

硕士学位论文

求解 AllDifferent 约束的局部搜索算法

作者姓名: _____

指导教师: _____

学位类别: 电子信息硕士

学科专业: 软件工程

培养单位: _____

2024 年 6 月

Local search algorithm for solving AllDifferent constraints

A thesis submitted to

University

in partial fulfillment of the requirement

for the degree of

Master of Engineering

in Software Engineering

By

June, 2024

大学
学位论文原创性声明

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。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：

日 期：

大学
学位论文授权使用声明

本人完全了解并同意遵守 xxx 有关保存和使用学位论文的规定，即 xxx 有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：

日 期：

导师签名：

日 期：

摘要

约束满足问题是人工智能和运筹学中的一个数学问题定义。在约束满足问题中,需要找出满足一组约束的解,其中约束是对可能解的限制条件。约束满足问题可以用来描述许多实际问题,例如时间表编制、资源分配、电路设计等。约束编程是一种解决约束满足问题的编程范式,其核心思想是将问题建模为一组变量和约束,用高级、声明式的方式描述问题,然后使用约束传播、回溯搜索等算法寻找满足所有约束的解。其中, AllDifferent 约束是一种常见且基本的全局约束,它要求一组变量的取值都不相同,从而有效地减少搜索空间,排除许多显然不可能的解,使搜索过程更加高效。这种约束在许多经典的约束满足问题中都有应用,例如数独问题,八皇后问题等。针对此类约束,目前已经提出了许多过滤算法和搜索算法,以剪枝和加速求解。然而由于 AllDifferent 约束的内在复杂性,对于现有的基于搜索的求解器来说,解决包含大规模 AllDifferent 约束的约束满足问题仍然具有挑战性。

在本文中,我们深入探讨 AllDifferent 约束的特性,提出了用于求解 AllDifferent 约束的高效的局部搜索算法。首先,我们将问题中的一组 AllDifferent 约束转化为图表示,该图将 AllDifferent 约束分解为多个子约束,从而更细粒度地描述了 AllDifferent 约束涉及的变量和变量表达式,以及变量表达式之间的关系。接下来,鉴于图中涉及的两类关系,我们设计了两个多项式时间的规约规则来简化这个图,对问题进行预处理来减小问题的规模。之后,我们通过局部搜索算法对问题进行求解。首先,由于修改候选解的操作涉及到对变量和赋值的选择,我们提出了一种分步选择操作的策略来修改候选解,对于选择的两个步骤分别设计新的打分函数,并通过图中的邻域关系在平局出现时打破平局。除此之外,我们还精心设计了禁忌和重启策略。禁忌策略可以防止算法陷入局部最优,而重启策略则可以在搜索陷入困境时及时调整搜索的方向,这两种策略的设计都使算法能更有效地探索搜索空间。在禁忌策略中,我们将操作的禁忌分为了三部分,从而适配分步选择操作的策略。而在重启策略中,我们维护了候选解池,并在获得次优解时对约束图进行加权操作,使算法能更有效地探索搜索空间。

利用上述算法,我们实现了一个名为 AllDiff-LS 的高效约束求解工具,专门用于处理 AllDifferent 约束。该工具支持各种算术表达式,并能处理包含其他二元约束的约束满足问题。通过将问题编码为约束,我们可以利用 AllDiff-LS 来求解这些约束,并提供满足所有约束的解。我们在多种经典的 CSP 问题上进行了实验,如数独、N 皇后、全间隔和正交拉丁方问题。实验结果显示,与针对性的启发式求解算法以及基于启发式和完备的通用求解器相比,我们的方法在求解效率上表现更优,特别是在大规模实例上,我们能够解决更多的挑战性问题。

关键词: 约束编程, AllDifferent 约束, 局部搜索

Abstract

Constraint satisfaction problem is a mathematical problem definition in artificial intelligence and operations research. In constraint satisfaction problems, the goal is to find a solution that satisfies a set of constraints, where the constraints are conditions that limit the possible solutions. Constraint satisfaction problems can be used to describe many real-world issues such as scheduling, resource allocation, circuit design, etc. Constraint programming is a programming paradigm for solving constraint satisfaction problems, which involves modeling the problem with a set of variables and constraints, describing the problem in a high-level, declarative manner, and then using algorithms like constraint propagation and backtracking search to find solutions that satisfy all constraints. Among them, the AllDifferent constraint is a common and fundamental global constraint that requires the values of a group of variables to all be different, effectively reducing the search space, eliminating many obviously impossible solutions, and making the search process more efficient. This type of constraint is applied in many classic constraint satisfaction problems such as Sudoku and the eight queens puzzle. For such constraints, numerous filtering and search algorithms have been proposed to prune and accelerate the solving process. However, due to the inherent complexity of the AllDifferent constraint, solving constraint satisfaction problems with large-scale AllDifferent constraints remains challenging for existing search-based solvers.

