) 设置最大线程数目为 36 线程时的运行结果如表 4.2 所示:

表 4.2　最大线程数目为 36 线程时的测试结果

Instance	sRealTime(sec)	sNodeCount	pRealTime(sec)	pNodeCount	tRatio	nRatio
inputFile1	19.00	219433	3.00	219433	6.33	1.00
inputFile2	704.00	6386731	110.00	6386731	6.40	1.00
inputFile3	30.00	260491	4.00	260491	7.50	1.00
inputFile4	23.00	195823	4.00	195823	5.75	1.00
inputFile5	142.00	1111741	19.00	1111741	7.47	1.00
inputFile6	1649.00	11856665	217.00	11856665	7.60	1.00
inputFile7	timeout	–	3001.00	154886985	–	–
inputFile8	2132.00	15054457	276.00	15054457	7.72	1.00
inputFile9	45.00	329041	5.00	329041	9.00	1.00
inputFile10	timeout	–	timeout	–	–	–
inputFile11	timeout	–	timeout	–	–	–
inputFile12	timeout	–	timeout	–	–	–
inputFile13	timeout	–	timeout	–	–	–
inputFile14	timeout	–	timeout	–	–	–
inputFile15	timeout	–	timeout	–	–	–
inputFile16	6.00	41899	1.00	41899	6.00	1.00
inputFile17	timeout	–	timeout	–	–	–

续下页

续表 4.2　最大线程数目为 36 线程的测试结果

Instance	sRealTime(sec)	sNodeCount	pRealTime(sec)	pNodeCount	tRatio	nRatio
inputFile18	timeout	–	5170.00	257917933	–	–
inputFile19	601.00	3918381	72.00	3918381	8.35	1.00
inputFile20	timeout	–	timeout	–	–	–
inputFile21	3.00	19931	1.00	19931	3.00	1.00
inputFile22	6.00	44781	1.00	44781	6.00	1.00
inputFile23	timeout	–	timeout	–	–	–
inputFile24	timeout	–	timeout	–	–	–
inputFile25	4.00	31473	0.00	31473	–	1.00
inputFile26	447.00	2925201	53.00	2925201	8.43	1.00
inputFile27	timeout	–	4966.00	243827291	–	–
inputFile28	20.00	141263	3.00	141263	6.67	1.00
inputFile29	timeout	–	timeout	–	–	–
inputFile30	2.00	19943	1.00	19943	2.00	1.00
inputFile31	99.00	693071	13.00	693071	7.62	1.00
inputFile32	12.00	93047	2.00	93047	6.00	1.00
inputFile33	timeout	–	timeout	–	–	–
inputFile34	timeout	–	1015.00	51792541	–	–
inputFile35	181.00	1287593	22.00	1287593	8.23	1.00
inputFile36	0.00	2689	0.00	2689	–	1.00
inputFile37	1793.00	11881239	223.00	11881239	8.04	1.00
inputFile38	1.00	7517	0.00	7517	–	1.00
inputFile39	1.00	6589	0.00	6589	–	1.00
inputFile40	timeout	–	timeout	–	–	–
inputFile41	timeout	–	timeout	–	–	–
inputFile42	timeout	–	timeout	–	–	–
inputFile43	3112.00	20518643	391.00	20518643	7.96	1.00
inputFile44	5872.00	38011971	735.00	38011971	7.99	1.00
inputFile45	timeout	–	timeout	–	–	–
inputFile46	timeout	–	timeout	–	–	–
inputFile47	timeout	–	timeout	–	–	–
inputFile48	timeout	–	timeout	–	–	–
inputFile49	timeout	–	timeout	–	–	–
inputFile50	timeout	–	3473.00	176321643	–	–

统计结果:

- 在规定时间内, 串行算法能够求解的例子为 25 个, 并行算法能够求解的例子为 30 个. 相比于 8 线程, 增添了 2 个例子 inputFile18、inputFile27.
- 对于两种算法均能求解的例子, 串行算法在节点处理上所花费的平均时间为 676.16 秒, 并行算法为 86.24 秒, 平均时间比 (串行时间/并行时间) 为 7.84.

4. 结果分析

(a) 在求解能力方面, 并行算法较串行算法有一定的优势, 可以在相同的时间内求解一些串行算法无法求解的例子.

