北京大学

博士研究生学位论文

题目：面向组合优化问题的程序自动生成关键技术研究

姓　　名：林舒

学　　号：1501111310

院　　系：信息科学技术学院

专　　业：计算机软件与理论

研究方向：人工智能

导　　师：李文新 教授

二〇二一年六月

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。

摘要

程序自动生成的目标是根据用户对实际问题的描述，自动生成可以求解该问题的计算机程序。组合优化问题求解的目标是找到决策变量的赋值，满足所有约束关系且目标函数的值最优。如何高效求解组合优化问题一直是人工智能领域的重要问题。本文试图针对组合优化问题给出有效的程序自动生成方法，主要创新性工作包括：

第一，提出组合优化问题自动求解新方法，包括组合优化问题描述语言 *COPDL* 的设计和基于 *COPDL* 的求解程序自动生成工具 *COPDL2C* 的设计与实现。相比于传统求解器的针对问题实例求解，本文提出了一种针对整类问题求解的新方法。

第二，提出自动静态分析问题性质的方法，并根据问题性质在求解程序中自动加入搜索剪枝和动态规划优化以提高求解效率。在 *COPDL2C* 的基础上自动设计并实现相应的搜索剪枝和动态规划优化，大幅提高了求解程序的执行效率。

第三，建立包含 40 个用自然语言描述的组合优化问题及其测试数据的数据集，并在此数据集上设计实现基于模板规则和基于框架的将自然语言问题描述自动转换为 *COPDL* 描述的方法——*NL2COPDL*。将 *NL2COPDL* 与 *COPDL2C* 相连接，实现了在自建的数据集上由自然语言描述问题自动生成其求解程序的系统。

第四，基于 *COPDL2C* 设计开发问题建模训练平台 *COPDLOpenJudge*，用于提升程序员的培养效率。*NL2COPDL* 和 *COPDL2C* 两个工具将从问题的自然语言描述到生成问题求解程序的过程自然地分成问题建模和模型求解两个阶段。实验表明这样的任务分解可以使学生提振信心并提升学习效率。

综上所述，本文以组合优化问题求解程序的自动生成作为研究目标，调查研究了组合优化问题自动求解和程序自动生成的最新进展和面临的主要问题，分析了组合优化问题求解程序自动生成中的挑战和重点，提出了一种区别于传统的约束求解器的求解程序自动生成新方法，解决了关键挑战并实现了几个相应的软件工具，包括（1）组合优化问题描述语言 *COPDL* 及求解程序生成工具 *COPDL2C*；（2）自然语言到 *COPDL* 的自动转换工具 *NL2COPDL*；（3）问题建模训练平台 *COPDLOpenJudge*（已经应用于实际教学中）。论文工作成果不仅可以作为组合优化问题求解程序的自动生成的软件工具，还可以作为辅助教学工具被应用到程序员的培养中。

未来工作可向两个方向推进：一是扩展 *COPDL* 和优化 *COPDL2C* 以支持对更多问题的高效求解；二是引入最新自然语言理解研究成果，扩大 *NL2COPDL* 的适用范围。

关键词：组合优化问题，求解程序自动生成，求解程序自动优化，问题模型描述语言，

问题建模训练工具

Research of Key Technologies in Program Synthesis for Combinatorial Optimization Problems

Shu Lin (Computer Software and Theory)
Directed by Prof. Wenxin Li

ABSTRACT

The goal of program synthesis is to automatically generate a computer program for solving a practical problem described by the user. The goal of combinatorial optimization problem solving is to find an assignment to all decision variables, so that all the constraints are satisfied and the value of the objective function is optimal. How to solve combinatorial optimization problems efficiently is an important topic in artificial intelligence. This thesis proposes an effective program synthesis method for solving combinatorial optimization problems. The major innovations of this thesis are:

1. This thesis proposes a new approach for automatically solving combinatorial optimization problems, including the design of a combinatorial optimization problem description language *COPDL*, and the design and implementation of a solving program synthesis tool *COPDL2C* based on *COPDL*. Compared with traditional solvers which focus on solving an instance of a problem, the new approach proposed by this thesis focuses on solving the whole class of a problem.

2. This thesis proposes a method that automatically analyzes problem properties statically, and inserts branch pruning and dynamic programming optimizations into the solving programs according to these properties, in order to improve the efficiency of solving. In addition to the original *COPDL2C*, this method automatically designs and implements the corresponding branch pruning and dynamic programming optimizations, so that the efficiency of a solving program can be significantly improved.

3. This thesis establishes a dataset of natural language descriptions of 40 combinatorial optimization problems, along with their test cases. This thesis also proposes a natural language-*COPDL* automatic translation method—*NL2COPDL*—for this dataset, which is based on templates and rules, as well as sketches. Combining *NL2COPDL* and *COPDL2C*, each problem in this dataset can be translated from its natural language description

into a correct solving program.

4. This thesis designs and develops a problem modeling training system *COPDLOpenJudge* based on *COPDL2C*, which is used for improving the efficiency of training programmers. The whole solving process from understanding the natural language description of the problem to implementing the solving program can be naturally divided into two stages by *NL2COPDL* and *COPDL2C*: the problem modeling stage and the model solving stage. The experiment results show that such task division can enhance students' confidence and improve learning efficiency.

In conclusion, this thesis selects combinatorial optimization problems as the research object, investigates the research progress and major problems of both combinatorial optimization problem solving and program synthesis, analyzes the challenges and key points of program synthesis for combinatorial optimization problems, and proposes a new approach for solving problem synthesis, which differs from traditional combinatorial optimization problem solvers. This approach tackles the major challenges and derives several software tools, including (1) a combinatorial optimization problem description language *COPDL* and the solving program synthesis tool *COPDL2C*; (2) a natural language-*COPDL* automatic translation tool *NL2COPDL*; (3) a problem modeling training platform *COPDLOpenJudge* (has already been applied to teaching practical courses). The products of this thesis are not only software tools for automatically generating solving programs for combinatorial optimization problems, but also aided educating tools for training programmers.

In the future, the research can be further improved in two aspects: first, extending *COPDL* and optimizing *COPDL2C* in order to efficiently solve more problems; second, extending the application scope of *NL2COPDL* by applying the state-of-the-art technologies of natural language understanding.

KEYWORDS: Combinatorial optimization problem, automatic solving program synthesis, automatic solving program optimization, problem model description language, problem modeling training tool

目录

第一章 引言 **1**
1.1 程序自动生成概述 1
1.1.1 程序自动生成的定义 1
1.1.2 程序自动生成的发展历史 2
1.1.3 程序自动生成的研究方向 4
1.1.4 现有程序自动生成研究与本文研究的关系 5
1.2 组合优化约束求解概述 5
1.2.1 组合优化约束模型概述 5
1.2.2 组合优化问题自动求解算法 8
1.2.3 现有约束求解研究与本文研究的关系 15
1.3 基于组合优化约束模型的程序自动生成的引入和研究意义 16
1.4 本文结构 18

第二章 相关工作的研究进展 **21**
2.1 程序自动生成相关工作 21
2.1.1 终端用户编程 21
2.1.2 程序变换和代码生成 22
2.2 约束求解优化技术相关工作 22
2.2.1 通用约束求解优化技术 23
2.2.2 动态规划求解器 23
2.2.3 自动存表技术 23
2.3 自然语言程序设计相关工作 24
2.3.1 语义分析 24
2.3.2 自动约束编程 24
2.4 计算机科学教育相关工作 25
2.4.1 问题求解能力与计算机科学教育 25
2.4.2 支架式教学工具 25
2.5 本章小结 26

第三章 问题描述语言 *COPDL* 和程序自动生成方法 *COPDL2C* **27**
3.1 从组合优化约束模型到程序源代码 27

3.2 程序源代码生成示例 28
3.2.1 问题描述 29
3.2.2 模型分析和程序生成 30
3.3 组合优化约束模型描述语言设计 31
3.3.1 *COPDL* 的核心语法 33
3.3.2 *COPDL* 描述各组成部分 33
3.4 经典问题的 *COPDL* 描述示例 36
3.4.1 最大公约数问题 36
3.4.2 最短路径问题 36
3.5 从组合优化约束模型自动产生求解程序源代码的算法 36
3.5.1 变量类型推导 37
3.5.2 独立变量选取 38
3.5.3 程序代码生成 40
3.6 *COPDL* 及 *COPDL2C* 的有效性验证实验设计与结果 42
3.6.1 数据集 43
3.6.2 实验设计及实验结果 43
3.7 本章小结 44

第四章 程序自动生成方法 *COPDL2C* 中的搜索剪枝和动态规划优化 45
4.1 组合优化问题求解的搜索剪枝和动态规划优化 45
4.2 搜索剪枝优化 46
4.2.1 搜索剪枝在 0/1 背包问题上的应用示例 46
4.2.2 可行性剪枝优化 48
4.2.3 最优性剪枝优化 49
4.3 动态规划优化 51
4.3.1 模型预处理 52
4.3.2 约束模型性质检查 53
4.3.3 动态规划优化的生成 56
4.3.4 不同动态规划求解方法的比较和选择 58
4.3.5 特殊情况处理 61
4.4 优化后的 *COPDL2C* 的性能测试实验设计与结果 63
4.4.1 优化后的 *COPDL2C* 的有效性评估 63
4.4.2 *COPDL2C* 在求解经典组合问题上的效率 64
4.4.3 *COPDL2C* 优化模型与原始模型在求解效率上的比较 65

4.5 本章讨论 69
4.6 本章小结 70

第五章 从自然语言问题描述产生约束模型的翻译方法 *NL2COPDL* 73
5.1 从自然语言描述到约束模型 73
5.2 组合优化约束模型自然语言描述的特点 74
5.2.1 组合优化约束模型自然语言描述的受限性 74
5.2.2 组合优化约束模型自然语言描述的翻译难点 74
5.2.3 组合优化约束模型自然语言描述的翻译方法 74
5.3 建立组合优化问题的自然语言描述数据集 75
5.4 从自然语言描述自动产生 *COPDL* 模型的算法 77
5.4.1 0/1 背包自然语言描述示例 78
5.4.2 描述处理 79
5.4.3 元素提取 79
5.4.4 框架生成 86
5.4.5 框架模型填充 87
5.5 *NL2COPDL* 正确性和性能测试实验设计与结果 88
5.5.1 研究问题及实验设计 88
5.5.2 实验结果 89
5.6 本章小结 92

第六章 问题建模训练平台 *COPDLOpenJudge* 的关键技术和应用 95
6.1 程序设计类与算法分析类课程中的问题求解能力 95
6.2 问题建模训练平台 *COPDLOpenJudge* 的设计与实现 96
6.2.1 问题建模训练平台 *COPDLOpenJudge* 的技术架构 96
6.2.2 问题建模训练平台 *COPDLOpenJudge* 的使用方法 97
6.2.3 错误提示与反馈信息 99
6.3 *COPDL* 及 *COPDL2C* 辅助教学效果评估实验设计与结果 99
6.3.1 实验设计 100
6.3.2 实验结果 102
6.4 本章小结 104

第七章 结论和展望 107
7.1 本文工作总结 107
7.2 未来工作展望 108

参考文献 **111**

附录 A *COPDL* 手册 **123**

A.1 *COPDL* 简介 123

A.2 *COPDL* 实例 123

A.2.1 最大公约数 123

A.2.2 0/1 背包问题 124

A.3 *COPDL* 语法 125

A.3.1 基本框架 125

A.3.2 变量定义 126

A.3.3 基本运算 127

A.3.4 复合运算 127

A.3.5 量词 128

A.3.6 特殊约束关系 128

A.4 常见错误 129

A.4.1 编译错误 Compile Error 129

A.4.2 其他错误 130

附录 B 基础课程期末试题数据集信息及 *COPDL2C* 有效性测试结果 **131**

B.1 计算概论 A 课程期末上机考试题详情及实验结果 131

B.2 程序设计实习课程期末上机考试题详情及实验结果 131

B.3 数据结构与算法 A 课程期末上机考试题详情及实验结果 133

附录 C 包含 40 个用自然语言描述的组合优化问题的数据集 **135**

C.1 来源于课程例题、作业与测试的 33 个问题的自然语言描述 135

C.2 来源于 MiniZinc 教程或标准测试集的 7 个问题的自然语言描述 138

个人简历及博士期间研究成果 **141**

致谢 **143**

北京大学学位论文原创性声明和使用授权说明 **145**

图 目 录

1.1 对手绘几何图形的自动识别和美化系统 4
1.2 回溯搜索算法伪代码 9
1.3 Gecode 内部实现结构 10
1.4 DPLL 算法伪代码 12
1.5 人类通过编程解决实际问题的过程 16
1.6 本文核心章节的体系结构图 19

3.1 *N* 皇后问题的 *COPDL* 描述 28
3.2 *N* 皇后问题的 MiniZinc 描述 28
3.3 *N* 皇后问题的 Essence 描述 29
3.4 立方体问题 29
3.5 立方体问题的 *COPDL* 描述 30
3.6 对立方体问题生成的基于循环的程序 32
3.7 *COPDL* 核心语法：主体框架 33
3.8 *COPDL* 核心语法：变量声明和类型范围约束 33
3.9 *COPDL* 核心语法：语句 33
3.10 *COPDL* 核心语法：表达式 34
3.11 最大公约数问题的 *COPDL* 描述 36
3.12 最短路径问题的 *COPDL* 描述 37
3.13 *COPDL2C* 处理步骤 37
3.14 一个特殊的 `#required` 部分示例 38
3.15 寻找最小独立变量集的算法伪代码 39
3.16 基于循环的程序的实现逻辑 40
3.17 基于递归的程序的实现逻辑 41
3.18 0/1 背包问题的自然语言描述与 *COPDL* 描述 42
3.19 0/1 背包问题对应的基于递归的程序 43

4.1 添加高级优化技术后的 *COPDL2C* 框架 46
4.2 应用了剪枝优化的 `_find_sel_bp(...)` 47
4.3 应用了动态规划优化的 `_find_sel_dp(...)` 57
4.4 0/1 背包问题的优化模型 59

4.5 嵌套 0/1 背包问题的 MiniZinc 约束模型 60
4.6 嵌套 0/1 背包问题动态规划版本的 MiniZinc 约束模型 62

5.1 数据集中自然语言描述的符号数分布 77
5.2 数据集中自然语言描述的关键词数分布 77
5.3 数据集中自然语言描述的提及数分布 77
5.4 *NL2COPDL* 处理步骤 77
5.5 *NL2COPDL* 自动产生的 0/1 背包问题的 *COPDL* 描述 78
5.6 0/1 背包问题自然语言描述中第一句的短语结构树 80
5.7 0/1 背包问题自然语言描述中第一句的依赖树 81
5.8 0/1 背包问题自然语言描述中的提及和共指关系 82
5.9 数词转化为数值算法 83
5.10 TOTAL-OF 和 DO-NOT-EXCEED 模板和类型推导逻辑 84
5.11 模板匹配示例（分别匹配 TOTAL-OF 与 DO-NOT-EXCEED 模板） 85
5.12 基于 TOTAL-OF 模板和 DO-NOT-EXCEED 模板的框架生成过程 87
5.13 数据集中自然语言描述对应 *COPDL* 模型的符号数分布 89
5.14 数据集中自然语言描述对应 *COPDL* 模型的语句数分布 89
5.15 数据集中自然语言描述对应 *COPDL* 模型的变量数分布 89
5.16 *NL2COPDL* 产生 *COPDL* 模型的运行时间 90
5.17 *COPDL2C* 求解产生的 *COPDL* 模型的平均时间 91

6.1 *COPDLOpenJudge* 技术架构 96
6.2 *COPDLOpenJudge* 关键操作的交互图 98
6.3 *COPDLOpenJudge* 提交 *COPDL* 描述界面 98
6.4 *COPDLOpenJudge* 反馈结果界面 99
6.5 学生在实验前的求解问题信心程度 102
6.6 学生对使用 *COPDL* 和 C 编写程序难易程度的评价 103
6.7 学生对使用 *COPDL* 和 C 查错和修改难易程度的评价 103
6.8 学生在阶段 1 结束后的求解问题信心程度 104

表 目 录

1.1 三类求解器的比较 15

3.1 *COPDL2C* 的内置运算符和累积函数表 35
3.2 立方体问题的标识符表 37
3.3 *COPDL* 及 *COPDL2C* 在数据集中 49 个问题上的有效性评估 44

4.1 优化后的 *COPDL2C* 在数据集中 49 个问题上的有效性评估 63
4.2 *COPDL2C*，Gecode，和 Chuffed 求解各问题实例的时间和空间消耗 . . 66
4.3 不同求解工具在各问题模型上的求解时间（秒） 68

5.1 来源于 MiniZinc 教程或标准测试集的 7 个问题 76
5.2 *NL2COPDL* 预定义的关键词 82
5.3 自动生成模型与人工设计模型的平均求解时间（秒）对比 91

6.1 评测结果分类及含义 97
6.2 用于不同组学生进行对比实验的编程任务 100
6.3 各组学生需要完成的任务 101
6.4 各组学生求解 6 个组合优化问题的情况统计 102

B.1 计算概论 A（2018 年秋）期末试题列表及有效性测试结果 132
B.2 计算概论 A（2019 年秋）期末试题列表及有效性测试结果 132
B.3 程序设计实习（2019 年春）期末试题列表及有效性测试结果 132
B.4 程序设计实习（2020 年春）期末试题列表及有效性测试结果 133
B.5 数据结构与算法 A（2018 年秋）期末试题列表及有效性测试结果 133
B.6 数据结构与算法 A（2019 年秋）期末试题列表及有效性测试结果 133

C.1 来源于课程例题、作业与测试的 33 个问题的自然语言描述 135
C.2 来源于 MiniZinc 教程或标准测试集的 7 个问题的自然语言描述 139

第一章　引言

程序自动生成技术起源于人工智能，兴起于软件工程，它很好地结合了这两个领域的相关知识和技术。程序自动生成的目标是根据用户对实际问题的描述，自动生成可以求解该问题的计算机程序。本文选定组合优化问题作为研究对象，试图针对这类问题给出一种高效的求解程序自动生成方法。

组合优化问题是人工智能领域的经典问题，可用组合优化约束模型（combinatorial optimization constraint model）进行准确描述。这类问题广泛存在于各种管理、规划相关的实际问题或理论计算中，例如寄存器分配问题、旅行商问题、组合拍卖的赢家确定问题、绿色物流规划问题、地下水流速估计问题、基因组映射问题等等。

本章首先概述了程序自动生成和组合优化问题求解这两个与本文密切相关的领域的背景知识。接着，引出本文的研究主题——基于组合优化约束模型的程序自动生成问题，并分析阐述其研究意义和难点。最后，介绍本文的组织结构和各章节内容。

1.1　程序自动生成概述

程序自动生成（program synthesis），通俗来说，就是用户向计算机描述待求解的问题，而计算机需要自动生成能够求解该问题的程序（Basin et al 2004）。一般来说，用户使用抽象层次更高的描述方法，比如输入-输出样例对、形式化的数学语言、图表、甚至是自然语言等（Gulwani 2010b），而计算机根据用户给出的输入自动产生抽象层次更低、或者优化过的机器可理解的表达形式（如程序、测试数据、规格说明等）。

1.1.1　程序自动生成的定义

Bodik et al（2012）给出了一种形式化的程序自动生成定义：找到一个程序 P 满足式 (1.1)。

$$\exists P.\forall x.\Phi(x, P(x)) \tag{1.1}$$

其中 $P(x)$ 为程序 P 在输入 x 下的输出，Φ 为用户给定的输入和输出之间需要满足的关系，即问题的规范化说明；x 为问题中的变量。

一个程序自动生成技术，至少要满足以下两个条件之一，才是有实际应用价值的：

- 提高生产率：用户使用 Φ 描述问题比直接编写 P 更快；

- 保证正确性：验证 Φ 的正确性比验证 P 的正确性更容易。

此外，也有不少人认为，程序自动生成实际上是一种广义的自动定理证明（Bodik et al 2012）：即将程序自动生成的过程，视为证明给定命题是否存在合法的解的过程，而证明的方式，就是找到一种合法解的构造方法，而这个构造方法即为程序。这种思想可以形式化描述为证明命题 1.2 是否为真。

$$\forall x.\exists y.\Phi(x,y) \tag{1.2}$$

其中 Φ 和 x 的含义同前述，y 为问题的一个合法解。而构造合法解 y 的过程，即为 P。

随着研究的不断深入以及程序自动生成技术的不断进步，许多程序自动生成技术被应用到了其他领域。程序自动生成，从广义上说，从程序的自动生成，扩展到所有利用了与程序自动生成技术相似的技术的其他计算机输出内容的自动生成，例如数学试题（Alvin et al 2014）、图表（Cheema et al 2012）。

1.1.2 程序自动生成的发展历史

程序自动生成的发展历史可分为早期、中期、以及现代三个阶段。

1.1.2.1 早期

对程序自动生成技术的研究可以一直追溯到二十世纪五六十年代。

1957 年，阿隆佐·邱奇（Alonzo Church）定义并研究了一类根据数学约束产生电路的问题（Church 1957）。虽然这项工作的目标是生成电路而非程序，但它仍然被视为程序自动生成领域最初的研究之一。正因为如此，一些研究者也将程序自动生成称为“邱奇问题”。

随着人工智能的兴起，以冯·诺依曼为代表的研究者们，就开始尝试运用人工智能领域的一些与自动定理证明相关的理论和技术，来搭建一种能够对给定的数学问题，自动生成针对性的求解程序的系统（von Neumann 1956）。在人工智能领域中，这类系统被称为“自动化程序员（automatic programmer）”。这些研究主要围绕于如何建立完整的程序设计理论，并将其与现有的数学理论，或者相对已经成熟的自动机理论联系起来。因此，这一段时期的程序自动生成方法，主要有两大类：

- **演绎式综合方法**：起源于人工智能领域的自动定理证明，该方法将程序自动生成的过程视为对命题的证明，而证明过程即为目标程序（G. Robinson et al 1968，J. A. Robinson 1966）。
- **基于自动机的综合方法**：一种程序化综合方法，首先将约束条件转化为自动机，然后利用自动机理论进行合并化简，最后根据规则将简化后的自动机翻译为目标程序（Church 1962，von Neumann 1956）。

然而，受到计算机硬件资源等各种因素的限制，这些研究基本停留在理论上，使用的各种数学方法虽然正确，但效率较低。对于当时的计算机使用者来说，程序自动生成技术并不能起到太多的帮助，因此很少有成功的应用系统。

1.1.2.2　中期

二十世纪七十年代前后，计算机技术快速发展，计算机应用也日益普及。由于计算机软件的规模越来越大，而传统的软件开发技术并没有重大突破，使得软件质量差，生产效率低，造成了重大的“软件危机”，也直接导致了软件工程学科的兴起。

为了解决在大型软件开发过程中遇到的种种问题，对于程序自动生成技术的研究在八十年代末期又重新活跃了起来。在软件工程中，程序自动生成及其相关技术被运用于各种软件开发辅助工具的研制，包括：

- 程序错误定位和修复（Caballero et al 2007，Shapiro 1982）；
- 字符串处理操作（Gulwani 2010a，Lau et al 2003）
- 多线程垃圾回收处理（Vechev、Yahav、Bacon 2006，Vechev、Yahav、Bacon、Rinetzky 2007）。

与上一时期以抽象的数学问题为问题域不同的是，在以实际问题，特别是以软件问题为问题域的时候，需要考虑的情况以及演绎推理时使用的规则就少了很多。再加上计算机硬件性能的迅速提升，使得将程序自动生成技术用于实际应用系统成为可能。

在这一段时期内，各种程序自动生成方法不断涌现并快速发展：基于理论自动机的综合方法被进一步完善，并应用于一些实际问题（Büchi et al 1969）；演绎式综合方法在原有基础上，通过引入一些强有力的启发函数，大幅度提高了程序自动生成的效率（Manna、Waldinger 1980，Manna、Waldinger et al 1992）。

值得一提的是，现代高抽象层次的程序语言的发展，也被认为是程序自动生成的一种研究方向。

1.1.2.3　现状

随着布尔可满足性求解器（Davis et al 1962）和可满足性模块理论求解器（Bryant et al 2001）的广泛应用，出现了将实际问题转化为布尔可满足性问题，然后利用相应的求解器（有关这两类求解器的详细描述分别参见第 1.2.2.2 节和第 1.2.2.3 节）完成程序自动生成的思路（Solar-Lezama 2008）；间接利用求解器的求解器辅助综合方法也逐渐开始兴起（Torlak et al 2014）。

2013 年，一种基于语法指导的程序自动生成（Syntax-guided Synthesis，SyGuS）统一框架被提出（Alur、Bodik et al 2013），吸引了来自国内外不同研究机构的研究者

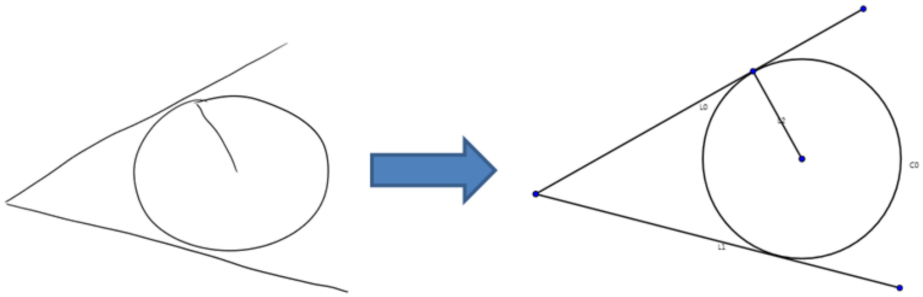

图 1.1　对手绘几何图形的自动识别和美化系统

的关注。不过，大部分可用的算法只能针对简单程序片段（例如单个表达式）进行综自动生成（Alur、Singh et al 2018，Becker et al 2015，W. Lee et al 2018）。2015 年，Volkstorf（2015）提出了用公理证明的方法自动生成 PHP 语言程序思路，在求解诸如判断质数、因数分解等简单数学问题取得一定的成功。

在归纳式综合方法（也称示例编程）的研究中，Polozov et al（2015）设计了一种程序空间表示方法 VSA（version space algebra），并在其基础上设计了归纳式的程序自动生成框架 FlashMeta。Ji et al（2020）在 FlashMeta 的基础上进入概率模型，使得综合效率得到极大提升。DeepCoder 将深度学习用在基于输入输出样例对的程序自动生成任务中，能够从已有其他程序中挑选代码片段并组合成程序，但目前仅在短程序（5 行以内）的自动生成上有少量的成果（Balog et al 2017）。

随着研究的不断深入，一些研究者发现，在问题域不具有特殊性的情况下，现有的程序自动生成方法已经很难突破目前的理论框架，几乎无法进一步提高效率。因此，部分研究者将自己的研究范围缩小到一个特定领域的特定问题域上，从而通过利用领域知识以及研究者自身对该领域问题的求解经验，根据问题域特点选择最适合的程序自动生成技术，搭建出更成功的应用系统（Gulwani、Jha et al 2011，Le et al 2013，Singh et al 2012）。

与此同时，程序自动生成的概念，也已经不再仅仅局限于程序（特指可执行代码片段）的自动生成，而是扩展到了任何可以利用程序自动生成技术实现的计算机输出内容的自动生成，例如图 1.1 所示的对手绘几何图形的美化（Cheema et al 2012）。

1.1.3　程序自动生成的研究方向

程序自动生成技术起源于人工智能，兴起于软件工程，它很好地结合了这两个领域的相关知识和技术。在 2012 年 ACM Computing Classification System （CCS）（见 `https://www.acm.org/about/class/2012/`）中，程序自动生成技术分别出现在软件类下的编程技术类中，以及计算方法学类下的人工智能类中。

程序自动生成技术在两个领域中的研究侧重点并不完全相同。在软件工程中，对

程序自动生成的研究着重于对描述语言的设计及完备性验证（Gulwani、Korthikanti et al 2011）、利用输出程序语言的特性进行优化和验证（Kuncak et al 2010）等方面，其目标用户群主要为软件开发工程师；而在人工智能领域中，则偏重于利用问题域的性质，对产生的程序细节进行优化，其目标用户群主要为非计算机专业的普通用户（Ahmed et al 2013，Alvin et al 2014）。

此外，在人机交互领域的研究中，程序自动生成技术也被应用于一些图表等非文本化内容的自动生成（Cheema et al 2012）。

1.1.4　现有程序自动生成研究与本文研究的关系

现有的程序自动生成相关研究已经能够自动产生求解程序解决许多实际问题。然而，面对组合优化问题，现有方法并不适用。主要原因包括：

- **模型描述复杂**：组合优化问题约束形式多样，没有固定模式，不容易进行自动分析。
- **算法设计复杂**：组合优化问题的求解算法复杂，而许多现有程序自动生成技术和系统并不需要自动设计或优化算法，在用户没有额外指定求解方法的情况下，并不能很好地实现此类问题的自动求解。
- **程序实现复杂**：组合优化问题的求解程序相对较长，程序空间大，而现有程序自动生成技术和系统基本只能在较小程序空间上实现对简单代码段的自动生成。

而本文专注于研究针对组合优化问题的程序自动生成方法。

1.2　组合优化约束求解概述

约束求解（constraint solving），是针对给定的由一系列约束定义的约束模型，在庞大的搜索空间中找到一组变量赋值，使得所有约束均被满足且最大化目标函数值的过程。组合优化约束求解则是将问题进一步限定为组合问题，并在此限制下完成上述求解过程。在组合优化约束求解领域的研究中，通常先将问题抽象成一个组合优化约束模型，然后基于搜索等算法框架，探究如何引入新的优化策略提高求解效率。

1.2.1　组合优化约束模型概述

组合优化约束模型（combinatorial optimization constraint model 或 combinatorial optimization constrained model）是通过变量及变量间约束关系来形式化描述某个实际问题的一种数学模型，一般用于求解问题的可行解或最（较）优解。使用计算机求解实际问题时，一般需要先对问题建立组合优化约束模型，再使用求解算法得到目标

解。组合优化约束模型可广泛应用于各种管理、规划相关的实际问题或理论计算中（Gonzalez et al 2014，Papadimitriou et al 2000），如：

- **寄存器分配问题**（*Lozano et al 2016*）：将大量的程序变量合理分配给少量寄存器，以提高程序执行效率。
- **旅行商问题**（*Applegate et al 2011*）：给定城市列表和城市间距离，找到一条回路访问每座城市恰好一次并返回起点，使得移动路线最短。
- **组合拍卖的赢家确定问题**（*Sandholm et al 2002*）：给定拍卖中的出价集合，将各拍卖品合理分配给出价者，使得拍卖者的收益最大化。
- **绿色物流规划问题**（*Sbihi et al 2010*）：根据供需关系、运输能力等，合理规划物流活动中的运输、存储、配送等环节，降低物流成本，同时通过降低能源消耗和废物排放来减少对环境的污染。
- **地下水流速估计问题**（*Hobe et al 2018*）：给定地下水道网络结构，水流入口和出口等信息，估计每段区间水流的流速。
- **基因组映射问题**（*Blazewicz et al 2005*）：给定一组基因片段及它们之间的距离约束，要求在一段 *DNA* 序列中定位这些基因片段。

相比于一般的约束模型，组合优化约束模型额外限定了问题必须是**组合问题**，即变量数量有限且各变量取值范围有限——换言之，问题的搜索空间是有限的。这一限制为设计和实现通用的自动求解工具提供了可能。

1.2.1.1 组合优化约束模型的定义

组合优化约束模型 $\mathcal{M}$ 一般包含以下几个要素：

- 若干个参数变量 $\mathcal{A} = \{a_1, a_2, \ldots\}$，变量值由问题实例确定；
- 若干个决策变量 $\mathcal{V} = \{v_1, v_2, \ldots\}$，变量定义域为有限集，变量值需要通过求解获得；
- 若干个约束关系 $\mathcal{R} = \{r_1(\mathcal{A}, \mathcal{V}), r_2(\mathcal{A}, \mathcal{V}), \ldots\}$，定义了变量值需要满足的条件；
- 零个或一个目标函数 $O(\mathcal{A}, \mathcal{V})$，用于比较不同可行解的优劣。

求解组合优化约束模型，就是在给定参数变量值的条件下，找到一组决策变量的赋值，在满足所有约束关系的同时，尽可能优化目标函数的值。其中，如果没有指定目标函数（或者认为目标函数值恒为常数 $O \equiv C$），那么求解该模型只需找到一组可行解即可，即该模型对应一个约束满足问题（constraint satisfaction problem，CSP）；否则，该模型对应一个组合优化问题（combinatorial optimization problem，COP）。因此，本文将**约束满足问题视为组合优化问题的一个子集**，不做特殊区分。

使用组合优化约束模型描述问题，对问题求解有诸多好处：

- 通过问题抽象，忽略与问题求解无关的干扰内容（如背景故事、事物名称等），问题描述得以简化，变量之间的关系也更清晰。

  例如，社交网络里包含大量人名、消息等信息，若将每个人抽象成一个点，将有用的关系抽象成边，可以便于进一步分析和处理。
- 使用数学语言形式化描述变量之间的关系后，可以方便地利用各种数学定理和公式进行手工计算和证明。

  例如，对于立体几何问题，建立空间坐标系并采用向量的形式对其进行描述，可以充分利用与向量有关的定理对其进行求解。
- 约束模型容易表示成能够被计算机理解的描述形式，从而可以充分利用自动求解工具对其进行求解。

  相对于自然语言，约束模型使用的语法规则和要素更少更严格，使得模型容易被计算机自动解析的同时，不存在二义性，从而计算机对模型的理解与用户保持一致。

1.2.1.2　组合优化约束模型的应用局限性

随着计算机性能的提升以及算法的发展，人们越来越依赖于使用计算机来求解问题。组合优化约束模型作为人类与计算机沟通的桥梁之一，对问题建立组合优化约束模型能力显得越发重要。

然而，在实践中经常可以发现，对同一问题的不同约束模型，使用同样的求解算法进行求解，效率可能大相径庭。换言之，约束模型设计的好坏，往往是能否最终被高效求解的决定性原因之一。

要设计出好的组合优化约束模型，建模者一般需要具备以下知识：

- 对实际问题本身有正确的认识。

  建模者需要正确理解问题所有细节，不遗漏任何一个关键的约束条件。
- 对该领域的知识有一定的了解。

  例如，对与图有关的问题建模，需要知道图论中的基本概念和表示方法。
- 对求解算法有一定的了解。

  例如，对于基于搜索的求解算法，若能够在模型中根据搜索的顺序合理安排各变量和约束的顺序，可以使得求解算法优先处理更有可能存在最优解的分支，从而更快找到目标解。

尤其是作为专家知识的后两点要求，对于普通用户来说掌握难度较大。

现有的组合优化约束模型相关研究主要集中在如何对通用求解算法进行优化，缺乏对模型的分析和模型特性的挖掘，导致模型的好坏直接影响求解算法的效率。建模者只有对求解算法有一定了解，才能设计出较好的模型，这无形中提高了用户的门

槛。

1.2.2 组合优化问题自动求解算法

1970 年代以来，研究者们设计并实现了许多约束求解器（constraint solver），用于求解通用组合优化问题。同时，软件工程领域的研究者也尝试引入布尔可满足性求解器（satisfiability solver，简称 SAT solver）和可满足性模理论求解器（satisfiability modulo theories solver，简称 SMT solver）来完成类型推导、程序正确性验证、基于符号执行的软件测试、基于搜索的程序片段生成等工作。

1.2.2.1 约束求解器

自动化的约束求解方法是人工智能领域的一个重要研究方向。

1972 年，Waltz（1972）在其博士论文中提出并实现了一种算法，能够根据二维图像和阴影信息来还原一个三维图像结构。该算法中的核心技术**约束传播**，后来被 Mackworth（1977）泛化为弧一致性（AC-3）算法，成为约束求解的关键技术之一。二十世纪七十年代中后期，关于约束求解搜索技术的研究开始兴起。初期的研究主要采用反向树搜索算法，在此基础上，后来又发展出了各种变体（Gaschnig 1977a,b、1979），形成了一套完整的**回溯搜索**技术。约束传播和回溯搜索是所有穷尽式求解算法的关键技术，目前主流求解器的核心算法采用的依然是这两种技术的融合：以回溯搜索方式遍历搜索空间，各结点展开时应用约束传播减少分支数（Chu et al 2012，Perron et al 2019，Schulte et al 2006）。

二十世纪八十年代，局部搜索被引入约束求解，成为一种用于提高搜索效率的优化策略。局部搜索是非穷尽式的搜索算法，其主要基于一些启发式的近似算法，例如爬山法、模拟退火、遗传算法等（Hoos et al 2004，Hromkovic 2010）。

关键技术：约束求解算法使用的关键技术包括：

- **回溯搜索**：在搜索过程中，若当前的部分变量赋值会使得某个约束无法满足，则撤销对其中某个变量的赋值。图 1.2 展示了回溯搜索的伪代码。回溯搜索一般可分为时间顺序回溯（只能撤销最近的变量赋值）和非时间顺序回溯（可以根据冲突原因撤销任意变量赋值）。此外，确定搜索顺序（包括变量顺序和赋值顺序）可以采用不同的启发式策略，比如，由此也产生了许多不同的回溯搜索变体（Gaschnig 1977a,b、1979）。
- **约束传播**：在搜索过程中，将当前的部分变量赋值代入约束并化简，以此来减少未赋值变量的合法取值范围。约束传播的核心思想是约束的局部相容性，根据参与变量的数量可分为结点相容（一元约束），弧相容（二元约束），路径相容（多元约束）等（Mackworth 1977）。

输入: 组合优化问题模型 $\mathcal{M}$，当前变量赋值集合 $\hat{A}$
输出: 使得 $\mathcal{M}$ 成立的变量赋值 A，或者 "NoSolution"

```
1  Function Backtracking (M, Â):
2      if Â 对所有变量赋值 then
3          return Â;
4      end
5      v ← 启发式选择一个未赋值变量（根据模型 M 和当前变量赋值集合 Â）;
6      foreach value ∈ v 的定义域 do
7          if v = value 与 Â 在 M 中不会产生矛盾 then
8              result ← Backtracking (M, Â ∪ {v = value});
9              if result ≠ NoSolution then
10                 return result;
11             end
12         end
13     end
14     return NoSolution;
```

图 1.2　回溯搜索算法伪代码

- **局部搜索**：局部搜索是从一个完全的变量赋值出发，每一步尝试通过改变少量变量的赋值来减少冲突，不断迭代直至找到可行解。局部搜索一般比回溯搜索的求解效率高，但它是非穷尽式的搜索，是一种近似算法，在可行解很稀疏的问题中不一定能找到解，因此往往作为回溯搜索中的一种优化策略（Hoos et al 2004，Hromkovic 2010）。

主流约束求解器基本结构：约束求解器的结构一般都比较复杂，不同求解器的具体内部实现也不尽相同。不过，由于约束求解器均是基于上述三种关键的约束求解技术，故一般包含如下几个部分：模型处理，变量模块，约束传播器，搜索分支器，搜索引擎，以及求解器内核。以目前最常用的 Gecode（Schulte et al 2006）为例，其内部结构表现为图 1.3。其中各模块功能为：

- **模型处理**：对输入的模型进行编译，转化为内部表示。该模块的主要任务是将模型中的变量和约束提取出来交给相应的变量子模块，并将搜索参数传递给搜索引擎。
- **变量模块**：每个子模块处理一种类型变量的范围推导、约束变形等。
- **约束传播器**：实现约束传播技术：在搜索过程中，根据约束中的矛盾，缩小变量范围。约束传播器根据参与约束的变量数量，可分为一元、二元、多元、混合等不同类型的约束传播器。
- **搜索分支器**：确定搜索顺序：根据特定策略改变搜索顺序，构建搜索树。常用的有改变变量顺序的变量选择器，改变变量值搜索顺序（增序、降序、二分等）

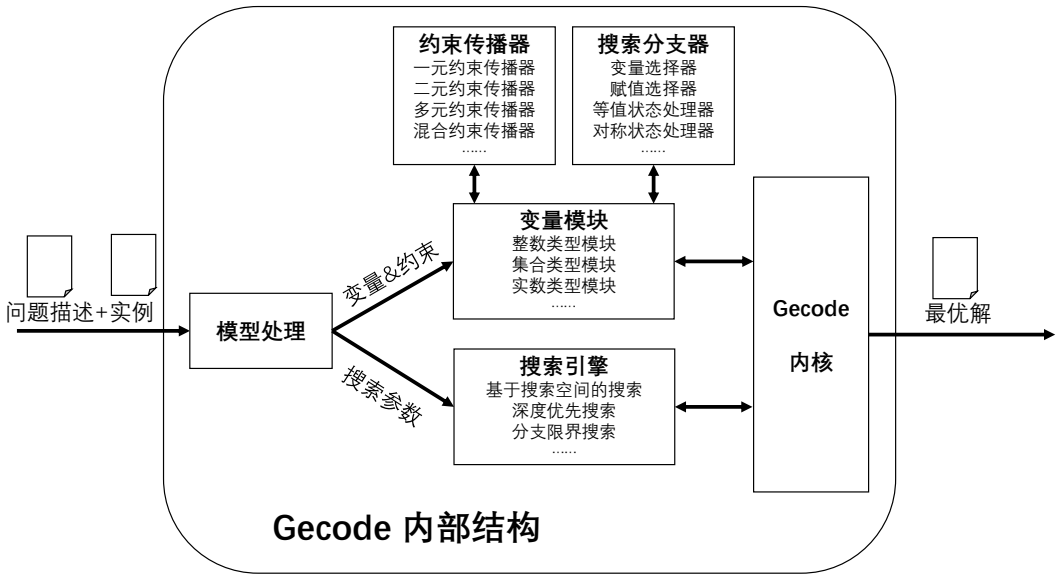

图 1.3　Gecode 内部实现结构

的赋值选择器，用于合并等价状态的等值状态处理器和对称状态处理器等。

- **搜索引擎**：实现具体的搜索技术。常用的搜索策略包括：深度优先搜索、广度优先搜索、迭代加深搜索等。此外，搜索引擎内也提供了对各种通用优化策略（包括局部搜索）的支持，用户可以通过设置搜索参数来控制是否应用这些策略。
- **Gecode 内核**：提供基本功能，同时，在求解过程中，通过变量模块生成和更新搜索树，并调用搜索引擎实现对搜索树的搜索。