In this paper, we delve into the characteristics of the AllDifferent constraint and propose an efficient local search algorithm for solving the AllDifferent constraint. Initially, we transform a set of AllDifferent constraints in the problem into a graph representation, breaking down the AllDifferent constraint into multiple sub-constraints to describe the variables and variable expressions involved in the AllDifferent constraint more finely, as well as the relationships between variable expressions. Subsequently, considering the two types of relationships involved in the graph, we devise two polynomial-time reduction rules to simplify this graph, preprocessing the problem to reduce its scale. Following this, we solve the problem using a local search algorithm. Firstly, as the operation of modifying candidate solutions involves selecting variables and assignments, we propose a stepwise selection strategy to modify candidate solutions, designing new scoring functions for each of the two steps and breaking ties when they occur using the neighborhood relationships in the graph. Additionally, we carefully design taboo and restart strategies. The taboo strategy prevents the algorithm from getting stuck in local optima, while the restart strategy adjusts the search direction promptly when the search gets stuck, enabling the algorithm to explore the search space more effectively. In the taboo strategy, we divide the taboo of operations into three parts to

adapt to the stepwise selection strategy. In the restart strategy, we maintain a pool of candidate solutions and perform weighted operations on the constraint graph when obtaining suboptimal solutions, allowing the algorithm to explore the search space more effectively.

Using the algorithm mentioned above, we have developed an efficient constraint solving tool called AllDiff-LS, specifically designed for handling AllDifferent constraints. This tool supports various arithmetic expressions and can deal with constraint satisfaction problems containing other binary constraints. By encoding problems as constraints, we can utilize AllDiff-LS to solve these constraints and provide solutions that satisfy all constraints. We conducted experiments on various classical CSP problems such as Sudoku, N-Queens, All-Interval, and Orthogonal Latin Squares. The experimental results demonstrate that our method outperforms targeted heuristic algorithms, as well as heuristic-based and complete general solvers in terms of solving efficiency, particularly excelling in solving more challenging problems on a large scale.

Keywords: Constraint Programming, AllDifferent Constraints, Local Search

目 录

第 1 章 绪论	1
1.1 研究背景及意义	1
1.1.1 约束满足问题	1
1.1.2 AllDifferent 约束	3
1.2 论文主要工作	4
1.3 论文组织	5
第 2 章 相关技术及研究现状	7
2.1 约束满足问题求解现状	7
2.1.1 约束编程求解	8
2.1.2 其他求解方法	10
2.2 AllDifferent 约束求解现状	11
2.2.1 图论基础	11
2.2.2 AllDifferent 约束和二部图匹配	12
2.2.3 AllDifferent 约束的过滤算法	14
2.3 局部搜索算法现状	16
2.4 本章小结	18
第 3 章 AllDiff-LS 的基础组件和算法框架	19
3.1 问题介绍和求解思路	19
3.2 AllDifferent 约束转化为图及其优化	20
3.3 分步选择操作策略	23
3.4 禁忌策略	25
3.5 打破平局策略	26
3.6 局部搜索算法框架	27
3.7 本章小结	30
第 4 章 针对 AllDifferent 约束的加权和重启策略	31
4.1 解池技术	31
4.2 约束加权图和动态迭代策略	32
4.3 AllDiff-LS 工具设计	34
4.4 本章小结	35

第 5 章 实验设计及结果分析	37
5.1 实验设置	37
5.2 AllDiff-LS 求解 AllDifferent 约束的能力	39
5.3 AllDiff-LS 涉及的策略的有效性	43
5.4 本章小结	45
第 6 章 总结与展望	47
6.1 工作总结	47
6.2 下一步的工作	48
参考文献	49
作者简历及攻读学位期间发表的学术论文与研究成果	53

图形列表

1.1 工作的整体框架。	5
2.1 AllDifferent 约束和其二部图表示。	13
2.2 值图与残差有向图的转换。	16
3.1 例子中约束转化得到约束图。	21
3.2 例子中约束图的化简。	22
3.3 Local Search 的求解流程。	28
4.1 名为 “AI Escargot” 的数独实例。	35
5.1 四种方法在给定基准测试上的结果。当 <i>avg_time</i> 为 1000s 时，意味着 超时。	41
5.2 在四类实例上 AllDiff-LS（横坐标）和其他两个版本（纵坐标）的平 均运行时间比较。	44
5.3 在数独上 AllDiff-LS（横坐标）与其他三个版本（纵坐标）的平均运 行时间比较。	44

表格列表

5.1 AllDiff-LS 和其他最先进的基线方法在数独实例上的结果。	40
5.2 AllDiff-LS 和其他最先进的基准方法在 N 皇后, 全间隔和 2-MOLS 问题上的结果。	40
5.3 OR-Tools、LocalSolver 和 SAT 求解器 Kissat 在数独实例上的结果。 .	42
5.4 OR-Tools 和 LocalSolver 在其他实例上的结果。	42
5.5 在给定数独实例上使用不同化简规则对应的变量简化比率。	43

第1章 绪论

本章将简要介绍本文研究主题的背景和意义。首先，本文会给出一些基础概念，这些概念在后续章节中会频繁使用，主要包括约束满足问题、AllDifferent 约束以及其主要的求解算法，并引出研究该问题的动机。接着，本文会详细概述论文的主要工作内容。最后，本文将介绍论文的组织结构和框架。

1.1 研究背景及意义

在当下信息时代，随着计算机科学和信息技术的飞速发展，约束满足问题 (Constraint Satisfaction Problem, CSP) 在各个领域中的应用也越来越广泛。无论是在工业生产、交通调度、金融决策，还是在人工智能、数据挖掘等高科技领域，我们都可以看到约束满足问题的身影。随着问题规模的增大和复杂度的提升，如何有效、高效地解决约束满足问题，已经成为了一个亟待解决的重要挑战。若未能及时有效地处理这些约束满足问题，可能会对相关领域的运作造成一定的影响。因此，研究一种高效求解约束满足问题的算法具有一定的理论意义和实践价值。