(b) 在相同例子的求解时间上, 并行算法明显优于串行算法. 如 8 线程程序的平均时间比 (串行时间/并行时间) 为 4.70, 36 线程的平均时间比 (串行时间/并行时间) 为 7.84, 符合预期效果.

(c) 在节点数方面, 两种算法的结果相同. 猜测, 这是因为两者的节点选择顺序相同, 但根据章节 4.2.2 末尾的分析, 串行算法中的节点选择顺序在并行时将被打乱. 分析造成这种巧合的可能原因是: 例子规模较大, 求解对应的 LP 松弛需要花费一定的时间, 这就导致在为下一个线程分配节点时, 前面的线程中尚未有新节点产生, 此时分配的节点仍是最优节点, 保证了求解顺序.

(d) 令人较为意外的是, 当线程数量由 8 线程增至 36 线程后, 相同时间内能够求解的算例并没有明显增多, 并行求解时间也没有成比例的下降. 可能的原因是: 线程数量的增加导致原本巨大的线程创建、销毁成本也随之增加, 拉低了求解效率.

第五章　总结与展望

5.1　总结

1. 本文成功在节点层面上实现了 0-1 背包问题的并行分支定界算法求解, 并达到预期效果;
2. 通过实际编程, 对分支定界方法有了更加细致深入的理解, 并切身感受到并行编程与串行编程的巨大不同.

5.2　展望

1. 在本文中, 我们采用*互斥锁 (mutex)* 对公共内存进行保护. 但这种方式的最大缺点是: 当一个线程对共同内存进行操作时, 其他线程只能等待而不能做其他事情, 造成了计算资源的浪费. 之后考虑使用其他方式, 如*信号量 (semaphore)* 机制, 处理内存冲突;
2. 如结果分析中提到的, 线程的创建和销毁需要耗费巨大的时间及内存成本. 之后考虑使用*线程池 (thread pools)* 机制, 这种机制的主要思想是: 预先创建一定数量的线程, 放在线程池中, 等待任务的分配. 当任务出现时, 挑选一个线程进行执行, 执行完毕后回到线程池继续等待;
3. 考虑实现章节 4.1.2 提到的负载均衡、确定性并行以及其他层面的并行, 如子树并行和次节点并行;
4. 努力实现一般整数规划问题的并行分支定界算法求解, 而不只是 0-1 背包问题;
5. 对于分支定界算法, 考虑实现其他分支策略及节点选择策略.

参考文献

[1] Hane C A, Barnhart C, Johnson E L, et al. The fleet assignment problem: Solving a large-scale integer program[J]. Mathematical Programming, 1995, 70(1):211–232. Publisher: Springer.

[2] Bertsimas D, Patterson S S. The air traffic flow management problem with enroute capacities[J]. Operations research, 1998, 46(3):406–422. Publisher: INFORMS.

[3] Lambert W B, Brickey A, Newman A M, et al. Open-pit block-sequencing formulations: a tutorial[J]. Interfaces, 2014, 44(2):127–142. Publisher: INFORMS.

[4] Gryffenberg I, Lausberg J L, Smit W J, et al. Guns or butter: decision support for determining the size and shape of the south african national defense force[J]. Interfaces, 1997, 27(1):7–28. Publisher: INFORMS.

[5] Wagner H M, Whitin T M. Dynamic version of the economic lot size model[J]. Management science, 1958, 5(1):89–96. Publisher: INFORMS.

[6] Pochet Y, Wolsey L A. Production planning by mixed integer programming[M]. Springer Science & Business Media, 2006.

[7] Sinha G P, Chandrasekaran B, Mitter N, et al. Strategic and operational management with optimization at Tata Steel[J]. Interfaces, 1995, 25(1):6–19. Publisher: INFORMS.

[8] Yaman H, en A. Manufacturer’s mixed pallet design problem[J]. European Journal of Operational Research, 2008, 186(2):826–840. Publisher: Elsevier.

[9] Lee E K, Zaider M. Mixed integer programming approaches to treatment planning for brachytherapy–application to permanent prostate implants[J]. Annals of operations research, 2003, 119(1):147–163. Publisher: Springer.