常见通用优化：在约束求解技术的基本框架上，为了提高求解效率，研究者们提出了许多通用的优化策略来避免或减少求解过程中的冗余搜索。每种优化策略都会对某一类具有特定性质问题的求解上起到较为显著的优化效果，但也可能引入额外的时间和资源开销影响求解效率。用户可以通过手动设置搜索参数来控制这些策略的应用与否。一些典型的优化策略包括：

- **缓存**技术通过存储搜索状态来避免相同状态被重复计算（B. M. Smith 2005）。
- **对称求解**和**子问题支配**技术分别分析不同状态之间的等价性和支配关系，以减少无用的状态探索（Chu et al 2012，Gent et al 2006）。
- **问题分解**技术根据变量赋值情况，将原问题分解成不相关的子问题，从而可以独立求解各子问题，最后再组合成完整的解（Kitching et al 2007）。
- **惰性语句生成**技术从搜索过程中遇到的冲突中发现新的约束条件，以减少搜索空间（Ohrimenko et al 2009）。

常见约束求解器：各种约束求解器在学术界主要用于教学和研究，同时也在工业界用于求解一些实际问题，包括程序验证，智能网格管理、时间安排、任务规划、路线导航等。目前较为活跃的约束求解器有（大致按照诞生时间从早到晚排序）：

- **Gecode**：C++ 语言的一套开源工具库（Schulte et al 2006），官方网址 https://www.gecode.org/。
- **SICStus Prolog**：目前最先进的 Prolog 开发环境，采用了高性能的 Prolog 求解引擎（Carlsson et al 2010），官方网址 https://sicstus.sics.se/。
- **chuffed**：最先进的基于惰性从句生成的求解器（Chu et al 2012），官方网址 https://github.com/chuffed/chuffed。
- **JaCoP**：由 Java 语言实现的约束求解器（Kuchcinski et al 2013），官方网址 https://sourceforge.net/projects/jacop-solver/。
- **Yuck**：由 Scala 语言实现的**局部搜索**求解器（Bjordal et al 2015），官方网址 https://github.com/informarte/yuck/。
- **Choco-solver**：Java 语言的一套约束求解工具库（Choco-solver dev team 2021），官方网址 https://choco-solver.org/。
- **OR-tools**：由 Google 公司开发的运筹学开源研究工具（Perron et al 2019），官方网址 https://developers.google.com/optimization/。

1.2.2.2　布尔可满足性求解器

布尔可满足性求解器是用于求解布尔可满足性问题的一类工具，较为著名的有：

- **GRASP** 实现了一种通用的搜索求解算法（Marques-Silva、Sakallah 1999），官方网址 http://vlsicad.eecs.umich.edu/BK/Slots/cache/sat.inesc.pt/~jpms/grasp/。
- **zChaff** 实现了布尔可满足性求解算法 Chaff（Mahajan et al 2004，Moskewicz et al 2001），官方网址 https://www.princeton.edu/~chaff/。
- **MiniSAT** 是极简的开源求解器，旨在帮助引导研究者或开发者从事布尔可满足性问题求解的研究或开发工作（Een 2005），官方网址 https://minisat.se/。

布尔可满足性问题是为每个布尔变量赋值为“$TRUE$” 或 “$FALSE$”，使得给定布尔公式的值为“$TRUE$”。它是第一个被证明是 NP 完全的问题（Cook 1971）。布尔可满足性求解器采用的主要求解算法是 DPLL（Davis-Putnam-Logemann-Loveland）算法，伪代码如图 1.4所示。DPLL 算法（Davis et al 1962） 是一个完全的、基于回溯的搜索算法。该算法能够针对给定的合取范式（CNF）形式表示的约束，找到一组满足约束的变量赋值。该算法于 1961 年被提出，时至今日仍是几乎所有高效的完全布尔可满足性求解器的基础框架。

输入: 合取范式中的从句集 Φ，当前变量赋值 $\hat{A} \subset V \times \{TRUE, FALSE\}$
输出: 变量赋值函数 $A : V \to \{TRUE, FALSE\}$，使得 Φ 成立

```
1  Function DPLL (Φ, Â):
2      将 Â 中的变量赋值代入 Φ 中每个从句并化简;
3      Â′ ← Â∪ 所有只有唯一可选值变量的相应赋值;
4      if Φ 中存在矛盾的子句 then
5          return FALSE;
6      end
7      if dom(Â) = V then
8          输出 Â;
9          return TRUE;
10     end
11     v ← 启发式选择一个未赋值变量（根据参与约束数量和重要程度等）;
12     return DPLL (Φ, Â′ ∪ {⟨v, TRUE⟩}) or DPLL(Φ, Â′ ∪ {⟨v, FALSE⟩});
```

图 1.4　DPLL 算法伪代码

DPLL 算法每一步首先将 $\hat{A}$ 中已被赋值变量的具体值代入并化简合取范式 Φ 中的每一个从句，并将只有唯一合法取值的变量的相应赋值直接加入 $\hat{A}$；若在此期间产生冲突（存在一个永远为假的从句），则停止当前分支并回溯。否则，判断 $\hat{A}$ 是否已对所有变量赋值，若是，将 $\hat{A}$ 作为最终的可行解输出，并结束算法。若既没有产生矛盾也没有完成对所有变量的赋值，则启发式地选择一个未被赋值的变量 v，在 $\hat{A}$ 中添加其赋值“$v = TRUE$”或“$v = FALSE$”，分别递归进行下一步搜索。

进入二十一世纪以来，研究者们在 DPLL 算法的基础上，提出并实现了如下三类优化技术，使得布尔可满足性求解器的求解效率迅速提高：

- 优化变量搜索顺序，即研究如何更好地选取下一个未被赋值的变量，其中较为著名的是独立变量状态衰减和（variable state independent decaying sum，VSIDS）技术（Moskewicz et al 2001）；
- 优化约束传播，即引入高级数据结构提高代入变量值和从句化简步骤的效率（Een 2005，Mahajan et al 2004）；
- 使用的非时间顺序回溯（non-chronological backtracking）和从句学习（clause learning）技术，避免重复遇到相同的冲突。冲突驱动从句学习（conflict-driven clause learning，CDCL）布尔可满足性求解器是采用此技术的最具代表性的一类求解器（Marques-Silva、Lynce et al 2009，Zhang et al 2001）。

布尔可满足性求解器的求解效率相对较高，然而它只能用于求解**所有变量均为布尔类型**的**约束满足**问题。

1.2.2.3　可满足性模理论求解器

可满足性模理论求解器是用于求解可满足性模理论问题的一类工具，常用的包括：

- **Z3 定理验证工具**是由微软公司开发的开源定理验证工具（De Moura et al 2008），是目前使用人数最多，最稳定的求解器，官方网址 `https://github.com/Z3Prover/z3/`。
- **CVC4** 是一套开源的定理验证工具（Barrett、Conway et al 2011）。官方网址 `https://cvc4.github.io/`。
- **Yices** 是由斯坦福国际研究院（SRI International）研究开发的形式化验证工具（Dutertre 2014），官方网址 `https://yices.csl.sri.com/`。

可满足性模理论问题是在布尔逻辑的基础上，引入了以一阶逻辑形式描述的理论（包括数学理论、数据结构理论等），因此能够比布尔可满足性问题具有更强的表达能力，对人工智能、形式化方法领域问题的表达上更具优势（金继伟 et al 2015）。而可满足性模理论求解器也是以布尔可满足性求解器为核心，并在其基础上加入针对各类特殊约束形式的模理论及相应的求解算法。

可满足性模理论求解器的核心算法：现有的可满足性模理论求解器的核心算法是 DPLL(T) 算法（Ganzinger et al 2004）——DPLL 算法的扩展版本，并采用了冲突驱动从句学习（CDCL）技术。DPLL(T) 算法的基本步骤是：

1. 将各原子一阶逻辑公式替换为布尔变量使得问题转化为布尔可满足性问题并用 DPLL 算法求得一组可行赋值；若不存在可行赋值则证明原问题无解，结束求解过程；
2. 用理论和相应的理论求解技术（包括线性规划、高斯消元等）计算符合各原子公式赋值的解；若存在可行解，输出解并结束求解过程；否则将冲突作为约束加入原问题并返回步骤 1。

可满足性模理论库：可满足性模理论库（SMT-LIB）于 2003 年创建，其收集、整理、规范了一系列在实际应用（程序验证、符号执行等）中常用的理论，包括数学理论和计算机基本数据结构理论等，因此成为所有可满足性模理论求解器的基础和标准（Barrett、Fontaine et al 2016）。可满足性模理论库中定义的理论主要包括如下几类：

- **核心理论**：定义了基础布尔逻辑运算符（与、或、非、蕴含等）的性质。例如：A 蕴含 B 等价于非 A 或 B。
- **整数理论**：定义了整数相关运算符（四则运算、比较运算、取绝对值等）的性质。例如：整除操作遵循欧几里得定义，即在除数为正数时向下取整，除数为负数时向上取整（Boute 1992）。

- **位向量理论**：定义了位运算符（按位取与、或、非、移位运算等）的性质。例如：左移操作在不会溢出的情况下等于让原数值乘以 2。
- **实数理论**：定义了实数相关运算符（四则运算、比较运算、取绝对值等）的性质。例如：负数不存在开方根。
- **浮点数理论**：定义了浮点数相关操作（无穷大、非数、舍入等）的性质，当前采用的是 IEEE-754 二进制浮点数算术标准（Brain et al 2015）。例如：0 除以 0 为非数（NaN）。
- **实数整数转化理论**：定义了实数整数之间转换函数的性质。例如：整数转换为实数再转换回整数，应与原整数相等。
- **数组理论**：定义了数组相关操作（存、取元素等）的性质。例如：在数组特定位置存入数据然后立刻读取同一位置内容，将得到与存入数据相同的结果。
- **字符串理论**：定义了 Unicode 字符串相关运算（比较运算、判断子串等）及正则表达式相关操作（查找、替换等）的性质。例如：字符串 S 是两个字符串 S 和 T 连接产生字符串的一个前缀。

可满足性模理论求解器的优劣势：相比于约束求解器，可满足性模理论求解器的优点是，如果原子公式的形式适合被某种特定的理论求解技术计算，就可以对问题进行高效求解；但与此同时，如果没有合适的理论求解技术用于计算，可满足性模理论求解器的求解效率就比较低，甚至完全无法求解。特别是对于优化类问题，现有的可满足性模理论求解器仅能够处理优化目标为线性函数的情况（金继伟 et al 2015）。

1.2.2.4 不同求解器的比较

本节介绍了三种类型的求解器：约束求解器，布尔可满足性（SAT）求解器，以及可满足性模理论（SMT）求解器。表 1.1 列出了这三类求解器的区别，具体来说：

- **针对问题**：布尔可满足性求解器只能判定变量均为布尔类型的公式是否存在可行解；可满足性模理论求解器则是在布尔可满足性求解器的基础上，引入一阶逻辑及模理论，从而具备判定一阶逻辑公式可满足性的能力；约束求解器则能够对任何使用数学约束（包括一阶逻辑公式）表示的组合优化问题进行求解，特别是遇到缺少模理论、优化函数非线性的问题时，可满足性模理论求解器求解效率低下甚至无法求解，而约束求解器依然能够适用。
- **核心算法**：布尔可满足性求解器的核心是 DPLL 算法；可满足性模理论求解器的核心是在 DPLL 算法的基础上发展出来的 DPLL(T) 算法；约束求解器的核心是用回溯搜索遍历搜索空间，并在结点展开时使用约束传播。
- **发展开端**：布尔可满足性求解器的发展起源于二十世纪六十年代 DPLL 算法的提出；可满足性模理论求解器则是在进入二十一世纪后才逐渐受到研究者的重

表 1.1　三类求解器的比较

求解器	约束求解器	SAT 求解器	SMT 求解器
针对问题	组合优化	布尔可满足性	一阶逻辑可满足性
核心技术	回溯搜索，约束传播	DPLL	DPLL(T)
发展开端	二十世纪七十年代	二十世纪六十年代	二十一世纪初
实际应用	任务规划，科学计算	定理证明	程序验证，符号执行

视；约束求解器的研究是从二十世纪七十年代开始兴起，并逐步形成了一套完整的求解框架。

- **实际应用**：布尔可满足性求解器最初是为了完成定理证明（Cook 1971，Davis et al 1962），后来研究者们将其引入软件工程领域并由此扩展出了可满足性模理论求解器，用于程序验证、符号执行等（Barrett、Moura et al 2011）；约束求解器在人工智能领域则被用于对任务规划、科学计算等问题的形式化建模和自动求解工作中（Gonzalez et al 2014，Papadimitriou et al 2000）。

约束求解器的研究群体主要为算法理论、人工智能等方向的学者，对约束求解器的研究从求解一般的组合优化问题的回溯搜索和约束传播技术出发，根据具体问题的特殊性质，引入一系列搜索优化来提高求解效率，即约束求解器是沿着**从一般到特殊**的路径发展的。而布尔可满足性求解器和可满足性模理论求解器的研究群体主要为软件工程方向的学者，对这两类求解器的研究则从求解特殊问题——布尔可满足性问题的 DPLL 算法出发，不断叠加实际应用所需要的新模理论以扩大求解问题的范围，即布尔可满足性求解器和可满足性模理论求解器是沿着**从特殊到一般**的路径发展的。

随着求解算法的发展和用户需求的提高，各类求解器的研究者们均开始尝试引入其他类型求解器的技术来弥补自身的不足，以获得更好的求解效果。约束求解器将可满足性模理论求解器的一些模理论，甚至整个求解器封装为一个子模块，用于高效求解具有特定性质（线性规划、整数规划等）的子问题（Schulte et al 2006）；布尔可满足性求解器和可满足性模理论求解器将约束求解器中采用的回溯搜索和约束传播等关键技术与 DPLL 和 DPLL(T) 结合，提高搜索效率（De Moura et al 2008，Een 2005）。

1.2.3　现有约束求解研究与本文研究的关系

现有的约束求解技术及系统虽然能在一定程度上有效解决组合优化问题，但在实际应用中，约束求解器往往是作为黑盒直接作为软件系统的一个模块，这种做法会造成额外时间和空间资源的浪费，同时也可能存在某些兼容性问题。而本文致力于研究如何直接产生能够求解组合优化问题的轻量级程序源代码，方便用户更好地了解求解程序的工作原理，或将该程序集成到更大的软件系统中。

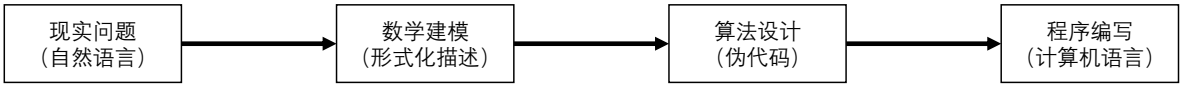

图 1.5　人类通过编程解决实际问题的过程

1.3　基于组合优化约束模型的程序自动生成的引入和研究意义

人类通过编程解决实际问题的过程，以及在过程中表达方式的变化，一般可以表示为图 1.5。

程序自动生成技术，往往以第二步形式化描述（极个别以第一步自然语言）为输入，使用问题领域的相关知识对其进行理解和处理，采用一些核心的自动生成技术，完成算法的自动设计，最后再根据目标程序的相关知识，自动生成相应的程序，从而完成自动化程序设计。

在实际的程序自动生成应用中，作为问题形式化描述的方法有很多，其中，

- **形式化一阶谓词逻辑，高阶函数式语言**等描述方式虽然对于计算机来说最容易处理，但不易于用户理解和书写；
- **领域特定语言**等描述方式的适用性比较局限，表达能力不足；
- **输入-输出样例对，程序及程序框架**等描述方式难以完备地覆盖所有逻辑，查错难度较大；
- **非文本化数据，自然语言**等描述方式具有非常好的用户体验，但可能存在二义性，计算机正确处理的难度较大。

组合优化约束模型作为数学上常用的形式化描述方法，相较于上述描述方法，具有诸多优势：

- 用户只需基本的数学基础即可理解和书写组合优化约束模型；
- 适用范围广，表达能力强，能够方便地描述所有约束满足问题及组合优化问题；
- 能够抽象、完备地描述复杂的约束关系，查错难度较低；
- 无二义性，计算机可理解。

因此，将组合优化约束模型作为程序自动生成的起点和基础，用户可以较容易地向计算机准确描述一个较为复杂的问题，计算机也能够正确理解用户意图，并将复杂的算法和技术应用到程序自动生成中。

然而，采用合适的方法对组合优化问题进行求解并不容易。虽然组合优化问题理论上一定可以使用暴力搜索的办法找到最优解，但这种办法求解效率低下，往往难以在用户可容忍的时间限制内得出结果。人工设计并实现组合优化问题的高效算法，用户需要具备较强的能力和技能依次完成如下工作：

1. **对问题进行形式化建模：**将一个非形式化方式（如自然语言、图表、示例等）描述的问题用形式化的语言完整描述，正确定义所有的变量、变量之间的约束关系、以及优化目标，为后续深入分析问题提供基础。
2. **实现基本求解程序：**编写完整且正确的基本求解程序。
3. **对问题本身特性做深入分析并个性化设计优化算法：**分析变量之间的约束关系是否具有某些性质（如可行性剪枝性质，最优子结构性质等），然后针对问题所具有的性质，结合相应的优化策略（搜索剪枝、动态规划等），设计优化算法并应用到基本求解程序中。

前人已有的程序自动生成相关技术和系统已经能够自动产生求解程序解决各种实际问题。然而，现有的工作均无法很好地应用于组合优化问题。主要原因包括：

- **模型描述复杂：**组合优化问题一般使用表达能力较强的约束模型进行建模，相对于领域特定模型（如数据库查询模型、字符串处理模型等），约束模型的操作符种类多，语句形式灵活，没有相对固定的求解模式，自动化分析和处理较为困难。
- **算法设计复杂：**组合优化问题的求解和优化算法复杂，而约束模型本身并不包含任何算法描述，因此程序自动生成系统需要自动完成算法设计。相较而言，许多现有程序自动生成技术和系统并不需要自动设计或优化算法，只是将用户给定的一种算法描述形式（触发器-动作规则、伪代码等）直接翻译成程序，或者基于规格说明（状态机、类图等）产生不含具体处理细节的代码框架。
- **程序实现复杂：**组合优化问题的求解程序（1）语句数量多且相互关联，既（2）包含多次反复执行的循环语句，又（3）包含只有在边界条件下才会遇到的判断语句。这三个特点的存在意味着无法单纯使用基于机器学习的技术来实现组合优化问题的程序自动生成，因为要实现具备这种程序自动生成能力的系统，需要用海量的数据做训练，且数据必须覆盖并突出体现所有的特殊情况处理方式。

于此同时，现有的约束求解技术及系统虽然能够有效解决组合优化问题，但在实际应用中，存在下列问题：

- 求解器对于用户来说是黑盒，用户无法知道求解器如何求解问题，亦难以从中发现潜在的导致求解效率降低的问题描述缺陷。
- 求解器自身的额外时间和空间开销较大，直接集成到软件系统中会造成一定的资源浪费。
- 求解器对编程语言有依赖，与既有系统可能存在兼容性问题。
- 布尔可满足性求解器和可满足性模理论求解器的适用范围受到现有理论的局限，不足以求解任意组合优化问题。

为了实现面向组合优化问题的程序自动生成，本文首先在主流约束模型描述语言的基础上，设计了问题描述语言 *COPDL* 并实现了相应的程序自动生成工具。*COPDL* 能够表达任何组合优化问题（包括约束满足问题），表达能力与常用的约束模型描述语言（如 MiniZinc 语言）完全相同。但由于 *COPDL* 最终是为了生成能够求解某个问题所有实例的程序，而非只是计算该问题给定示例的解，*COPDL* 在设计上引入了两个其他约束模型描述语言所不具备的特性，在不降低表达能力的前提下，为后续程序自动生成技术的应用提供了支持：

- 语法上更好地支持使用常量、变量或表达式对变量的定义域、数组和集合的维度进行显式声明；语义上要求各变量的范围均受到显式或隐式的限制。
- 语法上要求参数变量定义、决策变量定义既约束关系、优化目标、输出表达式四个部分独立分开，这样虽然略微损失了描述的灵活性，但既提高了可读性，又方便了最终生成程序时对输入和输出的处理。

接着，本文以 *COPDL* 为中间语言，通过研究、设计和实现从组合优化问题产生程序的下列三个步骤的算法和技术，分别使得人工编程求解组合优化问题的三个重要且困难的步骤自动化/半自动化，从而降低了用户的学习和使用门槛。

- **对问题进行形式化建模**：结合基于模板的语义分析技术，基于规则的模板翻译技术，以及基于框架的程序自动生成技术，设计并实现了从自然语言描述自动产生 *COPDL* 描述模型的算法和工具，方便用户描述问题。
- **实现基本求解程序**：根据预设规则，结合启发式搜索等方法，针对给定问题的 *COPDL* 描述产生正确的基于循环或递归的基本求解程序，作为之后优化的基础。
- **对问题本身特性做深入分析并个性化设计优化算法**：利用专家知识对组合优化约束模型进行等价变换，自动分析模型本身性质，根据性质设计候选优化策略，并通过自动估计时间和空间复杂度对候选优化策略进行筛选，最终产生高效的优化算法并实现于基本求解程序之中。

此外，本文设计并实现了基于组合优化约束模型程序自动生成技术的程序在线评测平台，将其作为教学辅助工具，用于培养程序员的问题求解能力。

1.4 本文结构

本文第二章总结了相关工作的研究进展，第三章到第六章是本文的核心章节，分别介绍了本文工作的四个主要创新性工作（图 1.6 展示了这些章节之间的体系结构），第七章总结了全文工作并展望了未来工作。这些章节的具体内容组织如下：

第二章综述了相关研究工作的进展及与本文工作的关系，包括程序自动生成相关

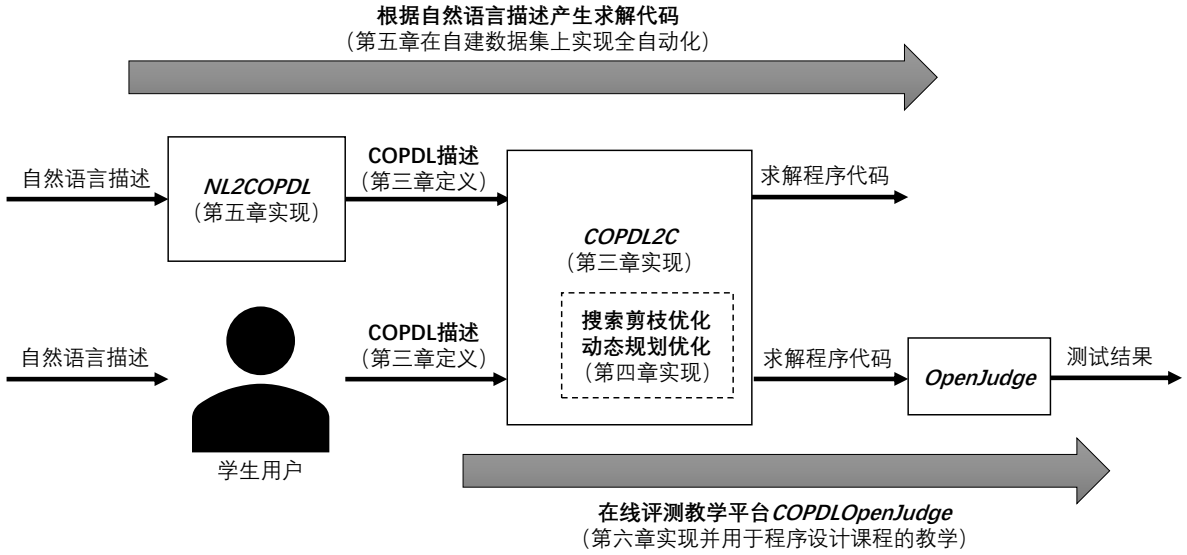

图 1.6　本文核心章节的体系结构图

工作，约束求解优化相关工作，自然语言程序设计相关工作，以及计算机科学教育相关工作。

第三章定义了一种适合用于程序自动生成的组合优化约束模型描述语言 *COPDL*，该语言表达能力较强，能够有效描述各种类型、采用 1.2.1.1 节的形式定义的**单优化目标的组合优化问题或无优化目标的可满足性问题**；同时提出并实现了从 *COPDL* 描述到 C 程序代码的自动生成新方法 *COPDL2C*，该方法基于预设规则，结合启发式搜索等算法，快速自动产生基于循环或递归枚举的求解程序的基本框架。

第四章提出并实现了对组合优化约束模型求解程序进行自动**搜索剪枝**和**动态规划**的优化技术。这些优化技术利用专家知识对求解算法进行等价变换，并根据搜索、动态规划等相关知识对求解程序的基本框架进行优化，产生新的正确且高效的求解程序。

第五章给出了一个包含 40 个用自然语言描述的组合优化问题及其测试数据的数据集，并在此数据集上给出了一种基于模板和规则的将自然语言问题描述自动转换为 *COPDL* 描述的方法——*NL2COPDL*。将 *NL2COPDL* 与 *COPDL2C* 相连接，实现了**在自建的数据集上根据自然语言描述问题自动生成其求解程序**的目标。

第六章介绍了 *COPDL2C* 在程序员培养中的应用。作者基于 *COPDL2C* 开发了一个在线评测教学平台 *COPDLOpenJudge*，将其**应用于计算机专业学生和非专业程序员的组合优化问题求解能力的训练中**，提升了程序员的培养效率。

第七章总结了本论文的主要工作和创新点，并对未来工作进行了展望。

第二章　相关工作的研究进展

本文从多个方面讨论了面向组合优化约束模型的程序自动生成关键技术，相关工作可以总结为如下四个大类：（1）程序自动生成技术相关工作，包括终端用户编程和代码生成；（2）约束求解优化技术相关工作，包括约束求解通用优化技术、基于动态规划的求解技术和自动存表技术；（3）自然语言程序设计相关工作，包括语义分析和自动约束编程；（4）计算机科学教育相关工作，包括对问题求解能力培养方法的研究和支架式教学工具。本章综述这几类研究方向的研究进展和特点，并说明了这些工作与本文各章节研究内容的关系。

2.1　程序自动生成相关工作

本节介绍程序自动生成的相关工作，包括终端用户编程、程序变换和代码生成。

2.1.1　终端用户编程

Barricelli、Cassano et al（2019）总结了 14 类终端用户编程技术，包括基于规则，基于模板，基于模型的技术，以及示例编程（programming by example，PbE）。

大多数基于规则的技术目标是让用户能够自定义智能设备、物联网应用等的行为（Barricelli、Valtolina 2017，Ghiani et al 2017，Guo et al 2011，Houben et al 2016，Korpipää et al 2006，Su et al 2014）。例如，Ghiani et al（2017）实现了技术和工具支持用户声明触发器-动作规则（`IF/WHEN` ⟨*trigger_expression*⟩ `DO` ⟨*action_expression*⟩）。要使用这些技术，用户必须自己设计并实现基础的算法，尽管这些算法比较简单。

基于模板的技术提供模板帮助用户自定义应用特性 （Abraham et al 2006，Berenz et al 2014，Bhatti et al 2015，Carmien et al 2008，Eagan et al 2008，Riss 2012）。例如，Eagan et al（2008）实现了应用 Cocoa Buzz，支持用户使用模板来个性化页面布局，选择感兴趣的信息。这类用户编程更接近于对软件的组件做个性化配置，几乎不涉及算法和实现。

基于模型的技术通常需要使用一个领域特定语言来描述模型，而代码生成器就可以分析这个模型并输出代码实现 （Bogdan et al 2008，Díaz et al 2011，Faravelon et al 2014，Garzotto et al 2011，Hartmann et al 2009，Maleki et al 2014）。例如，Bogdan et al（2008）实现了一个工具，支持用户采用类似对话的方式为交互界面建模，然后这个工具根据模型自动生成相应的界面。这类技术所面向的模型一般只局限于很小的领域，相对来说比较简单，自动设计的算法也比较单一。

示例编程允许终端用户使用非常轻松的方式进行编程：通过具体示例表达需求，系统自动推测出泛化的程序 （Bates 2003，Gulwani 2010a，Meng et al 2011、2013，Menon et al 2013，Miller et al 2001，Nix 1984）。例如，SMARTEdit （Bates 2003） 自动从用户的演示中学得重复性的文本编辑工作，例如“将鼠标指针移动到行末”。Gulwani（2010a）从用户提供的输入-输出对中自动生成关于字符串操作的正则表达式。但通过输入-输出样例不容易完备地描述复杂的约束关系，因此不适合用于解决诸如组合优化问题等复杂问题。

2.1.2 程序变换和代码生成

程序变换类的工具根据用户提供的原始程序和转换规则，自动生成新的代码（An et al 2018，Bravenboer et al 2008，Burson et al 1990，Cordy et al 1988，Ladd et al 1994）。例如，Stratego/XT （Bravenboer et al 2008） 提供了一种用于方便描述变换规则和重写策略的 Stratego 语言，而 XT 工具可以根据声明的规格说明自动实现程序变换。An et al（2018）实现了 tInferer 工具，能够在 Java 和 Swift 语言编写的源程序之间进行自动翻译。这类工作均是基于翻译规则实现代码生成，一般不需要在翻译过程中添加新的算法或优化。

基于模型的软件工程工具可以从 UML 模型（如类图） （J. Klein 2015，Steinberg et al 2015），状态机 （Dion 2004），或自然语言描述的系统需求（Franců et al 2009） 自动生成代码。例如，Eclipse 提供了“Uml to Java Generator” （Obeo 2021） 支持从 UML 模型生成 Java 源代码。这些模型通常只是定义了程序的结构，生成的代码一般也只是框架，不包含具体的业务逻辑。

一些工具可以基于代码模板和领域特定语言生成程序 （Popma 2004，Team 2017，Xpand 2021）。例如，JET （Popma 2004） 为用户提供了一种类似于 JSP 的语法用于定义生成代码的模板，同时还提供了通用模板引擎，能够基于模板生成 SQL，XML，Java 代码等。这类工具的主要目标是将用户模板根据要求组合在一起，而非求解一个组合优化问题。

2.2 约束求解优化技术相关工作

本节介绍约束求解优化技术的相关工作，包括：约束求解的通用优化技术，基于动态规划的求解技术，以及自动存表技术。

2.2.1　通用约束求解优化技术

对于约束求解的基于搜索的通用求解算法，研究者们提出了许多自动化技术来避免或减少求解过程中的冗余搜索。

- 缓存技术通过存储搜索状态来避免相同状态被重复计算（B. M. Smith 2005）。
- 对称求解和子问题支配技术分别分析不同状态之间的等价性和支配关系，以减少无用的状态探索（Chu et al 2012，Gent et al 2006）。
- 问题分解技术根据变量赋值情况，将原问题分解成不相关的子问题，从而可以独立求解各子问题，最后再组合成完整的解（Kitching et al 2007）。
- 惰性语句生成技术从搜索过程中遇到的冲突中发现新的约束条件，以减少搜索空间（Ohrimenko et al 2009）。

这类优化技术均是在求解过程中动态实现的，只能针对于某个具体实例的求解而非针对整类问题的求解。

2.2.2　动态规划求解器

许多研究者也曾经尝试过使用动态规划来加速约束求解。

- 一些领域相关的语言（KOLMOGOROV 和 GAP 等）支持用户对某些领域的问题建立动态规划模型，然后应用特定方法，在特定问题上实现基于动态规划的约束求解（Prestwich et al 2018，Sauthoff et al 2011）。然而，这些方法均不能用于对通用模型的自动分析和处理。
- 有些研究者致力于研究动态规划性质和适用性的理论基础（De Moor 1994），但这些理论与实际应用之间仍有一定的距离。
- 函数式编程语言的一些函数库允许用户以枚举-选择的形式描述问题，然后使用动态规划的方法计算结果（Morihata et al 2014）。不过这些库只能处理较为简单的问题，无法解决复杂的（多变量、多约束等）或含有特殊情况的问题。

2.2.3　自动存表技术

自动存表技术将求解过程中的中间结果存储与特定的数据结构中，以达到减少或消除重复计算的目的。

- 用户可使用 MiniZinc 注解和 Picat 等工具明确说明哪些中间值需要被复用以及如何存储（Dekker et al 2017，Zhou et al 2015）。这些技术无法自动发现可被优化的内容。
- 一些优化技术可以利用等价性探测出子问题，根据用户指定的顺序建立多值决策图（MDD）或确定性可分解否定范式（d-DNNF），从而实现自动存表（Chu

et al 2012，Uña et al 2019）。这些技术需要用户的额外输入。

2.3 自然语言程序设计相关工作

本节介绍自然语言程序设计的相关工作，包括语义分析和自动约束编程。

2.3.1 语义分析

语义分析（semantic parsing）是自然语言处理中的一项常见的基础任务，其目标是将自然语言描述转化为语义等价的机器可理解的表示形式。之前的研究者实现了许多语义分析工具，用于将自然语言描述转化为测试数据，领域相关的规则或约束，以及可执行程序等。例如，Litmus（Dwarakanath et al 2012） 使用基于模板匹配规则的方法，从功能性需求文档直接生成功能性测试用例。Sawant et al（2013）实现了一个从软件需求中提取知识表示图的工具，从而能够根据知识表示图进一步自动产生测试数据。UMTG（Wang et al 2015） 能够从使用自然语言描述的用例规格说明中生成可执行的系统测试用例。该工具使用自然语言处理技术根据规格说明构造测试用例模型，通过与软件工程师不断交互来完善模型，最终使用约束求解产生测试数据。Bajwa、Bordbar、M. G. Lee（2010）的工作分别实现了将自然语言规格说明转化为 OCL 约束（Bajwa、Bordbar、M. G. Lee 2010），SBVR 商务规则（Bajwa、M. G. Lee et al 2011），以及 Alloy 模型（Bajwa、Bordbar、M. G. Lee、Anastasakis 2012） 的工具。这些工具均采用基于规则的语义分析来解释自然语言描述，并应用基于规则的翻译规则产生最终的软件制品。SQLNet（Xu et al 2017） 使用基于框架的自动生成方法，利用神经网络预测框架中每个占位符需要填入的内容，从而实现从自然语言描述到 SQL 查询语句的自动翻译。Lei et al（2013）提出一种贝叶斯生成模型，用于将自然语言描述的输入格式转化为可执行的 C++ 语言输入处理代码段。NL2Bash（Lin et al 2019）使用基于编码器-解码器模型的方法，将英文句子转化为 Linux shell 脚本。GANCoder（Zhu et al 2019） 使用生成对抗网络（generative adversarial networks，GAN）实现了从自然语言描述到 Python 代码的单行翻译。

2.3.2 自动约束编程

有些辅助工具能够自动或半自动地帮助用户更好地进行约束编程。例如，领域特定语言 ELAN（Kirchner et al 1998） 允许用户使用抽象程度更高的表示方式声明规则和策略，而它的解释器能够自动从用户提供的规格说明产生约束模型。Flener et al（1998）提出了一种从一阶谓词逻辑表示的形式化规格说明自动产生约束模型的方法。Conjure（Wetter et al 2015） 以用 Essence 语言描述的抽象约束规格说明为输入，自动

完善规格说明并生成具体的约束模型。

2.4　计算机科学教育相关工作

本节介绍计算机科学教育的相关工作，包括对问题求解能力与计算机科学教育关系的研究，以及现有的支架式教学工具。

2.4.1　问题求解能力与计算机科学教育

许多研究表明，问题求解能力对于学生能够在计算机科学专业具有优秀表现非常重要（Beaubouef et al 2005，Butler et al 2007）。例如，Beaubouef et al（2005）报告称，在许多高等院校中，30%–40% 的计算机科学专业本科生无法完成学业。产生这种高退学率现象的原因有很多，其中常见的一条就是“学生的数学能力和问题求解能力不足”。

另外有一些研究表明，编程工作可以很好地训练学生的计算思维，提升问题求解能力（Buitrago Flórez et al 2017，Salehi et al 2020，Tu et al 1990）。例如，Salehi et al（2020）通过观察发现，在无需额外领域特定知识的情况下，计算机科学专业的学生在求解问题时的表现显著强于其其他专业的学生，同时，编程任务可以有效地帮助学生对特定问题的求解能力进行针对性训练。然而，根据之前的教学经验，如果编程任务过于困难，学生可能会对解决问题失去信心和动力，不但不能起到训练作用，反而成为了一个负面经历，让学生对编程产生恐惧甚至厌恶。

2.4.2　支架式教学工具

支架式（Scaffolding）教学指的是使用辅助工具为学生的学习提供支持和指导的一种教学方法（Hammond 2001）。支架式的教学工具可以临时性地辅助学生完成任务，等到学生有足够的信心和能力之后，再独立完成整项工作。

常用的一类支架式教学工具（例如 C0（Pfenning 2010） 和 Ironclad C++（Delozier et al 2013））将通用的程序语言（如 C，C++）简化为只有安全操作的子语言。学生使用这些简化版本的语言进行编程时，不会遇到初学者难以理解和处理的复杂情况（例如不安全的内存访问），因此学生可以将注意力集中在学习基本编程概念上。另一类支架式教学工具则是通过可视化的编程语言来降低编程难度（Bart et al 2017，Hermans et al 2017，I. Lee et al 2011，Maloney et al 2010，MIT App Inventor 2021，Ruthmann et al 2010）。例如，Scratch（Maloney et al 2010） 是一种基于代码块的面向对象程序语言。学生可以通过拖拽代码块来学习基本的编程技巧和概念。

不过，这些支架式教学工具均不能自动进行算法设计，因此无法将问题理解与程序设计完全分开。

2.5 本章小结

本章综述了程序自动生成技术，约束求解优化技术，自然语言程序设计，计算机科学教育四个领域的相关工作及研究进展。这四类相关工作分别与本文第三至六章的研究内容紧密相关。

本文第三章讨论了问题描述语言的设计与基本求解程序的自动生成方法。与现有的程序自动生成技术相关工作相比，首先，本文设计的问题描述语言为一种约束模型描述语言，因而描述的问题模型具有更好的通用性和完备性，容易描述包括组合优化问题在内的更复杂的问题；其次，本文提出的程序自动生成方法能够自动为组合优化问题设计求解算法并生成相应程序，通用性更强，且无需用户提供算法描述。

本文第四章研究了如何自动设计并实现搜索剪枝和动态规划优化技术。与之前的通用约束求解优化工作相比，这些优化技术基于对问题性质的静态分析，因此可以针对一整类问题而非仅针对单个问题实例。同时，若问题适用动态规划优化，优化效果远远高于传统的基于搜索的优化技术。而相比于其他基于动态规划的优化方法来说，本文提出的动态规划优化方法可以自动判定问题能否适用动态规划，并生成相应优化程序，不需要用户的额外输入。

本文第五章研究了从自然语言产生组合优化约束模型的关键技术。与现有的自然语言程序设计工作不同的是，本文从自然语言问题描述自动产生为组合优化问题的形式化描述。由于组合优化约束模型的自然语言描述中关键的符号、操作符密度大，翻译精确性要求较高，在没有足够训练数据和计算能力的情况下，很难采用基于机器学习的语义分析方法；另一方面，组合优化约束模型的操作符种类有限、模式相对固定，因此本文建立了由 40 道问题组成的自然语言描述数据集，并针对该数据集，设计并实现了基于模板规则和基于框架的翻译方法。

本文第六章尝试将面向组合优化约束模型的程序自动生成的关键技术应用到对问题求解能力的培养和训练中。之前关于问题求解能力与计算机科学教育关系的研究结果证明了问题建模训练在问题求解能力培养中的重要性，以及将问题求解过程分解为问题建模和模型求解，能够降低学习难度并提升学习效果。本文实现了之前相关教学工具所不具备的对于算法设计的自动化，因而成功地将这两个阶段完全分离，在实现对学生问题建模能力独立训练的同时，提升了学生对于编程求解问题全过程的信心和学习效率。

第三章　问题描述语言 *COPDL* 和程序自动生成方法 *COPDL2C*

本文研究的核心目标是为用户给定的组合优化问题自动生成求解程序。为了达成这一核心目标，（1）首先，要为用户提供描述待求解的组合优化问题的形式化语言；（2）其次，为给定的问题生成基本的求解程序框架；（3）最后，进一步优化基本的求解框架以提高求解效率。本章主要研究的是前两个部分。