1.1.1 约束满足问题

约束满足问题是一类在各种领域中广泛应用的运筹学问题，它涉及到在一组特定的限制条件下，寻找一种满足所有约束的解决方案。在这类问题中，每一个约束都是对一组变量的取值范围的限制，只有当所有变量的取值都满足所有的约束时，我们才称这个赋值方案为一个解。求解约束满足问题的过程就是寻找这样的解的过程。约束满足问题的复杂性主要来自于两个方面：一方面，由于涉及的变量数量可能非常大，使得解空间的规模呈指数级增长；另外，由于约束的复杂性可能非常高，使得检查一个解是否满足所有约束的过程可能比较耗时。

约束满足问题在各个领域中都有广泛应用，其解决方案对于优化决策、提高效率等方面起着重要作用。约束满足问题在生产调度领域得到了广泛应用，它被用来优化生产流程和资源分配。例如，工厂在制定最优生产计划时，需要满足各种生产约束，如设备使用时间、原材料供应等，以此提高生产效率和降低成本。而在人工智能领域，约束满足问题则成为了实现自动推理、知识表示等核心任务的重要工具。智能机器人需要在满足物理规则、任务需求等环境约束的情况下，做出最优的决策和行动。此外，约束满足问题在网络安全领域也有重要应用，它被用于安全策略的制定和漏洞检测。网络管理员需要在满足权限控制、安全规则等安全约束的情况下，制定出最优的安全策略，以保护网络系统的安全。

为了有效解决约束满足问题，学术界和工业界已经提出和发展了许多不同类型的算法。这些算法大致可以分为两类：完备搜索算法^[1]和启发式搜索算法^[2]。

完备搜索算法，如回溯搜索和深度优先搜索等，是最早被提出和使用的求解约束满足问题的算法。这类算法的主要思想是对解空间进行系统的遍历，直到找到一个满足所有约束的解，或者确定不存在这样的解为止。然而，由于约束满足问题的解空间通常非常大，这类算法在处理大规模问题时会面临严重的时间和空间复杂性问题。因此，研究者们提出了启发式搜索算法，这类算法的主要思想是通过一些启发式的策略，如随机性、局部搜索等，来引导搜索过程，以期在较短的时间内找到满足约束的解，或者尽可能好的解。这类算法在处理大规模和复杂的约束满足问题时，表现出了较好的效果。除了前述两种算法，神经网络的研究进展也引发了新的研究热点，也即利用神经方法解决约束满足问题。然而，尽管这种新方法在某些特定结构的问题上表现出色，但在泛化能力和求解准确率方面，仍然存在一定的局限性。接下来，我们将重点介绍前两类算法的求解思路，并介绍算法涉及的各种求解策略。

完备搜索算法，也称为穷举搜索算法，完备搜索算法能够保证找到问题的所有解。最常见的完备搜索算法包括回溯搜索算法等。回溯搜索算法是一种基于试错的策略，它在搜索过程中会记住已经访问过的状态，当发现当前的选择不能导致解的时候，就会撤销上一步的选择，回溯到前一状态，然后再进行选择。深度优先搜索算法则是一种基于堆栈的搜索策略，它总是对最新发现的状态进行扩展，直到找到解或者发现该状态无法扩展为止。虽然完备搜索算法在理论上能够解决所有的约束满足问题，但是在实际应用中，由于解空间的规模通常非常大，导致完备搜索算法在处理大规模问题时面临严重的时间和空间复杂性问题。因此，目前主流的完备搜索算法都结合了约束传播^[3]、学习子句归结^[4]以及重启策略^[5]等，进一步缩小搜索的范围和提高求解的效率。

局部搜索算法是求解约束满足问题的另一种重要方法。相比于完备搜索算法，局部搜索算法不再对所有可能的解进行遍历，而是从一个初始解出发，通过不断地在解的邻域中搜索，寻找满足约束的解，或者使得某种评价函数值更优的解。局部搜索算法的主要优点是能够在较短的时间内找到满足约束的解或者较优的解，特别是在处理大规模问题时，局部搜索算法通常能够提供更高的效率。然而，局部搜索算法也存在一些缺点，例如可能陷入局部最优解，或者在解空间中的搜索过程缺乏方向性等。同时，局部搜索算法一般是不完备的，也即无法判断问题是否是不可满足的（无解）。常见的局部搜索算法包括模拟退火^[6]、遗传算法^[7]、粒子群算法^[8]等算法。这些算法都通过引入一定的随机性和启发性，以期在全局范围内寻找到更优的解。例如，模拟退火算法模拟物理退火过程中的随机搜索，遗传算法则模拟自然界中的进化过程进行搜索。在实际应用中，人们通常会根据问题的特性和需求，选择合适的局部搜索算法，或者将多种算法进行组合，以期得到更好的求解效果。

除了完备搜索算法和局部搜索算法外，还可以将约束满足问题转换为其他类型的问题，如布尔可满足性问题 (SAT)^[9]、可满足性模理论问题 (SMT)^[10]或线性规划问题 (LP)^[11]，然后使用相应的求解器来求解。其中，布尔可满足性