[10] Preciado-Walters F, Rardin R, Langer M, et al. A coupled column generation, mixed integer approach to optimal planning of intensity modulated radiation therapy for cancer[J]. Mathematical Programming, 2004, 101(2):319–338. Publisher: Springer.

[11] Althaus E, Caprara A, Lenhof H P, et al. A branch-and-cut algorithm for multiple sequence alignment[J]. Mathematical Programming, 2006, 105(2):387–425. Publisher: Springer.

[12] Ford Jr L R, Fulkerson D R. A suggested computation for maximal multi-commodity network flows[J]. Management Science, 1958, 5(1):97–101. Publisher: INFORMS.

[13] Dantzig G, Fulkerson R, Johnson S. Solution of a large-scale traveling-salesman problem[J]. Journal of the operations research society of America, 1954, 2(4):393–410. Publisher: INFORMS.

[14] Markowitz H M, Manne A S. On the solution of discrete programming problems[J]. Econometrica, 1957, 25(1):84–110. Publisher: JSTOR.

[15] Willard Lawrence Eastman. Linear programming with pattern constraints[D]. Cambridge, MA, USA: Harvard University, 1958.

[16] Land A, Doig A. An Automatic Method of Solving Discrete Programming Problems[J]. Econometrica, 1960, 28:497–520.

[17] Cook W. Markowitz and Manne+ Eastman+ Land and Doig= branch and bound[J]. Optimization Stories, 2012. 227–238.

[18] Gomory R E. Outline of an algorithm for integer solutions to linear programs[J]. Bulletin of the American Mathematical Society, 1958, 64(5):275–278.

[19] Bixby R E. A brief history of linear and mixed-integer programming computation[J]. Documenta Mathematica, 2012, (2012):107–121.

[20] Gurobi. https://www.gurobi.com/.

[21] IBM CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer.

[22] SCIP: Solving Constraint Integer Programs. https://www.scipopt.org/.

[23] Nemhauser G L, Wolsey L A. Integer and Combinatorial Optimization[M]. New York: Wiley, 1988.
[24] Schrijver A. Theory of Linear and Integer Programming[M]. New York: Wiley, 1986.
[25] Laurence A Wolsey. Integer Programming[M]. First ed., New York: Wiley, 1998.
[26] Jünger M, Liebling T M, Naddef D, et al. 50 Years of integer programming 1958-2008: From the early years to the state-of-the-art[M]. Springer Science & Business Media, 2009.
[27] Achterberg T, Wunderling R. Mixed integer programming: Analyzing 12 years of progress[M]. . Proceedings of Facets of combinatorial optimization. Springer, 2013: 449–481.
[28] 孙小玲, 李端. 整数规划 [M]. 北京: 科学出版社, 2010.
[29] 孙小玲, 李端. 整数规划新进展 [J]. 运筹学学报, 2014, 18(1):39–68.
[30] Zheng X, Sun X, Li D, et al. Lagrangian decomposition and mixed-integer quadratic programming reformulations for probabilistically constrained quadratic programs[J]. European Journal of Operational Research, 2012, 221(1):38–48. Publisher: Elsevier.
[31] Zheng X, Sun X, Li D. Improving the performance of MIQP solvers for quadratic programs with cardinality and minimum threshold constraints: A semidefinite program approach[J]. INFORMS Journal on Computing, 2014, 26(4):690–703. Publisher: INFORMS.
[32] Chen W K, Dai Y H. Combinatorial separation algorithms for the continuous knapsack polyhedra with divisible capacities[J]. arXiv preprint arXiv:1907.03162, 2019..
[33] Chen W K, Dai Y H. On the complexity of sequentially lifting cover inequalities for the knapsack polytope[J]. Science China Mathematics, 2021, 64(1):211–220. Publisher: Springer.
[34] IBM100 - Power 4 : The First Multi-Core, 1GHz Processor, March, 2012. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/. Publisher: IBM Corporation.
[35] Boehning R L, Butler R M, Gillett B E. A parallel integer linear programming algorithm[J]. European Journal of Operational Research, 1988, 34(3):393–398. Publisher: Elsevier.
[36] Eckstein J. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5[J]. SIAM journal on optimization, 1994, 4(4):794–814. Publisher: SIAM.
[37] Linderoth J T. Topics in parallel integer optimization[D]. Atlanta, GA, USA: Georgia Institute of Technology, 1998.
[38] Ralphs T K, Guzelsoy M, Mahajan A. SYMPHONY Version 5.6, 2020. https://github.com/coin-or/SYMPHONY.
[39] johnjforrest, Vigerske S, Santos H G, et al. CBC Version 2.10.5, 2020. https://github.com/coin-or/Cbc.
[40] BCP: Branch-Cut-Price Framework. https://github.com/coin-or/Bcp.
[41] Xu Y, Ralphs T, Ladányi L, et al. BLIS Version 0.94, 2020. https://github.com/coin-or/CHiPPS-BLIS.
[42] Bulut A, Ralphs T. DisCO Version 1.0, 2018. https://github.com/coin-or/DisCO.
[43] Xu Y, Ralphs T, Ladányi L, et al. ALPS Version 2.0, 2020.
[44] Xu Y, Ralphs T, Ladányi L, et al. BiCEPs Version 0.99, 2020.
[45] Xu Y, Ralphs T K, Ladányi L, et al. Alps: A framework for implementing parallel tree search algorithms[M]. . Proceedings of The next wave in computing, optimization, and decision technologies. Springer, 2005: 319–334.
[46] Xu Y, Ralphs T K, Ladányi L, et al. Computational experience with a software framework for parallel integer programming[J]. INFORMS Journal on Computing, 2009, 21(3):383–397. Publisher: INFORMS.
[47] Shinano Y, Heinz S, Vigerske S, et al. FiberSCIP-a shared memory parallelization of SCIP[J]. ZIB-Report, 2013. 13–55.
[48] UG: Ubiquity Generator framework. https://ug.zib.de/.
[49] Eckstein J, Hart W E, Phillips C A. Pebbl: an object-oriented framework for scalable parallel branch and bound[J]. Mathematical Programming Computation, 2015, 7(4):429–469. Publisher: Springer.