在本章中，首先给出了一种自定义的组合优化问题描述语言 *COPDL*。*COPDL* 与主流的组合优化问题描述语言 MiniZinc 具有相同的问题描述能力和表述难度，但不同的是，*COPDL* 要求用户必须为每一个参数变量明确指定取值范围。这一特性使得用 *COPDL* 描述的是一类问题——参数变量取值不同对应不同的问题实例，而非一个特定的问题实例。之后，本章给出了以 *COPDL* 描述的问题为输入，自动生成 C 语言求解程序的软件工具 *COPDL2C* 的设计与实现方法。第 3.4 节给出了一些经典问题的 *COPDL* 描述示例，并给出了生成求解程序的正确性和效率分析。本章 *COPDL2C* 的实现为下一章求解程序的进一步优化提供了基础。

3.1　从组合优化约束模型到程序源代码

为了能够求解复杂的组合优化问题，本章尝试为用户提供便利的问题描述语言用于描述约束模型，同时利用程序自动生成技术，得到正确的求解算法，并生成相应的最终 C 程序。

本文设计了一种新的问题描述语言 *COPDL*，作为中间的问题模型描述语言，同时，设计并实现了 *COPDL* 的程序自动生成工具 *COPDL2C*。相比于现有的主流约束模型描述语言，例如 MiniZinc（Nethercote et al 2007），Essence（Frisch et al 2008）等，*COPDL* 具有一个其他语言所不具备的重要特点：**参数变量的取值范围需要由用户明确指定**，这一特点使得：

- **基于 *COPDL* 的工具能够在不知道实际输入参数的情况下，根据参数取值范围，针对该类问题的所有实例而非特定问题实例实现程序自动生成；**
- **通过静态分析，找到问题特性，并充分利用这些性质实现高级程序优化算法，例如搜索剪枝和动态规划。**

现有的约束求解器是在求解过程中动态调整搜索策略，此时求解器已经知道实际输入参数，因此并不需要用户预先给出它们的取值范围；而程序自动生成是通过静态分析

```
#input
    N of int in [1,1000];

#required
        col of (int in [1,N])[1~N];
        alldiff col;
        alldiff [col[i] + i : forall i];
        alldiff [col[i] - i : forall i];

#output
        col;
```

图 3.1　*N* 皇后问题的 *COPDL* 描述

```
int: n;

include "alldifferent.mzn";
array [1..n] of var 1..n: col;
constraint alldifferent(col);
constraint alldifferent([col[i] + i | i in 1..n]);
constraint alldifferent([col[i] - i | i in 1..n]);

solve satisfy;
```

图 3.2　*N* 皇后问题的 MiniZinc 描述

过程完成的，因此无论在生成基础框架还是实现进一步优化上，都需要根据参数的取值范围来选定搜索策略。而本文后续部分的程序自动生成算法，均是基于 *COPDL* 这一重要特性设计和实现的。

同时，*COPDL* 与其他约束模型描述语言在描述形式、基本功能上非常相似，都能够支持用户描述组合优化约束模型的参数变量、决策变量、约束关系等要素，具有相同的表达能力，只是在语法、符号等细节上有所差异。以 *N* 皇后问题为例，图 3.1，3.2，3.3 分别展示了该问题在 *COPDL*，MiniZinc，Essence 三种语言下的约束模型。由于这种相似性的存在，具有使用其他约束模型描述语言建模经验的用户可以很容易的掌握和使用 *COPDL*。另一方面，*COPDL* 要求用户明确给出参数变量的取值范围，例如 *N* 皇后问题中参数变量 *N* 的取值范围，使得基于 *COPDL* 的工具可以针对整个问题类实现程序自动生成。

3.2　程序源代码生成示例

本节通过立方体问题实例来介绍如何针对一个 *COPDL* 描述自动生成出基本的求

```
language ESSENCE' 1.0

given n: int
letting DOM be domain int(1..n)
find col : matrix indexed by [ DOM ] of DOM

such that
    allDiff(col),
    forAll i,j : DOM .
        (i > j) -> (col[i] - i != col[j] - j),
    forAll i,j : DOM .
        (i > j) -> (col[i] + i != col[j] + j)
```

图 3.3　*N* 皇后问题的 Essence 描述

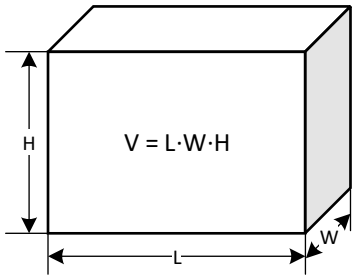

图 3.4　立方体问题

解程序。

3.2.1　问题描述

立方体问题：给定一个整数 V（$\leq 10,000$），找到表面积最小的立方体，满足：体积为 V 且各边长（即图 3.4 中的长 L，宽 W 和高 H）均为整数。

通过分析，可以从问题描述中识别出约束模型的各个要素，并根据题目中省略的常识（例如物体的体积、边长均为正数，立方体的体积、表面积计算公式）补全取值范围、约束关系等：

- 一个输入参数：立方体体积 V，$1 \leq V \leq 10,000$；
- 三个需要被求解的正整数变量：立方体的长 L，宽 W 和高 H，各变量均为正整数；
- 一个约束关系：立方体长、宽、高的乘积应等于其体积；
- 目标函数：最小化立方体的表面积。

本章设计并实现了一种组合优化约束模型描述语言 *COPDL*（Problem Description Language，问题描述语言），支持形式化地描述问题。图 3.5 展示了用 *COPDL* 描述的

```
#input
    V of int in [1,10^4];

#required
    L of int in [1,?];
    W of int in [1,?];
    H of int in [1,?];
    V = L * W * H;

#objective
    minimize (2 * (L * W + L * H + W * H));
```

图 3.5　立方体问题的 *COPDL* 描述

立方体问题模型。在这个 *COPDL* 描述中，包含了三个部分：

- `#input` 部分定义了所有的输入参数（V）及其取值范围。
- `#required` 部分声明了需要被求解的变量，同时列举了变量与输入参数之间的所有约束关系。在本例中，L，W 和 H 均被声明为正整数，其中“`?`”表示未显式声明的边界，此处等价于正无穷（$+\infty$）。表达式 `V = L * W * H` 表示了立方体体积与其长、宽和高的约束关系。
- `#objective` 部分定义了一个目标函数，作为可行解优劣的评价标准。

3.2.2　模型分析和程序生成

针对给定的 *COPDL* 问题描述，*COPDL2C* 将分三步实现模型分析和程序生成：

1. 变量类型推导；
2. 独立变量选取；
3. 程序自动生成。

3.2.2.1　变量类型推导

COPDL2C 首先使用 FBBT 算法 （Belotti et al 2010） 迭代计算出每个变量更紧的取值范围。例如，在第一次迭代时，将原模型中的约束关系转换为 $L = V \div W \div H$，FBBT 算法更新 L 的定义域为：

$$\begin{aligned} D'_L &= D_L \cap ([1, 10^4] \div [1, +\infty] \div [1, +\infty]) \\ &= [1, +\infty] \cap (0, 10^4] \\ &= [1, 10^4] \end{aligned} \tag{3.1}$$

而在第二次迭代时，FBBT 用类似的方法，根据 L 的新定义域更新了 W 的定义域。迭代过程不断重复，直至所有变量的定义域均固定。

3.2.2.2　独立变量选取

首先给出变量依赖的定义 3.1 和独立变量集的定义 3.2。

定义 3.1 (变量依赖)**.** 对于约束关系中的方程，若某个变量的值能够被其他某些变量唯一确定，那么该变量依赖于这些变量组成的一个集合。

定义 3.2 (独立变量集)**.** 独立变量集是变量集 $\mathcal{V}$ 的一个子集，且其中的变量能够唯一确定所有不在独立变量集里的其他变量的值。

在示例问题中，L，W，H 分别依赖于另外两个变量，因为只要给定任意两个变量的值，就能唯一确定第三个变量的值。*COPDL2C* 通过启发式搜索的方式，选取最优的独立变量集，用于产生基于循环或基于递归的枚举程序。

3.2.2.3　程序自动生成

对于求解组合优化问题，一般存在两种算法设计策略 （Lenstra et al 1981）：

- **显式枚举**：显式枚举所有可行解，从中找出目标函数值最优的解。显式枚举一般采用穷尽式或暴力搜索的方式实现。
- **隐式枚举**：所有的可行解都会被考虑到，但不是所有的可行解都会被显式枚举。典型的方法是分支限界 （Marinescu et al 2009），在枚举过程中，自动删除不可能成为最优解的搜索分支。

对应这两种策略，*COPDL2C* 生成两类求解程序：基于循环的程序和基于递归的程序：

定义 3.3. 基于循环的程序: 通过嵌套的 `for` 循环结构，显式枚举所有可行解。

定义 3.4. 基于递归的程序: 通过递归函数调用枚举所有可行解，通过加入 `if` 条件判断甚至动态规划优化实现剪枝操作，减少搜索空间，提高搜索效率。

图 3.6 是对立方体问题产生的基于循环的求解程序。该程序枚举独立变量 H 和 L 的所有可行赋值组合，计算非独立变量 W，检查所有约束是否满足，并计算目标函数值，一旦目标函数值优于当前最佳答案，则更新。最终输出最佳答案。

3.3　组合优化约束模型描述语言设计

本节列出了 *COPDL* 的核心语法，并详细介绍了各主要组成部分。

```c
#include<stdio.h>

int V;
int H;
int L;
int _result;
int W;
int _best__result;
// 获取输入
void _input() {
    scanf("%d", &V);
}
// 输出最优解
void _output() {
    printf("%d\n", _best__result);
}
// 更新最优解
void _update() {
    if (_result >= _best__result)
        return;
    _best__result = _result;
}
// 枚举所有变量赋值组合并检查是否满足约束
void _solve() {
    _best__result = 600000001;
    for (H = 1; H <= 10000; H++) {
        for (L = 1; L <= 10000; L++) {
            if (V % L != 0)
                continue;
            if (V / L % H != 0)
                continue;
            W = V / L / H;
            _result = 2 * (L * W + L * H + W * H);
            _update();
        }
    }
}
// 调用函数获取输入，求解问题，并输出最优解
int main() {
    _input();
    _solve();
    _output();
    return 0;
}
```

图 3.6　对立方体问题生成的基于循环的程序

描述　$DESC \rightarrow IP\ RQ\ OBJ\ OUT$
输入部分　$IP \rightarrow$ **#input** $(VD\ ;)*$
约束部分　$RQ \rightarrow$ **#required** $(Stmt\ ;)*$
目标部分　$OBJ \rightarrow$ **#objective** $((\textbf{minimize} \mid \textbf{maximize})\ Exp\ ;)?$
输出部分　$OUT \rightarrow$ **#output** $(ExpStmt\ ;)*$

图 3.7　*COPDL* 核心语法：主体框架

变量声明	VD	$\rightarrow$	Id **of** $Type$	
类型声明	$Type$	$\rightarrow$	$PrimType \mid CompType \mid FuncType$	
基本类型	$PrimType$	$\rightarrow$	(**int** | **real** | **char** | **bool**) (**in [** B**,** B **]**)?	
复合类型	$CompType$	$\rightarrow$	$Type$ **[** B **~** B **]**	// 列表
		|	$Type$ **{}**	// 集合
		|	**(** $TypeList$ **)**	// 元组或结构
函数类型	$FuncType$	$\rightarrow$	**(** $TypeList$ **) -> ** $Type$ **= (** Exp **)**	
类型列表	$TypeList$	$\rightarrow$	$Type(\textbf{,}\ Type)*$	
取值范围	B	$\rightarrow$	$RValue \mid \textbf{?}$	// **?** 表示未定义
右值	$RValue$	$\rightarrow$	$Num \mid Exp$	

图 3.8　*COPDL* 核心语法：变量声明和类型范围约束

3.3.1　*COPDL* 的核心语法

本章定义了一种组合优化约束模型描述语言 *COPDL*，为用户描述组合优化问题提供遍历。*COPDL* 允许用户分别定义输入参数、变量、约束关系、以及目标函数。图 3.7–3.10 展示了 *COPDL* 的核心语法，使用巴克斯–瑙鲁范式（BNF）定义。

3.3.2　*COPDL* 描述各组成部分

一个 *COPDL* 描述可分为四个主要组成部分：`#input`，`#required`，`#objective`，以及 `#output`。

3.3.2.1　输入部分（`#input`）

这一部分声明了所有（零个或多个）输入参数及其定义域。最终的程序将根据输入部分声明参数的顺序，生成对应的获取输入的语句。在 *COPDL* 中，一个变量可以是一个基本类型（如 `int`）或一个复合类型（如数组或集合）。

语句	$Stmt$	$\rightarrow$	VD	// 变量声明
		|	$ExpStmt$	// 表达式语句
		|	$ForEnum$	// “Forall”约束
表达式语句	$ExpStmt$	$\rightarrow$	Exp	
`forall` 约束	$ForEnum$	$\rightarrow$	Exp **: forall** $Id(\textbf{,}\ Id)*$ **(** $QExp$ **)**	

图 3.9　*COPDL* 核心语法：语句

表达式	*Exp*	→	**(** *Exp* **)**	
		|	*AExp*	// 原子表达式
		|	**not** *Exp*	// 一元表达式
		|	*BExp*	// 二元表达式
		|	*GExp*	// 累积函数表达式
		|	*FuncCall*	// 函数调用表达式
		|	**exists** *Id*(**,** *Id*) ∗ **(** *QExp* **)**	
		|	**if** *Exp* (*Exp*) **else (** *Exp* **)**	
原子表达式	*AExp*	→	*Id* | *Exp* **[** *RValue* **]**	
二元表达式	*BExp*	→	*Exp BinOp Exp*	
累积函数表达式	*GExp*	→	*AggOp* [*ForEnum*]	
函数调用表达式	*FuncCall*	→	*id* **(** *ParamList*? **)**	
参数列表	*ParamList*	→	*Exp*(**,** *Exp*)∗	
量词表达式	*QExp*	→	*EExp*	
		|	**not** *QExp*	
		|	*QExp* (**and**|**or**|**xor**) *EExp*	
枚举表达式	EExp	→	*Id* **in [** *B***,** *B* **]**	// 枚举范围内数值
		|	*Id* **in** *Id*	// 枚举集合元素
		|	**(** *Elem*(**,** *Elem*) ∗ **) in** *Id*	// 枚举元组
元素	*Elem*	→	*Id* | **?**	
二元运算符	*BinOp*	→	**+** | **-** | ***** | **/** | . . .	
累积函数	*AggOp*	→	**summation** | **product**	
		|	**min** | **max**	
		|	**count** | **alldiff**	

图 3.10　*COPDL* 核心语法：表达式

表 3.1　*COPDL2C* 的内置运算符和累积函数表

类别	符号
关系运算符	=, !=, >, <, >=, <=
逻辑运算符	and, or, not, xor
算术运算符	+, -, *, /, mod
指数运算符	^
累积函数	min, max, summation, product, count, alldiff

3.3.2.2　约束部分（#required）

这一部分声明了所有变量及其定义域，以及变量和输入参数之间的约束关系。约束部分包含零个或多个语句，每个语句可以是变量声明、约束关系、或者是 forall 约束。其中，forall 约束是一个高阶函数，它将指定的函数同时作用在某个复合类型结构里的每一个元素上。

COPDL 支持多种表达式形式，包括：

- 括号表达式；
- 原子表达式（如变量 A）；
- 一元表达式（如 not A）；
- 二元表达式（如 A+B）；
- forall 量词得到的累积表达式（如 summation [A[i] : forall i in [1, 10]]）；
- 函数调用（foo()）；
- exists 量词对应的表达式（如 exists a (a in [1, 10])）；
- 条件表达式（如 if a (b=1) (else (b=0))）。

同时，*COPDL* 支持许多内置运算符和累积函数（见表 3.1）。

关于 *COPDL* 的更多说明和样例，可以参考附录 A。

3.3.2.3　目标部分（#objective）

这一部分声明了零个或一个目标函数。若没有目标函数，则该问题是约束满足问题，最终自动生成的程序将在找到第一个可行解时立即停止并返回；否则，程序将搜索使得目标函数值最优的解。

3.3.2.4　输出部分（#output）

这一部分声明了零个或多个表达式，作为额外的信息输出。

```
#input
        A of int in [1,10^6];
        B of int in [1,10^6];

#required
        G of int in [0,?];
        X of int;
        Y of int;
        A = G * X;
        B = G * Y;

#objective
        maximize G;
```

图 3.11　最大公约数问题的 *COPDL* 描述

3.4　经典问题的 *COPDL* 描述示例

在 3.1 节和 3.2 节中已分别展示了两个经典问题的 *COPDL* 描述示例，包括 *N* 皇后问题（图 3.1）以及立方体问题（图 3.5），此外，在后续的 3.5 节中还将展示另一个经典规划问题——0/1 背包问题的描述示例（图 3.18）。本节额外给出了两个经典问题的 *COPDL* 描述示例，分别是数论中的**最大公约数问题**，以及图论中的**最短路径问题**。

3.4.1　最大公约数问题

最大公约数问题的自然语言描述：给定两个正整数 *A* 和 *B*，求最大的正整数 *G*，满足 *G* 同时是 *A* 和 *B* 的约数。图 3.11 给出了该问题的 *COPDL* 描述。

3.4.2　最短路径问题

最短路径问题的自然语言描述：图中包含 *n* 个结点，用邻接矩阵 *d* 的形式给出结点之间的边长，求从结点 1 到结点 *n* 的一条最短路径 *p*。图 3.12 给出了该问题的 *COPDL* 描述。

3.5　从组合优化约束模型自动产生求解程序源代码的算法

图 3.13 展示了 *COPDL2C* 将 *COPDL* 描述转化为 C 程序的处理过程。整个过程主要分为三步，分别是变量类型推导（3.5.1 节），独立变量选取（3.5.2 节）以及程序自动生成（3.5.3 节）。

```
#input
        n of int in [1,100];
        d of (int in [0,100])[1~n][1~n];

#required
        l of int in [1,n];
        p of (int in [1,n])[1~l];
        p[1] = 1;
        p[l] = n;

#objective
        minimize summation [d[p[i]][p[i+1]] : forall i];
```

图 3.12　最短路径问题的 *COPDL* 描述

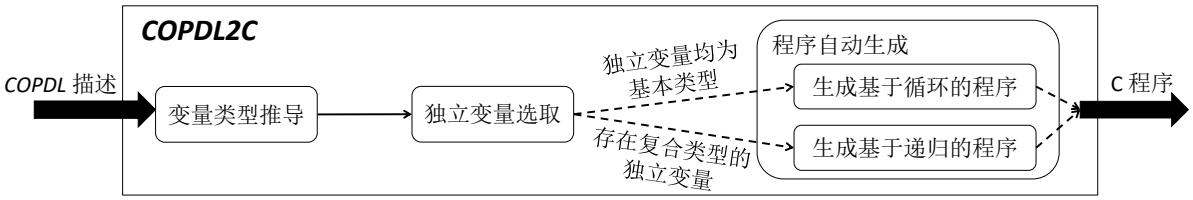

图 3.13　*COPDL2C* 处理步骤

3.5.1　变量类型推导

给定一个 *COPDL* 描述，*COPDL2C* 首先通过词法分析和语法分析创建**标识符表**。具体来说，*COPDL2C* 基于 JavaCC （Kodaganallur 2004） 和 Java Tree Builder (JTB)（UCLA Compilers Group 2005） 构建了 *COPDL* 的词法和语法分析器，通过遍历抽象语法树，提取所有标识符及其类型，产生标识符表。

类似于传统编译器的符号表，*COPDL2C* 的标识符表记录了每个参数和变量的名字以及类型。不同的是，标识符表额外记录了（1）每个标识符的类型是输入参数还是变量；（2）每个标识符的取值范围，这些取值范围将被 FBBT 算法用于计算每个变量更紧的上下界。表 3.2 是立方体问题的标识符表。

表 3.2　立方体问题的标识符表

标识符	参数 / 变量	类型 & 范围
V	参数	int in $[1,10^4]$
L	变量	int in $[1,10^4]$
W	变量	int in $[1,10^4]$
H	变量	int in $[1,10^4]$

```
#required
    A of int in [1, 5];
    B of int in [1, 10];
    C of int in [1, 500];
    D of int in [1, 250000];
    C = A * B ^ 2;
    D = B ^ 2 * C ^ 2;
```

图 3.14　一个特殊的 #required 部分示例

3.5.2　独立变量选取

有了标识符表之后，本步骤的目标是从变量集中选取最小的独立变量集 $\mathcal{I}_{opt}$。一个独立变量集是最小的，当且仅当集合内所有变量取值范围的笛卡尔积最小。$\mathcal{I}_{opt}$ 实际上决定了目标程序的搜索空间大小，因此，独立变量集越小，生成程序的运行效率一般就越高。

为了找到 $\mathcal{I}_{opt}$，首先需要识别出所有的变量依赖关系（3.5.2.1 节），然后采用启发式搜索的方式，从 $\mathcal{V}$ 的所有子集中找到最小的独立变量集（3.5.2.2 节）。

3.5.2.1　变量依赖识别

本节使用一个具体的例子说明整个处理过程。

图 3.14 定义了 4 个变量（A，B，C，D）和两个约束关系。根据第一个约束，C 对集合 {A, B} 具有依赖，因为一旦这两个变量值确定，就能唯一确定 C 的值。这个依赖关系可以形式化地表示为：

$$C \lhd P_A \wedge P_B \tag{3.2}$$

类似的，通过将第一个约束变形为 $A = C/B^2$，可以得到 A 的一个依赖关系

$$A \lhd P_B \wedge P_C \tag{3.3}$$

不过，B 并不依赖集合 {A, C}，因为即使给定这两个变量的取值，也无法通过变形后的方程 $B = \pm\sqrt{C/A}$ 唯一确定 B 的值。

对于第二个约束，还可以得到变量 D 的一个依赖关系：

$$D \lhd P_B \wedge P_C \tag{3.4}$$

3.5.2.2　变量子集枚举

COPDL2C 将所有变量按照取值范围从小到大排序，然后采用启发式回溯搜索的方式找到最小的独立变量集，算法伪代码如图 3.15。

输入: 依赖关系集 DEP，变量集 $\mathcal{V}$
输出: 最小独立变量集 $\mathcal{I}_{opt}$

```
1  Function MAIN(DEP, V):
2      I_opt ← V;
3      GetIndependentVar (DEP, ∅, V);
4      return I_opt;
5  Function GetIndependentVar (DEP, I, V'):
       /* 若不存在未定值变量，则 I 为更小的独立变量集                    */
6      U ← undetermined(I, DEP);
7      if U = ∅ then
8          I_opt ← I;
9          return;
10     end
       /* 若待选变量集 V' 为空，该分支无可行解，剪枝                     */
11     if V' = ∅ then
12         return;
13     end
       /* 若添加范围最小的变量 v 都会导致 I 不小于 I_opt，剪枝           */
14     v ← V' 中取值范围最小的变量;
15     if |I ∪ {v}| ≥ |I_opt| then
16         return;
17     end
       /* 尝试将 v 加入 I                                               */
18     GetIndependentVar (DEP, I ∪ {v}, V' ∩ undermined(I ∪ {v}, DEP));
       /* 尝试不将 v 加入 I                                             */
19     GetIndependentVar (DEP, I, V' − {v});
```

图 3.15　寻找最小独立变量集的算法伪代码

以图 3.14 定义的变量和约束关系为例。按照取值范围从小到大排序后的变量列表为 `[A, B, C, D]`。*COPDL2C* 首先尝试将 `A` 放入候选集合 $\mathcal{I}_1$。由于 `A` 自身无法唯一确定其他任何变量的值，*COPDL2C* 接着将 `B` 也放入候选集合。根据依赖关系式 (3.2) 和 (3.4)，变量 `C` 和 `D` 的值均能够被唯一确定。至此，所有不在候选集合中的变量也能被唯一确定，因此 $\mathcal{I}_1 = \{A, B\}$ 成为当前的最优独立变量集。

在后续的探索中，`A` 被移出候选集合而 `B` 被加入候选集合。此时没有任何其他变量能被唯一确定。之后无论在候选集合中加入任何变量，候选集合都会大于 $\mathcal{I}_1$，因此这一分支被提前剪掉。继续重复上述搜索过程，最终，对于该示例来说，$\mathcal{I}_{opt}$=$\mathcal{I}_1$ = $\{A, B\}$。

3.5.3 程序代码生成

根据 $\mathcal{I}_{opt}$ 中独立变量的类型情况，*COPDL2C* 生成基于循环的程序（3.5.3.1 节）或基于递归的程序（3.5.3.2 节）。

输入: 参数变量 $\mathcal{A}$，约束集 $\mathcal{R}$，目标函数 O
输出: 找到的目标函数最优值 *_best__result*

```
1  Function IterationBasedSearch (A, R, O):
       /* 1. 将 _best__result 初始化为一个不合法的值                    */
2      _best__result = ∅;
       /* 2. 多重循环枚举最小独立变量集 I_opt = {iv_1, ..., iv_p} 中各变量取值
          */
3      foreach iv_1 = vaule | value ∈ iv_1 的定义域 do
4          ...
5          foreach iv_p = value | value ∈ iv_p 的定义域 do
               /* 3. 计算非独立变量的值                                 */
6              computeValues(V − I_opt, iv_1, ..., iv_p);
               /* 4. 检查是否所有约束均被满足                           */
7              if ∀r ∈ R, r 为真 then
                   /* 5. 计算目标函数值                                 */
8                  _result ← O(A, V);
                   /* 6. 如果找到了更好的解，更新 _best__result          */
9                  if _best__result = ∅ or _result 优于 _best__result then
10                     _best__result ← _result;
11                 end
12             end
13         end
14     end
```

图 3.16 基于循环的程序的实现逻辑

3.5.3.1 生成基于循环的枚举算法程序

若独立变量集中的所有独立变量均为基本类型，*COPDL2C* 将每个独立变量当作循环变量，然后采用嵌套的 `for` 循环枚举每个赋值组合，计算所有非独立变量值，检查所有约束是否都能满足，计算目标函数值并更新最优解，从而实现基于循环的程序。由于 $\mathcal{I}_{opt}$ 理论上对应的搜索空间最小，因此生成的程序在没有剪枝的显式枚举程序中效率是最高的。

最终生成的基于循环的程序将包含如下 5 个部分：

- `_input(...)`：从命令行读取 `#input` 部分定义的输入参数值；
- `_output(...)`：打印最优解以及 `#output` 部分的表达式；

- `_update(...)`：比较当前解与之前最优解的好坏，若当前解更优，则更新最优解`_best__result`；
- `_solve(...)`：枚举独立变量的所有赋值组合，计算所有非独立变量值，检查所有约束是否满足，计算目标函数值等主体工作，其实现逻辑参见图 3.16；
- `main()`：将上述函数整合到一个完整的程序中。

3.5.3.2　生成基于递归的程序

动机：若一个独立变量 iv 是复合类型的（数组或集合），假设其包含 N 个元素。如果强行生成基于循环的程序，需要定义 N 个循环变量对应 iv 的每个元素，并使用 N 重嵌套的 `for` 循环来实现显式枚举。这种形式不仅会导致求解程序代码过长，还会导致编译过慢、最终的可执行文件体积过大等问题，显然是不好的。因此，当某个独立变量 iv 是复合类型的时候，*COPDL2C* 生成基于递归的程序，使用递归函数调用来实现隐式枚举。图 3.17 展示了基于递归搜索的算法伪代码。

输入: 参数变量 $\mathcal{A}$，约束集 $\mathcal{R}$，目标函数 O
输出: 找到的目标函数最优值 *_best__result*

Function *MAIN*$(\mathcal{A}, \mathcal{R}, O)$**:**
    /* 1.　将 *_best__result* 初始化为一个不合法的值　*/
    _best__result $= \emptyset$;
    RecursionBasedSearch $(\mathcal{A}, \mathcal{R}, O, 1)$ **return** *_best__result*;
/* 2.　递归枚举最小独立变量集 $I_{opt} = \{iv_1, \ldots, iv_p\}$ 中各变量取值　*/
Function RecursionBasedSearch $(\mathcal{A}, \mathcal{R}, O, step)$**:**
    /* 完成对所有独立变量的赋值　*/
    if $step > p$ **then**
        /* 3.　计算非独立变量的值　*/
        $computeValues(V - I_{opt}, iv_1, \ldots, iv_p)$;
        /* 4.　检查是否所有约束均被满足　*/
        if $\forall r \in \mathcal{R}$，$r$ 为真 **then**
            /* 5.　计算目标函数值　*/
            _result $\leftarrow O(\mathcal{A}, \mathcal{V})$;
            /* 6.　如果找到了更好的解，更新 *_best__result*　*/
            if *_best__result* $= \emptyset$ *or _result* 优于 *_best__result* **then**
                _best__result $\leftarrow$ *_result*;
            end
        end
        return;
    end
    foreach $iv_{step} = vaule \mid value \in iv_{step}$ 的定义域 **do**
        RecursionBasedSearch $(\mathcal{A}, \mathcal{R}, O, step + 1)$;
    end

图 3.17　基于递归的程序的实现逻辑

自然语言描述

将 N 个物品放入容量为 C 的背包，第 i 个物品（$i \in N$）的重量为 W_i，价值为 V_i。

将物品的子集 $sel \subseteq [1,\ldots,N]$ 放入背包，要求背包内物品的总重量不超过 C。

最大化背包内物品总价值。

***COPDL* 描述**

```
#input
    N of int in [1,100];
    C of int in [1,1000];
    W of (int in [1,1000])[1~N];
    V of (int in [1,1000])[1~N];

#required
    sel of (int in [1,N]){};
    summation
        [W[i] : forall i (i in sel)]
            <= C;

#objective
    maximize summation
        [V[i] : forall i (i in sel)];
```

图 3.18　0/1 背包问题的自然语言描述与 *COPDL* 描述

自动生成方法：与基于循环的程序相比，基于递归的程序只是在 `_solve(...)` 的实现上有所不同。具体来说，对于每个具有复合数据类型的独立变量 `VAR`，*COPDL2C* 定义了辅助函数 `_find_VAR(...)` 来枚举 `VAR` 里所有元素的赋值组合。具体来说，

- 若 `VAR` 是一个数组类型的变量，其下标范围为 $[1..N]$。`_solve(...)` 以“1”为参数调用 `_find_VAR(...)`，代表下一步需要为 `VAR[1]` 赋值。`_find_VAR(...)` 在获得一个参数 i 后，首先检查 i 是否超过下标的上界 N；若没有超过，使用单个 `for` 循环枚举 `VAR[i]` 的取值，并以 $(i+1)$ 为参数递归调用自身，表示需要为下一个元素赋值。不断递归直至参数 i 超过 N，此时就得到了 `VAR` 里所有元素的一个赋值组合。
- 若 `VAR` 是一个集合类型的变量，其全集的元素个数为 N。*COPDL2C* 简单地将其视为长度为 N 的布尔类型数组，数组中第 i 个元素为的值决定了全集中的第 i 个元素是否属于该集合。之后，就可以按照数组类型变量的处理方法枚举这个布尔类型数组的赋值组合。

对于图 3.18 给出的 0/1 背包问题，最小独立变量集为 {`sel`}，其中 `sel` 是一个集合类型的变量。*COPDL2C* 将 `sel` 转化为布尔数组变量 `sel2`，然后生成图 3.19 中展示的 `_find_sel(...)` 和 `_solve(...)` 两个函数，实现对 `sel2` 所有赋值情况的枚举。

3.6　*COPDL* 及 *COPDL2C* 的有效性验证实验设计与结果

本节通过实验评估 *COPDL* 及 *COPDL2C* 的有效性，即 *COPDL* 和 *COPDL2C* 能否

```
// _find_sel(...) 递归调用自身，枚举第 i 个元素是否要加入子集
void _find_sel(int _step) {
    if (_step > N) {
        _result = total value of selected items;
        if (!(total weight of selected items <= C))
            return;
        _update();
        return;
    }
    for (sel2[_step] = 0; sel2[_step] <= 1; sel2[_step]++)
        _find_sel(_step + 1);
}

// _solve() 调用 _find_sel(...) 去探索第一个物品是否加入子集
void _solve() {
    _best__result = -1;
    _find_sel(1);
}
```

图 3.19　0/1 背包问题对应的基于递归的程序

广泛应用于多种问题的建模与求解。

3.6.1　数据集

实验使用的数据集包含 49 个问题，由大一、大二计算机专业程序设计和算法分析类基础课程在 2018–2019 学年及 2019–2020 学年的所有期末考试题组成。这些课程包括计算概论 A（2018 年秋季学期，2019 年秋季学期），程序设计实习（2019 年春季学期，2020 年春季学期），以及数据结构与算法 A（2018 年秋季学期，2019 年秋季学期）。数据集具体信息及详细实验结果参见附录 B。

3.6.2　实验设计及实验结果

本文作者对数据集中的每个问题分别编写最接近题意的 *COPDL* 描述，然后使用 *COPDL2C* 生成相应的 C 语言程序，并提交到评测系统获得评测结果。

如表 3.3 所示，在全部 49 个问题中。*COPDL2C* 能够在 22 个（44.9%）问题上生成能够完全通过测试的正确程序，在计算概论课程的期末考试数据集上表现甚至略优于学生的平均水平，而在程序设计实习课程的期末考试上也能达到及格的水平。同时，*COPDL2C* 在 20 个（40.9%）问题上生成了正确但不够高效的求解程序，对于部分的测试数据能够在规定时间限制内给出正确的解。这 20 个问题都是比较具有挑战性的问题，高效求解需要在代码中实现特定的优化技术（例如，搜索剪枝、动态规划、贪

表 3.3 *COPDL* 及 *COPDL2C* 在数据集中 49 个问题上的有效性评估

课程及年份	总计	无法表达	部分可解	完全可解	学生平均
计算概论 2018 秋	8	1 (12.5%)	1 (12.5%)	6 (75.0%)	5.8 (72.5%)
计算概论 2019 秋	9	0 (0%)	2 (22.2%)	7 (77.8%)	6.3 (70.0%)
程序设计实习 2019 春	12	1 (8.3%)	7 (58.3%)	4 (33.3%)	4.5 (37.5%)
程序设计实习 2020 春	8	0 (0%)	5 (62.5%)	3 (37.5%)	4.8 (60.0%)
数据结构与算法 2018 秋	6	3 (50.0%)	3 (50.0%)	0 (0%)	2.4 (40.0%)
数据结构与算法 2019 秋	6	2 (33.3%)	4 (66.7%)	0 (0%)	3.2 (53.3%)
总计	49	7 (14.3%)	22 (44.9%)	20 (40.9%)	27.0 (55.1%)

心、分治等）。此外，有 7 个问题无法使用 *COPDL* 描述，原因是这些问题里包含复杂格式输出操作（如输出直方图）、对字符串的复杂操作（如字符串匹配），以及过程控制（模拟）类的操作，而 *COPDL* 暂时没有提供描述这类操作的支持；或者问题为数据结构应用题（例如使用堆栈计算中缀表达式的值）、程序填空等非组合优化问题，不在 *COPDL* 的问题域中。

结论 3.1：*COPDL* 具有一定的适用性，能够描述多种不同的问题。同时，*COPDL2C* 是可靠的，对于正确的问题描述，*COPDL2C* 总能生成正确的程序，只是在效率上可能不如人工设计的程序。

3.7 本章小结

本章主要介绍了 *COPDL* 问题描述语言的设计以及基于 *COPDL* 的基本程序自动生成算法 *COPDL2C* 的设计和实现。首先以立方体问题为例，展示了 *COPDL* 描述的形式和 *COPDL2C* 的程序自动生成流程，其次，介绍了 *COPDL* 的具体设计，包括核心语法和各部分细节，并展示了一些经典问题的 *COPDL* 描述示例，最后，详细介绍程序自动生成算法 *COPDL2C* 生成枚举算法程序和递归算法程序的方法。

本章的主要创新点包括：

1. 设计了适合用于程序自动生成的组合优化问题描述语言 *COPDL*，通过对参数变量取值范围的严格限定，使得后续算法和工具可以针对一类问题的静态特性完成程序自动生成工作，而非仅针对某个特定的问题实例；
2. 提出了基于 *COPDL* 的基础程序自动生成算法，能够根据用户提供的 *COPDL* 问题描述，通过启发式搜索和预设规则等，产生循环和递归算法程序，为下一节引入基于问题特性的高级优化算法奠定基础。

第四章　程序自动生成方法 *COPDL2C* 中的搜索剪枝和动态规划优化

上一章的 *COPDL2C* 能够对组合优化问题模型自动生成正确的基础求解程序。然而，相比于人工编写的程序以及现有的求解器，这些程序还不够高效。因为在使用枚举求解时，相同子问题会被重复计算，严重影响求解效率。本章在原始 *COPDL2C* 工具的基础上，提出了一种自动判定搜索剪枝和动态规划等优化性质的存在性的方法，并根据判定结果在所生成的求解程序中自动加入相应的优化语句，大幅提高了自动生成的求解程序的效率。本章首先分析了自动实现搜索剪枝和动态规划的难点所在。随后各节详细给出了自动实现搜索剪枝和动态规划优化的具体方法。最后，通过实验验证了优化算法的有效性。

4.1　组合优化问题求解的搜索剪枝和动态规划优化

求解组合优化问题时，一个通用的约束求解器往往采用搜索的办法去找到满足所有约束条件且是目标函数最优化的一组变量赋值。这种搜索不可避免地会对搜索树中不同分支下的等价子问题进行多次重复计算，导致在最坏情况下，搜索的复杂度高达 $O(M^N)$，其中 N 是待赋值变量的个数，M 是搜索树中每个结点的平均分支数。

约束求解器的研究者们已经提出了许多方法来尽可能避免或减少重复搜索，但经典的优化方法，均是已经给定问题实例的前提下，在求解的过程中动态应用的。而这些技术并不能迁移到针对一类问题所有实例求解的程序自动生成方法上。*COPDL2C* 通过采用基于静态分析的搜索剪枝优化来提高搜索效率。搜索剪枝优化是根据当前所处状态，自动估计该分支下是否存在目标解，并在无目标解时及时停止对无效分支的探索，从而减少搜索结点，提高搜索效率。搜索剪枝优化大体可以分为可行性剪枝和最优性剪枝两类，分别针对当前分支下无可行解、或无更优解的情况实现剪枝操作。在静态分析中自动实现搜索剪枝，存在两个主要挑战：

1. 如何从问题模型中提炼出可行性条件，作为可行性剪枝的判断标准；
2. 如何从问题模型中提炼出最优性条件，作为最优性剪枝的判断标准。

动态规划是另一种求解组合优化问题的经典方法。给定一个问题，动态规划将其分解为更简单的子问题，依次求解每个子问题至多一次，并将子问题的解以表格形式存储。当再次遇到已经解决过的子问题时，不再重复求解，而是直接从表格中获取解。动态规划作为一种有一定适用条件的方法，虽然不像搜索那样能够普遍适用于对任意

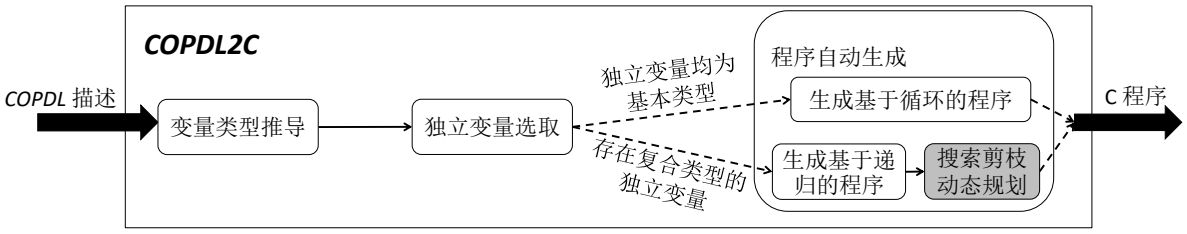

图 4.1　添加高级优化技术后的 *COPDL2C* 框架

组合优化问题的约束求解，但对于适用的问题来说，动态规划的求解效率要远高于搜索的求解效率。然而，现有的通用求解器均没有直接将动态规划应用于约束求解，因为这样做存在三个主要挑战：

1. 如何自动判断动态规划适用于给定的组合优化问题；
2. 针对适用问题，如何设计和实现动态规划求解；
3. 若对于同一个问题存在多种动态规划求解方法，如何自动比较它们的优劣并选择最优方法。

本章重点研究如何在 *COPDL2C* 的基础上，将搜索剪枝和动态规划这两大类高级优化技术自动应用于程序优化，从而大幅提高求解效率。图 4.1 展示了添加了高级优化技术（灰色部分）后的 *COPDL2C* 整体框架。

4.2　搜索剪枝优化

搜索剪枝优化是一种广泛应用于搜索算法的优化策略，通过在搜索过程中加入 `if` 判断，提前删去一定不可能得到可行解或更优解的分支，从而减少搜索空间，提高搜索效率。*COPDL2C* 能够对程序自动应用两类经典剪枝方法：

- **可行性剪枝**：若当前某个约束条件已经不可能被满足了，那么无论剩余未赋值独立变量如何赋值，也不可能找到可行解，因此可以将这一分支提前删去。
- **最优性剪枝**：若无论剩余未赋值独立变量如何赋值，都不可能使目标函数值比当前最优解更优，即在这一分支下不可能得到更优解，则将此分支提前删去。