问题是计算机科学中最基本的决策问题之一，通过将 CSP 中的每个变量和约束用多个布尔变量和布尔逻辑公式编码，我们可以将离散的 CSP 问题转化为 SAT 问题，并可以利用 MiniSAT^[12]、Kissat 等成熟的 SAT 求解器对其求解。SMT 问题在 SAT 问题的基础上，进一步引入了一些理论和谓词约束，从而能够编码更高阶的逻辑问题。通过将 CSP 约束转化为函数，我们可以轻松的将 CSP 转化为 SMT 问题，并利用 SMT 求解器（如 Z3^[13]、CVC5^[14] 等）来求解。LP 问题是一种优化问题，CSP 问题也可以视为一类有界的整数规划问题，因此可以利用 CPLEX、Gurobi 等线性规划求解器对其求解。这些方法的主要优点是能够利用已有的成熟工具和算法，将 CSP 问题的求解转化为已知问题的求解，从而提高求解的稳定性。然而，这些方法也存在一些局限性，例如转换过程可能会引入额外的复杂性，或者可能无法完备保留原问题的所有信息等。

1.1.2 AllDifferent 约束

AllDifferent 约束^[15]是 CSP 中的一种常见的全局约束，它要求一组变量的取值都不相同。这种约束在很多实际问题中都有应用，例如在数独游戏中，要求每行、每列和每个子网格内的数字都必须不同，这种限制就可以用 AllDifferent 约束来表示。通常，AllDifferent 约束还常常与其他类型的约束（如线性约束、区间约束等）结合使用，以描述更复杂的问题。由于 AllDifferent 约束要求所有变量的取值都必须不同，因此在搜索过程中，一旦有两个变量被赋予了相同的值，就可以立即判定当前的部分解不可行，从而避免无效的搜索。在许多实际应用中，我们都可以见到 AllDifferent 约束的影子。例如，在处理排班问题时，我们可以利用 AllDifferent 约束来保证在一个特定的时间段内，每个员工的工作任务都是独一无二的，这样就能避免任务冲突。而对于车辆路径问题，该约束可以帮助我们确保每辆车的行驶路径都是各不相同的，这样就能更好地优化路线，减少交通拥堵的可能性。

针对 AllDifferent 约束的求解，主流的方法是采用过滤算法对约束进行化简，之后再通过搜索和约束传播得到满足约束的赋值。最主流的过滤算法是基于匹配的过滤算法^[16]。这种算法将 AllDifferent 约束转化为一个二分图匹配问题：每个变量对应图的一个节点，每个可能的取值也对应一个节点，如果一个变量可以取某个值，则在对应的两个节点之间连一条边。然后，寻找这个图的一个完全匹配，即找到一种方式，使得每个变量节点都与一个取值节点相连，并且所有的边都不相交。如果找不到这样的完全匹配，那么原问题就无解。在实际操作中，通常使用网络流算法（如 Ford-Fulkerson 算法^[17]或 Edmonds-Karp 算法^[18]）来寻找完全匹配。目前关于 AllDifferent 约束的优化思路大多基于此类思想，通过减少或避免匹配的复杂度或者复用匹配过程的信息来优化化简的时间。此外，针对 AllDifferent 约束，还存在一些启发式的策略^[19]，该策略通过为部分约束涉及的变量赋不同的值，将这些约束视为隐含约束，从而只针对其他约束进行求解，之后再通过交换 AllDifferent 约束中变量的赋值来寻找整个问题的可行解。

虽然已有的求解 AllDifferent 约束的算法在很多情况下都能够工作得很好，

但是它们仍然存在一些局限性和挑战。对于大规模的问题，过滤算法可能会遇到效率问题。尽管过滤算法可以有效地剪枝搜索空间，但是在问题规模较大的情况下，过滤过程本身的计算量也可能会变得非常大。特别是当使用基于匹配的过滤算法时，需要解决的二分图匹配问题的规模可能会非常大，导致求解时间过长。启发式策略的性能同样受到问题规模和约束难度的影响，在问题规模变大或约束结构复杂时，无法同时保证多个隐含约束，从而导致算法的性能下降严重。最后，对于一些特殊的 AllDifferent 约束，例如包含了额外的条件或限制的 AllDifferent 约束，现有的算法可能无法直接应用，针对一类问题设计的策略可能无法泛化到更广的范围上去，需要进行额外的处理或修改，这同样增加了算法设计和实现的复杂性。因此，如何进一步提高算法的效率和通用性，仍然是一个重要的研究方向。

1.2 论文主要工作

本文重点关注的是包含 AllDifferent 约束的约束满足问题。本文的目标是研究如何利用 AllDifferent 约束的特性，从而设计高效求解此类问题的局部搜索算法。本文特别关注那些大规模的、困难的样例，因为这些样例在实际应用中很常见，但对现有的算法来说，求解这些样例往往是最具挑战性的。

本文考虑在几类具有代表性的问题上进行实验，期望取得超越目前已有的启发式算法和主流约束编程算法的效果。具体地，本文将数独^[20]、N 皇后^[21]、全间隔^[22]、正交拉丁方问题^[23] 使用 AllDifferent 约束进行了建模，并通过研究这几类问题的特点，设计一个较为完善的针对此类问题的求解算法。在算法部分，本文主要考虑以下几个方面：

- 考虑如何将问题转化为便于采用局部搜索算法的形式，并进行化简；
- 设计用于局部搜索算法的操作算子和代价函数，构造局部搜索框架；
- 设计针对性的补充策略，如禁忌策略、打破平局策略和重启策略等。