[50] Laurence A Wolsey. Integer Programming[M]. Second ed., New York: Wiley, 2020.
[51] Applegate D, Bixby R. Finding cuts in the TSP (A preliminary report)[R]. Technical report, Citeseer, 1995.
[52] Achterberg T. Constraint integer programming[D]. Berlin, Germany: Technische Universität Berlin, 2007.
[53] Linderoth J T, Savelsbergh M W. A computational study of search strategies for mixed integer programming[J]. INFORMS Journal on Computing, 1999, 11(2):173–187. Publisher: INFORMS.
[54] Little J D, Murty K G, Sweeney D W, et al. An algorithm for the traveling salesman problem[J]. Operations research, 1963, 11(6):972–989. Publisher: INFORMS.
[55] Dakin R J. A tree-search algorithm for mixed integer programming problems[J]. The computer journal, 1965, 8(3):250–255. Publisher: Oxford University Press.
[56] Johnson D S, Garey M R. Computers and Intractability: A Guide to the Theory of NP-completeness[M]. New York: WH Freeman, 1979.
[57] Hans K, Ulrich P, David P. Knapsack Problems[M]. Berlin: Spinger-Verlag, 2004.
[58] Martello S, Toth P. Knapsack Problems: Algorithms and Computer Implementations[M]. Chichester: Wiley, 1990.
[59] Alsuwaiyel M H. Algorithms: Design Techniques and Analysis[M]. Revised ed., World Scientific, 2016.
[60] Pisinger D. Where are the hard knapsack problems?[J]. Computers & Operations Research, 2005, 32(9):2271–2284.
[61] Gendron B, Crainic T G. Parallel branch-and-branch algorithms: Survey and synthesis[J]. Operations research, 1994, 42(6):1042–1066. Publisher: INFORMS.
[62] Ralphs T, Shinano Y, Berthold T, et al. Parallel solvers for mixed integer linear programming[J]. ZIB Report, 2016. 16–74.
[63] Silberschatz A, Galvin P B, Gagne G. Operating System Concepts[M]. Ninth ed., John Wiley & Sons, 2012.
[64] OpenMP. https://www.openmp.org/.
[65] pthreads(7) —Linux manual page. https://man7.org/linux/man-pages/man7/pthreads.7.html.
[66] POSIX Threads. https://en.wikipedia.org/wiki/POSIX_Threads.
[67] Message Passing Interface. https://en.wikipedia.org/wiki/Message_Passing_Interface.