为了实现这两类剪枝的自动生成，本章提出了**子约束表达式值域受限**和**目标函数值域受限**两个性质，分别用于判定给定问题模型能否应用可行性剪枝和最优性剪枝，并在静态分析过程中实现对输入问题模型的这两个性质的自动分析，使得 *COPDL2C* 可以利用搜索剪枝技术提高求解程序的求解效率。

4.2.1　搜索剪枝在 0/1 背包问题上的应用示例

COPDL2C 尝试从约束中自动发现可行性剪枝与最优性剪枝，并将剪枝以 `if` 语句的形式插入 `_find_VAR(...)` 函数，在搜索的过程中提前删去无效的分支。

```
void _find_sel_bp(int _step, int _sum1, int _sum2) {
    if (_step > N) {
        _result = _sum2;
        if (!(_sum1 <= C))
            return;
        _update();
        return;
    }
    int __sum1 = _sum1;
    int __sum2 = _sum2;
    for (sel2[_step] = 0; sel2[_step] <= 1; sel2[_step]++) {
        _sum1 = __sum1;
        _sum2 = __sum2;
        if (sel2[_step])
            _sum1 += W[_step];
        if (sel2[_step])
            _sum2 += V[_step];
// 可行性剪枝：若当前背包内物品总重量已超过 C，剪枝
        if (!(_sum1 <= C))
            continue;
// 最优性剪枝：若当前背包内物品总价值加上潜在最大价值仍不超过最优解，剪枝
        if (!(_sum2 + 1000 * (N - _step) > _best__result))
            continue;
        _find_sel_bp(_step + 1, _sum1, _sum2);
    }
}
```

图 4.2　应用了剪枝优化的 `_find_sel_bp(...)`

图 4.2 展示了应用剪枝优化的 0/1 背包问题（问题描述见图 3.18）的递归函数。在这个优化后的函数中，分别加入了一个可行性剪枝和一个最优性剪枝。其中，可行性剪枝为检查当前背包内重量和是否在背包的容量限制 C 之内；若这条约束不成立，即当前重量和已经超过 C，那么无论之后如何取舍物品，最终的重量和不会减少，因此这个分支下不存在可行解，可以直接跳过。类似的，加入的最优性剪枝预估了当前背包内物品的价值和加上未来的最大可能收益是否能够超过当前最优解；如果不能，那么这个分支下不存在更优解，同样可以直接跳过。

4.2.2 可行性剪枝优化

COPDL2C 自动分析问题模型的性质——子约束表达式值域受限性质 4.1，并利用该性质确定能否针对该问题进行可行性剪枝。

性质 4.1 (子约束表达式值域受限性质)**.** 约束中包含独立变量的子约束表达式的值域受限于参数变量值和其他独立变量的值域。

COPDL2C 先将约束拆成两个子约束表达式，然后分别利用变量的取值范围估计这两个子约束表达式的值域，最后用一个子约束表达式的值域范围去限制另一个子约束表达式的值域范围，并生成相应的可行性剪枝判定条件。

4.2.2.1 约束拆解

COPDL2C 首先将一个约束拆成如式 (4.1) 所示形式，

$$P(\mathcal{I}_1)\,\langle cmp\rangle\,Q(\mathcal{I}_2) \tag{4.1}$$

其中，P 和 Q 分别是原约束的子约束表达式，$\langle cmp\rangle$ 是一个比较符（大于 $>$、小于 $<$、大等于 $\geq$、小等于 $\leq$ 和等于 $=$），$\mathcal{I}_1$ 和 $\mathcal{I}_2$ 是两个互斥的独立变量集合，即满足$\mathcal{I}_1 \subseteq \mathcal{I}_{opt}, \mathcal{I}_2 \subseteq \mathcal{I}_{opt}, \mathcal{I}_1 \cap \mathcal{I}_2 = \emptyset$。

以 0/1 背包问题为例，原模型中的约束为式 (4.2)：

$$\sum_{i \in sel} W_i \leq C \tag{4.2}$$

在 *COPDL2C* 中，由于集合被转化为 0-1 数组，因此约束被表示为式 (4.3)：

$$\sum_{i=1}^{N} W_i \cdot sel_i \leq C \tag{4.3}$$

在不同的 k 处对该约束作拆分，可以得到不同子约束表达式之间的关系式 (4.4)：

$$P(sel_1, \ldots, sel_k) = \sum_{i=1}^{k} W_i \cdot sel_i \leq C - \sum_{i=k+1}^{N} W_i \cdot sel_i = Q(sel_{k+1}, \ldots, sel_N) \tag{4.4}$$

4.2.2.2　值域估计

COPDL2C 根据独立变量和参数变量的取值范围，结合初等函数的单调性、极值等，可以对子约束表达式 P 和 Q 的值域做一个粗略的估计。

在 0/1 背包问题的例子中，由于 $0 < W_i \leq maxW$，$sel_i \in [0,1]$，可以得到式 (4.5) 和 (4.6)：

$$P(sel_1,\ldots,sel_k) = \sum_{i=1}^{k} W_i \cdot sel_i \in [0, maxW \cdot k] \tag{4.5}$$

$$Q(sel_{k+1},\ldots,sel_N) = C - \sum_{i=k+1}^{N} W_i \cdot sel_i \in [C - maxW \cdot (N-k), C] \tag{4.6}$$

4.2.2.3　生成可行性剪枝判定条件

COPDL2C 根据比较符 $\langle cmp \rangle$，尝试用 Q 的值域去为 P 设置可行性约束。具体来说，有式 (4.7)–(4.11)：

$$P > Q \quad \Rightarrow \quad P > LB(Q) \tag{4.7}$$

$$P < Q \quad \Rightarrow \quad P < UB(Q) \tag{4.8}$$

$$P \geq Q \quad \Rightarrow \quad P \geq LB(Q) \tag{4.9}$$

$$P \leq Q \quad \Rightarrow \quad P \leq UB(Q) \tag{4.10}$$

$$P = Q \quad \Rightarrow \quad P \geq LB(Q) \wedge P \leq UB(Q) \tag{4.11}$$

其中 LB 和 UB 分别表示相应子约束表达式的下界和上界。

如果某子约束表达式 P 的值域因此受到限制，就可以产生相应的判断条件，用于进行可行性剪枝。

对于 0/1 背包问题来说，$P \leq Q$，根据式 (4.10) 得到新的约束式 (4.12)：

$$P \leq UB(Q) = C \tag{4.12}$$

这使得 P 的上界变得更紧了。因此，可以将式 (4.13)作为该问题的求解程序的可行性判定条件式，用于可行性剪枝优化。

$$P = \sum_{i=1}^{k} W_i \leq C \tag{4.13}$$

4.2.3　最优性剪枝优化

COPDL2C 自动分析问题模型的性质——目标函数值域受限性质 4.2，并利用该性质确定能否针对该问题进行最优性剪枝。

性质 4.2 (目标函数值域受限性质). 目标函数的值域受限于包含独立变量的子目标函数值以及参数变量值和其他独立变量的值域。

COPDL2C 先将约束拆成两个子目标函数，然后利用变量的取值范围估计其中一个子目标函数的值域，最后用这个被估计的子目标函数函数的值域范围以及另一个子目标函数的确定值去限制整个目标函数的值域范围，并生成相应的最优性剪枝判定条件。

4.2.3.1 约束拆解

COPDL2C 首先将目标函数拆成如式 (4.14) 所示形式：

$$minimize \mid maximize\ P(\mathcal{I}_1)\ \langle binop \rangle\ Q(\mathcal{I}_2) \tag{4.14}$$

其中，P 和 Q 分别是原约束的子函数，$\langle binop \rangle$ 是一个二元运算符（算术运算符、逻辑运算符、指数运算符），$\mathcal{I}_1$ 和 $\mathcal{I}_2$ 是两个互斥的独立变量集合，即满足$\mathcal{I}_1 \subseteq \mathcal{I}_{opt}, \mathcal{I}_2 \subseteq \mathcal{I}_{opt}, \mathcal{I}_1 \cap \mathcal{I}_2 = \emptyset$。

以 0/1 背包问题为例，原模型中的约束为式 (4.15)：

$$maximize \sum_{i \in sel} V_i \tag{4.15}$$

在 *COPDL2C* 中，由于集合被转化为 0-1 数组，因此约束被表示为式 (4.16)：

$$maximize \sum_{i=1}^{N} V_i \cdot sel_i \tag{4.16}$$

在不同的 k 处对该约束作拆分，可以得到子目标函数式 (4.17)–(4.19)：

$$P(sel_1, \ldots, sel_k) = \sum_{i=1}^{k} V_i \cdot sel_i \tag{4.17}$$

$$Q(sel_{k+1}, \ldots, sel_N) = \sum_{i=k+1}^{N} V_i \cdot sel_i \tag{4.18}$$

$$O = P + Q \tag{4.19}$$

4.2.3.2 值域估计

COPDL2C 根据独立变量和参数变量的取值范围，结合初等函数的单调性、极值等，可以对子目标函数 Q 的值域做一个粗略的估计。

在 0/1 背包问题的例子中，由于 $0 < V_i \leq maxV$，$sel_i \in [0, 1]$，可以得到式 (4.20)：

$$Q(sel_{k+1}, \ldots, sel_N) = \sum_{i=k+1}^{N} V_i \cdot sel_i \in [0, maxV \cdot (N - k)] \tag{4.20}$$

4.2.3.3　生成可行性剪枝判定条件

COPDL2C 根据二元运算符 $\langle binop \rangle$ 的单调性，尝试用 P 的值和 Q 的值域去估计当前搜索分支下目标函数 O 的值域。以算术四则运算为例，有式 (4.21)–(4.24)：

$$O = P + Q \Rightarrow LB(O) = P + LB(Q) \wedge UB(O) = P + UB(Q) \tag{4.21}$$

$$O = P - Q \Rightarrow LB(O) = P - UB(Q) \wedge UB(O) = P - LB(Q) \tag{4.22}$$

$$O = P \cdot Q \Rightarrow \begin{aligned} &LB(O) = \min\{P \cdot LB(Q), P \cdot UB(Q)\} \\ &\wedge UB(O) = \max\{P \cdot LB(Q), P \cdot UB(Q)\} \end{aligned} \tag{4.23}$$

$$O = P/Q \Rightarrow \begin{aligned} &LB(O) = \begin{cases} \min\{P/LB(Q), P/UB(Q)\} & 0 \notin (LB(Q), UB(Q)) \\ -\infty & 0 \in (LB(Q), UB(Q)) \end{cases} \\ &\wedge UB(O) = \begin{cases} \max\{P/LB(Q), P/UB(Q)\} & 0 \notin (LB(Q), UB(Q)) \\ +\infty & 0 \in (LB(Q), UB(Q)) \end{cases} \end{aligned} \tag{4.24}$$

其中 LB 和 UB 分别表示相应子约束表达式的下界和上界。

如果 O 的值域因此受到限制，就可以产生相应的判断条件，用于进行最优性剪枝。

对于 0/1 背包问题来说，目标为最大化 $O = P + Q$，根据式 (4.21) 得到 O 的更紧的上界估计式 (4.25)：

$$UB(O) = P + UB(Q) = \sum_{i=1}^{k} V_i + maxV \cdot (N - k) \tag{4.25}$$

要保证当前搜索分支的最优性，即当前搜索分支下存在比之前已找到的最优解更优的解，必要条件为 $UB(O) > O_{previous}$。因此，可以为该问题的求解程序产生相应的最优性判定条件式 (4.26)，作为最优性剪枝优化。

$$UB(O) = \sum_{i=1}^{k} V_i + maxV \cdot (N - k) > O_{previous} \tag{4.26}$$

只有当对目标函数的估计上界大于之前已经找到的最优解 $O_{previous}$ 时，才继续探索该分支；否则可以证明该分支下不可能存在更优的解，不必再进一步深入探索。

4.3　动态规划优化

动态规划适用于求解某个问题模型，前提是该模型具有以下两个性质（Cormen et al 2009）：

1. **最优子结构性质**：每个子问题的最优解可以通过其子问题的最优解构造而来。

这个性质保证了动态规划是可用的，因为动态规划只保存及使用每个子问题的一个最优解。

2. **重叠子问题性质**：当问题被递归分解成子问题时，有些相同的子问题会重复出现。这个性质保证了动态规划是有用的，因为只有存在多余的重复计算，动态规划才能够利用记忆子问题的解的办法减少时间开销，提高求解效率。

COPDL2C 在对产生程序进行动态规划优化时，分为四个步骤：

1. 模型预处理；
2. 检查约束模型性质；
3. 生成动态规划相关约束；
4. 比较不同动态规划求解方法，自动选择最优方法；

其中 2–4 步分别对应解决了 4.1 节里提到的三个主要挑战。

4.3.1 模型预处理

对于选定的独立变量集，其所有累积函数（连加、连乘等）以及与其相关的循环语句都需要被展开，转换为单步的迭代形式，以便后续优化算法能够定义子问题并进行性质检查。

在 0/1 背包问题中，$knapsack$ 被选定为独立变量，在原约束模型中存在两个约束关系/目标函数包含与其相关的累积函数（式 (4.27) 和式 (4.28)）。

$$\sum_{i=1}^{N} W[i] \cdot knapsack[i] \leq C \tag{4.27}$$

$$maximize \sum_{i=1}^{N} V[i] \cdot knapsack[i] \tag{4.28}$$

约束关系 (4.27) 将被展开成式 (4.29) 中的一系列约束关系。这里 $f_i(W, knapsack)$ 计算了前 i 个物品中，被选进 $knapsack$ 的物品的总重量；$c_i(W, knapsack)$ 是每一步需要进行约束检查的一个中间值，本例中为“剩余的背包容量”，根据题意，必须始终非负。

$$
\begin{aligned}
f_0(W, knapsack) &= 0 \\
f_1(W, knapsack) &= f_0(W, knapsack) + W[1] \cdot knapsack[1] \\
c_1(W, knapsack) &= C - f_1(W, knapsack) \\
c_1(W, knapsack) &\geq 0 \\
&\vdots \\
f_N(W, knapsack) &= f_{N-1}(W, knapsack) + W[N] \cdot knapsack[N] \\
c_N(W, knapsack) &= C - f_N(W, knapsack) \\
c_N(W, knapsack) &\geq 0
\end{aligned}
\tag{4.29}
$$

类似地，目标函数式 (4.28) 将被展开为式 (4.30) 中的一系列约束关系。其中，函数 $o_i(V, knapsack)$ 计算了前 i 个物品中，被选进 $knapsack$ 的物品的总价值；$opt_i(V)$ 是每一步的最优解。当 $knapsack$ 的所有赋值均被探查之后，可以通过 $opt_N(V)$ 得到最优的目标函数值。

$$
\begin{aligned}
o_0(V, knapsack) &= 0 \\
o_1(V, knapsack) &= o_0(V, knapsack) + V[1] \cdot knapsack[1] \\
opt_1(V) &= \max_{knapsack} o_1(V, knapsack) \\
&\vdots \\
o_N(V, knapsack) &= o_{N-1}(V, knapsack) + V[N] \cdot knapsack[N] \\
opt_N(V) &= \max_{knapsack} o_N(V, knapsack)
\end{aligned}
\tag{4.30}
$$

4.3.2　约束模型性质检查

COPDL2C 依次检查中间模型是否具有动态规划的两个性质。

4.3.2.1　检查最优子结构性质

定理 4.1. 给定两组函数集合（“$\cdot$”代表参数列表）：

$$O = \{o_0(\cdot), o_1(\cdot), \cdots, o_n(\cdot)\} \tag{4.31}$$

$$Opt = \{opt_0(\cdot), opt_1(\cdot), \cdots, opt_n(\cdot)\} \tag{4.32}$$

对于任意 $i \in [1, n]$，若

- $o_i(\cdot) = h(o_{i-1}(\cdot), b[i])$，其中 b 是被优化变量，h 是对于 $o_{i-1}(\cdot)$ 单调递增的函数；
- $opt_i(\cdot) = \max o_i(\cdot)$。

那么 $opt_i(\cdot)$ 对于 $opt_{i-1}(\cdot)$ 单调递增，即存在最优子结构性质。

证明. 对于任意 $i \in [1, n]$，

$$o_{i-1}(\cdot) \leq \max o_{i-1}(\cdot) \tag{4.33}$$

因为 h 对于 $o_{i-1}(\cdot)$ 单调递增，因此有

$$o_i(\cdot) = h(o_{i-1}(\cdot), b[i]) \leq h(\max o_{i-1}(\cdot), b[i]) = h(opt_{i-1}(\cdot), b[i]) \tag{4.34}$$

由于存在一组赋值使得等号成立，故

$$opt_i(\cdot) = \max o_i(\cdot) = h(opt_{i-1}(\cdot), b[i]) \tag{4.35}$$

即 $opt_i(\cdot)$ 对于 $opt_{i-1}(\cdot)$ 单调递增。 □

类似地，同样可以证明将定理 4.1 中的 max 换成 min，新的定理同样成立。

COPDL2C 检查中间模型中转换后的目标函数是否符合定理 4.1 的条件，若符合，意味着任意子问题的最优解可由其子问题的最优解计算而来，即该模型满足最优子结构性质。

对于 0/1 背包问题的模型，式 (4.30) 是其转换后的目标函数。将其对应到定理 4.1 得到：

$$o_i(\cdot) = h(o_{i-1}(\cdot)) = o_{i-1}(\cdot) + V[i] * knapsack[i] \tag{4.36}$$

可验证 h 对于 $o_{i-1}(\cdot)$ 单调递增。经检查，该目标函数符合定理 4.1 的条件，因此该模型具有最优子结构性质。

4.3.2.2 检查重叠子问题性质

定理 4.2. 给定两组函数集合（“$\cdot$”代表参数列表）：

$$F = \{f_0(\cdot), f_1(\cdot), \cdots, f_n(\cdot)\} \tag{4.37}$$

$$C = \{c_0(\cdot), c_1(\cdot), \cdots, c_n(\cdot)\} \tag{4.38}$$

对于任意 $i \in [1, n]$，若

- $f_0(\cdot) = a$，a 为常数；
- $f_i(\cdot) = p(f_{i-1}(\cdot), b[i])$，其中 b 是每个元素均有多种不同合法取值的被优化变量；

- $c_i(\cdot) = q(f_{i-1}(\cdot), b[i])$。

那么 $f_n(\cdot)$ 和 $c_n(\cdot)$ 的计算过程在 b 的不同赋值之间存在重叠子问题性质。

证明. 使用数学归纳法证明 $f_n(\cdot)$ 的计算过程在 b 的不同赋值之间存在重叠子问题性质。

- $n = 2$ **时**：假设有 b 的两个首元素相同的不同赋值 $b'_{A2} \neq b''_{A2}$ 但 $b'_{A2}[1] = b''_{A2}[1]$。对于 b'_{A2} 与 b''_{A2} 来说，计算

$$f_1(\cdot) = p(f_0(\cdot), b[1]) \tag{4.39}$$

  的过程是完全相同的。由于 $f_2(\cdot)$ 的计算依赖于 $f_1(\cdot)$ 的计算，因此对于 b'_{A2} 与 b''_{A2} 来说，在计算 $f_2(\cdot)$ 的过程中，对子问题 $f_1(\cdot)$ 的计算是重叠的，即 $f_n(\cdot)$ 的计算过程在 b'_{A2} 与 b''_{A2} 之间存在重叠子问题。
- $n = k$（$k > 2$）**时**：假设命题成立，即存在 b 的两个不同的赋值 $b'_{Ak} \neq b''_{Ak}$，使得 $f_n(\cdot)$ 的计算过程在 b'_{Ak} 与 b''_{Ak} 之间存在重叠子问题。
- $n = k + 1$ **时**：为了证明此命题，构造 b 的两个赋值 $b'_{A(k+1)}$ 与 $b''_{A(k+1)}$，其中 $b'_{A(k+1)}$ 与 b'_{Ak} 的前 k 项相同，$b''_{A(k+1)}$ 与 b''_{Ak} 的前 k 项相同。由于 $b'_{Ak} \neq b''_{Ak}$，$b'_{A(k+1)} \neq b''_{A(k+1)}$。$f_(k+1)(\cdot)$ 的计算依赖于$f_k(\cdot)$ 的计算，而$f_k(\cdot)$ 的计算过程在 b'_{Ak} 与 b''_{Ak} 之间存在重叠子问题，因此 $f_(k+1)(\cdot)$ 的计算过程也存在重叠子问题。

综上所述，$f_n(\cdot)$ 的计算过程在 b 的不同赋值之间存在重叠子问题性质。

同理可证，$c_n(\cdot)$ 的计算过程在 b 的不同赋值之间也存在重叠子问题性质。 □

基于定理 4.2，*COPDL2C* 将中间模型中转化后的约束关系匹配到 F 和 C 得出 p 和 q。如果匹配成功，则该模型具有重叠子问题性质。

对于 0/1 背包问题的模型，式 (4.29) 是转换后的约束关系。将其匹配到定理 4.2 中，得到函数 p（式 (4.40)）和函数 q（式 (4.41)）：

$$p(f_{i-1}(\cdot), b[i]) = f_{i-1}(\cdot) + W[i] \cdot knapsack[i] \tag{4.40}$$

$$q(f_{i-1}(\cdot), b[i]) = C - f_{i-1}(\cdot) + W[i] \cdot knapsack[i] \tag{4.41}$$

动态规划的核心思想是用空间换取时间，即通过存储和复用子问题的最优解来减少重复计算。如果一个问题模型没有重叠子问题性质，那么 *COPDL2C* 不会试图对其进行优化，因为这种模型无法通过复用中间结果来加速求解。不过，如果一个问题模型有重叠子问题性质但没有最优子结构性质，虽然原问题不适合直接使用动态规划，但 *COPDL2C* 会自动将其转化为可满足性问题，尝试存储子问题的每一个可行解以达到目的。

4.3.3 动态规划优化的生成

动态规划使用基于表的方法求解组合优化问题。给定一个问题，动态规划首先将其递归地拆分为子问题，然后分别求解。每遇到一个子问题时，若该子问题是首次遇到，则求解它并将结果保存在表内；否则直接从表中搜索并重用之间计算过结果，避免重复计算（D. K. Smith et al 1996）。

实现动态规划有两种途径：自顶向下和自底向上。

4.3.3.1 自顶向下的动态规划优化生成

COPDL2C 则在经过剪枝优化的 `_find_VAR_bp(...)` 的基础上实现基于表的动态规划 `_find_VAR_dp(...)`。为了实现动态规划，*COPDL2C* 向搜索函数加入了（1）创建表，和（2）读写表项两个部分的操作。在创建表时，*COPDL2C* 根据递归函数 `_find_VAR_bp(...)` 的参数，自动设定表的维度。具体来说，对于每个不是用于最优性剪枝的参数，*COPDL2C* 在表中增加一维对应该参数的值。最终，表中的每一项对应了 `_find_VAR_bp(...)` 的一种参数组合。在实际实现时，这个表会被定义成全局的多维数组，表中各项均被初始化为无效值。在 `_find_VAR_bp(...)` 中，*COPDL2C* 加入了两个代码块用于实现对表的访问操作。第一个代码块在函数刚进入的时候，检查当前的子问题是否已经被求解（相应表项不是无效值），若是，则直接提取表内存储的解，避免重复计算；第二个代码块在当前子问题找到更优解时，将这个最优解写入相应表项。

图 4.3 是应用了动态规划优化的 0/1 背包求解程序，是在 `_find_sel_bp(...)` 的基础上修改得到的。函数 `_find_sel_bp(...)`包含三个参数 `_step`，`_sum1`，`_sum2`，由于 `_sum2` 用于最优性剪枝不作为动态规划表的一维，因此动态规划表只有两维，分别对应参数 `_step` 和 `_sum1`。动态规划表被声明为全局数组 `_DP_sel[][]`，在搜索开始前被全部初始化为无效值（本题中设为 -1）。每个表项 `_DP_sel[_step][_sum1]` 的值表示“在前 (`_step`-1) 个物品中已经选取了总重量为 `_sum1` 的集合，那么在后面的 (`N-_step`+1) 个物品中，在满足约束的前提下，选到物品的总价值至多是多少”。

4.3.3.2 自底向上的动态规划优化生成

在模型预处理中展开得到的递归函数集（例如式 (4.29) 和式 (4.30)）并没有消除计算过程中的任何冗余计算。为了实现动态规划优化，*COPDL2C* 进一步将这些递归函数转化为递推函数，并将对子问题求解的中间结果存入动态规划表，用于避免重复计算。

具体来说，给定：

- 独立变量集 $\{b_i\}$，

```
int _find_sel_dp(int _step, int _sum1, int _sum2) {
    // 若该子问题曾被解决过，直接使用表中存储的解
    if (_DP_sel[_step][_sum1] != -1) {
        _sum2 += _DP_sel[_step][_sum1];
        _step = N + 1;
    }
    if (_step > N) {
        _result = _sum2;
        if (!(_sum1 <= C))
            return -1;
        _update();
        return _sum2;
    }
    int __sum1 = _sum1;
    int __sum2 = _sum2;
    for (sel2[_step] = 0; sel2[_step] <= 1; sel2[_step]++) {
        _sum1 = __sum1;
        _sum2 = __sum2;
        if (sel2[_step])
            _sum1 += W[_step];
        if (sel2[_step])
            _sum2 += V[_step];
        if (!(_sum1 <= C))
            continue;
        if (!(_sum2 + 1000 * (N - _step) > _best__result))
            continue;
        int _tmp0 =
            _find_sel_dp(_step + 1, _sum1, _sum2) - __sum2;
        // 若找到当前子问题的更优解，更新对应表项
        if (_tmp0 > _DP_sel[_step][__sum1])
            _DP_sel[_step][__sum1] = _tmp0;
    }
    return __sum2 + _DP_sel[_step][__sum1];
}
```

图 4.3　应用了动态规划优化的 `_find_sel_dp(...)`

- 一系列递归形式的约束函数 $\{c_i(\cdot)\}$，
- 一系列递归形式的目标函数 $\{o_i(\cdot)\}$，
- 最优化方向 extreme（最大化 maximize 或最小化 minimize），

COPDL2C 创建一个二维表 M，并通过式 (4.42) 自下而上构造各子问题的解。

$$\begin{aligned} M[i, c_i] &= extreme\ o_i(\cdot) \\ &= extreme\ h(extreme\ o_{i-1}(\cdot), b_i) \\ &= extreme\ h(M[i-1, c_{i-1}], b_i) \end{aligned} \tag{4.42}$$

在 M 表中，i 对应 b 的数组下标范围，c_i 对应相应的约束函数值，而表项 $M[i, c_i]$ 则记录了对应的子问题的最优解，其含义为："给定约束函数值 c_i，适当设定 b 的前 i 个元素值使得目标函数值 o_i 最优"。枚举不同的 b_i 并使用相应的 c_{i-1} 构造解，比较各构造产生的解，从中选择最优的存入表项 $M[i, c_i]$。通过这种表驱动的递推计算过程，*COPDL2C* 实现了自底向上的动态规划优化。

图 4.4 展示了 *COPDL2C* 在对原始 0/1 背包问题模型采用自底向上动态规划优化后的中间模型。

4.3.3.3 自顶向下与自底向上动态规划的区别

自顶向下的动态规划是基于递归的，能够与搜索剪枝优化兼容；同时只会求解从顶层原始问题向下分解可以达到的子问题，在子问题稀疏的情况下能够避免大量不必要的计算。因此，一般情况下，从求解效率上来说，自顶向下的动态规划要好于自底向上的动态规划。

另一方面，自底向上的动态规划不依赖于 *COPDL2C* 中的其他程序自动生成技术（基于循环/递归的程序框架、搜索剪枝），可以直接在原始模型上进行优化，因此中间结果可作为其他约束求解器的输入，与现有约束求解器融合。

4.3.4 不同动态规划求解方法的比较和选择

某些问题模型可能存在多个可选的被优化变量，因此存在多种动态规划求解方法。用户可以通过设定最大内存限制，让 *COPDL2C* 自动估计每种可选方法的时间和空间复杂度，并从中选择使用内存不超过用户约定的上限、具有最大理论加速比的模型。

本节以一个"嵌套 0/1 背包问题"为例来解释如何进行不同求解方法地比较和选择。嵌套 0/1 背包问题的问题描述如下：

```
% 输入参数
    ... % 保持不变

% 变量，表 dpvalue
    array[0..N, 0..C] of var int: dpvalue;

% 约束，表 dpvalue 各项的计算过程
    constraint forall(j in 0..C) (
        dpvalue[0,j] = if j==0 then 0 else -1 endif
    );
    function var int: calcValue(int: i,int: j,int: k) =
        if j-W[i]*k>=0 then
            if dpvalue[i-1,j-W[i]*k] != -1 then
                dpvalue[i-1,j-W[i]*k]+V[i]*k
            else
                -1
            endif
        else
            -1
        endif;
    constraint
        forall(i in 1..N, j in 0..C) (
            dpvalue[i,j] = max(k in 0..1)(calcValue(i,j,k))
        );

% 目标函数，从最后一行中选择最优值
    solve maximize max(j in 0..C)(dpvalue[N,j]);
```

图 4.4　0/1 背包问题的优化模型

```
% 输入参数
    int: N1;
    int: N2;
    int: C1;
    array[1..N1] of int: V1;
    array[1..N1] of int: W1;
    array[1..N2] of int: V2;
    array[1..N2] of int: W2;

% 变量
    var int: C2;
    var set of 1..N1: K1;
    var set of 1..N2: K2;

% 约束关系
    constraint sum (i in K1) (W1[i]) <= C1;
    constraint C2 = sum (i in K1) (V1[i]);
    constraint sum (i in K2) (W2[i]) <= C2;

% 目标函数
    solve maximize sum (i in K2) (V2[i]);
```

图 4.5　嵌套 0/1 背包问题的 MiniZinc 约束模型

有两组物品集合，分别包含 N_1 和 N_2 个物品，每个物品均有各自的重量和价值。从这两个集合中选取物品，分别用于填充两个背包 K_1 和 K_2。要求 K_1 中物品总重量不超过 C_1，而 K_2 中物品总重量不超过 K_1 中物品总价值。求 K_2 中物品总价值最大是多少？

图 4.5 是该问题原始的 MiniZinc 约束模型。

通过静态分析，*COPDL2C* 发现了两个可选的被优化变量——K_1 和 K_2，分别为两个背包内最终的填充物品集合。对于这两个被优化变量的候选，存在三种潜在的动态规划求解方法：

1. **只对 K_2 进行优化**：在采用搜索得到第一个背包的填充物品集合 K_1 后，根据得到的第二个背包的容量限制 C_2，剩下的问题变成了一个普通的 0/1 背包问题。因此，*COPDL2C* 将对 K_2 的求解过程转化为动态规划求解过程，空间复杂度和时间复杂度均为 $O(N_2C_2)$。由于对普通 0/1 背包问题的搜索算法的时间复杂度理论上为 $O(2^{N_2})$，因此估计该优化的加速比为 $2^{N_2}/(N_2C_2)$。
2. **只对 K_1 进行优化**：在采用搜索得到第二个背包的填充物品集合 K_2 后，第一个背包内物品总价值的下界为第二个背包内物品总重量，同时总重量的上界为 C_1。由于剩下的问题为无目标函数的可满足性问题，需要记录所有可行解，空

间复杂度较高，因而无法生成可行的动态规划求解方法。

3. **同时对 K_1 和 K_2 进行优化**：*COPDL2C* 将 K_1 和 K_2 合并转化为一个更大的被优化变量 K。对于整个问题来说，搜索的时间复杂度理论上为 $O(2^{N_1+N_2})$，相比较而言，动态规划的空间复杂度和时间复杂度均为 $O((N_1+N_2)C_1C_2^2)$。因此，可估计加速比为 $2^{N_1+N_2}/[(N_1+N_2)C_1C_2^2]$。

动态规划解法 1 和 3 的存储表空间大小均在内存限制以内，而解法 3 因有更大的加速比而被 *COPDL2C* 最终选中。图 4.6 展示了使用解法 3 优化得到的动态规划版本约束模型。

4.3.5　特殊情况处理

对于问题模型中的一些本不适合用动态规划求解方法处理的情况，*COPDL2C* 尝试对原问题做适当转化，使得本章的方法能够适用于更多的问题。

- **孤立约束**：孤立约束是指对被优化变量的某个或某些元素的特别约束，区别于循环或累积约束。对于孤立约束，*COPDL2C* 会生成相应的 `if` 语句，并将它们插入 `calcValue(...)` 函数的开头。例如，在 0/1 背包问题中，要求“物品 3 必须放入背包”，即“`3 in knapsack`”，此时 *COPDL2C* 会在 `calcValue(...)` 函数一开始加入一个 `if` 语句“`if i==3 /\ k!=1 then -1 else ...`”确保最终的赋值一定能够满足该约束。
- **局部后效性**：某些问题（如最长上升子序列）存在局部后效性。具体来说，每个子问题的最优解 $o_i(\cdot)$ 计算不仅仅依赖于它的子问题的最优解 $o_{i-1}(\cdot)$ 以及被优化变量当前元素 $b[i]$，还依赖于被优化变量之前枚举过的某些元素值（如 $b[i-1]$）。*COPDL2C* 通过保存更多数据的方式来避免此问题。如果对 $o_i(\cdot)$ 的计算还依赖于 b 的最后 T 个之前枚举过的元素 $b[i-T:i-1]$，那么 *COPDL2C* 会在 *dpvalue* 表中增加 T 维，分别保存这 T 个元素的值。
- **无最优子结构性质**：对于无目标函数的可满足性问题或无最优子结构的优化问题，*COPDL2C* 存储子问题的所有可行解来实现数据复用。*COPDL2C* 会在 *dpvalue* 表中新增一维，每个表项 $dpvalue[i,c_i,j]$ 存储一个二进制值，用于表示是否存在满足约束函数值为 $c_i(\cdot)$ 的对 b 前 i 个元素的一组赋值，使得目标函数值为 j，“1”表示存在，“0”表示不存在；同时，h 函数值等于其第一个参数值，extreme 为 max。这样设置保证了，一个子问题是可满足的，当且仅当它的至少一个子问题是可满足的。
- **多个被优化变量**：当原问题模型中含有多个可选的被优化变量时，*COPDL2C* 会枚举各种组合，尝试生成不同的动态规划解法，然后根据空间复杂度和时间复杂度，选择符合要求的最佳方法。若可选的被优化变量数量很多，这种分析

```
% 输入参数
    ... % 保持不变
    int: sum = 20; % 根据数据得到的 C2 的上界

% 变量，表 dpvalue
    array[0..N1+N2, 0..C1, 0..sum, 0..sum] of var int:
        dpvalue;

% 约束，表 dpvalue 各项的计算过程
    constraint forall (i2 in 0..C1,i3 in 0..sum,i4 in 0..sum)(
        dpvalue[0,i2,i3,i4] =
            if i2==0 /\ i3==0 /\ i4==0 then 0 else -1 endif
    );

    function var int:
        calcValue(int:i1,int:i2,int:i3,int:i4,int:i5) =
            if i1 <= N1 then
                if i2-W1[i1]*i5>=0 /\ i3-V1[i1]*i5>=0 then
                    dpvalue[i1-1,i2-W1[i1]*i5,i3-V1[i1]*i5,i4]
                else
                    -1
                endif
            elseif i4<=i3 /\ i4-W2[i1-N1]*i5>=0 then
                if dpvalue[i1-1,i2,i3,i4-W2[i1-N1]*i5]!=-1
                    then
                    dpvalue[i1-1,i2,i3,i4-W2[i1-N1]*i5]
                        +V2[i1-N1]*i5
                else
                    -1
                endif
            else
                -1
            endif;

    constraint
        forall(i1 in 1..N1+N2,i2 in 0..C1,
            i3 in 0..sum,i4 in 0..sum) (
                dpvalue[i1,i2,i3,i4] =
                    max(i5 in 0..1)(calcValue(i1,i2,i3,i4,i5))
            );

% 目标函数，从最后一行中选择最优值
    solve maximize max (i2 in 0..C1,i3 in 0..sum,i4 in 0..sum)
        (dpvalue[N1 + N2, i2, i3, i4]);
```

图 4.6　嵌套 0/1 背包问题动态规划版本的 MiniZinc 约束模型

表 4.1　优化后的 *COPDL2C* 在数据集中 49 个问题上的有效性评估

课程及年份	总计	无法表达	部分可解	完全可解	学生平均
计算概论 2018 秋	8	1	0 (-1)	7 (+1)	5.8
计算概论 2019 秋	9	0	0 (-2)	9 (+2)	6.3
程序设计实习 2019 春	12	1	2 (-5)	9 (+5)	4.5
程序设计实习 2020 春	8	0	1 (-4)	7 (+4)	4.8
数据结构与算法 2018 秋	6	3	1 (-2)	2 (+2)	2.4
数据结构与算法 2019 秋	6	2	2 (-2)	2 (+2)	3.2
总计	49	7	6 (-16)	36 (+16)	27.0

方法可能会比较慢。为了避免分析过于低效，在这种情况下，*COPDL2C* 会按照长度对这些被优化变量排序，然后只关注其中长度较长的变量。

4.4　优化后的 *COPDL2C* 的性能测试实验设计与结果

本节通过比较实验，客观评估了 *COPDL2C* 产生的求解程序的求解效率。首先，在上一章的程序设计与算法分析类课程期末考试题的数据集（第 3.6.1 节）上，评估经优化后的 *COPDL2C* 的有效性；其次，在 9 个经典的组合优化问题上，对比了由 *COPDL2C* 产生的求解程序与通用约束求解器 Gecode 以及人工编写程序的运行效率，从而评价引入搜索剪枝和动态规划优化后的 *COPDL2C* 在任意问题上的求解能力；最后，又有针对性地在 9 个不同的动态规划问题上，比较了 *COPDL2C*+Gecode 的工具组合与 Gecode 和 Chuffed 求解器的效率。

4.4.1　优化后的 *COPDL2C* 的有效性评估

本节采用与上一章有效性评估相似的实验设计（第 3.6.2 节），将同样的 *COPDL* 描述作为输入，使用优化后的 *COPDL2C* 生成相应的 C 语言程序，并提交到评测系统获得评测结果。数据集具体信息及详细实验结果参见附录 B。

表 4.1 展示了添加了搜索剪枝和动态规划优化后的 *COPDL2C* 在这些期末考试题上的表现，其中，括号内为与未优化的 *COPDL2C* 相比的数值变化。可以看到，在计算概论课程中，*COPDL2C* 可以达到接近满分的成绩；而在程序设计实习和数据结构与算法课程中，*COPDL2C* 的表现也有大幅的提升。特别是在计算概论和程序设计实习课程上，*COPDL2C* 的表现都远优于学生的平均表现。

结论 4.1：添加了搜索剪枝和动态规划优化后的 *COPDL2C*，能够有效解决许多程序设计和算法分析类课程的测试题，其表现甚至在一定程度上优于学生的平均水平。

4.4.2 *COPDL2C* 在求解经典组合问题上的效率

本节在 9 个经典的组合优化问题上，对比了由 *COPDL2C* 自动生成的求解程序与通用求解器 Gecode 和 Chuffed 的运行效率。

4.4.2.1 数据集

本实验选取了 9 个经典的组合优化问题作为数据集，包括三个枚举问题，三个搜索剪枝问题，和三个动态规划问题：

1. 最大公约数问题（枚举）：找到给定若干正整数的最大公约数；
2. 蛋糕烘培问题（枚举）：已知每类蛋糕对每种原料的需求以及利润，要求在给定各原料限制的条件下，确定各类蛋糕的制作量以最大化总利润；
3. 立方体问题（枚举）：见图 3.5；
4. 图染色问题（搜索）：给定无向图，为每个结点染色，要求相邻结点不同色，求至少需要多少种颜色；
5. 0/1 背包问题（动态规划）：见图 3.18；
6. 团队合作（动态规划）：已知每个人的合作能力和工作能力，从中挑选若干人组成合作团队，要求团队内总合作能力值为正，总工作能力值尽可能大；
7. N 皇后问题（搜索）：将 N 个国际象棋的皇后放置在 $N \times N$ 的棋盘上，要求皇后之间无法相互攻击；
8. 旅行商问题（搜索）：给定带权无向图，要求在图中找一条最短的汉密尔顿回路，即经过每个点恰好一次后返回起点；
9. 最短路径（动态规划）：给定带权图，找到从结点 1 到结点 N 的最短路径；

4.4.2.2 实验设置

本实验将 *COPDL2C*，Gecode 和 Chuffed 应用到对数据集中问题的求解中。由于三种工具分别是以 *COPDL* 和 MiniZinc 模型作为输入，为了避免因模型的差异造成的不公平，同一问题的 *COPDL* 和 MiniZinc 模型采用统一的表述方式，即采用相同的顺序定义变量，且所有表达式均具有相同结构，保证两个模型只是因为语法的区别会略有差异，且这些差异并不会影响求解效率。

整个实验均在同一台普通个人电脑（CPU：Intel Core i5-7300HQ 2.5GHz，内存：8G）上进行。每个问题预先设定了 5 组具有代表性的测试数据。在每个问题上分别运行这些 *COPDL2C*，Gecode，和 Chuffed，然后分别记录下在这 5 组输入上的求解时间。在每次求解时，设置了时间限制 1800 秒，求解程序运行超过时间限制仍未得出结果将被直接终止。