对此，本文设计了用于解决包含 AllDifferent 约束的约束满足问题的局部搜索算法，其整体过程如图1.1所示。此外，本文根据该算法开发了相应的求解工具，并通过实验展示了其高效的求解能力。

其中，本文将在图上的操作定义为修改一个变量的赋值，同时一个操作的代价函数定义为修改前后图中存在冲突的边的个数。这意味着我们可以将选择操作的步骤分为两步：首先选择变量，然后选择值。同时，我们可以相应地为每一步设计一个新的代价函数，即变量当前赋值导致的冲突边的个数和改变赋值后导致的冲突边的个数（这两者相减即操作的代价函数）。在定义完局部搜索中最关键的两个概念后，算法的整体框架基本就可以得到了。我们仍然需要设计一些新的策略来保证算法更少地陷入循环和局部最优解，这是局部搜索算法常见的两个挑战。补充的策略可分为禁忌策略、打破平局策略和重启策略三个部分：在禁忌策略中，为操作的两个步骤分别设计了禁忌策略和性能检测器；在打破平局策略中，本文设计了新的细粒度的打分函数；在重启策略中，会根据迭代得到的

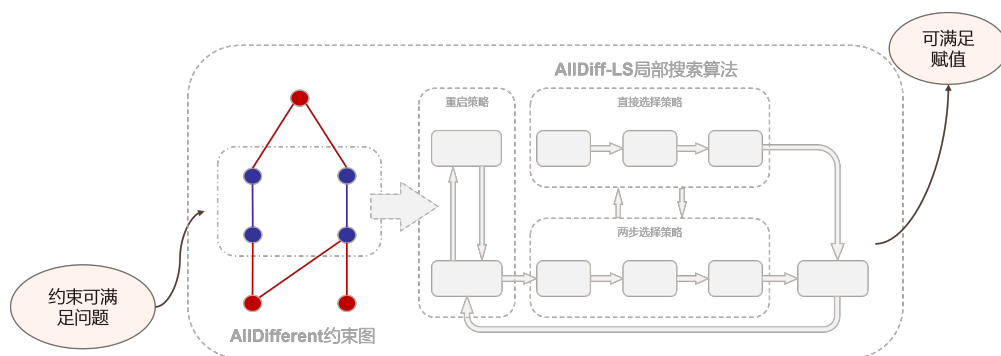


图 1.1 工作的整体框架。

Figure 1.1 The overall framework of the work.

当前解，来动态地调整下一次迭代的打分准则和迭代次数等参数。

具体的，本文介绍的工作主要包含三个方面的贡献：

- 首先，本文将约束满足问题用图进行表示，这是局部搜索擅长求解的问题类型。与以往的策略不同，我们并不会对每个 AllDifferent 约束建图，而是将它们组合在一起，通过提取变量、表达式之间的关系，用一张图表示整个问题。这样得到的异构图有利于我们设计局部搜索需要定义的状态、操作等术语，也便于我们提出更高效的弧一致性算法。

- 其次，本文将根据得到的异构图，设计合适的移动操作（它同时定义了邻域）和打分函数，构造了针对问题求解的局部搜索的整体框架。通过引入其他针对问题特征的优化策略，如重启策略、禁忌策略等，我们可以有保证算法的高效性，这些策略也具有一定的通用性，为局部搜索求解提供了新的思路。

- 最后，本文根据前面提到的求解算法 AllDiff-LS 设计了针对包含 AllDifferent 约束的 CSP 的求解工具，他可以在很短的时间内解决几乎所有问题。通过一系列比较实验，本文证明了算法的有效性，其求解效果在问题规模较小和较大的 CSP 上都极具竞争力，超越了我们找到的已知求解器。此外，本文通过消融实验也证明了算法的各个策略的有效性，这些策略结合在一起保证了算法的高效性。

1.3 论文组织

本文的后续章节按照以下方式组织：

第二章：介绍本文所涉及的相关技术及研究现状。

第三章：介绍本文关于 AllDifferent 约束求解算法的设计思路及实现过程。

第四章：介绍基于第三章算法设计的加权和重启策略。

第五章：介绍本文在实验评估阶段的实验设计及结果分析。

第六章：总结了本文的主要贡献并展望了进一步的研究工作。

第2章 相关技术及研究现状

目前,国内外有很多关于 AllDifferent 约束求解的研究,本章节将介绍一下相关知识。首先,我们将介绍一下 CSP 求解的现状,包括主流的约束编程 (CP) 求解,以及其他 SMT、SAT、LP 求解器的介绍。我们将重点介绍实验部分要进行比较的完备的或启发式的 CP 求解器。其次,我们将介绍 AllDifferent 约束求解的现状,包括经典的 AllDifferent 约束的过滤算法和近几年基于该算法的改进工作。最后,我们将介绍局部搜索算法的现状,以及此类算法在各类 CSP 求解上的进展和突破。

2.1 约束满足问题求解现状

首先,我们先介绍 CSP 的定义,它的特点是要求变量的值域有限且离散。首先我们给出一些前置概念的定义。

对于一个变量 x ,我们用 $D(x)$ 来表示 x 的域,即可以赋予 x 的一组可能的值。通常,我们会使用简写 $x \in \{d_1, \dots, d_m\}$ 来定义 $D(x) = \{d_1, \dots, d_m\}$ 。特别地,在本文的讨论中我们仅考虑那些具有有限域的变量。