4.4.2.3　实验结果

表 4.2 列出了各求解程序在每个问题上的平均求解时间。严格来说，*COPDL2C* 的求解过程应该包含生成程序的时间开销，但测试时发现，相比求解时间，*COPDL2C* 生成程序的时间开销非常短（小于 0.1 秒），而且对于每个问题，*COPDL2C* 只需产生一次程序，因此可以忽略不计。

可以看到，对于搜索剪枝类的问题，*COPDL2C* 生成的求解程序在求解效率上与通用求解器差距不大。可能原因是，虽然 *COPDL2C* 生成的求解程序远比通用求解器更轻量，占用资源更少，但毕竟是通过静态分析模型产生的，只能按照原定策略进行搜索，很难在搜索过程中根据数据的变化动态调整搜索策略，因此表现与通用求解器差异不大；

结论 4.2：*COPDL2C* 比 Gecode 和 Chuffed 在限制时间内能够解决更多的问题，并且求解效率高数百甚至数千倍。

4.4.3　*COPDL2C* 优化模型与原始模型在求解效率上的比较

本节实验单独评价 *COPDL2C* 的自底向上的动态规划优化效果，即在 9 个具有代表性的组合优化问题约束模型上，测量了经由 *COPDL2C* 优化模型在普通约束求解器上的求解效率，并与其他两个常用约束求解器直接求解原始模型的效率进行对比。

4.4.3.1　数据集

为了评价 *COPDL2C* 自底向上的动态规划优化的有效性，实验中使用了 9 个具有代表性的组合优化问题约束模型作为数据集：

1. 0/1 背包问题。
2. 完全背包问题：与 0/1 背包问题类似，但每个物品可以重复选取。
3. 嵌套 0/1 背包问题。
4. 最短路问题：给定含有 N 个点的图以及各点间的带权边，找到从点 S 到点 T 的权值和最小的路径。
5. 最长上升子序列问题：从长度为 N 的序列中找出最长的上升子序列。
6. 最长公共子序列问题：从两个长度为 N 的序列中找出最长的公共子序列。
7. 放射治疗问题（Baatar et al 2007）：给定一个 $M \times N$ 的整数矩阵以描述每个区域的辐射剂量，将该矩阵分解成一组矩阵的整数线性组合。要求最小化权值和及矩阵数量。
8. 个位最小背包问题：与 0/1 背包问题类似，但目标函数是最小化被选取物品的总价值和的个位数。

表 4.2　*COPDL2C*，Gecode，和 Chuffed 求解各问题实例的时间和空间消耗

题号	运行时间（秒）			内存开销（MB）		
	COPDL2C	**Gecode**	**Chuffed**	***COPDL2C***	**Gecode**	**Chuffed**
P1	[0.19, 0.30, 0.38, 0.41, 4.09]	[0.66, 1.15, 2.00, 10.51, 96.50]	[0.90, 2.54, 2.79, 20.54, 325.89]	3–12	16–136	20–139
P2	[0.02, 0.03, 0.03, 0.05, 0.24]	[0.63, 0.63, 0.65, 0.81, 1.31]	[0.61, 0.67, 0.68, 3.25, 147.80]	3	13	13
P3	[0.02, 0.02, 0.04, 0.05, 0.36]	[0.61, 0.60, 0.82, 1.24, 5.72]	[1.49, 7.56, 48.05, >1800, >1800]	3	13-21	N/A
P4	[0.10, 4.23, 47.52, >1800, >1800]	[0.66, 1.14, 6.88, 395.32, >1800]	[0.59, 0.79, 3.25, 247.52, >1800]	N/A	N/A	N/A
P5	[0.03, 0.06, 0.11, 0.32, 0.56]	[4.35, 363.49, >1800, >1800, >1800]	[2.95, 191.22, 1523.72, >1800, >1800]	3–5	N/A	N/A
P6	[0.02, 0.03, 0.06, 0.11, 0.28]	[0.73, 3.24, 60.53, 134.24, 863.72]	[0.70, 1.92, 5.83, 14.98, 71.29]	3–4	15–21	16–22
P7	[0.02, 0.02, 0.14, 0.36, 0.55]	[0.67, 0.74, 3.99, 6.07, 45.41]	[0.72, 0.75, 1.86, 2.08, 23.50]	3	13–17	13–19
P8	[0.02, 0.07, 9.23, 56.24, 842.73]	[0.64, 0.74, 24.87, 532.91, >1800]	[0.60, 3.56, 72.13, >1800, >1800]	3–8	N/A	N/A
P9	[0.02, 0.02, 0.03, 0.04, 0.06]	[0.78, 0.95, 4.52, 65.36, 242.91]	[0.79, 1.59, 82.67, 1252.01, >1800]	3–4	13–220	N/A

注：“N/A”表示求解过程因超时而强制结束，导致内存开销无法准确估计

9. 黑洞游戏问题：一种单人扑克游戏。首先将黑桃 A 放入黑洞，并将剩余的 51 张牌随机分成 17 堆，每堆 3 张。每一轮只能将某一堆堆顶的牌放入黑洞，且必须与上一次放入黑洞的牌点数相邻。请找到一种方法将所有牌移入黑洞。

其中，前七个模型均满足动态规划的两个性质，而后两个模型只有重叠子问题性质而没有最优子结构性质。选择这些问题作为实验数据集的原因是：

- 这些模型的动态规划求解方法两两不同；
- 这些模型所使用的求解方法代表并覆盖了大部分经典的动态规划求解方法；
- 许多问题以背包问题变种的形式出现，方便说明和理解。

4.4.3.2　实验设置

本实验将 *COPDL2C*+Gecode，Gecode（Schulte et al 2006）和 Chuffed（Chu et al 2012）应用到对数据集中模型的求解中。特别地，对于 Chuffed 求解器，实验测量了的三种不同工作模式下的性能表现，分别是

- 原始求解模式（记为 Chuffed）；
- 带自动缓存的求解模式（记为 ChuffedC），通过重用中间结果来减少冗余搜索（B. M. Smith 2005）；
- 带惰性语句生成的求解模式（记为 ChuffedL），通过从冲突中产生新约束减少冗余搜索（Ohrimenko et al 2009）。

整个实验均在同一台普通个人电脑（CPU：Intel Core i5-7300HQ 2.5GHz，内存：8G）上进行。对于实验中每个问题模型的给定输入规模，随机产生了 10 组输入参数，分别运行这些求解工具，然后记录下在这 10 组输入上的平均求解时间。在每次求解时，设置了时间限制 10000 秒，求解器运行超过时间限制仍未得出结果将被直接终止。

4.4.3.3　实验结果

表 4.3 列出了各工具在不同问题模型上的求解时间。从中可以发现，Gecode 和 Chuffed 在四个模型（第 4，6，7，8 个模型）上不能在时间限制内得到最优解，主要原因是这两个求解器使用的一些基本优化不足以高效求解这些复杂问题。与之相反，*COPDL2C*+Gecode 能在 100 秒内求解每个模型。对于其他模型，*COPDL2C*+Gecode 比 Gecode 快约 698 倍，比 Chuffed 快约 790 倍。

结论 4.3：相比于只采用基本优化技术的通用求解器，*COPDL2C* 的求解速度要快数百倍。

考虑具有高级优化的两个求解器 ChuffedC 和 ChuffedL。在所有模型的求解上，

表 4.3　不同求解工具在各问题模型上的求解时间（秒）

问题模型	特殊情况	输入规模	Gecode	CHUFFED	CHUFFEDC	CHUFFEDL	*COPDL2C*+Gecode 总	*COPDL2C*+Gecode 分析
1. 0/1 背包	-	$N = 10^3$ $C = 10^3$	975	1143	47	1598	2	0.05
2. 完全背包	-	$N = 10^3$ $C = 10^3$	3985	4262	125	>10000	2	0.05
3. 嵌套 0/1 背包	多变量	$N = 100$ $C_1 = 100$ $C_2 = 20$	357	496	78	672	1	0.03
4. 最短路	局部后效性	$N = 10^3$	>10000	>10000	784	>10000	97	0.15
5. 最长上升子序列	局部后效性	$N = 10^3$	1274	1653	34	1751	5	0.03
6. 最长公共子序列	多变量，局部后效性	$N = 10^3$	>10000	>10000	55	>10000	10	0.08
7. 放射治疗	多变量，局部后效性	$N = 10$ $M = 8$	>10000	>10000	40	4	4	0.18
8. 个位最小背包	无最优子结构	$N = 10^3$ $C = 10^3$	>10000	>10000	58	>10000	53	0.05
9. 黑洞游戏	无最优子结构，多变量	$N = 52$	168	74	46	267	43	0.16

COPDL2C+Gecode 比 ChuffedC 快约 23 倍。ChuffedL 在四个模型（第 2，4，6，8 个模型）上不能在时间限制内得到最优解，其可能的原因是其采用的惰性语句生成优化技术在搜索过程中，根据遇到的冲突来添加新的约束关系。然而在这些问题中，这些新的约束关系并不能有效地缩小搜索空间，反而增加了处理开销，导致效率严重降低。对于其他模型，*COPDL2C*+Gecode 比 ChuffedL 快约 345 倍。

结论 4.4： 相比于采用缓存和惰性语句生成优化技术的通用求解器，*COPDL2C*+Gecode 在同时具有两个动态规划性质、以及某些无最优子结构性质的问题上均具有更快的求解速度。

COPDL2C+Gecode 的求解过程包含两个部分：约束模型优化和约束模型求解。为了分析模型优化对约束求解的影响，在实验时特别记录了约束模型优化步骤的时间开销。如表 4.3 最后一列所示，*COPDL2C*+Gecode 在对每个模型优化上的时间开销在 0.03-0.18 秒之间。相比于总求解时间以及通过模型优化节省的时间，模型优化的时间开销完全可以忽略不计。

结论 4.5： *COPDL2C*+Gecode 通过约束模型优化，在显著提高求解效率的同时，几乎没有引入额外的时间开销。

4.5 本章讨论

理论上，*COPDL2C* 可对满足下列两个条件的任意模型 $\mathcal{M}$ 进行优化：

(1) $\mathcal{M}$ 具有重叠子问题。

(2) $\mathcal{M}$ 存在至少一个潜在的被优化变量。

其中，条件 (1) 保证该模型的求解过程中存在用空间换取时间的可能，因为只有子问题之间存在重复计算，才可以通过存储和复用中间结果的方式消除这些冗余的时间开销。条件 (3) 则确保基于表的动态规划能够适用，因为只有多个变量之间有线性顺序关系，才可以通过下标来区分不同的子问题。

在实际应用中，由于实现上的困难，*COPDL2C* 的适用性还受到其他两个条件的限制：

(3) $\mathcal{M}$ 中所有表达式只能使用约束模型描述语言（*COPDL2C* 或 MiniZinc）内置的运算符、全局约束、非递归函数等。

(4) 动态规划方法需要的空间开销不能超过用户或运行环境要求的内存上限。假设：

- 被优化变量长度为 N；

- 被优化变量每个元素的定义域大小为 M；
- 与被优化变量直接相关的累积函数有 L 个；
- 对于第 i 个直接相关的累积函数，其约束值的值域大小为 R_i；
- 后效性常量值为 T。

那么总的空间复杂度为 $O(NM^T \prod_{i=1}^{L} R_i)$。根据这个式子，*COPDL2C* 可以估计出实际的空间开销，比较其与内存上限的大小关系，从而决定是否对模型进行优化。

COPDL2C 不局限于求解“纯”动态规划问题模型。对于一些具有重叠子问题的非动态规划问题模型，*COPDL2C* 也可以对其进行优化。例如，图染色问题，要求用 C 种不同颜色对图中 N 个点染色，要求任意相邻的点不能同色。该问题具有后效性（每个点的颜色可能受到之前每个点染色情况的限制），因此不被认为是动态规划问题。不过，当点数 N 比较小时（如 $N = 10$），*COPDL2C* 可通过设置后效性常量值 $T = N$，使其转化为一个能够被动态规划方法求解的问题模型并对其进行优化。但当点数 N 比较大时（如 $N = 50$），由于空间开销过大，*COPDL2C* 无法实现模型优化。

4.6 本章小结

本章在 *COPDL2C* 的基础上，实现了在组合优化问题模型上的搜索剪枝和动态规划两类高级优化技术的自动生成，使得产生的求解程序具有较高的求解效率。在实施高级优化的过程中，*COPDL2C* 首先在静态分析的过程中，发现问题模型本身的特殊性质，然后根据这些性质，构造搜索剪枝（可行性剪枝和最优性剪枝）或动态规划（自顶向下和自底向上的动态规划）所需的判定条件或转移函数，最终将其应用于求解程序。

本章首先通过实验评估了 *COPDL2C* 的有效性；其次，比较 *COPDL2C* 产生求解程序，约束求解器 Gecode，与 Chuffed 在求解经典问题上的效率差异。实验结果表明，由于 *COPDL2C* 产生的程序比较轻量，虽然没有采用动态的搜索策略，但在引入搜索剪枝和动态规划后，求解效率优于约束求解器 Gecode 和 Chuffed。而在动态规划问题上，*COPDL2C* 产生的程序的表现显著优于 Gecode 和 Chuffed。同时，本章还通过实验比较经由 *COPDL2C* 自底向上动态规划优化的模型与原始模型在求解效率上的差异。结果表明，对于动态规划问题，在底层约束求解器不变的情况下，求解优化模型的效率远高于直接求解原始模型；而对于准动态规划问题，普通求解器求解优化模型的效率与加入针对性优化的求解器求解原始模型的效率相当。

本章的主要创新点包括：

1. 提出并实现了问题关键性质的自动静态分析方法，包括子约束值域受限、目标

函数值域受限、最优子结构、重叠子问题性质；

2. 提出并实现了高级优化技术的自动设计和应用方法，包括搜索剪枝和动态规划。

搜索剪枝和动态规划是人们在求解组合优化问题过程中广泛使用的两种优化技术。然而，由于在使用时具有一定的限制条件，同时设计和实现上也比较灵活，因此之前的研究均没有在一般的组合优化模型上应用对这两种优化技术。而本章通过研究并实现上述两个创新点，首次实现了这两种优化技术的完全自动化生成。

未来可以在 *COPDL2C* 中实现更多优化策略，包括分治、贪心，甚至引入模拟退火等概率算法，从而能够对更多问题生成更高效的代码。

第五章　从自然语言问题描述产生约束模型的翻译方法 *NL2COPDL*

第三章和第四章提出并实现了 *COPDL* 语言及带优化的求解程序自动生成工具 *COPDL2C*，有了这两个工具，用户只需将待求解的问题用 *COPDL* 描述，就可以运行 *COPDL2C* 获得相应的求解程序。本章尝试进一步降低用户的使用门槛，让用户直接用自然语言而非形式化语言描述问题。为此，本章建立了一个包含 40 个用自然语言描述的组合优化问题及其测试数据的数据集，并在此数据集上实现从自然语言问题描述到 *COPDL* 描述的自动转换方法——*NL2COPDL*。*NL2COPDL* 结合了基于模板规则和基于框架的技术，实现了将受限自然语言问题描述自动转换为 *COPDL* 描述的翻译过程。将 *NL2COPDL* 与 *COPDL2C* 相连接，就实现了在自建的数据集上由自然语言描述问题自动生成其求解程序的系统。*NL2COPDL* 的设计基于以下几个基本假设：（1）*COPDL* 是一种精简的语言，它能描述的问题框架的类型是可以通过逐一列举的方式给出的；（2）自然语言中描述约束关系的句式和关键词是有限的并可以通过逐一列举的方式给出；（3）变量存在于特定的有限的约束关系描述句式中，可以通过这些句式找到变量并推断其类型。因此 *NL2COPDL* 的设计采用了“预定义模板+规则”的方式从自然语言描述中提取变量和约束框架，然后使用约束求解技术对框架进行填充，最终得到完整的 *COPDL* 描述模型。

5.1　从自然语言描述到约束模型

组合优化问题一般可以用组合优化约束模型来精确描述。相对于形式化的约束模型来说，自然语言往往存在歧义性、不完备性、不一致性等缺点，导致计算机不能正确地理解问题，使得求解过程出错，无法得到用户想要的结果。换言之，自然语言与形式化的约束模型之间存在着一定的鸿沟。为了增强 *COPDL* 及相关工具的适用性，帮助用户更好地求解组合优化问题，本章设计并实现了能够从受限的英文自然语言描述出发，自动生成出等价 *COPDL* 模型的方法 *NL2COPDL*。

NL2COPDL 通过四个处理步骤，解决了研究过程中的三个主要挑战：

- 如何识别和区分问题描述中的变量；
- 如何从自然语言描述的约束中提取约束模板；
- 如何用变量填充模板，使之成为完整的约束模型。

5.2 组合优化约束模型自然语言描述的特点

本节讨论了处理描述组合优化约束模型的自然语言与处理任意文本自然语言的差异，包括（1）自然语言受限性，（2）组合优化约束模型翻译难点，以及（3）处理方法。

5.2.1 组合优化约束模型自然语言描述的受限性

用于描述组合优化约束模型的自然语言是相对受限的，因此与普通文本自然语言有很大的不同，主要原因有以下两点：

- 约束模型里的操作符是有限的，而每种操作符对应的自然语言短语相对受限。例如，描述“≤”可以用“less than or equal to”，“not greater than”，“not exceed”等。
- 问题描述有相对固定的模式。例如，在描述组合优化约束模型的自然语言中，常用“given”关键词引出参数变量，“find”关键词引出决策变量，以及“so that”引出约束关系等。

5.2.2 组合优化约束模型自然语言描述的翻译难点

要正确将组合优化约束模型的自然语言描述翻译为形式化的模型描述（*COPDL* 描述），相比于普通文本自然语言翻译任务，需要克服以下困难：

- 指代关系复杂：同一变量在上下文中可能以不同形式（实词、代词）多次出现；
- 逻辑关系复杂：一条短句可能涉及到大量实体、操作符，而且各元素的出现位置往往决定其在最终约束中的角色
- 高精确性要求：任何一个细节的错误都可能严重影响整个模型的正确性

5.2.3 组合优化约束模型自然语言描述的翻译方法

目前自然语言理解和翻译技术主要分两类：（1）基于统计和机器学习的方法，（2）基于模板规则的方法。

对于本章要解决的组合优化约束模型自然语言描述翻译问题来说，目前已知的基于统计和机器学习的方法中，并没有特别有效的解决方法，主要原因包括：

- 描述组合优化模型的自然语言与普通文本自然语言相差较大，没有合适的预训练模型，也缺乏足够的训练数据；
- 相比普通文本自然语言，组合优化模型的自然语言描述中经常存在密集且复杂的指代关系和逻辑关系，其规律不容易通过统计来发现；

- 翻译组合优化约束模型的精确性要求极高，基于统计和机器学习方法的准确性无法满足要求。

因此，本章采用基于模板规则的翻译方法，确保逻辑关系的正确抽取。其中所采用的模板和匹配规则由人工从各种操作符对应的常见描述方式总结整理而来。同时，结合基于框架的翻译技术，提高指代关系的识别率，实现多条关联语句的正确翻译。

5.3　建立组合优化问题的自然语言描述数据集

本章建立了一个组合优化问题的自然语言描述数据集，数据集中收集了 40 个问题的自然语言描述，附录 C 列出了这些问题的详细内容。其中，33 个问题来自大学本科一、二年级的程序设计类与算法设计类课程的例题、作业与测试；7 个问题来自 MiniZinc 教程（Marriott et al 2014） 或 MiniZinc 标准测试集（Stuckey et al 2014）。

这些问题的原始自然语言描述并不总能精确描述期望的问题。主要原因由如下几点：

- 许多描述本身并不包含所有信息，一些常识性的知识或领域相关概念的具体含义会被直接省略，而这些常识或概念对生成组合优化约束模型至关重要。例如，在鸡兔同笼问题中，省略了“每只鸡有 2 条腿，每只兔有 4 条腿”的常识；在四色地图问题中，省略了“图染色要求相邻结点颜色不同”的领域相关概念。让计算机自动理解所有常识、概念，几乎是不可能实现的。考虑到用户表达的便利性，翻译工具可以将最常用的图论元素、数学术语等硬编码为翻译规则，但其他的常识或概念仍需要用户给出详细说明。
- 许多描述存在语法错误、标点符号使用不规范等问题。这些问题虽然对于人来说不会影响理解，但会导致自然语言处理错误，最终无法生成正确的组合优化约束模型。
- 原问题经常包含复杂的输入输出交互格式，而组合优化约束模型本身并不能很好地支持格式化的输入输出。

针对这些存在的问题，用于实验的数据集内的自然语言描述在原描述的基础上做了一些小的修改，包括：补充常识或领域相关概念的具体说明，改正语法错误，简化输入输出格式。

图 5.1，5.2，和 5.3 分别展示了数据集中 40 个问题自然语言描述的数据分布情况，包括 (a) 符号数，(b) 关键词数，以及 (c) 提及数。其中，有 36 个问题的自然语言描述至少包含 20 个符号以及至少 3 个关键词；37 个问题的描述内至少包含 5 个提及。这些数值表明大部分问题的自然语言描述都是有一定复杂程度的，不能通过简单的翻译实现组合优化约束模型的自动生成。

表 5.1　来源于 MiniZinc 教程或标准测试集的 7 个问题

编号	题目名	题目描述
P_A	地图着色	Given edges of a graph with N nodes. There are three colors that can be used for coloring nodes. Determine the color of each node so that the colors of nodes of each edge should be different.
P_B	杂货店问题	Find the prices of 4 items so that the sum of prices is 711 and the product of prices is 711000000. The prices of items are in ascending order.
P_C	魔幻序列	Find a sequence of N numbers. Each number is equal to the count of (its index minus 1) in the sequence.
P_D	0/1 背包问题	Given the weights and values of N items, put a subset of items into a knapsack of capacity C to get the maximum total value in the knapsack. The total weight of items in the knapsack does not exceed C.
P_E	蛋糕烘培	A banana cake takes 250 flour, 2 bananas, 75 sugar and 100 butter, and a chocolate cake takes 200 flour, 75 cocoa, 150 sugar and 150 butter. The profit of a chocolate cake is 45 and the profit of a banana cake is 40. And we have 4000 flour, 6 bananas, 2000 sugar, 500 butter and 500 cocoa. The question is how many of chocolate cakes and banana cakes we should bake for the fete to maximize the total profit.
P_F	N 皇后	Determine the columns where N queens should be placed. Each column should be greater than or equal to 1 and less than or equal to N. Columns should be all different. For each column, the sum of its value and its index should be not equal to that of any other column; the difference of its value and its index should be not equal to that of any other column.
P_G	最短路	There is a graph with N nodes. Given the length of each edge between the nodes, find the shortest path from S to E.

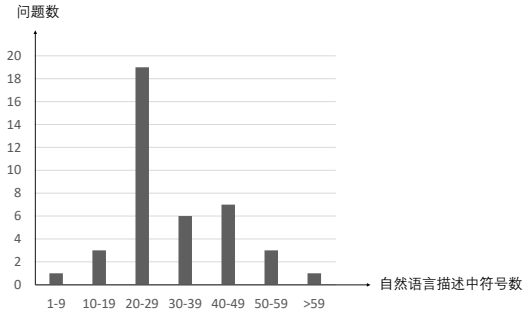

图 5.1　数据集中自然语言描述的符号数分布

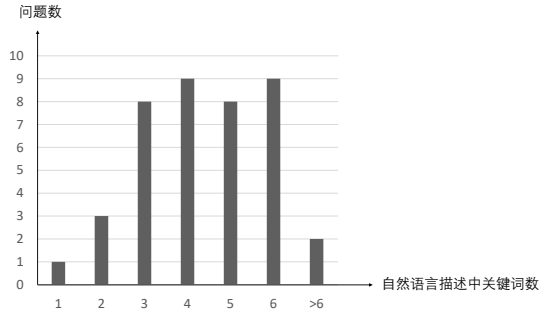

图 5.2　数据集中自然语言描述的关键词数分布

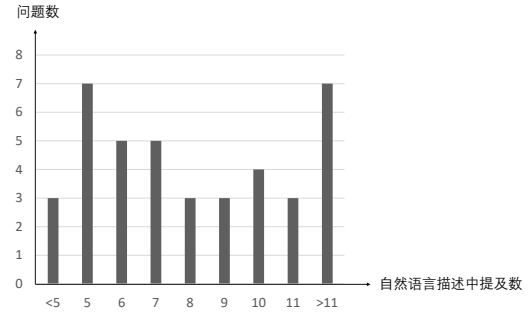

图 5.3　数据集中自然语言描述的提及数分布

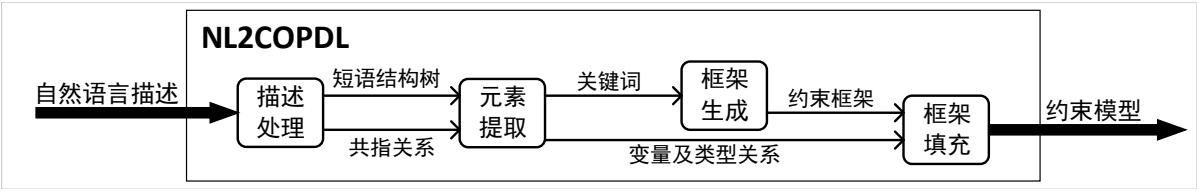

图 5.4　*NL2COPDL* 处理步骤

5.4　从自然语言描述自动产生 *COPDL* 模型的算法

如图 5.4 所示，*NL2COPDL* 通过四个处理步骤实现从英文自然语言描述生成等价的 *COPDL* 约束模型：

1. **描述处理**：采用前人工具进行自然语言输入的预处理（分词、分句、短语结构分析、依赖分析、共指消解）；
2. **元素提取**：提取关键词（包括 7 类共 91 个关键词）、提取变量（根据 2 条规则和 75 个树形关键词模板），并对变量进行类型推导；
3. **框架生成**：根据预定义的 36 个树形结构模板及翻译规则，从自然语言文本产生约束框架；
4. **框架填充**：根据变量及类型关系填充上一步得到的约束框架，使之成为完整组合优化约束模型。具体包括：（1）将框架中的每个抽象变量替换为实际变量，（2）区分参数变量和决策变量，并为每一个变量添加相应的声明语句。

本章将以 0/1 背包问题为例（5.4.1 节），依次阐述各处理步骤的实现算法（5.4.2

```
#input
    N of int in [1,100];
    weight of (int in [1,1000])[1~N];
    value of (int in [1,1000])[1~N];
    C of int in [1,1000];

#required
    subset1 of (int in [1,N]){};
    summation [weight[i] : forall i (i in subset1)] <= C;

#objective
    maximize summation [value[i] : forall i (i in subset1)];
```

图 5.5 *NL2COPDL* 自动产生的 0/1 背包问题的 *COPDL* 描述

节至 5.4.5 节）。

5.4.1 0/1 背包自然语言描述示例

为了便于讨论，本章以处理 0/1 背包问题的自然语言描述为例，展示 *NL2COPDL* 的实现细节。该问题的英文自然语言描述如下：

> Given the weights and values of N items, put a subset of items into a knapsack of capacity C to get the maximum total value in the knapsack. The total weight of items in the knapsack does not exceed C. 1 <= N <= 100, 1 <= C <= 1000, the weight of each item is positive and less than 1000, the value of each item is positive and less than 1000.

0/1 背包问题是一个典型的组合优化问题，它约束了放入背包的物品总重量不超过 *C*（“the total weight of items in the knapsack does not exceed C”），同时设定了优化目标为最大化放入背包的物品总价值（“get the maximum total value in the knapsack”）。最后一句关于各输入参数的范围说明，并非问题描述原本就有的，而是为了在 *COPDL* 描述中产生各参数变量正确的上下界额外添加的。在之前第 3.1 节中曾提到，参数变量的取值范围对于从 *COPDL* 描述产生程序尤为重要。图 5.5 是 *NL2COPDL* 工作从该自然语言描述自动产生的正确 *COPDL* 模型，与图 3.18 所示的人工书写的 *COPDL* 描述基本等价。

用户在没有自动化工具辅助的情况下，需要手工编写类似图 5.5 的 *COPDL* 模型，才能让通用求解器理解问题并自动完成求解工作。具体来说，用户需要指定约束模型中的参数变量（即本例中的 “N”，“C”，“weight” 和 “value”），决策变量（即本例中的“subset1”），各变量类型及取值范围，约束关系，目标函数等。这项工作对于普通

用户来说，有时是比较困难且耗时耗力的。本章的研究目的，就是为了降低约束建模的复杂度，帮助用户更好地利用约束编程求解问题。

5.4.2　描述处理

对于用户提供的自然语言问题描述，*NL2COPDL* 首先调用工具 CoreNLP（Manning et al 2015） 4.1.0 对输入进行自然语言预处理，获得语法结构树（constituent tree）以及实体之间的共指关系（coreference）。*NL2COPDL* 主要利用了 CoreNLP 的以下五个步骤完成预处理工作。

1. **分词（tokenization）**：将英文文本划分为符号（单词，标点等）序列。
2. **分句（sentence splitting）**：利用标点符号，将符号序列划分为独立的句子。
3. **短语结构分析（constituency parsing）**：对于每个句子，使用概率上下文无关文法（probabilistic context-free grammar，PCFG）分析器（D. Klein et al 2003） 产生**短语结构树（constituent tree）**。图 5.6 是 0/1 背包问题自然语言描述中第一句的短语结构树，其直观展示了句子的语法结构。树中的各叶子结点是原文本中的一个符号及其词性（part of speech，PoS）标记，例如“weights”的词性为 NNS（名词复数形式），“of”的词性为 IN（介词）。每个非叶子结点则是一个可作为独立成分的短语结构，例如词性为 NP（名词短语）的短语“weights and values”。
4. **依赖分析（dependency parsing）**：对于每个句子，通过分析语法结构，得到符号之间的依赖树（dependency tree）（Chen et al 2014）。图 5.7 是 0/1 背包问题自然语言描述中第一句的依赖树。树中每条边从一个短语的中心词指向它的依赖词。每个短语的中心词决定了这个短语的语法类型，而其他依赖词则是对中心词的修饰或限定（Nichols 1986）。
5. **共指消解（coreference resolution）**：在自然语言文本中，一个提及（mention）是对一个实体或物体的指代或表示，其表现形式可以为名词、名词短语或名词从句（Peng et al 2015）。若两个提及指代同一个实体，那么它们之间存在共指关系（coreference）。在自然语言转化为约束模型的任务中，共指消解的主要目的是将多次出现的同一实体链接在一起，确保每个变量与实体间的一一对应。图 5.8 展示了 0/1 背包问题自然语言描述中的 14 个提及（用圆圈表示）和 7 对共指关系（用实线表示）。

5.4.3　元素提取

利用在描述处理步骤中从 CoreNLP 获得的短语结构树和共指关系，*NL2COPDL* 从中提取出关键词、变量以及变量的数据类型。

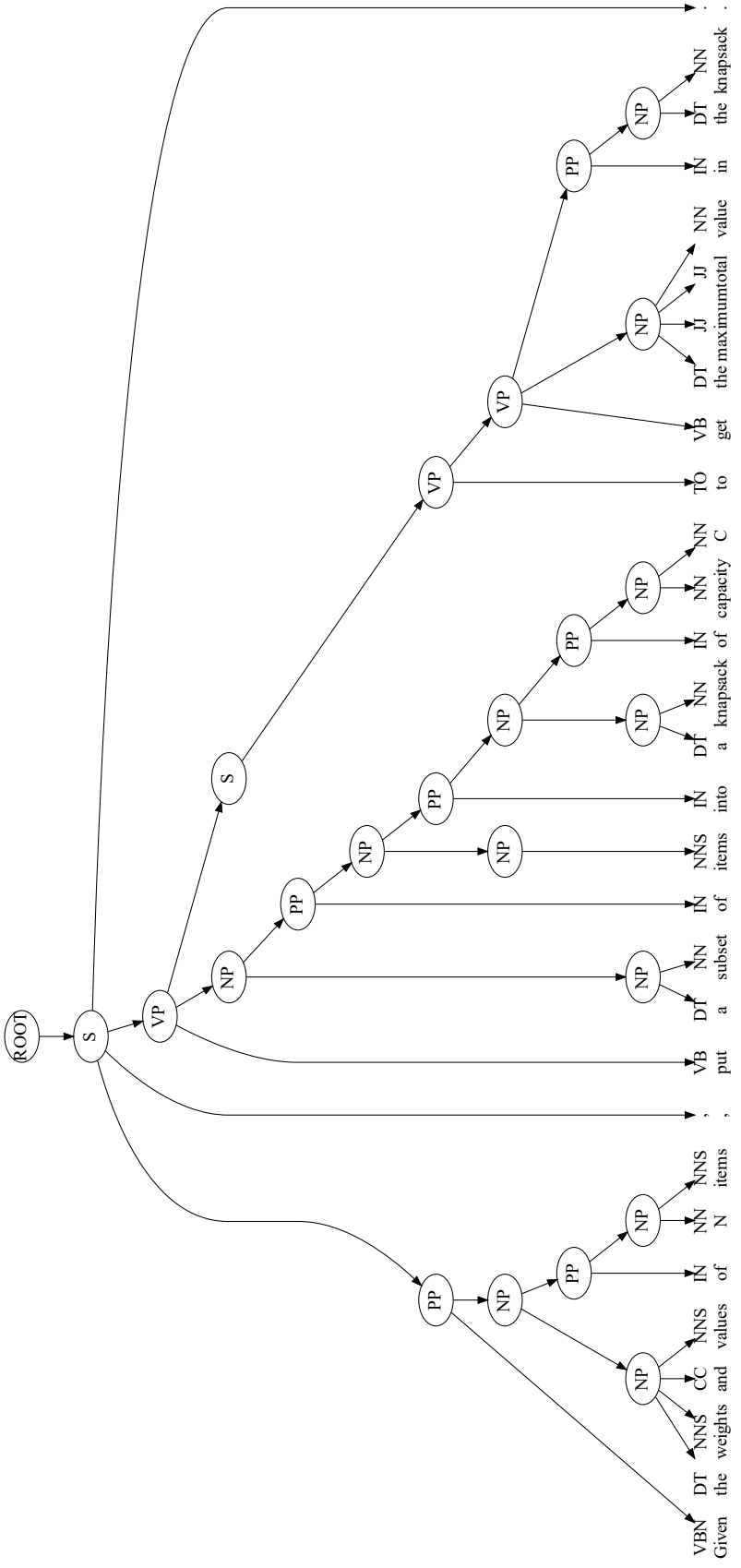

图 5.6 0/1 背包问题自然语言描述中第一句的短语结构树

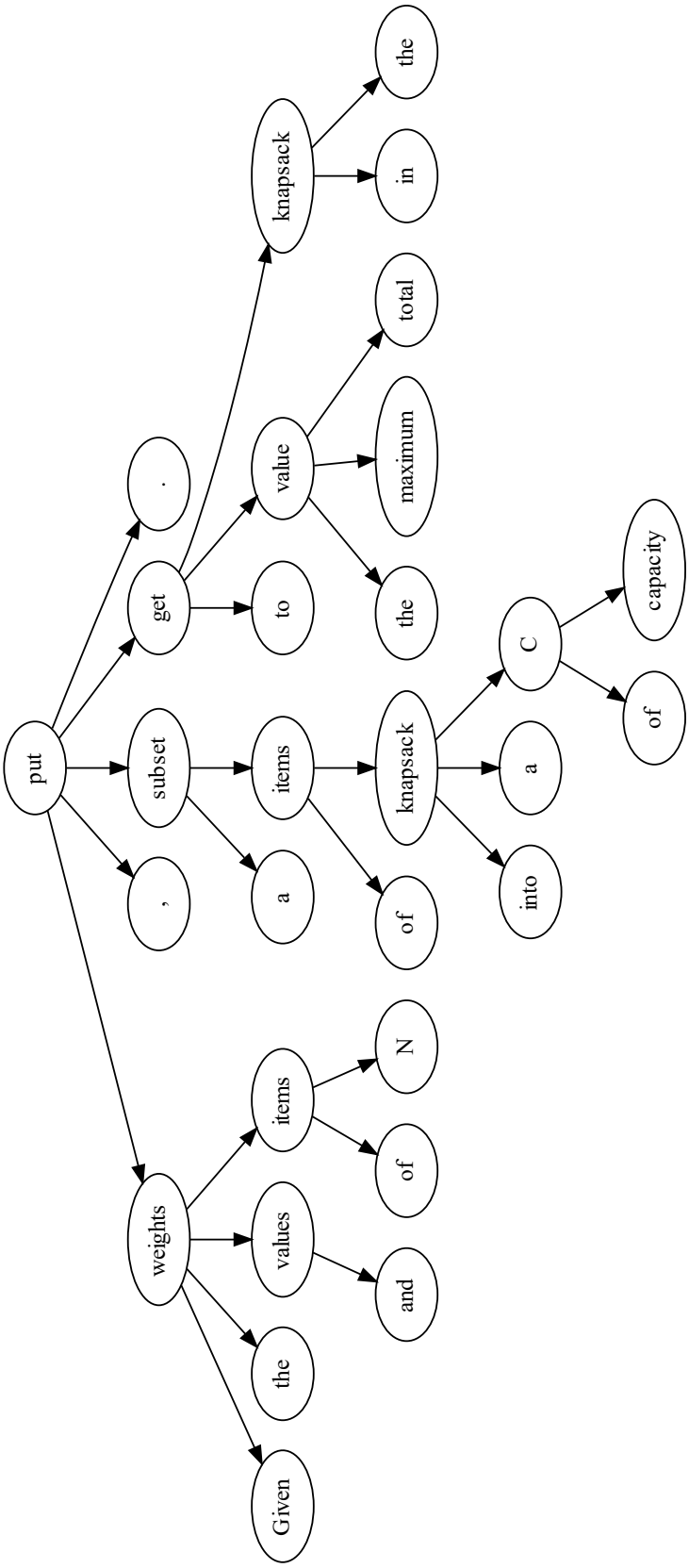

图 5.7　0/1 背包问题自然语言描述中第一句的依赖树

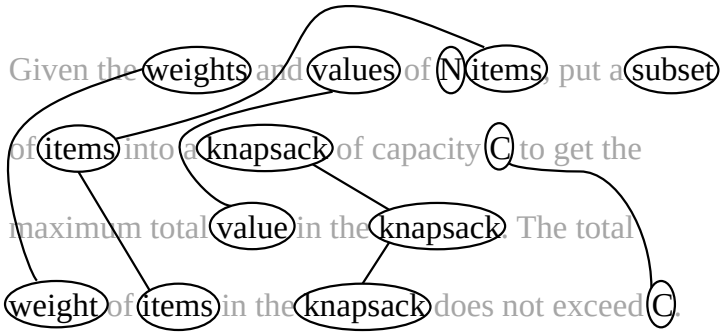

图 5.8　0/1 背包问题自然语言描述中的提及和共指关系

表 5.2　*NL2COPDL* 预定义的关键词

类别	描述	示例	数量
常量	使用单词或词组表示的数词	one, fifteen, hundred, thousand	27
基本操作符	表示运算符的词语	less than, not, plus	23
目标指示符	指示优化目标的词语	maximum, shortest, best	13
复杂函数	累积函数和全局约束	sum of, number of, all different	5
图论元素	常用的图论术语	node, edge, path, length	7
数学术语	常用的数学术语	odd, prime, multiple	11
结构指示符	指示数据结构的词语	each has, set of, NNS of NNS	6
总计			91

5.4.3.1　关键词提取

NL2COPDL 预定义了一些领域相关的、在自然语言问题描述中经常被使用到的关键词，这些关键词对于理解描述的语义起到了至关重要的作用。利用基于关键词的规则和模板，*NL2COPDL* 可以识别出不同文本块之间的关系，进而生成正确的约束模型。

如表 5.2 所示，*NL2COPDL* 中预定义的关键词可分为 7 大类。

- **常量**：使用单词或词组表示的数词。*NL2COPDL* 使用 WordToNum 算法（伪代码见图 5.9）将这类数词直接转化为数值，作为整型常量。
- **基本运算符**：用于描述变量与常量之间的大小关系、基本运算等操作符的词语，例如，“less than”表示“<’，“plus”表示“+”。*NL2COPDL* 的关键词汇表中包含 23 类描述这些基本运算常用的词语，这些单词或短语将会被用于对句子分块以及产生框架中的表达式。
- **目标指示符**：用于指示组合优化问题中的优化目标的词语，例如“maximum”表示问题的目标是最大化后续表达式的值。这些指示符将用于产生目标函数框

输入: 单词序列 $S = \{s_1, s_2, \ldots, s_n\}$
输出: 对应的数值 val

Function WordToNum *(S)*:
  $val \leftarrow 0$;
  $tmp \leftarrow 0$;
  foreach $i \in \{1, 2, \ldots, n\}$ **do**
    if $s_i \in$ *{"billion", "million", "thousand"}* **then**
      $val \leftarrow val +$ WordToValue$(s_i) \cdot tmp$;
      $tmp \leftarrow 0$;
    end
    else if $s_i =$ *"hundred"* **then**
      $tmp \leftarrow 100 \cdot tmp$;
    end
    else
      $tmp \leftarrow tmp +$ WordToValue(s_i);
    end
  end
  $val \leftarrow val + tmp$;
  return val;