设 $Y = y_1, y_2, \dots, y_k$ 是一组有限的变量序列,其中 $k > 0$ 。一个关于 Y 的约束 $c \in C$ 是变量序列 Y 中变量的域的笛卡尔积的子集,即 $c \subseteq D(y_1) \times D(y_2) \times \dots \times D(y_k)$ 。约束可以写作 $c(Y)$ 或 $c(y_1, y_2, \dots, y_k)$ 。如果约束定义在两个变量上,我们称之为二元约束。如果约束定义在两个以上的变量上,我们称之为全局约束。

定义 2.1 (约束满足问题). 一个 CSP,由一组有限的变量序列 $X = x_1, x_2, \dots, x_n$ 及其各自的域 $D = D(x_1), D(x_2), \dots, D(x_n)$ 组成,同时伴有一组约束 C ,每个约束都在 X 的一个子序列上。为了简化表示,我们在表示特定的约束集合时经常省略大括号 “{}”,从而一个 CSP 被表示为 $P = (X, D, C)$ 。

之后,我们给出 CSP 解的定义。设 $P = (X, D, C)$ 为一个 CSP,其中 $X = \{x_1, x_2, \dots, x_n\}$ 且 $D = \{D(x_1), D(x_2), \dots, D(x_n)\}$ 。如果元组 $(d_1, \dots, d_n) \in D(x_1) \times \dots \times D(x_n)$,并且 $(d_{i_1}, d_{i_2}, \dots, d_{i_m}) \in c$, 其中 $c \in C$,那么,我们称关于 X 的这组赋值满足在变量 $x_{i_1}, x_{i_2}, \dots, x_{i_m}$ 上的约束 C 。

对于变量序列 K ,定义 $D(K) = \bigcup_{x \in K} D(x)$ 。当变量 x 的域 $D(x)$ 是单元素集,即 $D(x) = \{d\}$ 时,我们将其简写为 $x = d$,记作单域。

我们称一个 CSP 是一致的,如果这个 CSP 存在解。相反,不存在解的 CSP 是不一致的。一个失败的 CSP 指具有空域(也即存在变量的值域为空)的 CSP,或者只有单域的 CSP 且这些值并不能共同构成 CSP 的解。一个 CSP 是已解决的,若该 CSP 只有单域,并且其中的赋值可以共同构成 CSP 的解。需要注意的是,一个失败的 CSP 也是不一致的,但并非所有不一致的 CSP 都是失败的。

设 $P = (X, D, C)$ 和 $P' = (X, D', C')$ 都是 CSP。如果 P 和 P' 有相同的解集，那么它们被称为等价的。如果 P 和 P' 等价，且对所有的 $x \in X$ ，都有 $D(x) \subseteq D'(x)$ ，那么我们说 P 小于 P' 。这种关系记作 $P \leq P'$ 。如果 $P \leq P'$ 并且至少存在一个 $x \in X$ 使得 $D(x) \subset D'(x)$ ，那么我们说 P 严格小于 P' 。这种关系记作 $P < P'$ 。当 $P \leq P'$ 和 $P' \leq P$ 都成立时，记作 $P \equiv P'$ 。

2.1.1 约束编程求解

约束编程的目标是寻找给定的 CSP 的一个解（或所有解）。求解过程中，值域过滤、约束传播和搜索是交织进行的。

对 CSP 的解（或所有解）的寻找是通过迭代地将一个 CSP 分解为更小的 CSP 来进行的。这个分解过程被称为构建搜索树。树的一个节点代表一个 CSP。最初，在根节点，我们得到一个待解决的 CSP，记作 P_0 。如果 P_0 既没有被解决也没有失败，我们就将 P_0 分解为两个或更多的 CSP，记作 P_1, P_2, \dots, P_k ($k > 1$)。同时，我们必须确保 P_0 的所有解都被保留，而且没有多余的解被添加到 P_0 中。从而， P_1, P_2, \dots, P_k 的解集的并集等于 P_0 的解集。此外，每个 CSP P_i ($i > 0$) 应该严格小于 P_0 ，以确保分解过程能够终止。接下来，我们根据上述相同的标准分解每个 CSP P_1, P_2, \dots, P_k 。分解会一直进行，直到所有 CSP 被分解为失败的或已解决的 CSP。失败和已解决的 CSP 是搜索树的叶子节点。

搜索树的大小与 P_0 中变量数量呈指数关系。为了减小搜索树的大小，约束编程使用了一个叫做约束传播的过程。给定一个约束 C ，一个值域过滤算法会从 C 中的变量的值域中移除与 C 不一致的值。算法必须保留所有的解并且不向 C 中添加任何解。每当一个不一致的值域值被移除，其效果会通过所有其他共享相同对应变量的约束进行传播。这个迭代过程持续进行，直到在 CSP 中的所有约束都未检测到更多的不一致值。此时，我们就说 CSP 是局部一致的。局部一致性反映了算法并没有得到一个全局一致的 CSP，而是得到了一个所有约束都是局部的，也即单独的、一致的 CSP。这里我们将介绍四个常见的局部一致性：弧一致性、超弧一致性、边界一致性和区间一致性。