图 5.9　数词转化为数值算法

架。

- **复杂函数**：用于描述累积函数、全局约束等组合优化约束模型中常见复杂函数的词语，例如"sum of"表示对一个序列或集合求和，"all different"约束序列中元素两两不同。这些单词或短语将用于生成组合优化约束模型内置的一些作用于序列或集合的常用复杂函数。
- **图论元素**：在图论问题中经常使用到的、具有特殊意义的词语，例如"edge"表示一个结点对，"path"表示一个结点序列且序列中相邻结点有边相连。这些单词或短语将会被根据预设知识翻译为相应的数据结构以及约束框架。
- **数学术语**：常用的表示数学概念的词语，例如"odd"和"even"分别表示奇偶数，"prime"表示质数。这些单词或短语将会被翻译为显式的数学约束作为框架的一部分。
- **结构指示符**：用于表示复杂数据结构的词语，例如"set of"指示了主语是一个集合类型，"NNS_1 of NNS_2"（"NNS"表示名词的复数形式）则需要对"NNS_1"创建一个长度等于"NNS_2"数量的数组，来记录每个元素的值。这些单词或短语将被用于构造具有集合、数组等复杂数据类型的变量。

5.4.3.2　变量提取

正如图 5.5 所示，一个组合优化约束模型可以看作以参数变量值为输入、决策变

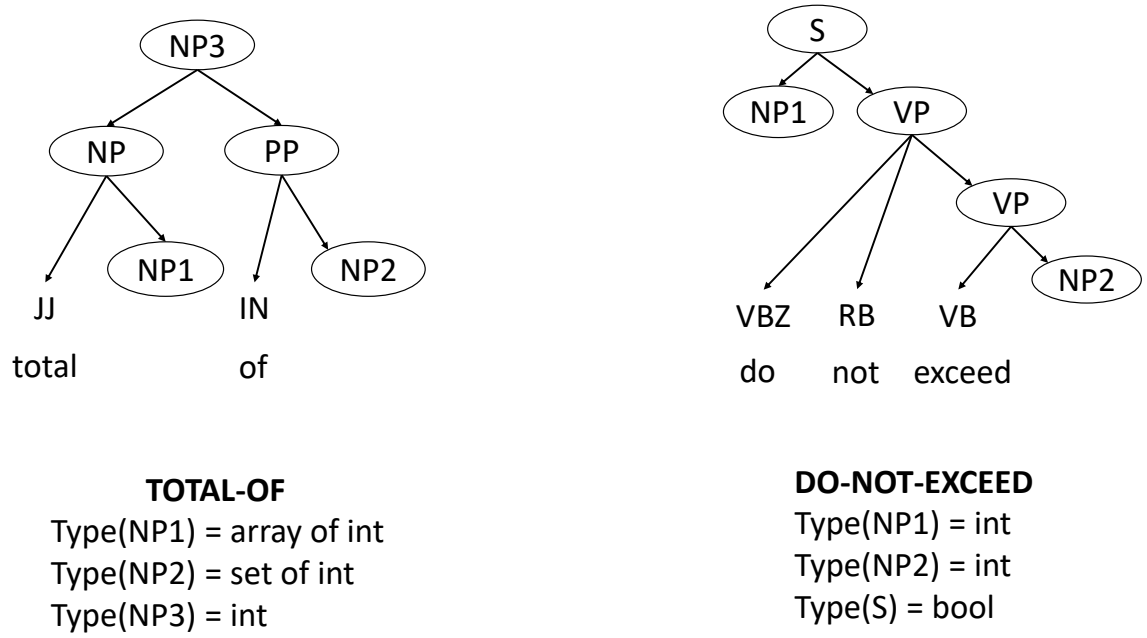

图 5.10　TOTAL-OF 和 DO-NOT-EXCEED 模板和类型推导逻辑

量值为输出的一个程序。为了进行约束建模，*NL2COPDL* 首先需要在自然语言描述中识别出所有的变量（包括参数变量和决策变量）。根据自然语言的性质和用户的使用习惯，*NL2COPDL* 用如下两条规则寻找候选变量：

- 若自然语言描述的某个短语与某个预定义的关键词模板匹配，而这个关键词要求其中某个或某些名词或名词短语必须是变量或表达式（例如“NN_1 is less than NN_2”要求“NN_1”和“NN_2”均为变量或表达式），如果相应名词或名词短语不可再分，那么它就应该被提取出来作为候选变量。
- 若一个名词很短（由不超过三个字母组成），且全为大写（例如“N”，“M”）或无意义（例如“val”），那么它将被提取出来作为候选变量。

NL2COPDL 预定义了 75 条关键词模板用于判定名词或名词短语是否是候选变量，其中每个模板均表示为树形结构。若短语结构树的某个子树匹配某个模板结构，则可以根据该模板的规则提取候选变量。图 5.10 展示了 TOTAL-OF 和 DO-NOT-EXCEED 两个模板的结构。其中，每个模板包含一些关键词（例如“total of”和“do not exceed”），以及用于匹配名词或名词短语的非终结符（例如“NP”）。这些非终结符所对应的名词或名词短语若不可再分，就将会被提取为候选变量。

对于每个从自然语言描述中发现的关键词，*NL2COPDL* 尝试使用相关的模板匹配相应子树。如果匹配成功，*NL2COPDL* 将子树中相应的候选变量提取出来。如图 5.11 所示，文本“total weight of items in the knapsack”成功匹配模板 TOTAL-OF 模板，因此 *NL2COPDL* 可以将“weight”和“items in the knapsack”提取出来作为候选变量。提取变量

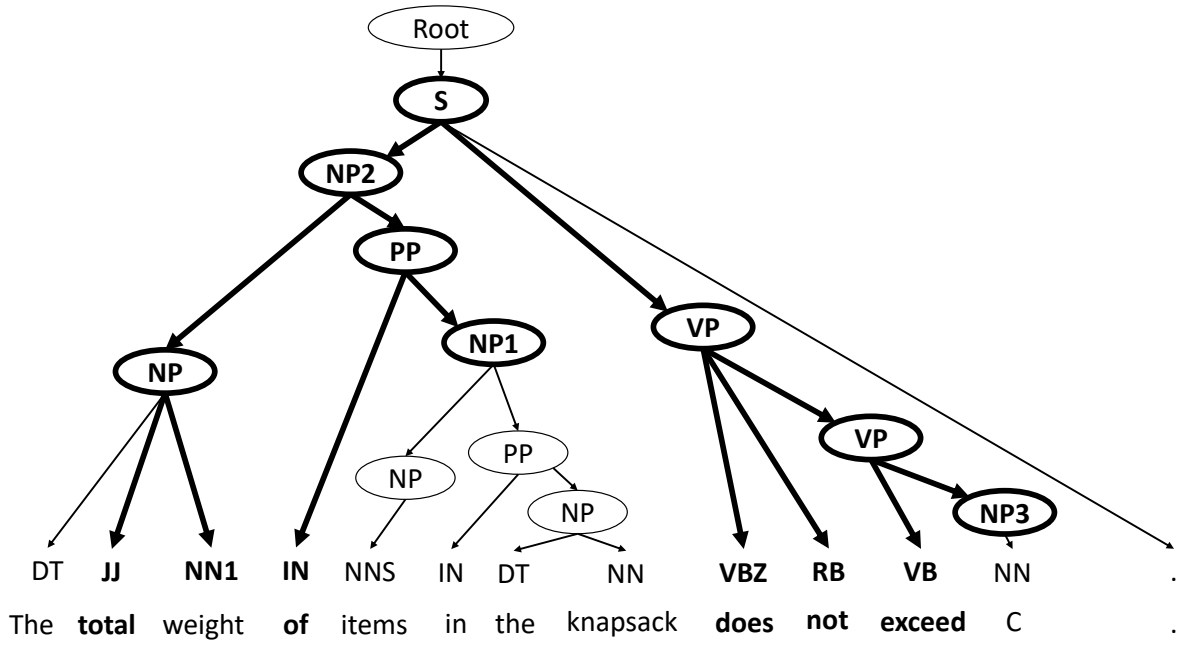

图 5.11　模板匹配示例（分别匹配 TOTAL-OF 与 DO-NOT-EXCEED 模板）

的匹配算法基于树形结构模板而非线性结构模板，因此可以根据句子的语法结构，有效避免误匹配的同时，使用自底向上的方式实现递归搜索匹配。

5.4.3.3　类型推导

上述 75 个模板除了定义了哪些非终结符需要作为变量或表达式存在，还规定了它们的数据类型。因此，在使用这些模板对自然语言描述进行匹配时，*NL2COPDL* 不但提取了变量，还进一步分析并推导了各变量、表达式的数据类型。如图 5.11 中的 TOTAL-OF 模板，就限定了终结符“NP1”，“NP2”，“NP3”分别为整数数组类型，整数集合类型，以及整数类型。对于文本“total weight of items in the knapsack”来说，“weight”和 “items in the knapsack”就会分别被推导为整数数组类型和整数集合类型，而整个短语所对应的表达式则为整数类型。

对于被推导为数组或集合两类具有复合类型的变量，由于组合优化约束模型要求数组长度和集合范围在变量声明时作为参数传入（参见图 5.5），因此需要额外推导数组长度或集合范围值。在推导数组长度或集合范围值时，*NL2COPDL* 首先检查数组类型变量的主体或集合类型变量的全集在自然描述中的提及是否被数词或变量约束，如果找到这样的数词或变量，则将其作为数组长度或集合范围值；否则，遍历共指关系图，检查主体或全集的其他共指提及是否被数词或变量约束。在 0/1 背包问题示例中，通过短语“the weights and values of N items”识别出的数组变量“weight”的主体是“items”，而“items”的数量被变量“N”约束，因此“N”即为数组“weight”的长度；通过短语“a subset

of items”识别出的集合变量“subset1”的全集也是“items”，此处的“items”并没有被数词或变量约束，*NL2COPDL* 通过遍历“items”的其他共指提及，得知“items”的数量为“N”，因此全集的范围也被设为 1–*N*。

在完成对每个被模板匹配的子树上的变量和非终结符的类型推导后，*NL2COPDL* 利用共指关系图和短语结构树传播数据类型并迭代地更新直至所有变量和非终结符地类型均不再变化。如果一个候选变量仍然没有获得任何类型，则默认将其限定为整数类型。

在类型推导过程中，如果产生了类型冲突（例如具有共指关系的两个提及本应是同一变量，但却被推导为不兼容的两种类型），*NL2COPDL* 将向用户报告错误，提醒用户检查自然语言描述是否有误或存在歧义。

5.4.4 框架生成

一个**框架（sketch）**是一个不完整的表达式，可以是一个约束，目标函数，或者它们的子表达式。由于 *NL2COPDL* 最终输出 *COPDL* 语言表示的组合优化约束模型，本章讨论的框架均为不完整的 *COPDL* 表达式。一个框架具有如下特点：

- 包含表达式的完整结构；
- 可能包含一个或多个抽象变量；
- 全局变量没有具体变量名。

对于一个自然语言描述，*NL2COPDL* 将会构造相应的包含一个或多个框架的框架模型。在这一过程中，*NL2COPDL* 同样以模板驱动的方式实现自然语言描述到框架模型的翻译工作。为了实现这项任务，*NL2COPDL* 预定义了 36 个树形结构模板及翻译规则。这些模板有一部分与类型推导步骤中使用的模板相同，但服务于不同的目标——用于产生框架而非提取变量或推导类型。

NL2COPDL 以自底向上的方式，在短语结构树上尝试进行模板匹配。若子树 *st* 匹配了某个模板，*NL2COPDL* 将生成一个框架取代 *st*。接下来，若 *st* 的某个祖先结点 *pt* 匹配了某个模板，*NL2COPDL* 同样生成一个框架取代 *pt* ，其中 *st* 对应的框架将成为 *pt*对应框架的一部分。这种“匹配-替换”的过程将被不断重复直至不再有新的子树被匹配，此时，树中所有框架均被作为独立语句加入框架模型。例如，基于图 5.11 的模板匹配结果，“NN1”，“NP1”，和“NP3”作为不可再分的提及，分别被替换为抽象变量 `$V1`，`$V2`，和`$V3`；之后，*NL2COPDL* 根据 `TOTAL-OF` 模板定义的翻译规则，将以“NP2”为根的子树翻译成求和表达式，其中子结点“NN1”和“NP1”被相应的框架（抽象变量）所替代；最后，*NL2COPDL* 将以“S”为根的子树根据 `DO-NOT-EXCEED` 模板的翻译规则，翻译为比较两个表达式大小关系的框架，由于不再有更多子树被匹配，该框架作为独立的约束框架加入最终的框架模型。整个框架转换和生成过程如图 5.12 所

```
SK(NN1) = $V1
SK(NP1) = $V2
SK(NP3) = $V3
SK(NP2) = sum (i in SK(NP1)) (SK(NN1)[i])
        = sum (i in $V2) ($V1[i])
  SK(S) = SK(NP2) <= SK(NP3)
        = sum (i in $V2) ($V1[i]) <= $V3
```

图 5.12　基于 TOTAL-OF 模板和 DO-NOT-EXCEED 模板的框架生成过程

示，其中 SK(N) 表示非终结符 N 所对应的框架。

在翻译过程中，若某个提及所对应的原始文本为常量数值，则这个数值将直接填入框架的相应位置；若模板对各变量的数据类型有约束，则这些类型约束将会转换为对应抽象变量之间的类型约束，保存用于后续分析。

5.4.5　框架模型填充

通过框架生成步骤得到框架模型后，*NL2COPDL* 根据变量及类型关系填充框架，使之成为完整组合优化约束模型。具体来说，框架模型填充分为两个部分：（1）将框架中的每个抽象变量替换为实际变量；（2）区分参数变量和决策变量，并为每一个变量添加相应的声明语句。

5.4.5.1　基于框架的模型自动生成

对于任意一个框架 *s*，*NL2COPDL* 首先从 *s* 对应的原自然语言描述文本中找出相关变量，作为填空的备选。相关变量包括直接出现在该文本中的变量，以及出现在该文本中的某个名词的其他共指提及的修饰变量（例如大小，子集等）。*NL2COPDL* 从相关变量集合中搜索具体变量来替换 *s* 中的每个抽象变量，替换要求满足如下约束条件：

- **有序**：如果 *s* 中的一个多元运算符 *o* 是顺序敏感的（比如“<”，“-”等），那么该运算符的操作数的出现顺序必须与原自然语言描述中对应提及的出现顺序保持一致。
- **类型正确**：*s* 中每个运算符的操作数类型必须符合类型要求；同一变量的类型不能存在冲突。

上述寻找框架填充方案的问题本质上就是一个约束满足问题。*NL2COPDL* 自动将框架填充问题建模为 *COPDL* 组合优化约束模型，并调用通用求解器 Gecode 找到填充方案。若存在多个不同方案，*NL2COPDL* 自动选择变量在框架中的填充顺序与变量提及在原自然语言描述中的出现顺序差异最小的，即逆序对数最少的一种填充方案。

5.4.5.2 区分参数变量与决策变量

将框架内的所有抽象变量全部替换为具体变量后，实际上已经得到了所有的约束和目标函数。但目前的模型还并不完整，因为求解器无法根据模型判断那些变量的值是由用户指定的（参数变量），那些变量的值是需要被求解输出的（决策变量）。

NL2COPDL 采用贪心的方法，根据下列规则不断迭代将所有变量分类，直至无法再应用任何规则确定变量类别：

- 若变量 v 在原自然语言描述中的某个提及是“given”，“there be”等单词或短语的宾语，那么 v 是一个参数变量；另一方面，若 v 的某个提及是“find”，“assign”等单词或短语的宾语，那么 v 是一个决策变量。
- 若变量 v 在所有的约束和目标函数中只出现过一次，那么 v 是一个参数变量。
- 若变量 v 决定了一个数组的长度或集合的范围，那么 v 是一个参数变量。
- 若变量 v 是一个子集（“subset”）或路径（“path”）等，那么 v 是一个决策变量。
- 每个约束或目标函数通常至少包含一个决策变量。因此，如果一个约束或目标函数包含 N 个变量，且其中 $(N-1)$ 个变量已经被定为参数变量，那么剩余的那个变量是一个决策变量。

迭代完成后，所有未被分类的变量均视为决策变量。

最终，*NL2COPDL* 将每个变量的声明语句插入模型，便得到完整的 *COPDL* 模型。如果某个（些）变量被声明却没有在约束和目标函数中使用过，*NL2COPDL* 会生成警告信息反馈给用户，用户可以依此来检查自然语言问题描述是否存在拼写错误、语法错误等常见问题。

5.5 *NL2COPDL* 正确性和性能测试实验设计与结果

本节在自建数据集（见 5.3 节）上评价 *NL2COPDL*，第 5.5.1 节实验需要研究的问题，第 5.5.2 节展示相应的实验结果并进行分析。

5.5.1 研究问题及实验设计

本节中将设计实验来研究以下三个问题，以此来评价 *NL2COPDL* 的好坏：

1. ***NL2COPDL* 能否高效地从自然语言描述自动产生组合优化约束模型？**实验通过测量 *NL2COPDL* 对数据集中自然语言描述进行处理和转换的时间开销来评价 *NL2COPDL* 的运行效率。
2. ***NL2COPDL* 产生的组合优化约束模型的正确性如何？**实验通过 *COPDL2C* 产生求解程序，并使用预先设计的测试数据来验证每个组合优化约束模型是否正确。具体来说，对于数据集中的每个问题，都事先准备了 1–5 个具有代表性的

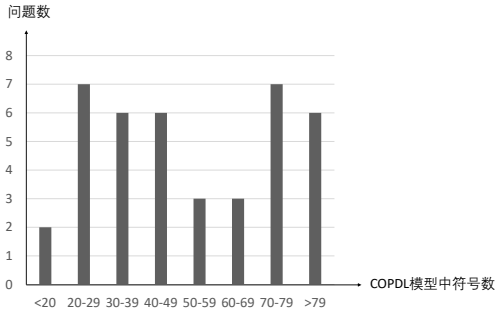

图 5.13　数据集中自然语言描述对应 *COPDL* 模型的符号数分布

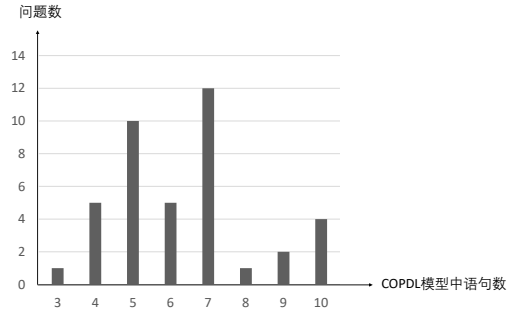

图 5.14　数据集中自然语言描述对应 *COPDL* 模型的语句数分布

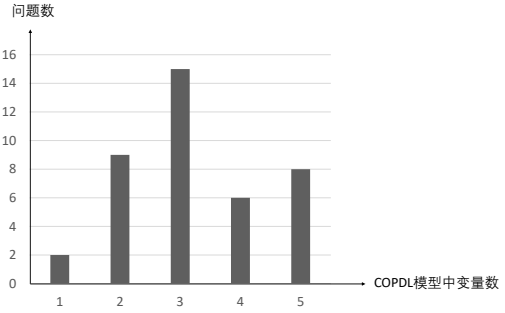

图 5.15　数据集中自然语言描述对应 *COPDL* 模型的变量数分布

测试数据，每个测试数据包括输入（参数变量值）和期望的标准输出（决策变量值）。将自动生成的组合优化约束模型交给 *COPDL2C* 产生求解程序，将每个测试数据的输入分别传入求解程序，求解得到最优的决策变量值，再将得到的决策变量值与标准输出进行比较。一个组合优化约束模型是正确的，当且仅当相应的求解程序在该问题所有测试数据的输入上都能求得期望的输出。

3. ***NL2COPDL* 产生的组合优化约束模型的质量如何？**由于数据集中 7 个来自 MiniZinc 手册或标准测试集的问题存在由专家设计的标准组合优化约束模型，实验通过比较 *NL2COPDL* 自动生成的组合优化约束模型以及专家编写的模型在自动求解效率上的差异，来评价自动生成的模型的质量。

5.5.2　实验结果

整个实验均在同一台普通个人电脑（CPU：Intel Core i5-7300HQ 2.5GHz，内存：8G）上进行。

对于数据集中的每个问题，*NL2COPDL* 均能生成语法正确的完整 *COPDL* 模型。图 5.13，5.14，和 5.15 分别展示了生成的 *COPDL* 模型的数据分布情况，包括 (a) 符号数，(b) 语句数，以及 (c) 变量数。其中，有 38 个模型至少包含 20 个符号。每个模型都包含 3–10 个 *COPDL* 语句以及 1–5 个变量。这些数值表明，大部分的模型都具有一

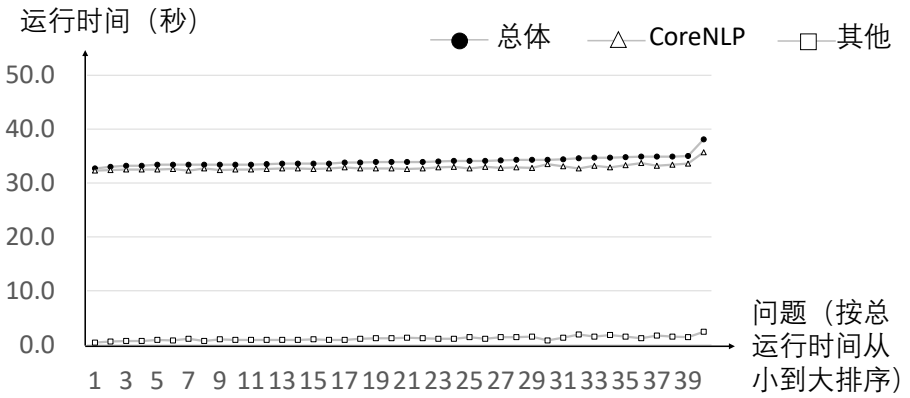

图 5.16　*NL2COPDL* 产生 *COPDL* 模型的运行时间

定的复杂性。

图 5.16 展示了 *NL2COPDL* 对数据集中每个问题产生相应 *COPDL* 模型的运行时间。对于每个问题，*NL2COPDL* 大约需要花费 32.7–38.1 秒的时间完成组合优化约束模型的自动生成工作，其中 93.7%–98.8% 的时间花费在了 CoreNLP 工具处理自然语言描述产生短语结构树和共指关系图上，后续的分析和生成工作只花费了总时间的 1.2%–6.3%。

结论 5.1：*NL2COPDL* 在分析自然语言描述和生成组合优化约束模型上具有较高的效率。

对于每个生成的组合优化约束模型，实验使用相应问题预先设计的所有测试数据测试其能否求出正确结果。实验表明，这些组合优化约束模型均能够顺利通过所有测试，即 *NL2COPDL* 对数据集中的 40 个问题的自然语言描述均能够产生正确的组合优化约束模型。图 5.17 展示了 *COPDL2C* 求解这些组合优化约束模型所需要的平均求解时间。其中，有 12 个模型能够在 1 秒内被求解，15 个模型在 1–10 秒内被求解，10 个模型在 10–100 秒内被求解，只有 3 个模型需要花费平均超过 100 秒才能被求解。

结论 5.2：对于数据集中的所有问题，*NL2COPDL* 均能生成正确的组合优化约束模型，且大部分模型对应的由 *COPDL2C* 产生的求解程序具有较高的求解效率。

表 5.1 列举了来源于 MiniZinc 教程或标准测试集的 7 个问题。由于这些问题本身没有对参数变量的范围限制，因此在 *NL2COPDL* 产生的 *COPDL* 模型上，人工将参数变量声明中描述上下界的“?”替换为测试实例中相应参数变量的最小/最大值。在这些问题上，实验比较了自动生成模型与人工设计模型在求解上的时间开销，结果见表

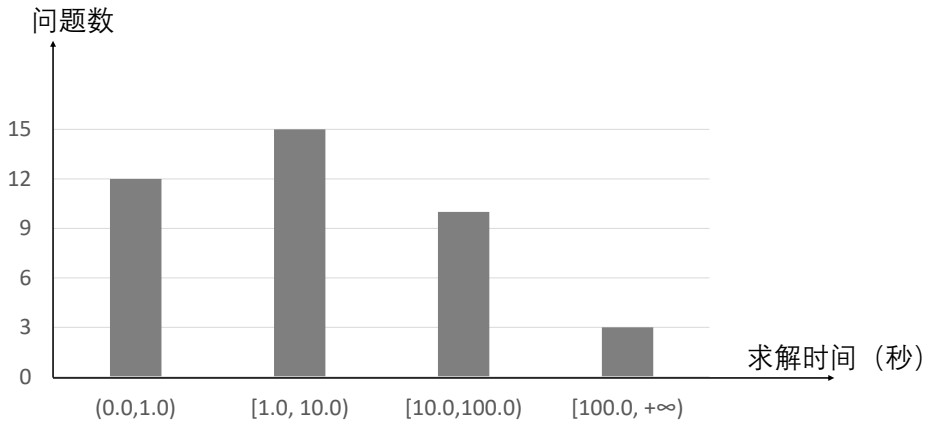

图 5.17　*COPDL2C* 求解产生的 *COPDL* 模型的平均时间

表 5.3　自动生成模型与人工设计模型的平均求解时间（秒）对比

题目编号	自动生成模型	人工设计模型
P_A	7.3	7.3
P_B	2.7	2.4
P_C	3.7	3.6
P_D	9.4	9.3
P_E	>100	1.4
P_F	47.8	22.6
P_G	>100	12.5

5.3。对于 P_A–P_D，自动生成模型与人工设计模型的表现基本一致，更进一步说，两种模型只有在变量名、语句顺序上略有差别，这些差别并不会对求解时间造成太大的影响。不过，对于 P_E–P_G，自动生成模型的求解时间要明显多于人工设计模型的求解时间。经过仔细研究发现，造成自动生成模型低效的原因可分为两点：

- *NL2COPDL* 缺乏对于决策变量取值范围限制的常识或领域知识。这类知识的缺失虽然不会影响模型的正确性，但由于无法对取值范围作严格的限制，导致搜索空间过大，最终造成求解效率低下。例如，P_E 要求搜索出两种蛋糕的数量。由于 *NL2COPDL* 没有“数量不能小于 0”的常识，只能将数量的范围设置为默认范围 $-\infty$ 到 $+\infty$，使得搜索空间很大。而专家设计的模型限定了数量为非负，极大减少了搜索空间，求解效率也有显著提升。
- *NL2COPDL* 并没有对组合优化约束模型有针对性地应用额外的优化策略，而一些专家设计的模型则可以根据问题的特性对变量的表示、搜索策略等进行优化。例如，P_F 的模型可以通过将搜索策略设置为“first_fail”（优先尝试搜索定义域较小的变量）和“indomain_min”（对于每个变量，从小到大顺序尝试定义域内可行值），提前发现冲突和矛盾，尽早舍弃不可行的分支，减少搜索量，从而提高搜索效率；在 P_G 中，使用边集合而非结点列表来表示路径，利用这种更有利于求解当前问题的特殊的数据结构，可以提升求解效率。这两种结合专家知识优化得到的模型，在求解效率上都要远远高于未经任何优化的模型。

结论 5.3：*NL2COPDL* 自动产生的组合优化约束模型有时会比专家设计的模型低效，主要原因是缺少常识或领域知识，或者没有应用领域相关的模型优化策略。这些缺点对于自动生成来说是难以克服的。

5.6 本章小结

本章介绍了 *NL2COPDL*，一种从自然语言问题描述自动产生组合优化约束模型的方法和工具。*NL2COPDL* 的创新之处包括：

1. *NL2COPDL* 是首个尝试将自然语言描述翻译成组合优化约束模型的工具。这种自动的模型生成工具，可以有效降低约束编程求解约束满足问题和组合优化问题的难度。
2. *NL2COPDL* 利用共指关系联系同一变量的不同提及，从而实现多条相关语句的正确生成。
3. *NL2COPDL* 结合了基于模板规则的翻译技术以及基于框架的翻译技术，以提高翻译的准确性。

这项研究课题非常具有挑战性，因为在自然语言描述中，组合优化的约束和优化目标的表达方式是非常灵活的：（1）有大量的领域特定关键词；（2）形式多样的名词或名词短语；（3）名词或名词短语之间具有复杂的共指关系。为了克服这些困难，*NL2COPDL* 采用了（1）自然语言处理，（2）基于模板的元素提取，（3）基于关键词和规则的框架生成，（4）基于框架的模型自动生成等技术，最终实现了从自然语言描述自动产生组合优化约束模型。实验结果表明，*NL2COPDL* 有能力对多种问题的自然语言描述生成正确的组合优化约束模型，虽然某些自动产生模型会比专家设计并优化过的模型在求解上要低效一些。

在后续工作中，可以通过多种途径提高 *NL2COPDL* 的适用性。例如，设计并实现开放接口，允许用户根据自身的需求和领域特定知识，添加自定义的关键词和翻译规则；使用机器学习技术，从错误中学习，自动扩充关键词库，完善规则；尝试自动分析问题特性并有针对性地应用高级搜索策略，以提高生成模型的求解效率。

第六章　问题建模训练平台 *COPDLOpenJudge* 的关键技术和应用

第三至五章介绍的 *COPDL2C* 和 *NL2COPDL* 这两个工具，将从问题的自然语言描述到生成问题求解程序的完整求解过程自然地分成问题建模和模型求解两个阶段。这样的任务分解在程序员的培养中可以起到提高效率的作用。

在计算机专业学生的培养方案中，对组合优化问题求解能力的训练和考察，在离散数学、程序设计、算法设计与分析等课程的作业和考试中，均占有相当大的比重。在世界大学生程序设计竞赛（ACM/ICPC）中，来自各大高校的顶尖计算机专业学生们比拼的主要也是组合优化问题的建模和高效求解能力。然而学生学会编写高效的组合优化问题求解程序并非易事。研究表明，很多学生因为受挫而中途退出计算机专业的学习。在现行的教学方法中，通常是要求学生从自然语言描述的问题直接编写出求解程序。此过程叠加了问题建模和模型求解两阶段的难度。如果能将这两个阶段分开来训练，则可极大降低学习的难度。另外，现有的教学方法中鲜有对建模能力的单独训练，而建模能力是决定学生能否学以致用的关键能力。

因此，本章基于 *COPDL2C* 设计开发了一个在线评测平台 *COPDLOpenJudge*，用于提升计算机专业和非专业程序员的培养效率。平台上放置了第五章第 5.3 节所描述的自建的自然语言描述的组合优化问题数据集。在传统的求解流程下，用户通过阅读自然语言描述的问题，编写并提交问题的 *COPDL* 描述来直接求解相应问题。而在基于 *COPDLOpenJudge* 的教学中，对于非计算机专业的程序员来说，只需要学会使用 *COPDL* 描述问题，就可以通过 *COPDL2C* 获得相应问题的 C 代码求解程序；对于计算机专业的程序员来说，可以通过 *COPDL* 单独训练问题的建模方法，同时也可以通过阅读由 *COPDL2C* 产生的具有一定可读性的 C 代码学习针对问题模型的高效求解算法。由此，在 *COPDLOpenJudge* 平台上，一个复杂的学习任务被分解成了两个相对较小的任务。实验表明，这样的任务分解可以使学生提振信心并提升学习效率。

6.1　程序设计类与算法分析类课程中的问题求解能力

在程序设计类和算法分析类课程中，**问题求解能力**是学生能够取得优秀表现的关键能力之一。先前的研究表明，计算机科学专业的高失败率，很多时候是由于初学者缺乏问题求解能力导致的（Beaubouef et al 2005，Butler et al 2007）。Tu et al（1990）研究发现，学生可以通过编程任务训练和提高问题求解能力。然而，如果这些任务过

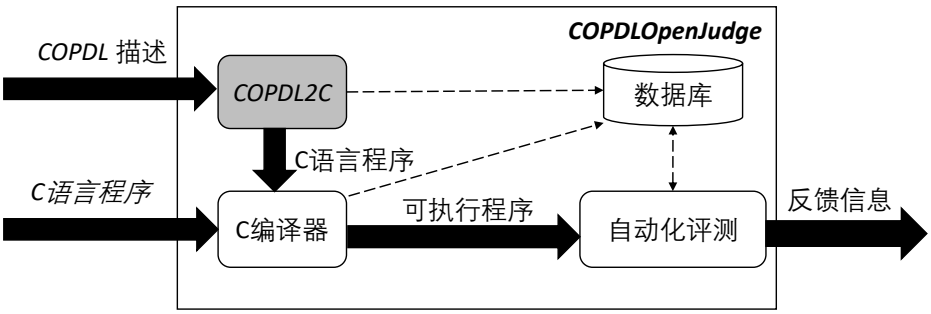

图 6.1 *COPDLOpenJudge* 技术架构

于困难，学生会丧失解决这些任务的信心和动力，反而失去了训练问题求解能力的机会，甚至对编程产生负面情绪。

根据之前程序设计类和算法分析类课程的教学经验以及与学生的交流发现，组合优化问题在编程任务里属于较难的一类问题。其原因主要有如下两点：

- 学生有时会遗漏题面中的重要条件，或者错误理解题目意义，导致最终编写的程序不符合题意；
- 学生正确理解了题意，但在设计和实现程序的时候犯错导致结果不正确。

在提交程序却被判定为不正确时，学生们往往会觉得很沮丧，特别是当无法判断错误是源于题意误解还是编码失误时，在修改时会觉得无从下手。

本章尝试在程序设计类和算法分析类课程的教学中引入 *COPDL* 语言以及相关的 *COPDL2C* 工具，辅助学生完成编程任务，从而使问题求解能力得到充分锻炼。作为一个“**支架式（scaffolding）**”教学工具，*COPDL* 将初学者还没有准备好面对的困难问题（比如算法设计、程序调试）暂时隐藏起来，先训练学生形式化描述问题的能力，直至学生有足够的信心和能力面对困难问题，再让学生舍弃工具，独立完成编程任务。

6.2 问题建模训练平台 *COPDLOpenJudge* 的设计与实现

本节介绍了问题建模训练平台 *COPDLOpenJudge* 的整体架构，使用方法，以及能够提供给用户的错误提示与反馈信息内容。

6.2.1 问题建模训练平台 *COPDLOpenJudge* 的技术架构

为了让学生能够更方便地使用 *COPDL* 语言，*COPDL2C* 工具被集成到北京大学在线评测系统（OpenJudge，`http://openjudge.cn/`）上，形成完整的 *COPDL* 语言在线学习、模型评测平台 *COPDLOpenJudge*（`http://pdl.openjudge.cn/`）。

表 6.1　评测结果分类及含义

评测结果	缩写	含义
Accepted	AC	程序通过测试（正确且高效）
Wrong Answer	WA	程序输出结果错误
Time Limit Exceeded	TLE	程序运行时间超出规定时间上限
Compile Error	CE	程序编译错误，无法生成可执行文件
Runtime Error	RE	程序在执行时抛出了未处理的异常
Presentation Error	PE	程序基本正确，只是输出格式与标准输出略有不同
Memory Limit Exceeded	MLE	程序使用内存超出规定内存使用上限
Output Limit Exceed	OLE	程序输出过量内容

在线程序评测系统（online judge，OJ）是用于判断程序的正确性的高效的自动评测系统。用户在使用在线程序评测系统时，首先阅读问题题面（自然语言），思考设计算法并编写程序，然后将程序源代码提交至系统。系统自动使用预先精心设计的一组或多组测试数据对用户提交的程序进行黑盒测试，以评估程序的正确性和效率。最终，根据运行结果，向用户反馈评测结果，评测结果主要分为 8 类，如表 6.1 所示。用户可根据评测结果修改自己的程序并重复提交。

图 6.1 展示了在既有在线程序评测系统上搭建的问题建模训练平台 *COPDLOpenJudge* 测试 *COPDL* 描述正确性的整体流程和技术架构。对于用户提交的 *COPDL* 描述，*COPDLOpenJudge* 首先使用 *COPDL* 的程序自动生成工具 *COPDL2C* 产生 C 程序，然后使用 C 编译器产生可执行文件。随后，自动化评测机使用预先设定的测试数据验证程序是否正确，并将评测结果返回给用户。同时，*COPDLOpenJudge* 也支持直接对 C 程序进行评测，从而方便进行对比实验。

图 6.2 展示了本章设计的 *COPDLOpenJudge* 关键操作的交互过程。学生首先会选择并打开一道试题，由前端服务器从自身的数据库中获取用自然语言描述的题面并返回给学生。学生每次完成编辑并提交 *COPDL* 描述后，前端服务器会异步地向后端评测机发送一个评测任务；后端评测机收到评测任务后，依次完成C代码生成、编译和黑箱测试等工作，并将错误提示和评测结果及时返回给前端服务器。大约在 *COPDL* 描述提交后 2–5 秒，学生就可从前端服务器上查询到相应的反馈信息。

6.2.2　问题建模训练平台 *COPDLOpenJudge* 的使用方法

图 6.3 展示了提交 *COPDL* 描述的界面。用户只需将写好的 *COPDL* 描述复制粘贴到文本框中，单击确认即可完成提交。*COPDLOpenJudge* 收到 *COPDL* 描述后，只需 2–5 秒便可完成整个评测过程，并将结果反馈给用户（如图 6.4）。

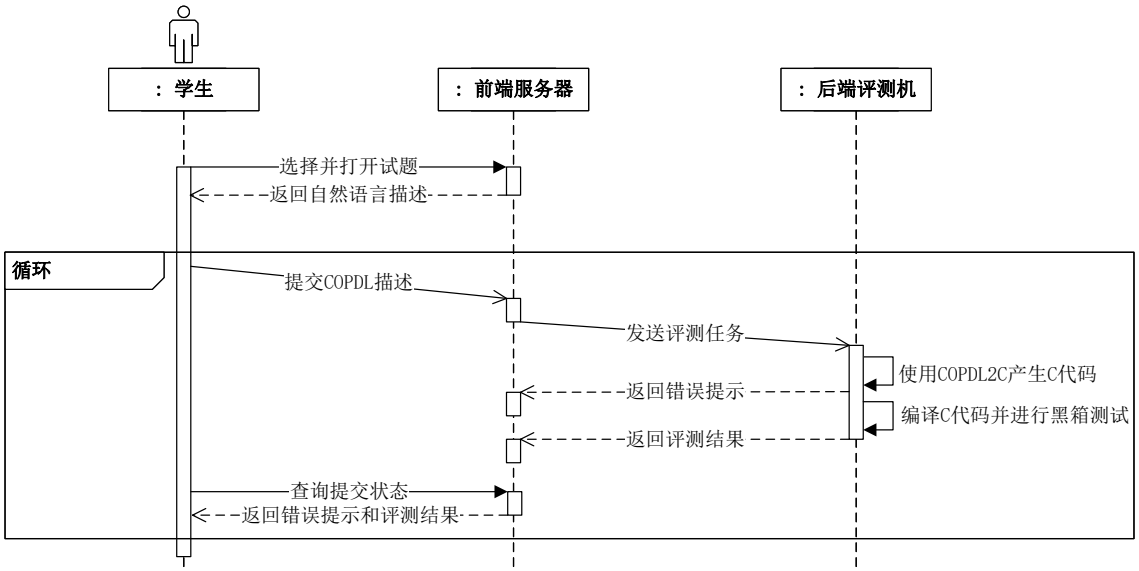

图 6.2　*COPDLOPENJUDGE* 关键操作的交互图

图 6.3　*COPDLOPENJUDGE* 提交 *COPDL* 描述界面

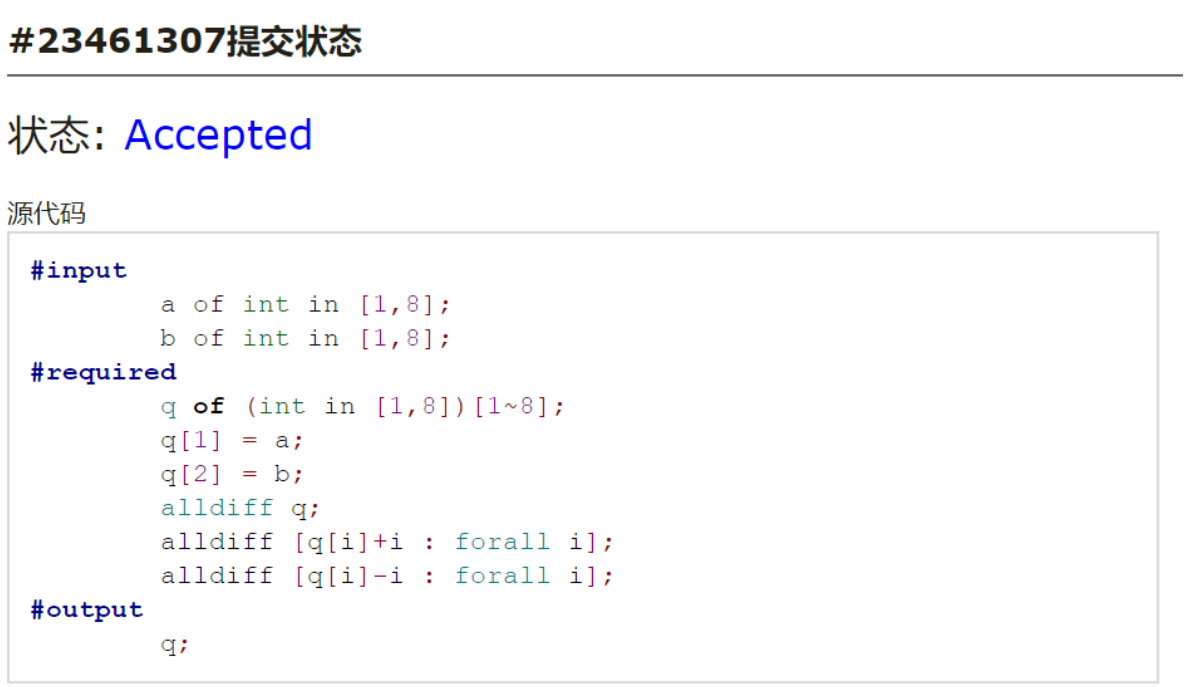

图 6.4　*COPDLOpenJudge* 反馈结果界面

6.2.3　错误提示与反馈信息

为了帮助用户更好地定位和改正 *COPDL* 描述中的错误，*COPDL2C* 在程序自动生成过程中，根据 *COPDL* 描述中可能存在的问题，提供了类似于编译信息的四类错误提示：

- **语法分析错误**：*COPDL* 描述中存在语法或拼写错误。反馈信息中还会包含首次出现错误的行列位置。
- **类型错误**：*COPDL* 描述中的表达式之间存在类型冲突。反馈信息中还会包含最小的存在类型冲突的子表达式，以及各操作数分别被推断成何种类型。
- **无界变量错误**：某些变量缺少必要的值域限制，导致搜索空间无界。反馈信息中还会包含这些缺少上界或下界的变量名。
- **未使用变量警告**：报告被定义但未被使用的变量。

同时，*COPDL2C* 还会将生成的 C 语言程序也一并返回给学生。学生提交的 *COPDL* 描述有问题时，可以自行设计一些样例数据测试这些程序，以更快地发现错误并进行修改；同时，学生也可以通过阅读代码，学习基本的程序实现和优化策略。

6.3　*COPDL* 及 *COPDL2C* 辅助教学效果评估实验设计与结果

本章通过在不同组学生之间的对比实验，评估 *COPDL* 及 *COPDL2C* 的适用性，

表 6.2　用于不同组学生进行对比实验的编程任务

编号	题目名	算法类型	题目描述
P1	重复的数	枚举	给定 $N(1 \le N \le 1000)$ 个整数，找出其中唯一一个重复出现的数。
P2	八皇后	搜索剪枝	将八个国际象棋里的皇后放置在象棋棋盘上，使得每个皇后都无法直接攻击别的皇后。
P3	最短路	动态规划	一个国家有 $N(1 \le N \le 100)$ 座城市。给出城市间的距离，找到从城市 1 到城市 N 的最短路。
P4	和为 K	枚举	从 $N(1 \le N \le 1000)$ 个给定整数中找到和为 K 的两个数。
P5	信使问题	搜索剪枝	一个国家有 $N(1 \le N \le 10)$ 座城市。给出城市间的距离，找到一条最短路经过每座城市恰好一次。
P6	团队合作	动态规划	有 $N(1 \le N \le 100)$ 名候选者。每名候选者具有一个合作值 v（$-50 \le v \le 50$）和一个工作值 w（$-50 \le w \le 50$）。要求从中选取任意数量的候选者组成团队，满足团队成员的总合作值为正。如何选取能够使得团队成员的总工作值最大？

即探究 *COPDL* 是否对程序设计类和算法分析类课程的教学有帮助。整个实验均在 *COPDLOpenJudge* 平台上完成。

实验与计算机科学专业本科的一门程序设计与算法分析相关课程合作，并作为课程的一项作业布置给学生。通过这门课之前的学习，学生已较为熟练地掌握了 C 语言编程并具有了枚举、搜索剪枝、动态规划等算法的设计和实现基础。

6.3.1　实验设计

在实验开始之前，所有学生都会收到 *COPDL* 的参考手册（见附录 A）。手册直观且详细地介绍了 *COPDL* 的语法和使用方式，同时提供了丰富的示例帮助学生理解。学生在实验前有充足的时间阅读手册并学习使用 *COPDL*。

为了进行对比实验，所有同学都被要求独立完成相同的六个编程任务。表 6.2 列举了这些编程任务的具体细节。其中，每种算法类型（枚举，搜索剪枝，动态规划）的任务各有两个。一般而言，三种算法类型的难度应该是“枚举 < 搜索剪枝 < 动态规划”。这些题目的难度与课程讲义的例题相当，因此大部分学生都能很好地掌握。

参与实验的所有 187 名学生被分成 4 组进行对比试验，分组时充分考虑了学生之前学期的计算机专业课成绩，使得各组总体编程能力基本接近。每组学生需要完成特定的任务。表 6.3 列出了对于每一组学生的任务设计。简单来说，每名学生的任务均

表 6.3　各组学生需要完成的任务

组编号	任务设计
组 I	先用 *NL2COPDL* 实现 P1–P3，再用 C 实现 P4–P6
组 II	先用 C 实现 P4–P6，再用 *NL2COPDL* 实现 P1–P3
组 III	先用 *NL2COPDL* 实现 P4–P6，再用 C 实现 P1–P3
组 IV	先用 C 实现 P1–P3，再用 *NL2COPDL* 实现 P4–P6

由两个阶段组成：在阶段 1，学生需要使用 *COPDL* 或 C 完成三个编程任务；在阶段 2，学生需要使用另一种工具完成另外三个编程任务。

这样的实验设置使得每个编程任务上使用 *COPDL* 和 C 的学生数大致相当，以便于保证后续实验比较的公平性。

每位参与学生在实验开始前，首先对自己求解问题的信心水平主观地做了评价。在实验的过程中，学生将有至多 30 分钟的时间用于完成每个编程任务。*COPDL* 描述或 C 程序的正确性均由在线评测系统实时判定并将结果返回给学生，学生可以在时间限制内无限次地进行提交。在每个阶段结束后，学生将填写如下问卷：

1. 你在每个编程任务上分别花费了多长时间？
2. 使用 *COPDL* 描述问题 / C 编写程序的难易程度如何？
3. 在 *COPDL* 描述 / C 程序上查错并修改的难易程度如何？
4. 你对求解问题的信心如何？

在完成编程任务的过程中，学生记录下自己解决每个问题所花费的分钟数作为问题 1 的填写内容。对于问题 2–4，使用五级李克特量（five-level Likert scale）（Allen et al 2007）进行主观评分。

实验结束后，根据学生的反馈信息以及在线评测系统记录的评测结果对下列研究问题进行深入探究：

1. **学生使用 *COPDL* / C 完成编程任务的表现如何？**通过比较学生解决每个问题所花费时间以及在线评测系统上记录的完成情况分析使用 *COPDL* 对学生完成编程任务的帮助。
2. **编写 *COPDL* 描述与编写 C 程序的难度是否有差异？**通过对比学生在问卷问题 2 上的评分，分析二者差异。
3. **对 *COPDL* 描述 / C 程序查错并修改的难度是否有差异？**通过对比学生在问卷问题 3 上的评分，分析二者差异。
4. ***COPDL* 能否帮助提升学生求解问题的信心？**通过对比学生在问卷问题 4 上的评分，分析使用 *COPDL* 的情况下，学生的自信心是否有提升。

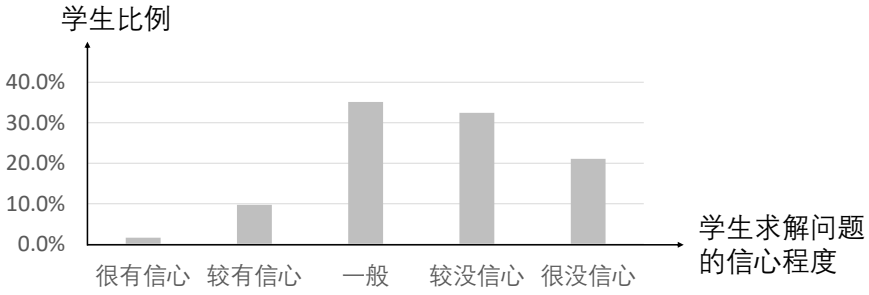

图 6.5　学生在实验前的求解问题信心程度

表 6.4　各组学生求解 6 个组合优化问题的情况统计

	使用 *COPDL*						使用 C					
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**
放弃率（%）	0.0	1.1	2.3	0.0	2.1	1.0	1.0	5.2	18.6	0.0	4.5	23.9
平均时间（分）	5.5	8.3	9.2	5.0	8.1	6.4	5.9	14.6	22.6	7.1	15.4	26.7
错误率（%）	0.0	3.4	2.3	0.0	2.1	0.0	2.1	9.8	17.7	0.0	11.9	17.9

6.3.2　实验结果

在本节的实验结果分析中，由于各组数据样本数量足够大（$n > 30$），且不具有方差齐性，因此均采用曼-惠特尼 U 检验（Mann-Whitney U test）判断两组数据均值是否存在显著差异（Mann et al 1947），单侧检验，设置显著水平为 $\alpha = 0.01$。零假设 H_0 和备择假设 H_1 分别定义为式 (6.1) 和式 (6.2)。

$$H_0 \ : \ \mu_1 \geq \mu_2 \tag{6.1}$$

$$H_1 \ : \ \mu_1 < \mu_2 \tag{6.2}$$

若计算得到的 $p \leq \alpha$，则拒绝 H_0，接受 H_1，即一组数据的均值显著小于另一组数据的均值；否则不能拒绝 H_0，即一组数据的均值不显著小于另一组数据的均值。

图 6.5 展示了学生在实验前对自己求解问题信心的主观评价。其中，53.5% 的学生对求解问题没有信心，而只有 11.4% 的学生有信心。大部分学生缺乏信心，也从侧面反映出求解组合优化问题是一项有难度的任务。

表 6.4 统计了学生在实验过程中编程任务的完成状态。其中，**放弃率**显示了该编程任务下在规定时间限制内没有向在线程序评测系统提交任何文件的学生比例；**平均时间**是该编程任务下所有成功完成的学生的平均完成时间；**错误率**是最后一次提交不正确的学生占未放弃学生人数的比例。根据此表可以看出，使用 *COPDL* 的学生的放弃率要低于使用 C 的学生，尤其是对于难度最大的两个问题 P3 和 P6，使用 *COPDL*

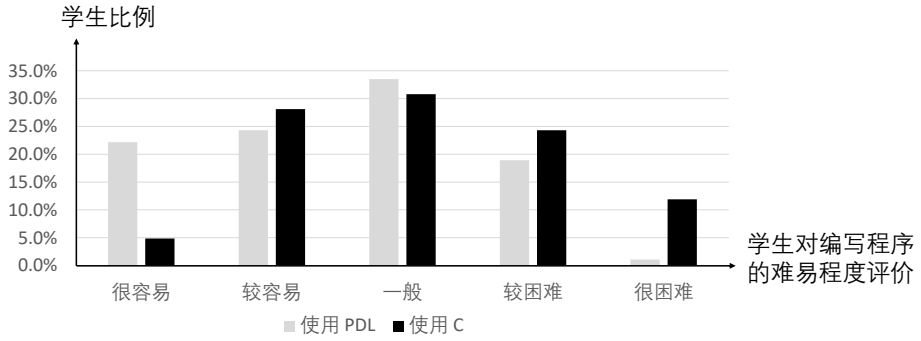

图 6.6　学生对使用 *COPDL* 和 C 编写程序难易程度的评价

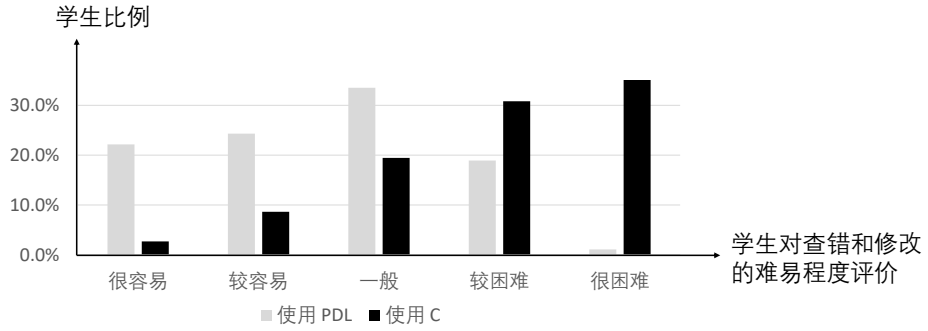

图 6.7　学生对使用 *COPDL* 和 C 查错和修改难易程度的评价

的学生中，分别只有 2.3% 和 1.0% 的学生选择放弃；然而，使用 C 的学生中，有多达 18.6% 和 23.9% 的学生选择了放弃。而对于没有放弃的学生，除了在 P1 和 P4 这两道简单题上，使用 *COPDL* 的学生和使用 C 的学生在平均求解时间上没有显著优势外（分别是 $p = 0.38209 > \alpha$ 和 $p = 0.01743 > \alpha$），在其他较难的题目上，使用 *COPDL* 的学生的平均求解时间要显著更短（$p < 0.00001 < \alpha$），且随着编程任务难度的的增加，二者的差异也会随之增大。此外，使用 *COPDL* 的学生的错误率也会更低一些，因为 *COPDL* 并不要求学生设计和实现求解算法，因此避免了因编码失误造成的错误。

结论 6.1：使用 *COPDL* 可以有效地降低学生求解问题时的放弃率，同时，也帮助学生在更少的时间内更正确地求解问题。

图 6.6 展示了学生对使用 *COPDL* 和 C 编写程序难易程度的评价。46.5% 的学生认为编写 *COPDL* 描述比较容易，而只有 20.0% 的学生认为比较困难。与之相反的，33.0% 的学生认为编写 C 程序比较简单，而 36.2% 的学生认为比较困难。平均来看，编写 *COPDL* 描述显著易于编写 C 程序（$p < 0.00001 < \alpha$）。

类似的反差同样出现在对使用 *COPDL* 和 C 查错和修改难易程度的评价上（图 6.7）。对 *COPDL* 描述查错和修改时，46.5% 的学生认为比较容易，而 20.0% 的学生

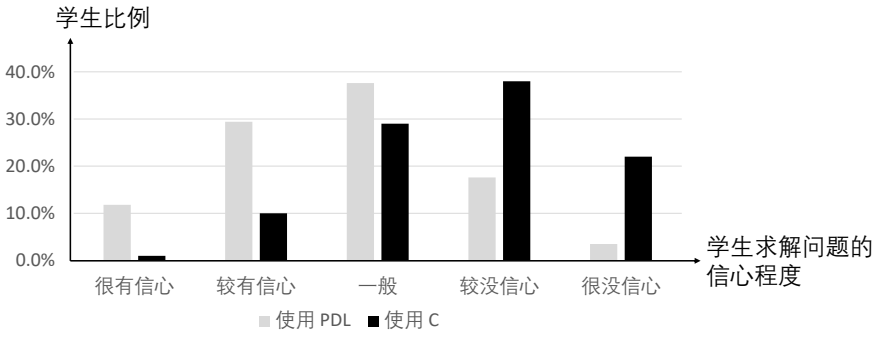

图 6.8　学生在阶段 1 结束后的求解问题信心程度

认为比较困难。而对 C 程序查错和修改时，只有 11.4% 的学生觉得容易，而有多达 69.2% 的学生觉得困难。这些结果也从侧面解释了为什么使用 *COPDL* 能够在更短的时间内更好地解决编程任务。平均来看，对 *COPDL* 描述查错和修改显著易于对 C 程序查错和修改（$p < 0.00001 < \alpha$）。

结论 6.2：相较于对 C 程序编写、查错和修改，对 *COPDL* 描述进行编写、查错和修改更为容易。对于初学者来说，*COPDL* 更容易上手。

图 6.8 展示了学生完成阶段 1 的三个编程任务后的求解问题信心程度。将其与实验前的信心程度（图 6.5）比较后可以发现，在使用 *COPDL* 的学生中，报告自己对求解问题有信心的人数大大增加（从 11.4% 增长至 41.2%），而缺乏自信的学生人数也明显减少（从 53.5% 降至 21.1%）。换言之，使用 *COPDL* 作为求解编程任务的辅助工具，学生的求解问题信心会有显著地提升，这有利于问题求解能力的训练。平均来看，使用 *COPDL* 的经历能够显著提高自信（$p < 0.00001 < \alpha$）；相反使用 C 的经历并不会对自信心提升有明显帮助（$p = 0.44433 > \alpha$）。

结论 6.3：使用 *COPDL* 的经历可以明显提升学生对求解问题的信心。

6.4　本章小结

组合优化问题通常是比较困难的，不仅是建模困难，而且实现求解程序也有一定困难。对于没有很多编程经验的初学者来说，独立求解这类问题可能会留下负面的经历，导致学习的信心和动力丧失。本章将 *COPDLOpenJudge* 作为支架式教学工具引入程序设计类和算法分析类课程的教学。通过将问题求解过程划分为问题建模和程序实现两个独立的部分，隐藏较困难的部分，降低问题求解难度，提升学生问题求解的信心，从而使学生的问题求解能力得到更好的锻炼。实验表明，*COPDL* 能够显著降低问

题求解的复杂性，并且让使用者更有信心去尝试求解组合优化问题。

第七章　结论和展望

让计算机学会像人一样做需要具备一定智能的工作，一直都是人工智能领域不懈追求的目标。下棋机器人、高考机器人、聊天机器人等等，计算机正在学会越来越多人类认为需要智能的事情。编写正确而高效的计算机程序，无疑是一项高难度的、需要复杂智能的工作。本文沿着这一思路出发，力图让计算机学会自动生成给定问题的求解程序。本文将问题域聚焦在了一类极具挑战性而又广泛存在的问题——组合优化问题上。论文工作围绕三个关键问题展开，即（1）如何描述待求解的组合优化问题？（2）如何提高解决组合优化问题的求解效率？（3）如何将自然语言描述的组合优化问题自动转化为形式化约束模型，进而自动产生相应的求解程序？

7.1　本文工作总结

本文首先探究了如何从组合优化约束模型产生程序源代码。为了实现这一目标，本文定义了新的问题描述语言 *COPDL* 并实现了相应的程序自动生成工具 *COPDL2C*。在 *COPDL2C* 中，经过类型推导，独立变量选取、基于循环/递归的程序自动生成等操作，成功将 *COPDL* 描述的问题转化为正确的基本求解程序。

本文进一步研究了如何静态分析问题性质，并在基础求解算法上自动应用搜索剪枝、动态规划两类高级优化技术，以提高求解程序的求解效率。这项工作的步骤包括自动化的问题性质识别，优化技术设计和实现，以及模型质量评估等。实验结果表明，通过应用这些高级优化技术，产生的程序比通用求解器消耗资源少，且在搜索问题和动态规划问题的求解效率上优于主流的约束求解器。

此外，本文进一步设计实现了从自然语言描述到 *COPDL* 的翻译算法 *NL2COPDL*，以降低用户描述组合优化问题的门槛。在转换过程中，本文首先利用 CoreNLP 自然语言处理工具对自然语言描述进行处理，然后从中提取出关键词、变量以及变量类型；之后，基于关键词和翻译规则产生了约束框架，并使用基于框架的程序自动生成产生了完整的 *COPDL* 约束模型。这一方法在本文自建的 40 个组合优化问题上取得了 100% 的正确率，但是其泛化能力还有待提升。后续将继续推广到其他公开数据集，以及更广泛的用自然语言描述的组合优化问题上。

最后，本文基于 *COPDL2C* 设计并实现了程序在线评测教学平台 *COPDLOPEN-JUDGE*，并将其应用在程序设计类与算法分析类课程的教学上。由于定义了 *COPDL* 并实现了 *NL2COPDL* 和 *COPDL2C* 两个工具，从自然语言问题描述到生成求解程序代码的工作被自然地分割成两个子任务，即问题建模和模型求解。这样的任务分割同样适

用于程序员的培养过程。让学生将自然语言问题描述重写成等价的 *COPDL*，可以训练其建模能力而无需纠结于模型求解代码的具体细节，规避了复杂的实现问题过早暴露给初学者，从而降低了问题求解的难度，这有助于增强学生的信心和学习动力，避免学生对编程产生恐惧。

综上所述，本文的主要创新点如下：

- **设计了适合用于程序自动生成的组合优化约束模型描述语言 *COPDL*，并设计实现了 *COPDL* 编译工具 *COPDL2C*，可用于产生基本求解算法；**
- **提出了问题模型性质的静态分析方法，并在 *COPDL2C* 的基础上，根据问题性质设计并应用搜索剪枝和动态规划两种高级优化技术；**
- **提出将描述组合优化问题的自然语言转化为形式化描述的算法 *NL2COPDL*，该算法使用基于模板规则和基于框架的方法生成 *COPDL* 描述；**
- **将 *COPDL* 及相关工具作为"支架式"教学工具应用于教学工作中，作为问题建模训练的工具，同时提高了学生求解问题的信心。**

7.2 未来工作展望

本文从多个角度研究了面向组合优化约束模型的程序自动生成的方法和技术，以及相关工具在教学上的应用，并通过实验验证了这些算法的有效性。基于本文工作，未来可以进一步优化和细化现有的成果和工作，包括：

1. **推广 *COPDL* 及相关工具**
    - 将 *COPDL* 及相关工具开源，吸引更多的程序自动生成方法研究人员在本文工作基础上继续深化和泛化现有研究，进而形成一套高效而实用的组合优化问题求解程序自动生成工具。
    - 将 *COPDL* 及相关工具更广泛地应用于程序设计与算法分析类课程的教学当中，提高对学生问题求解能力，特别是问题建模能力的培养效率。
2. **增强 *COPDL* 的表达能力及 *COPDL2C* 的程序优化能力**
    - 完善问题描述语言 *COPDL*：简化语法，提高抽象程度，让普通用户更容易学习和使用；或者引入常用的元素（例如字符串、高级数据结构等）增加表达能力，方便用户描述问题的同时，也能让 *COPDL2C* 更好地发现问题性质并运用优化算法。
    - 探究如何自动挖掘更多的问题性质，并自动应用其他优化算法，包括分治、贪心，甚至模拟退火、遗传算法等概率算法。
3. **扩大 *NL2COPDL* 的适用范围**
    - 进一步降低使用者门槛：尝试用自然语言理解领域的最新研究成果，增

强 *NL2COPDL* 的处理能力，使更多类型的问题能被直接转化为求解程序；

- 设计并实现 *NL2COPDL* 的扩展接口，支持用户自定义领域相关的模板库和规则库，从而提高 *NL2COPDL* 的可扩展性。

本文作者将继续现有的面向组合优化约束模型的程序自动生成研究工作，期望让计算机能够更好地提高人类的工作效率。

参考文献

金继伟，马菲菲，张健 (2015)。“*SMT求解技术简述*”。计算机科学与探索。

Abraham Robin and Erwig Martin (2006). “*Inferring templates from spreadsheets*”. In: *Proceedings - International Conference on Software Engineering*. New York, NY, USA: ACM, 182–191.

Ahmed Umair Z., Gulwani Sumit and Karkare Amey (2013). “*Automatically Generating Problems and Solutions for Natural Deduction*”. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. Beijing, China: AAAI Press, 1968–1975.

Allen I. Elaine and Seaman Christopher A. (2007). “*Likert scales and data analyses*”. *Quality Progress*, *40*(7): 64–65.

Alur Rajeev, Bodik Rastislav, Juniwal Garvit *et al.* (2013). *Syntax-guided synthesis*. 1–8.

Alur Rajeev, Singh Rishabh, Fisman Dana *et al.* (2018). “*Search-based program synthesis*”. *Communications of the ACM*, *61*(12): 84–93.

Alvin Chris, Gulwani Sumit, Majumdar Rupak *et al.* (2014). “*Synthesis of geometry proof problems*”. In: *Proceedings of the National Conference on Artificial Intelligence*.

An Kijin, Meng Na and Tilevich Eli (2018). “*Automatic inference of Java-to-swift translation rules for porting mobile applications*”. In: *Proceedings - International Conference on Software Engineering*. 180–190.

Applegate David L., Bixby Robert E., Chvátal Vašek *et al.* (2011). *The traveling salesman problem: A computational study*. Princeton, NJ, USA: Princeton University Press, 1–593.

Baatar Davaatseren, Boland Natashia, Brand Sebastian *et al.* (2007). “*Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches*”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 1–15.

Bajwa Imran Sarwar, Bordbar Behzad and Lee Mark G. (2010). “*OCL constraints generation from natural language specification*”. In: *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*. 204–213.

Bajwa Imran Sarwar, Bordbar Behzad, Lee Mark G. and Anastasakis Kyriakos (2012). “*NL2 Alloy: A tool to generate alloy from NL constraints*”. *Journal of Digital Information Management*, *10*(6): 365–372.

Bajwa Imran Sarwar, Lee Mark G. and Bordbar Behzad (2011). “*SBVR business rules generation from natural language specification*”. In: *AAAI Spring Symposium - Technical Report*. 2–8.

Balog Matej, Gaunt Alexander L., Brockschmidt Marc *et al.* (2017). *DeepCoder: Learning to Write Programs*.

Barrett Clark, Conway Christopher L., Deters Morgan *et al.* (2011). “*CVC4*”. In: Gopalakrishnan Ganesh and Qadeer Shaz, eds. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 171–177.

Barrett Clark, Fontaine Pascal and Tinelli Cesare (2016). *The Satisfiability Modulo Theories Library (SMT-LIB)*.

Barrett Clark, Moura Leonardo de, Ranise Silvio *et al.* (2011). "*The SMT-LIB Initiative and the Rise of SMT*". In: Barner Sharon, Harris Ian, Kroening Daniel *et al.*, eds. *Hardware and Software: Verification and Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 3–3.

Barricelli Barbara Rita, Cassano Fabio, Fogli Daniela *et al.* (2019). "*End-user development, end-user programming and end-user software engineering: A systematic mapping study*". *Journal of Systems and Software*, *149*: 101–137.

Barricelli Barbara Rita and Valtolina Stefano (2017). "*A visual language and interactive system for end-user development of internet of things ecosystems*". *Journal of Visual Languages and Computing*, *40*: 1–19.

Bart Austin Cory, Tibau Javier, Tilevich Eli *et al.* (2017). "*BlockPy: An Open Access Data-Science Environment for Introductory Programmers*". *Computer*, *50*(5): 18–26.

Basin David, Deville Yves, Flener Pierre *et al.* (2004). "*Synthesis of programs in computational logic*". *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

Bates Jane (2003). "*Your wish is my command*". In: *Nursing Standard*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 18–19.

Beaubouef Theresa and Mason John (2005). "*Why the high attrition rate for computer science students*". *ACM SIGCSE Bulletin*, *37*(2): 103–106.

Becker Andrew, Maksimovic Djordje, Novo David *et al.* (2015). "*FudgeFactor: Syntax-Guided Synthesis for Accurate RTL Error Localization and Correction*". In: *Haifa Verification Conference*.

Belotti Pietro, Cafieri Sonia, Lee Jon *et al.* (2010). "*Feasibility-based bounds tightening via fixed points*". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 65–76.

Berenz Vincent and Suzuki Kenji (2014). "*Targets-Drives-Means: A declarative approach to dynamic behavior specification with higher usability*". *Robotics and Autonomous Systems*, *62*(4): 545–555.

Bhatti Ghasan, Brémond Roland, Jessel Jean Pierre *et al.* (2015). "*Design and evaluation of a user-centered interface to model scenarios on driving simulators*". *Transportation Research Part C: Emerging Technologies*, *50*: 3–12.

Bjordal Gustav, Monette Jean Noel, Flener P. *et al.* (2015). "*A constraint-based local search backend for MiniZinc*". *Constraints*, *20*(3): 1–21.

Blazewicz Jacek and Kasprzak Marta (2005). "*Combinatorial optimization in DNA mapping — a computational threadof the Simplified Partial Digest Problem*". *RAIRO - Operations Research*, *39*(4): 227–241.

Bodik Rastislav and Torlak Emina (2012). "*Synthesizing Programs with Constraint Solvers*". In: *Computer Aided Verification*. 3–3.

Bogdan Cristian, Kaindl Hermann, Falb Jürgen *et al.* (2008). "*Modeling of interaction design by end users through discourse modeling*". In: *International Conference on Intelligent User Interfaces, Proceedings IUI*. New York, NY, USA: ACM, 305–308.

Boute Raymond T. (1992). "*The Euclidean Definition of the Functions Div and Mod*". *ACM Trans. Program. Lang. Syst.* 1992-04, *14*(2): 127–144.

Brain Martin, Tinelli Cesare, Rümmer P *et al.* (2015). "*An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic*". *IEEE Computer Society*.

Bravenboer Martin, Kalleberg Karl Trygve, Vermaas Rob *et al.* (2008). "*Stratego/XT 0.17. A language and toolset for program transformation*". *Science of Computer Programming*, *72*(1-2): 52–70.

Bryant Randal E., German Steven and Velev Miroslav N. (2001). "*Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic*". *ACM Transactions on Computational Logic*.

Büchi J. Richard and Landweber Lawrence H. (1969). "*Solving Sequential Conditions by Finite-State Strategies*". *Transactions of the American Mathematical Society*, *138*: 295–311.

Buitrago Flórez Francisco, Casallas Rubby, Hernández Marcela *et al.* (2017). "*Changing a Generation's Way of Thinking: Teaching Computational Thinking Through Programming*". *Review of Educational Research*, *87*(4): 834–860.

Burson Scott, Kotik Gordon B. and Markosian Lawrence Z. (1990). "*A program transformation approach to automating software re-engineering*". In: *Proceedings - IEEE Computer Society's International Computer Software & Applications Conference*. 314–322.

Butler Matthew and Morgan Michael (2007). "*Learning challenges faced by novice programming students studying high level and low feedback concepts*". In: *ASCILITE 2007 - The Australasian Society for Computers in Learning in Tertiary Education*. Australasian Society for Computers in Learning in Tertiary Education, 99–107.

Caballero Rafael, Hermanns Christian and Kuchen Herbert (2007). "*Algorithmic Debugging of Java Programs*". *Electronic Notes in Theoretical Computer Science*, 2007-06, *177*: 75–89.

Carlsson Mats and Mildner Per (2010). "*SICStus Prolog – the first 25 years*". *CoRR*, *abs/1011.5640*.

Carmien Stefan Parry and Fischer Gerhard (2008). "*Design, adoption, and assessment of a socio-technical environment supporting independence for persons with cognitive disabilities*". In: *Conference on Human Factors in Computing Systems - Proceedings*. New York, NY, USA: ACM, 597–606.

Cheema Salman, Gulwani Sumit and LaViola Joseph J. (2012). "*QuickDraw: Improving drawing experience for geometric diagrams*". In: *Conference on Human Factors in Computing Systems - Proceedings*.

Chen Danqi and Manning Christopher D. (2014). "*A fast and accurate dependency parser using neural networks*". In: *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*. Doha, Qatar: Association for Computational Linguistics, 740–750.

Choco-solver dev team (2021). *Choco Documentation*. `https://choco-solver.org`.

Chu Geoffrey, De La Banda Maria Garcia and Stuckey Peter J. (2012). "*Exploiting subproblem dominance in constraint programming*". *Constraints*, *17*(1): 1–38.

Church Alonzo (1957). "*Applications of recursive arithmetic to the problem of circuit synthesis*". *Summaries of the Summer Institute of Symbolic Logic*, *1*: 3–50.

Church Alonzo (1962). "*Logic, Arithmetic, and Automata*". *International Congress of Mathematicians*, 23–35.

Cook Stephen A. (1971). "*The Complexity of Theorem-Proving Procedures*". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Shaker Heights, Ohio, USA: Association for Computing Machinery, 151–158.

Cordy James R., Halpern Charles D. and Promislow Eric (1988). "*TXL: A rapid prototyping system for programming language dialects*". In: *Proceedings. 1988 International Conference on Computer Languages*. 280–285.

Cormen Thomas H., Leiserson Charles E., Rivest Ronald L. *et al.* (2009). "*Introduction to Algorithms, Third Edition*". *The MIT Press*.

Davis Martin, Logemann George and Loveland Donald (1962). "*A Machine Program for Theorem-Proving*". *Commun. ACM*, 1962-07, *5*(7): 394–397.

De Moor Oege (1994). "*Categories, Relations and Dynamic Programming*". *Mathematical Structures in Computer Science*, *4*(1): 33–69.

De Moura Leonardo and Bjørner Nikolaj (2008). "*Z3: An efficient SMT Solver*". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 337–340.

Dekker Jip J., Björdal Gustav, Carlsson Mats *et al.* (2017). "*Auto-tabling for subproblem presolving in MiniZinc*". *Constraints*, *22*(4): 512–529.

Delozier Christian, Eisenberg Richard, Nagarakatte Santosh *et al.* (2013). "*Ironclad C++ a library-augmented type-safe subset of C++*". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. 287–304.

Díaz Paloma, Malizia Alessio, Navarro Ignacio *et al.* (2011). "*Using recommendations to help novices to reuse design knowledge*". In: Costabile Maria Francesca, Dittrich Yvonne, Fischer Gerhard *et al.*, eds. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 331–336.

Dion Bernard (2004). "*Correct-by-construction methods for the development of safety-critical applications*". In: *SAE Technical Papers*. SAE International.

Dutertre Bruno (2014). "*Yices2.2*". In: *International Conference on Computer Aided Verification*.

Dwarakanath Anurag and Sengupta Shubhashis (2012). "*Litmus: Generation of test cases from functional requirements in natural language*". In: Bouma Gosse, Ittoo Ashwin, Métais Elisabeth *et al.*, eds. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 58–69.

Eagan James R. and Stasko John T. (2008). "*The Buzz: Supporting user tailorability in awareness applications*". In: *Conference on Human Factors in Computing Systems - Proceedings*. New York, NY, USA: ACM, 1729–1738.

Een Niklas (2005). "*Minisat : A sat solver with conflict-clause minimization*". In: *Proc Sat-05: International Conference on Theory & Applications of Satisfiability Testing*.

Faravelon Aurélien and Céret Eric (2014). "*Privacy conscious web apps composition*". In: *Proceedings - International Conference on Research Challenges in Information Science*. 1–12.

Flener Pierre, Zidoum Hamza and Hnich Brahim (1998). "*Schema-guided synthesis of constraint logic programs*". In: *Proceedings - 13th IEEE International Conference on Automated Software Engineering, ASE 1998*. 168–176.

Franců Jan and Hnětynka Petr (2009). "*Automated Code Generation from System Requirements in Natural Language*". *e-Informatica*, *3*(1): 72–88.

Frisch Alan M., Harvey Warwick, Jefferson Chris *et al.* (2008). "*Essence: A constraint language for specifying combinatorial problems*". *Constraints*, *13*(3): 268–306.

Ganzinger Harald, Hagen George, Nieuwenhuis Robert *et al.* (2004). "*DPLL(T): Fast Decision Procedures*". In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*.

Garzotto Franca and Gonella Roberto (2011). "*An open-ended tangible environment for disabled children's learning*". In: *Proceedings of IDC 2011 - 10th International Conference on Interaction Design and Children*. New York, NY, USA: ACM, 52–61.

Gaschnig John (1977a). "*A General Backtrack Algorithm That Eliminates Most Redundant Tests*". In: Reddy Raj, ed. *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*. William Kaufmann, 457.

Gaschnig John (1977b). "*Exactly How Good Are Heuristics?: Toward a Realistic Predictive Theory of Best-First Search*". In: Reddy Raj, ed. *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*. William Kaufmann, 434–441.

Gaschnig John (1979). "*A Problem Similarity Approach to Devising Heuristics: First Results*". In: Buchanan Bruce G., ed. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI 79, Tokyo, Japan, August 20-23, 1979, 2 Volumes*. William Kaufmann, 301–307.

Gent Ian P., Petrie Karen E. and Puget Jean François (2006). "*Chapter 10 Symmetry in constraint programming*". In: Rossi Francesca, Beek Peter van and Walsh Toby, eds. *Foundations of Artificial Intelligence*. Elsevier, 329–376.

Ghiani Giuseppe, Manca Marco, Paterno Fabio *et al.* (2017). "*Personalization of context-dependent applications through trigger-action rules*". *ACM Transactions on Computer-Human Interaction*, *24*(2): 14:1–14:33.

Gonzalez Teofilo, Diaz-Herrera Jorge and Tucker Allen (2014). *Combinatorial optimization*. New York, NY, USA: John Wiley & Sons, Inc., 13–1–13–47.

Gulwani Sumit (2010a). "*Automating string processing in spreadsheets using input-output examples*". In: *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. 317–329.

Gulwani Sumit (2010b). "*Dimensions in Program Synthesis*". In: *PPDP'10 - Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming*.

Gulwani Sumit, Jha Susmit, Tiwari Ashish *et al.* (2011). "*Synthesis of loop-free programs*". *ACM SIGPLAN Notices: A Monthly Publication of the Special Interest Group on Programming Languages*, *46*(6): 62–73.

Gulwani Sumit, Korthikanti Vijay Anand and Tiwari Ashish (2011). "*Synthesizing geometry constructions*". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Guo Bin, Zhang Daqing and Imai Michita (2011). "*Toward a cooperative programming framework for context-aware applications*". *Personal and Ubiquitous Computing*, *15*(3): 221–233.

Hammond Jennifer (2001). "*Scaffolding : Teaching and Learning in Language and Literacy Education*". *Primary English Teaching Assoc.* (5): 1–128.

Hartmann Melanie, Schreiber Daniel and Mühlhäuser Max (2009). "*AUGUR: Providing context-aware interaction support*". In: *EICS'09 - Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 123–131.

Hermans Felienne and Aivaloglou Ehimia (2017). "*A controlled experiment comparing plugged first and unplugged first programming lessons*". In: *ACM International Conference Proceeding Series*. New York, NY, USA: Association for Computing Machinery, 49–56.

Hobe Alex, Vogler Daniel, Seybold Martin P. *et al.* (2018). "*Estimating Flow Rates through Fracture Networks using Combinatorial Optimization*". *Advances in Water Resources*, *122*(DEC.): 85–97.

Hoos Holger H. and Stützle Thomas G. (2004). *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann Publishers Inc.

Houben Steven, Golsteijn Connie, Gallacher Sarah *et al.* (2016). "*Physikit: Data engagement through physical ambient visualizations in the home*". In: *Conference on Human Factors in Computing Systems - Proceedings*. New York, NY, USA: ACM, 1608–1619.

Hromkovic Juraj (2010). *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer-Verlag.

Ji Ruyi, Sun Yican, Xiong Yingfei *et al.* (2020). "*Guiding dynamic programing via structural probability for accelerating programming by example*". *Proceedings of the ACM on Programming Languages*, 2020-11, *4*: 1–29.

Kirchner Claude and Ringeissen Christophe (1998). "*Rule-Based Constraint Programming*". *Fundamenta Informaticae*, *34*: 225–262.

Kitching Matthew and Bacchus Fahiem (2007). "*Symmetric component caching*". In: *IJCAI International Joint Conference on Artificial Intelligence*. 118–124.

Klein Dan and Manning Christopher D. (2003). "*Accurate unlexicalized parsing*". In: *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*. USA: Association for Computational Linguistics, 423–430.

Klein John (2015). *Model-Driven Engineering : Automatic Code Generation and Beyond* [techreport]. Pittsburgh, PA, 1–51.

Kodaganallur Viswanathan (2004). "*Incorporating language processing into Java applications: A JavaCC tutorial*". *IEEE Software*, *21*(4): 70–77.

Korpipää Panu, Malm Esko Juhani, Rantakokko Tapani *et al.* (2006). "*Customizing user interaction in smart phones*". *IEEE Pervasive Computing*, *5*(3): 82–90.

Kuchcinski Krzysztof and Szymanek Radoslaw (2013). "*JaCoP - Java Constraint Programming solver*". In: *CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming*.

Kuncak Viktor, Mayer Mikaël, Piskac Ruzica *et al.* (2010). "*Complete functional synthesis*". In: *ACM SIGPLAN Notices*.

Ladd David A. and Ramming J. Christopher (1994). "*A*: A language for implementing language processors*". In: *IEEE International Conference on Computer Languages*. 1–10.

Lau Tessa, Domingos Pedro M. and Weld Daniel S. (2003). "*Learning programs from traces using version space algebra*". In: *the international conference*.

Le Vu, Gulwani Sumit and Su Zhendong (2013). "*SmartSynth: Synthesizing smartphone automation scripts from natural language*". In: 2013-06: 193–206.

Lee Irene, Martin Fred, Denner Jill *et al.* (2011). "*Computational thinking for youth in practice*". *ACM Inroads*, *2*(1): 32–37.

Lee Woosuk, Heo Kihong, Alur Rajeev *et al.* (2018). "*Accelerating search-based program synthesis using learned probabilistic models*". 436–449.

Lei Tao, Long Fan, Barzilay Regina *et al.* (2013). "*From natural language specifications to program input parsers*". In: *ACL 2013 - 51st Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*. 1294–1303.

Lenstra J. K. and Kan A. H. G. Rinnooy (1981). "*A Recursive Approach to the Implementation of Enumerative Methods*". In: Ausiello G and Lucertini M, eds. *Analysis and Design of Algorithms in Combinatorial Optimization*. Vienna: Springer Vienna, 65–83.

Lin Xi Victoria, Wang Chenglong, Zettlemoyer Luke *et al.* (2019). "*NL2Bash: A corpus and semantic parser for natural language interface to the Linux operating system*". In: *LREC 2018 - 11th International Conference on Language Resources and Evaluation*. Miyazaki, Japan, 3107–3118.