定义 2.2 (弧一致性). 一个二元约束 $c(x_1, x_2)$ 是弧一致的，如果对于 x_1 的所有值 $d_1 \in D(x_1)$ ，存在一个值 $d_2 \in D(x_2)$ 使得 $(d_1, d_2) \in c$ ，并且对于 x_2 的所有值 $d_2 \in D(x_2)$ ，存在一个值 $d_1 \in D(x_1)$ 使得 $(d_1, d_2) \in c$ 。

定义 2.3 (超弧一致性). 一个约束 $c(x_1, \dots, x_m)$ ($m > 1$) 是超弧一致的，如果对于所有 $i \in \{1, \dots, m\}$ 和所有 $d_i \in D(x_i)$ ，存在值 $d_j \in D(x_j)$ (其中 $j \in \{1, \dots, m\} - i$)，使得 $(d_1, \dots, d_m) \in c$ 。

超弧一致性又称全局弧一致性 (Generalized Arc Consistency, GAC)，相较于弧一致性限制性更强，应用也更广。需要注意的是，弧一致性相当于应用于二元约束的超弧一致性。弧一致性和超弧一致性都确保每个域中的所有值都属于满足约束的元组，与当前变量域相关。另外两种局部一致性概念主要关注变量

域的边界。因此，当应用这些定义时，我们假设涉及的变量域是一个固定的、线性有序的、有限域的元素集的子集。设 D 是这样的一个域，我们定义 $\min D$ 和 $\max D$ 分别为其最小值和最大值。此外，我们使用大括号 “{ }” 和方括号 “[]” 来分别表示一组域值和一个域值区间。

定义 2.4 (边界一致性). 一个约束 $c(x_1, \dots, x_m)$ ($m > 1$) 是边界一致的，如果对于所有 $i \in \{1, \dots, m\}$ 和每个值 $d_i \in \{\min D(x_i), \max D(x_i)\}$ ，存在值 $d_j \in [\min D(x_j), \max D(x_j)]$ (其中 $j \in \{1, \dots, m\} - i$)，使得 $(d_1, \dots, d_m) \in c$ 。

定义 2.5 (区间一致性). 一个约束 $c(x_1, \dots, x_m)$ ($m > 1$) 是区间一致的，如果对于所有 $i \in \{1, \dots, m\}$ 和所有 $d_i \in D(x_i)$ ，存在值 $d_j \in [\min D(x_j), \max D(x_j)]$ (其中 $j \in \{1, \dots, m\} - i$)，使得 $(d_1, \dots, d_m) \in c$ 。

在分解一个 CSP 之后，值域过滤和约束传播被应用到更小的 CSP 上。值的移除导致搜索树变小，从而加速了解决过程。同时，花费在约束传播上的时间应该小于它引发的加速，以便改善效果显著。关于约束传播过程的详细描述可以在^[24]和^[25]中找到，这里不再展开。总之，我们希望应用高效的过滤算法，其效率通常由应用于约束的局部一致性的概念来决定，而在解决过程中何时应用哪种局部一致性的概念则取决于问题本身。

当前主流的求解器分为完备的和启发式的求解器，一些求解器是商用的，而一些则是开源的，他们具有不同的特点，这里我们介绍三个主流的 CP 求解器。

- Choco。Choco¹是一个基于 Java 的约束满足问题 (CSP) 求解库。它提供了一种声明式语言，用于描述和解决各种复杂的约束满足问题。Choco 常被用作研究和教学工具，并且支持各种类型的约束，包括数值约束、逻辑约束等。

- CPLEX Optimizer。CPLEX Optimizer²是 IBM ILOG 的一款产品，它包含一个称为 CP Optimizer 的模块，这是一个约束编程求解器。其主要优点是拥有丰富的 API，支持多种编程语言，如 C++、Java、Python 等，并且它的求解速度非常快，可以处理非常大规模的问题。

- Yuck。Yuck 是一个基于 Scala 的运筹学和约束编程库，它的主要特点是它采用了一种基于邻域搜索的算法，可以在解空间中快速找到优质的解。作为一款启发式求解器，他在近几年的 CP 求解启发式赛道中都是第一名。

这些求解器都支持集成在 Minizinc^[26] 中。MiniZinc 是一种中间级别的约束建模语言，设计用于描述各种组合优化问题。它的主要目标是提供一种简单、可移植且高效的方式来描述这些问题，以便可以使用各种不同的求解器来解决它们。并且，它支持数组、集合和函数等高级数据结构，以及各种数学和逻辑运算符，并且可以使用 CPLEX、Choco、Gecode 等求解器作为后端求解器，这使得用户可以在不改变模型的情况下尝试不同的求解器和算法。

¹<https://choco-solver.org/>

²<https://www.ibm.com/products/ilog-cplex-optimization-studio>

2.1.2 其他求解方法

除了 CP 求解器外，有许多其他的求解技术也可以用来处理 CSP，它们来自于不同的研究领域。其中，布尔可满足性 (SAT) 求解器、可满足性模理论 (SMT) 求解器和整数线性规划 (ILP) 求解器是最常用的三种。接下来，我们将介绍这三种求解技术，以及它们在处理 CSP 时的优势和应用情景。

SAT 是计算机科学中的一个经典问题，涉及到查找一个给定的布尔公式的满足赋值。SMT 是一个基于布尔可满足性 (SAT) 的求解技术，它在布尔可满足性 (SAT) 的基础上引入了更多的数学理论，如整数和实数的算术、数组、列表等，使其能够处理涉及多种数据类型和复杂函数的公式。常见的 SAT 求解器有 MiniSAT、Kissat 等，而主流的 SMT 求解器则包含 Z3、CVC5 等。