Lozano Roberto Castañeda, Carlsson Mats, Blindell Gabriel Hjort *et al.* (2016). "*Register allocation and instruction scheduling in Unison*". In: *Proceedings of CC 2016: The 25th International Conference on Compiler Construction*. New York, NY, USA: ACM, 263–264.

Mackworth Alan K. (1977). "*Consistency in Networks of Relations*". *Artificial Intelligence*, *8*(1): 99–118.

Mahajan Yogesh S., Fu Zhaohui and Malik Sharad (2004). "*Zchaff2004: An efficient SAT solver*". In: *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*.

Maleki Maryam M., Woodbury Robert F. and Neustaedter Carman (2014). "*Liveness, localization and lookahead: Interaction elements for parametric design*". In: *Proceedings of the Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques, DIS*. New York, NY, USA: ACM, 805–814.

Maloney John, Resnick Mitchel, Rusk Natalie *et al.* (2010). "*The scratch programming language and environment*". *ACM Transactions on Computing Education*, *10*(4).

Mann Henry B. and Whitney Donald R. (1947). "*On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other*". *The Annals of Mathematical Statistics*, *18*(1): 50–60.

Manna Zohar and Waldinger Richard (1980). "*A Deductive Approach to Program Synthesis*". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *2*(1): 90–121.

Manna Zohar, Waldinger Richard, Manna Zohar *et al.* (1992). "*Fundamentals of deductive program synthesis*". *IEEE Transactions on Software Engineering*.

Manning Christopher, Surdeanu Mihai, Bauer John *et al.* (2015). "*The Stanford CoreNLP Natural Language Processing Toolkit*". In: *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60.

Marinescu Radu and Dechter Rina (2009). "*AND/OR Branch-and-Bound search for combinatorial optimization in graphical models*". *Artificial Intelligence*, *173*(16-17): 1457–1491.

Marques-Silva João P., Lynce Inês and Malik Sharad (2009). "*Conflict-Driven Clause Learning SAT Solvers*". *Frontiers in Artificial Intelligence and Applications*, 2009-01, *185*.

Marques-Silva João P. and Sakallah Karem A. (1999). "*GRASP: a search algorithm for propositional satisfiability*". *IEEE Transactions on Computers*, *48*(5): 506–521.

Marriott Kim and Stuckey Peter J. (2014). *A MiniZinc Tutorial*. `https://www.minizinc.org/`.

Meng Na, Kim Miryung and McKinley Kathryn S. (2011). "*Systematic editing: Generating program transformations from an example*". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 329–342.

Meng Na, Kim Miryung and McKinley Kathryn S. (2013). "*Lase: Locating and applying systematic edits by learning from examples*". In: *Proceedings - International Conference on Software Engineering*. 502–511.

Menon Aditya Krishna, Tamuz Omer, Gulwani Sumit *et al.* (2013). "*A machine learning framework for programming by example*". *30th International Conference on Machine Learning, ICML 2013*, *28*(PART 1): 187–195.

Miller Robert C. and Myers Brad A. (2001). "*Interactive simultaneous editing of multiple text regions*". In: *Proceedings of the 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 161–174.

MIT App Inventor (2021). *MIT App Inventor*. `https://appinventor.mit.edu/explore/`.

Morihata Akimasa, Koishi Masato and Ohori Atsushi (2014). *Dynamic programming via thinning and incrementalization*. Springer International Publishing, 186–202.

Moskewicz Matthew W., Madigan Conor F., Zhao Ying *et al.* (2001). "*Chaff: Engineering an Efficient SAT Solver*". In: *Proceedings of the 38th Annual Design Automation Conference*. Las Vegas, Nevada, USA: Association for Computing Machinery, 530–535.

Nethercote Nicholas, Stuckey Peter J., Becket Ralph *et al.* (2007). "*MiniZinc: Towards a Standard CP Modelling Language*". In: *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. Berlin, Heidelberg: Springer-Verlag, 529–543.

Nichols Johanna (1986). "*Head-Marking and Dependent-Marking Grammar*". *Language*, *62*(1): 56–119.

Nix Robert (1984). "*Editing By Example.*" In: *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 186–195.

Obeo (2021). *Uml to Java Generator*. https://marketplace.eclipse.org/content/uml-java-generator.

Ohrimenko Olga, Stuckey Peter J. and Codish Michael (2009). "*Propagation via lazy clause generation*". *Constraints*, *14*(3): 357–391.

Papadimitriou Christos H. and Steiglitz Kenneth (2000). "*Combinatorial Optimization*". *IEEE Transactions on Acoustics Speech & Signal Processing*, *21*(3): 374–383.

Peng Haoruo, Chang Kai Wei and Roth Dan (2015). "*A joint framework for coreference resolution and mention head detection*". In: *CoNLL 2015 - 19th Conference on Computational Natural Language Learning, Proceedings*. University of Illinois, Urbana-Champaign, Urbana, IL, 61801: ACL, 12–21.

Perron Laurent and Furnon Vincent (2019). *OR-Tools*. 2019-07-19. https://developers.google.com/optimization/.

Pfenning Frank (2010). *C0 Reference*. https://www.cs.cmu.edu/~fp/courses/15122-f10/misc/c0-reference.pdf.

Polozov Oleksandr and Gulwani Sumit (2015). "*FlashMeta: a framework for inductive program synthesis*". In: *Acm Sigplan International Conference on Object-oriented Programming*.

Popma Remko (2004). *JET Tutorial*. https://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.

Prestwich Steven, Rossi Roberto, Tarim S. Armagan *et al.* (2018). "*Towards a Closer Integration of Dynamic Programming and Constraint Programming*". *EPiC Series in Computing*, *55*: 202–188.

Riss Uwe V. (2012). "*TAPIR: Wiki-based task and personal information management supporting subjective process management*". In: Oppl Stefan and Fleischmann Albert, eds. *Communications in Computer and Information Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 220–235.

Robinson G. and Wos L. (1968). "*Paramodulation and Theorem-Proving in First-Order Theories with Equality*". *Machine Intelligence*, 135–150.

Robinson J. A. (1966). "*A Machine-Oriented Logic based on the Resolution Principle*". *Journal of Symbolic Logic*, *31*(3): 515.

Ruthmann Alex, Heines Jesse M., Greher Gena R. *et al.* (2010). "*Teaching computational thinking through musical live coding in Scratch*". In: *SIGCSE'10 - Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 351–355.

Salehi Shima, Wang Karen D., Toorawa Ruqayya *et al.* (2020). "*Can majoring in computer science improve general problem-solving skills?*" In: *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*. New York, NY, USA: Association for Computing Machinery, 156–161.

Sandholm Tuomas, Suri Subhash, Gilpin Andrew *et al.* (2002). "*Winner determination in combinatorial auction generalizations*". In: *Proceedings of the International Conference on Autonomous Agents*. New York, NY, USA: ACM, 69–76.

Sauthoff Georg, Janssen Stefan and Giegerich Robert (2011). "*Bellman's GAP - A declarative language for dynamic programming*". In: *PPDP'11 - Proceedings of the 2011 Symposium on Principles and Practices of Declarative Programming*. 29–39.

Sawant Tushar, Parekh Bhagya and Shah Naineel (2013). "*Computer independent USB to USB data transfer bridge*". In: *International Conference on Emerging Trends in Engineering and Technology, ICETET*. USA: IEEE Computer Society, 40–45.

Sbihi Abdelkader and Richard W. Eglese (2010). "*Combinatorial optimization and Green Logistics*". *Annals of Operations Research*.

Schulte Christian, Lagerkvist Mikael and Tack Guido (2006). "*Gecode*". `https://www.gecode.org/`.

Shapiro Ehud Y. (1982). "*Algorithmic Program Diagnosis.*" In: *Acm Sigplan-sigact Symposium on Principles of Programming Languages*.

Singh Rishabh and Gulwani Sumit (2012). "*Synthesizing Number Transformations from Input-Output Examples*". In: *Proceedings of the 24th international conference on Computer Aided Verification*.

Smith Barbara M. (2005). "*Caching search states in permutation problems*". In: Beek Peter van, ed. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 637–651.

Smith David K. and Bertsekas Dimitri P. (1996). *Dynamic Programming and Optimal Control. Volume 1Dynamic Programming and Optimal Control. Volume 2*. 2nd. Athena Scientific, 833.

Solar-Lezama Armando (2008). *Program synthesis by sketching*. [phdthesis].

Steinberg Dave, Budinsky Frank, Paternostro Marcelo *et al.* (2015). *EMF: Eclipse Modeling Framework*. 2nd. Addison-Wesley Professional.

Stuckey Peter J., Feydy Thibaut, Schutt Andreas *et al.* (2014). "*The MiniZinc challenge 2008-2013*". *AI Magazine*, *35*(2): 55–60.

Su Jun Ming and Huang Chih Fang (2014). "*An easy-to-use 3D visualization system for planning context-aware applications in smart buildings*". *Computer Standards and Interfaces*, *36*(2): 312–326.

Team JAXenter Editorial (2017). *Eclipse Xtend - Compact, static, perfect for code generation [Pirates of the JVM]*. `https://jaxenter.com/xtend-pirates-jvm-efftinge-132385.html`.

Torlak Emina and Bodik Rastislav (2014). "*A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages*". *ACM SIGPLAN Notices*, *49*(6): 530–541.

Tu Jho Ju and Johnson John R. (1990). "*Can Computer Programming Improve Problem-Solving Ability?*" *ACM SIGCSE Bulletin*, *22*(2): 30–33.

UCLA Compilers Group (2005). *Java TreeBuilder*. `https://compilers.cs.ucla.edu/jtb/`.

Uña Diego de, Gange Graeme, Schachte Peter *et al.* (2019). "*Compiling CP subproblems to MDDs and d-DNNFs*". *Constraints*, *24*(1): 56–93.

Vechev Martin T., Yahav Eran and Bacon David F. (2006). "*Correctness-Preserving Derivation of Concurrent Garbage Collection Algorithms*". *SIGPLAN Not.* 2006-06, *41*(6): 341–353.

Vechev Martin T., Yahav Eran, Bacon David F. and Rinetzky Noam (2007). "*CGCExplorer: A Semi-Automated Search Procedure for Provably Correct Concurrent Collectors*". *SIGPLAN Not.* 2007-06, *42*(6): 456–467.

Volkstorf Charles (2015). "*Program Synthesis from Axiomatic Proof of Correctness*". *computer science*.

von Neumann John (1956). "*Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components*". *Automata Studies*, 43–98.

Waltz David L. (1972). *Generating Semantic Descriptions From Drawings of Scenes With Shadows* [phdthesis].

Wang Chunhui, Pastore Fabrizio, Goknil Arda *et al.* (2015). "*Automatic generation of system test cases from use case specifications*". In: *2015 International Symposium on Software Testing and Analysis, ISSTA 2015 - Proceedings*. New York, NY, USA: Association for Computing Machinery, 385–396.

Wetter James, Akgün Özgür and Miguel Ian (2015). "*Automatically generating streamlined constraint models with ESSENCE and CONJURE*". In: Pesant Gilles, ed. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Cham: Springer International Publishing, 480–496.

Xpand (2021). *Xpand Template Language*. `https://what-when-how.com/Tutorial/topic-225aimup/Eclipse-Modeling-Project-A-Domain-Specific-Language-Toolkit-629.html`.

Xu Xiaojun, Liu Chang and Song Dawn (2017). "*SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning*". *CoRR*, *abs/1711.0*.

Zhang Lintao, Madigan C. F., Moskewicz M. H. *et al.* (2001). "*Efficient conflict driven learning in a Boolean satisfiability solver*". In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*. 279–285.

Zhou Neng-Fa, Kjellerstrand Håkan and Fruhman Jonathan (2015). *Constraint Solving and Planning with Picat*. Springer.

Zhu Yabing, Zhang Yanfeng, Yang Huili *et al.* (2019). *GANCoder: An Automatic Natural Language-to-Programming Language Translation Approach Based on GAN*. 529–539.

附录 A　*COPDL* 手册

A.1　*COPDL* 简介

COPDL（Combinatorial Optimization Problem Description Language）是一种用于组合优化问题建模的语言。使用 *COPDL* 时，用户只需要列举变量之间的约束关系，而无需指明如何求解该问题——*COPDL* 编译工具将根据约束的性质自动生成可行的求解算法及相应的源代码。

COPDL 支持各种常用的运算符与数学函数，允许用户使用接近于数学公式甚至自然语言的形式描述问题。

相比于传统的使用程序设计语言编写程序求解问题，使用 *COPDL* 生成工具自动生成程序有诸多好处：

- 用户只需具备基本的数学知识即可完成求解工作，无需任何编程基础和算法知识；
- *COPDL* 描述可读性强，便于交流及查错；
- 用户只需考虑如何建模，而无需在思考如何求解问题上花费大量精力和时间；
- 使用自动工具生成程序，可以避免在编程过程中因人为失误产生 bug；
- 用户可以在自动生成的程序的基础上进行二次开发。

A.2　*COPDL* 实例

本节呈现了两个经典问题的 *COPDL* 描述示例，具体的细节将在下一章详细说明。**单词间的空白符（空格、换行、制表符等）可以随意使用。**

A.2.1　最大公约数

求 *A* 和 *B* 的最大公约数。

```
#input
        A of int in [1,10^6];
        B of int in [1,10^6];

#required
        G of int in [0,?];
        X of int;
```

```
        Y of int;
        A = G * X;
        B = G * Y;

#objective
        maximize G;
```

A.2.2 0/1 背包问题

选取若干物品放入背包，在物品总重量和不超过背包容量的条件下，物品总价值和尽可能大。

```
#input
        N of int in [1,100];
        capacity of int in [1,10000];
        profits of (int in [1,1000])[1~N];
        weights of (int in [1,1000])[1~N];

#required
        knapsack of (int in [1,N]){};
        summation
            [weights[i] : forall i (i in knapsack)]
            <= capacity;

#objective
        maximize summation
            [profits[i] : forall i (i in knapsack)];

#output
        knapsack;
```

A.3　*COPDL* 语法

A.3.1　基本框架

一个 *COPDL* 模型由四部分组成：#input，#required，#objetive 和 #output （**“#”后不能有空格**）。每个部分均可省略，但 **#objetive 和 #output 至少存在一个。所有语句均以分号（;）作为结束**。下面将以 0/1 背包问题的描述来说明 *COPDL* 的基本框架。

A.3.1.1　#input 部分

定义问题的所有输入变量，包括变量名、类型和取值范围等。**注意：*COPDL* 编译工具会按照 #input 中列出的变量顺序依次生成相应的读入语句，因此务必保证列出顺序与实际输入顺序一致。**

在示例中，*#input* 部分列出了该问题的四个输入变量，分别是两个整数变量 N 和 $capacity$，以及两个整数数组变量 $profits$ 和 $weights$；同时，定义了各变量的类型和取值范围，数组变量还需定义下标范围。

A.3.1.2　#required 部分

列举变量之间的关系，由若干陈述（statement）组成。陈述分为两类：

- 定义中间变量，格式与 #input 部分相同；
- 一个约束关系，用一个 bool 类型的表达式表示。

在示例中，*#required* 部分首先定义了一个集合变量 $knapsack$，即最终选取物品的下标集合。之后，描述了 $knapsack$ 需要满足的一个约束条件——选取物品的总重量和不超过 $capacity$。

A.3.1.3　#objective 部分

一个目标函数（objective function）。为最大化（maximize）/最小化（minimize）某个表达式的值。特别地，如果问题仅仅要求找到一组符合所有约束的可行解，则该部分为空。

注意：默认情况下，最终的目标函数值将被第一个输出。在 maximize/minimize 前加“@”可禁止输出该值。

在示例中，求解目标为最大化表达式选取物品的总价值和。

A.3.1.4　#output 部分

若需要输出其他结果，则可以在此部分依次列出相应的表达式。同样的，列出顺序与实际输出顺序必须保持一致。

在示例中，要求额外输出最终选取物品的下标集合。

A.3.2 变量定义

在 *COPDL* 中，使用如下语句定义输入变量或中间变量：

```
<标识符> of <类型>
```

A.3.2.1 标识符

变量的标识符是由一系列字母（区分大小写）、数字及下划线组成的字符串，且必须以字母开头。一些保留字不可以作为标识符，包括：

```
int, real, bool, char, function, of, in, true, false,
and, or, not, xor, mod, if, else, forall, exists,
summation, product, count, min, max, minimize, maximize
```

为避免冲突，**目标程序语言（C/C++/Java 等）的保留字亦不可使用**，如 `return`，`main` 等。

A.3.2.2 类型

COPDL 支持三种基本类型和两种复合类型。

基本类型包括：整型 `int`，实型 `real`，布尔型 `bool`。可对基本类型的取值范围做限定：假设基本类型为 *T*，取值范围在 *lb* 至 *ub* 之间，则可表示为 `T in [lb,ub]`。若取值范围的某一侧没有限制，可以用 `?` 表示。

复合类型包括：

- 数组：元素类型为 `T` 的数组可表示为 `T[]`，例如：整型数组 `int[]`，二维布尔型数组 `bool[][]`。若需要限定数组下标的范围在 *li* 至 *ui* 之间，则可表示为 `T[li~ui]`。同样可以用 `?` 表示某一侧没有限制。
- 集合：元素类型为 `T` 的集合可表示为 `T{}`。集合元素无序且不相同。

A.3.2.3 变量定义示例

以下是一些常见的变量类型定义：

```
// 整型变量
i of int
// 限定取值范围的整型变量
n of int in [1,10^5]
// 整型数组
a of int[]
```

```
// 限定下标范围的整型数组（范围可以是表达式）
b of int[1~2*n]
// 限定下标范围及各元素取值范围的整型数组
c of (int in [1,10^6])[1~100]
// 多维数组，各维下标范围不同
d of (int in [1,100])[0~100][-5~5]
// 整型集合
s of int{}
// 限定元素取值范围的整型集合
t of (int in [1,n]){}
```

A.3.3　基本运算

COPDL 支持基本类型上的各种数学运算符，按照优先级从高到低为：

1. 括号 ()
2. 幂运算 ^
3. 乘 *，除 /，取余 mod
4. 加 +，减 -
5. 类型判断 of，取值范围判断 in
6. 非 not，与 and
7. 或 or，异或 xor
8. 比较符号：=，!=，>，<，>=，<=

A.3.4　复合运算

针对复合类型，*COPDL* 内置如下运算：

- 元素选择：数组元素选择 []。优先级仅次于括号。示例：*a[5]*，*a[i+1]*，其中 *a* 是一个数组或元组，元组中元素编号从*1*开始。
- 集合运算：交 *，并 +，差 -，优先级与相同符号的基本运算一致。
- 集合比较：相等 =，不等 !=，被真包含 >，真包含 <，被包含 >=，包含 <=，优先级同比较符号。
- 属于运算 in，优先级同比较符号。表达式 *e in C* 返回一个*bool*值，表示集合类型变量 *C* 中是否存在元素 *e*。
- 聚集运算：连加 summation()，连乘 product()，统计数量 count()，最大值 max()，最小值 min()。这些运算的参数可以是数组或集合。例如：*summation(a)*

计算数组 *a* 所有元素之和，*max(s)* 得到集合 *s* 中的最大元素。

A.3.5 量词

COPDL 支持一阶谓词逻辑中的两个量词 ∃ 和 ∀。对于存在量词 ∃，由于 *COPDL* 本身的约束关系本质上就是表示“存在”，因此不再提供额外支持。对于全称量词 ∀，提供了相应的 forall 表达式。

A.3.5.1 全称量词 ∀

全称量词表达式语法：

```
[<表达式> : forall (<变量列表 v1,v2,...>) (<bool 表达式>)]
```

该语句枚举所有满足后面 bool 表达式的变量取值组合，分别带入前面的表达式得到每个元素值，所有元素构成最终的数组。如果变量列表只有一个变量，可以省略括号。

示例：选取数组 a 中所有下标属于集合 s 的元素。

```
[ a[i] : forall i (i in s) ]
```

A.3.6 特殊约束关系

在 #required 部分，*COPDL* 提供了两种特殊的约束语句来简化描述，并为其提供了针对性的优化。

A.3.6.1 forall 约束

对于一系列形式相似的约束，可用 forall 约束来描述。语法：

```
<bool 表达式> : forall (<变量列表 v1,v2,...>) (<bool 表达式>)
```

该语句枚举所有满足后面 bool 表达式的变量取值组合，分别带入前面的 bool 表达式得到一组约束。**注意：与全称量词生成数组的语法基本相似，区别是最外层没有方括号“[ ]”且最前面的表达式是 bool 类型的。**

示例1：要求数组 a 中元素单调递增。

```
a[i] < a[i+1] : forall i ( 1 <= i and i < N )
```

注意：若变量列表中的变量取值范围可直接由上下文推断而无须额外约束，forall 后面的 bool 表达式可以省略。上例可简写成：

```
a[i] < a[i+1] : forall i
```

示例2：要求数组 a 中元素两两不同。

```
a[i] != a[j] : forall (i, j) (i != j)
```

A.3.6.2　alldiff 约束

若要求一个数组内元素互不相同，可使用 alldiff 约束描述，语法：

```
alldiff <数组表达式>
```

其中数组表达式可以是一个数组变量，或是一个由全称量词生成的数组。

示例：

```
alldiff a
alldiff [ a[i] : forall i (i in s) ]
```

第一个约束是数组 a 中元素互不相同，第二个约束是数组 a 中下标属于集合 s 的元素互不相同。

A.4　常见错误

本章收集了以往用户在使用过程中遇到的一些典型错误、可能的原因以及解决办法，供参考。

A.4.1　编译错误 Compile Error

编译错误可分为四类：语法错误、输出错误、类型错误和语义错误。

A.4.1.1　语法错误

编译信息为：

```
Parse Error!
Encountered ... at line X, column Y.
...
```

遇到此类错误，请仔细检查 X 行 Y 列附近的语句是否符合语法。

A.4.1.2　输出错误

编译信息为：

```
No Output Error!
```

这由于同时缺失了 #objective 和 #output 部分，导致程序无输出结果。

A.4.1.3 类型错误

编译信息为：

```
Type Error!
...
```

造成此类错误的原因是子表达式的类型不符合操作符的要求。请检查各表达式是否正确。

A.4.1.4 语义错误

编译信息为：

```
Cannot be enumerated!
Unbounded variables: ...
```

导致此类错误的原因一般是缺少了对列出变量取值范围的必要限制，或取值范围设置错误。可通过加强/修正变量取值范围来解决。**注意：有时变量名错误也有可能产生此类错误。**

A.4.2 其他错误

其他错误包括答案错误（Wrong Answer），超时（Time Limit Exceeded）等，错误原因一般是 *COPDL* 描述与题意不完全相符。建议解决步骤：

1. 检查对题意的理解是否正确；
2. 检查是否遗漏某些重要约束条件；
3. 检查 *COPDL* 描述细节是否有误。

以下列举了 *COPDL* 描述细节的常见错误，在检查第三步时可以重点关注：

- 变量取值范围缺失或错误
- 变量名不一致
- 比较符号错误，例如“<=”写成“<”
- 输入/输出顺序与题目要求不符

此外，还可以**将生成的程序拷贝到本地运行，利用样例输入输出来定位错误。**

附录 B　基础课程期末试题数据集信息及 *COPDL2C* 有效性测试结果

本附录列出了由北京大学大一、大二计算机专业的程序设计和算法分析类基础课程在 2018–2019 学年及 2019–2020 学年的**所有** 49 道期末上机考试题组成的数据集。该数据集被用于验证未优化的 *COPDL2C* 的有效性（本文第三章）以及优化过的 *COPDL2C* 的有效性（本文第四章）。

数据集中试题来源于三门课程各两次的期末上机考试题，包括计算概论 A（2018 年秋季学期，2019 年秋季学期），程序设计实习（2019 年春季学期，2020 年春季学期），以及数据结构与算法 A（2018 年秋季学期，2019 年秋季学期）。各课程期末上机考试中使用的试题列表及 *COPDL2C*（未优化/已优化）在这些问题上的求解表现参见表 B.1–B.6。各表中的三种有效性测试结果标记含义如下：

- “√”**完全通过**：生成的求解程序能够完全通过所有测试数据；
- “○”**部分通过**：生成的求解程序因求解效率不够高效，仅能通过部分小规模测试数据；
- “×”**无法表达**：无法使用 *COPDL* 描述试题，原因是试题本身非组合优化问题或包含 *COPDL* 语法暂不支持的复杂元素（输出格式、字符串操作、过程控制等）。

B.1　计算概论 A 课程期末上机考试题详情及实验结果

表 B.1 和表 B.2 分别列出了计算概论 A 课程在 2018 年秋季学期和 2019 年秋季学期的两次期末上机考试试题列表及实验结果。两次上机考试在 OpenJudge 平台上的公开访问链接分别为 `http://ica.openjudge.cn/2018winter/` 以及 `http://ica.openjudge.cn/2019winter/`。

B.2　程序设计实习课程期末上机考试题详情及实验结果

表 B.3 和表 B.4 分别列出了程序设计实习课程在 2019 年春季学期和 2020 年春季学期的两次期末上机考试试题列表及实验结果。两次上机考试在 OpenJudge 平台上的公开访问链接分别为 `http://cxsjsx.openjudge.cn/2019finalexam/` 以及 `http://cxsjsx.openjudge.cn/2020finalexam/`。

表 B.1　计算概论 A（2018 年秋）期末试题列表及有效性测试结果

编号	题目名	未优化	已优化
1	集体照	√	√
2	石头剪刀布	√	√
3	侃侃而谈的四位朋友	√	√
4	护林员盖房子	√	√
5	垂直直方图	×	×
6	最佳凑单	√	√
7	分成互质组	√	√
8	佳佳的筷子	○	√

表 B.2　计算概论 A（2019 年秋）期末试题列表及有效性测试结果

编号	题目名	未优化	已优化
1	整数序列的元素最大跨度值	√	√
2	与 7 无关的数	√	√
3	好人坏人	√	√
4	谁能拿到最多的硬币	√	√
5	错误探测	√	√
6	循环数	√	√
7	小游戏	○	√
8	最大子矩阵	√	√
9	采药	○	√

表 B.3　程序设计实习（2019 年春）期末试题列表及有效性测试结果

编号	题目名	未优化	已优化
1	错误探测	√	√
2	移动办公	○	√
3	重启系统	√	√
4	苹果消消乐	○	√
5	课程小作业	○	√
6	宝藏	○	√
7	课程表	√	√
8	密室逃脱	○	√
9	控制公司	√	√
10	冠军之路	○	○
11	他和他的猫	○	○
12	多连块拼图	×	×

表 B.4　程序设计实习（2020 年春）期末试题列表及有效性测试结果

编号	题目名	未优化	已优化
1	矩形数量	√	√
2	邮票收集	○	√
3	田忌赛马	√	√
4	漫漫回国路	○	√
5	Project Summer	○	√
6	物资打包	√	√
7	删除数字	○	√
8	最长的环	○	○

B.3　数据结构与算法 A 课程期末上机考试题详情及实验结果

表 B.5 和表 B.6 分别列出了数据结构与算法 A 课程在 2018 年秋季学期和 2019 年秋季学期的两次期末上机考试试题列表及实验结果。两次上机考试在 OpenJudge 平台上的公开访问链接分别为 `http://dsa.openjudge.cn/final20181223/` 以及 `http://dsa.openjudge.cn/final2019/`。

表 B.5　数据结构与算法 A（2018 年秋）期末试题列表及有效性测试结果

编号	题目名	未优化	已优化
1	词典	×	×
2	Field Reduction	○	√
3	Ultra-QuickSort	×	×
4	表达式·表达式·表达式求值	×	×
5	Angry Cows	○	√
6	广告计划	○	○

表 B.6　数据结构与算法 A（2019 年秋）期末试题列表及有效性测试结果

编号	题目名	未优化	已优化
1	Huffman 编码树	×	×
2	宗教信仰	○	√
3	舰队、海域出击!	○	○
4	现代艺术	○	○
5	越野滑雪	○	√
6	寻找匹配	×	×

附录 C　包含 40 个用自然语言描述的组合优化问题的数据集

本附录列出了包含 40 个用自然语言描述的组合优化问题的数据集详细内容，包括来源于大学本科一、二年级程序设计类与算法设计类课程例题、作业与测试的 33 个问题，以及 7 个来源于 MiniZinc 教程或标准测试集的问题。

C.1　来源于课程例题、作业与测试的 33 个问题的自然语言描述

表 C.1 列出了来源于课程例题、作业与测试的 33 个问题的自然语言描述。

表 C.1　来源于课程例题、作业与测试的 33 个问题的自然语言描述

编号	题目名	题目描述
1	最大公约数	Given A and B. Find the greatest D which is a common divisor of both A and B.
2	最小公倍数	Given A and B. Find the least positive M which is a common multiple of both A and B.
3	鸡兔同笼	In the yard were C chickens and R rabbits. Each chicken has one head and two legs. Each rabbit has one head and four legs. There are total 27 heads and 86 legs in the yard. Calculate C and R.
4	哥德巴赫猜想	Given the sum of prime A and prime B, find A and B.
5	最大序列乘积	Find a sequence of M positive numbers with the maximum product, while the sum of them is N.
6	同余问题	Given A and B. Find the smallest X that X is greater than 1 and A modulo X equals to B modulo X.
7	分苹果	There are N children. Each child has a positive number, and the numbers are different from each other. Given the sum of all numbers of children. Find a possible assignment.

续表 C.1　来源于课程例题、作业与测试的 33 个问题的自然语言描述

编号	题目名	题目描述
8	和为 K	Given a sequence of N numbers. Find different numbers A and B in the sequence so that the sum of A and B equals to K.
9	不超过 K 的最大和	Given N numbers, find two different numbers from them such that the sum of the two numbers is maximum but not exceeding K.
10	附近最大的城市	There are N cities, the length of edges between cities is given. Please find the city with the largest index id satisfying that there exisits a path between city 1 and city id and the length of this path is no more than K.
11	金币问题	There are N kinds of coins. Each kind of coins has a value V and a weight W. Tony wants to go traveling. Unfortunately, he can only carry coins of which the total weight is not greater than K. How much total value of coins can he carry at most?
12	重复的数	Given a sequence of N numbers, find a number A that the count of A in the sequence is at least two.
13	信使问题	There are N nodes in the graph. Given the length of edges between the nodes, find the shortest path of size N and the nodes in the path are all different.
14	团队合作	There are N candidates. Each candidate has a cooperation value and a working value. Select a subset of candidates to form a team, such that the sum of cooperation values is positive and the sum of working values is maximum.
15	不定方程	Given A, B and C. Find X and Y so that A times X plus B times Y equals to C.

续表 C.1　来源于课程例题、作业与测试的 33 个问题的自然语言描述

编号	题目名	题目描述
16	3 或 5 的倍数	How many positive integers are there which is less than N and is a multiple of 3 or a multiple of 5.
17	约数和倍数	Given D and M. D is the divisor of both A and B, while M is the multiple of both A and B. Find A and B.
18	序列搜索	Given a sequence S of N numbers. Find the least index id so that S[id] equals to K.
19	最接近的数	Given a sequence of N number and K. Find the nearest number X in the sequence that the absolute value of (X minus K) is minimum.
20	众数	Given a sequence of N number. Find the mode M in the sequence so that the number of M's appear in the sequence is maximum.
21	和数	Given a sequence of N number. How many numbers in the sequence are there which equal to the sum of two different numbers in the sequence.
22	约数问题	Find the minimum positive number A, so that both A and (M minus A) are divisors of N.
23	恰好相等	Find a subset of N numbers. The sum of the numbers in the subset is exactly equal to T.
24	糖果问题	There are N bags. Each bag has some candies. You can take some of the bags. the total number of candies in these bags should be a multiple of K because you want to divide the candies equally between K friends. How many candies can you take at most?
25	逆序对	Given a sequence S of N numbers. How many pairs of indexes I and J are there satisfying that I is less than J and S[i] is greater than S[J]?

续表 C.1　来源于课程例题、作业与测试的 33 个问题的自然语言描述

编号	题目名	题目描述
26	判断是否为树	Given edges of a graph with N nodes. Check whether it is a tree.
27	旅行开销	There are N cities in the graph. Given the length and the cost of each edge. Find path from city 1 to city N. The length of the path should be shortest and the cost of it should be less than M.
28	股票买卖	Given the predictive stock prices P of N days. Find two days I and J to buy and sell the stock. I is less than J and (P[J] minus P[I]) should be maximum.
29	回家	There are N cities in the graph. Given the edges between the cities. Can you find a path from city 1 to city N?
30	判断质数	Is the given N a prime number?
31	最小质数	Find a minimum prime which is greater than N.
32	出现奇数次	Given a sequence of N numbers. Find X that the number of X's in the sequence is odd.
33	找不同的数	Select some of N values. The selected values are all different and the number of the selected values is maximum.

C.2　来源于 MiniZinc 教程或标准测试集的 7 个问题的自然语言描述

表 C.2 列出了来源于 MiniZinc 教程或标准测试集的 7 个问题的自然语言描述。

表 C.2　来源于 MiniZinc 教程或标准测试集的 7 个问题的自然语言描述

编号	题目名	题目描述
34	地图着色	Given edges of a graph with N nodes. There are three colors that can be used for coloring nodes. Determine the color of each node so that the colors of nodes of each edge should be different.
35	杂货店问题	Find the prices of 4 items so that the sum of prices is 711 and the product of prices is 711000000. The prices of items are in ascending order.
36	魔幻序列	Find a sequence of N numbers. Each number is equal to the count of (its index minus 1) in the sequence.
37	0/1 背包问题	Given the weights and values of N items, put a subset of items into a knapsack of capacity C to get the maximum total value in the knapsack. The total weight of items in the knapsack does not exceed C.
38	蛋糕烘培	A banana cake takes 250 flour, 2 bananas, 75 sugar and 100 butter, and a chocolate cake takes 200 flour, 75 cocoa, 150 sugar and 150 butter. The profit of a chocolate cake is 45 and the profit of a banana cake is 40. And we have 4000 flour, 6 bananas, 2000 sugar, 500 butter and 500 cocoa. The question is how many of chocolate cakes and banana cakes we should bake for the fete to maximize the total profit.
39	N 皇后	Determine the columns where N queens should be placed. Each column should be greater than or equal to 1 and less than or equal to N. Columns should be all different. For each column, the sum of its value and its index should be not equal to that of any other column; the difference of its value and its index should be not equal to that of any other column.
40	最短路	There is a graph with N nodes. Given the length of each edge between the nodes, find the shortest path from S to E.

个人简历及博士期间研究成果

个人简历

林舒，男，1990 年 7 月出生于福建省福州市。从中学开始接触编程，曾在全国青少年信息学奥林匹克竞赛上（NOI）获得二等奖，并因此于 2009 年保送进入北京大学信息科学技术学院学习。2013 年获得计算机科学与技术专业理学学士学位后，继续在北京大学信息科学技术学院攻读计算机软件与理论专业博士学位至今。

发表论文

- **Shu Lin**, Na Meng, Wenxin Li (2021). "*Generating Efficient Solvers from Constraint Models*". Twenty-Ninth ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21), Athens, Greece. （软件工程领域 CCF A 类会议，已接收）
  论文内容与 *COPDL* 语言的设计和实现（本文第三章）以及求解算法自动优化（本文第四章）相关。
- **Shu Lin**, Na Meng, Wenxin Li (2019). "*Optimizing Constraint Solving via Dynamic Programming*". Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI'19), Macao, China. （人工智能领域 CCF A 类会议，已发表）
  论文内容与动态规划优化在约束求解上的应用（本文第四章）相关。
- **Shu Lin**, Na Meng, Dennis Kafura, Wenxin Li (2021). "*PDL: Scaffolding Problem Solving in Programming Courses*". Twenty-Sixth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'21), Paderborn, Germany. （计算机教育类顶级会议，已接收）
  论文内容与 *COPDL* 语言在编程课程教学上的应用（本文第五章）相关。

参与科研项目

- 非完全信息条件下的博弈决策（科技创新 2030——"新一代人工智能"重大项目，项目号：2018AAA01009）。主要研究问题建模，并发表一篇相关论文。
- 北京大学、腾讯公司"青少年编程教育"项目。主要研究并提出基于问题建模的新教学方法，参与课程设计、平台建设，并发表一篇相关论文。

- 基于国家高性能计算环境的HPC教育实验平台 2.0（国家重点研发计划项目，项目号：2018YFB0204100）。主要参与教育实践平台设计和建设。
- 面向全流程智慧健康管理决策的多源异构大数据融合方法研究（国家自然科学基金重点项目，项目号：91646202）。主要研究约束求解优化算法，并发表一篇相关论文。
- 基于国家高性能计算环境的教育实践平台（国家重点研发计划项目，项目号：2016YFB0201900）。主要参与教育实践平台设计和建设。

已登记软件著作权

- 周昊宇, **林舒**, 张勤健, 李文新, 余华山. 临湖草堂·并行程序评测题库平台. 登记号：2017SR688148. 2017 年 10 月.

获得奖励

- 北大天网-明略科技创新奖学金（2019）
- 北大天网-搜狐研发奖学金（2015）
- 研究生专项奖学金（2015）

参与校内外工作

- 自 2014 年起担任 CCF NOI 科学委员会学生委员，参与中学生程序设计竞赛（NOIP，NOI）的命题工作。
- 自 2013 年至 2017 年担任北京大学程序设计竞赛命题组组长。
- 曾担任北京大学多门课程的助教，包括：《游戏AI中的算法》《通用人工智能和非公理推理系统》《算法设计与分析》《数据结构与算法》《计算机系统导论》《程序设计实习》。

致谢

我于 2013 年在北京大学获得学士学位后，选择保送本校研究生继续深造，并于 2015 年由硕士转为博士。今年，已是我研究生生涯的第八年，同时也是进入实验室的第十年和在北大生活的第十二年。在攻读博士学位期间，我经历过诸多困难，包括初期选题的迷茫，中期论文的屡投不中，以及后期因疫情等原因造成的种种不顺，直到今天，总算取得了较为满意的研究成果。在这过程中，我从一个只是单向汲取知识的学习者，逐渐蜕变为有独立思考、创造和实践能力的学术研究者，并在自己感兴趣的研究领域中有所建树，为未来从事科研工作奠定了坚实的基础。

回望这八年的研究生生涯，帮助过我的老师、同学、亲人、朋友有太多太多。

我首先要衷心感谢我的导师李文新老师。从本科生阶段开始，李老师指引我逐步完成了问题提出、理论推导、工程实践、文献检索、实验设计、论文写作投稿等研究工作中的各个环节，并全力支持我参加各相关领域顶级会议以及各类学术交流，拓宽了我的研究思路，也让我积累了许多经验和自信。同时，李老师也教会了我许多学术道路甚至是人生道路上的哲理，帮助我成为一名优秀的科研工作者。

感谢孟娜老师。孟老师在我的论文投稿过程中，对我研究工作的方法改进、理论总结、实验设计、论文写作等方面上给予耐心指导，并在 2019 年春季支持我到美国弗吉尼亚理工大学进行短期交流，让我体验到了不同的学术环境和氛围。

感谢熊英飞老师。熊老师在我研究过程的前期和中期，对研究中的语言设计、方法总结、相关工作收集等方面提供了大量的指导和帮助，让我明确了研究工作的主攻方向。

感谢许卓群老师。许老师在我一开始选择博士阶段研究方向的时候，认真听取我的想法并仔细分析可行性和创新性，为我研究工作的后续开展打下坚实的基础。感谢王千祥老师、陈一峯老师、金芝老师在研究工作初期的理论和实践上提供了许多研究思路和方向。同样感谢陈向群老师、邓小铁老师、张铭老师、张世琨老师、郭耀老师、罗国杰老师、周明辉老师、汪小林老师、罗英伟老师，各位老师参与了我的综合考试、论文选题、预答辩等研究生阶段的各个关键环节，并提出了许多宝贵的建设性修改意见。

感谢张勤健师兄和杜仲轩师兄在北京大学程序在线评测平台（POJ 以及 OpenJudge）的设计和实现上作出的巨大贡献。两位师兄带着我从入门到精通，逐渐熟悉整个在线评测平台的架构和运行流程，并帮助我搭建了本文所需的实验环境。感谢张海峰师兄同时也是三年室友。通过学习和研究师兄推荐的博弈、强化学习等课程，夯

实了我的理论基础，也丰富了我的研究方法，此外，师兄也在我今后的学术道路规划上提供了许多建议和帮助。感谢鲁云龙师弟在我的博士预答辩和答辩过程中协助我整理反馈信息，为我修改和完善博士论文提供了诸多便利。

还要感谢人工智能实验室的其他各位师兄弟姐妹们，包括许国雄师兄、黄贝宁师兄、刘石磊师兄、郑何师兄、洪星星师兄、叶亚鹏、倪燎、周昊宇、张艺、王政飞、王鑫超、周昱杉、李昂、汪永毅。虽然主要研究方向有所不同，但在每周组会的相互学习和交流过程中，伙伴们给予了我许多新的启发。同时，与实验室的伙伴们一同参加 LAB 杯比赛、网络所年会等活动的经历，也是我博士生涯的一块重要组件。

最后，感谢我的家人，特别是我的妻子郑杰。她一如既往地在生活上和精神上给予我莫大的支持和鼓励，是我在攻读博士学位道路上最坚强的后盾。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：[illegible]　日期：2021 年 4 月 7 日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校□一年/□两年/□三年以后，在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：[illegible]　导师签名：[illegible]

日期：2021 年 4 月 7 日