定义 2.6 (布尔可满足性). 一个 SAT 问题由一系列的布尔变量和逻辑运算符（如与、或、非）构成，对该问题的求解涉及到查找一个给定的布尔公式的满足赋值，其中满足赋值指一种将所有布尔变量赋值为真或假的方式，使得整个公式的结果为真。

定义 2.7 (可满足性模理论). 一个 SMT 问题可以表示为一个三元组：(V, F, ϕ)，其中 V 是变量集合，F 是函数符号集合， ϕ 是一个由约束和公式组成的公式集合。约束是对变量的限制条件，而公式则是对变量和函数的逻辑表达式。SMT 问题的目标是找到一个满足所有约束和公式的赋值。

SAT 问题是 NP 完全问题，但现代的 SAT 求解器已经能够处理数百万个变量和数千万个约束的问题。将 CSP 转化为 SAT 问题的关键步骤是将 CSP 的变量和约束转化为布尔变量和布尔公式。具体来说，对于 CSP 中的每一个变量和它的每一个可能的值，我们都定义一个对应的布尔变量。如果 CSP 变量被赋予该值，则对应的布尔变量为真；否则为假。然后，我们使用“at most one”和“at least one”的公式来保证 CSP 中的每个变量只能取一个值。同时，CSP 的其他约束也可以通过定义相应的布尔公式来实现。

将 CSP 编码为 SMT 问题的过程相对直接。由于 SMT 支持整数变量和约束，以及各种数学函数，因此可以直接将 CSP 中的变量、约束和函数映射到 SMT 问题中。这样，CSP 就可以被转化为一个等价的 SAT/SMT 问题，进而利用相应求解器求解，一般来言，在编码 CSP 时，SAT 的编码复杂度要高很多，但比涉及高阶逻辑的 SMT 求解器拥有更好的性能。

最后是 ILP，它是线性规划 (LP) 的一种扩展。线性规划问题包含一组线性约束条件和一个线性目标函数，问题的目标是在满足这些约束条件的前提下，找到使目标函数取得最大（或最小）值的变量的取值，而 ILP 则要求所有的决策变量都是整数。由于 CSP 中的变量通常也是整数，因此 CSP 可以很自然地映射为 ILP 问题。常见的 ILP 求解器有 Gurobi、SCIP 等。

这些算法为解决 CSP 提供了新的视角，尽管某些转换可能相对复杂，可能会影响性能或效率，但它们也弥补了 CP 求解器在特定问题时的不足。

2.2 AllDifferent 约束求解现状

由于目前 AllDifferent 约束的过滤算法及后续算法都要用到图论的相关知识, 所以我们先介绍图论以及相关的最大匹配算法, 之后介绍 AllDifferent 约束的过滤算法以及基于该算法的一些改进策略。

2.2.1 图论基础

我们用 $G = (V, E)$ 表示一个 (无向) 图, 其中 V 是一个有限的顶点集, 而 E 是来自 V 的无序对的多重集, 称为边。顶点 $u \in V$ 和 $v \in V$ 之间的边记作 uv 。如果存在一个分区 S, T , 使得 $E \subseteq \{st | s \in S, t \in T\}$, 那么图 $G = (V, E)$ 称为二部图, 我们写作 $G = (S, T, E)$ 。

在图 $G = (V, E)$ 中, 游走指一个序列 $P = v_0, e_1, v_1, \dots, e_k, v_k$, 其中 $k \geq 0$, $v_0, v_1, \dots, v_k \in V$, $e_1, e_2, \dots, e_k \in E$, 并且对于 $i = 1, \dots, k$, 有 $e_i = v_{i-1}v_i$ 。如果 v_0, \dots, v_k 是不同的, 那么这个游走就被称为一条路径。一个闭合的路径, 即 $v_0 = v_k$, 称为回路。

图 $G = (V, E)$ 的子图是一个图 $G' = (V', E')$, 它满足 $V' \subseteq V$ 和 $E' \subseteq \{uv | u \in V', v \in V', uv \in E\}$ 。我们称子图 $G' = (V', E')$ 为图 $G = (V, E)$ 的最大连通子图, 若对于 V' 中的每一对 u, v , 在 G' 中都存在一条 u 到 v 的路径。

有向图是一个对 $G = (V, A)$, 其中 V 是一个有限的顶点集, A 是来自 V 的有序对的多重集, 称为弧, 从 $u \in V$ 到 $v \in V$ 的弧记作 (u, v) 。类似于无向二部图, 如果存在一个分区 S, T , 使得 $A \subseteq \{(s, t) | s \in S, t \in T\}$, 那么有向图 $G = (V, A)$ 就是二部图。我们也写作 $G = (S, T, A)$ 。此外, 关于有向图的路径、回路、子图、最大连通子图等概念和无向图类似, 这里不再展开。

在有向图中, 如果存在两个顶点 s 和 t , 从 s 可以通过一条有向路径到达 t , 同时也可以从 t 通过一条有向路径到达 s , 那么我们就说两个顶点相互到达。如果在一个有向图中, 存在一个顶点集 V , 对于集合中的任意两个顶点 u 和 v 都可以相互到达, 我们称 V 为强连通分量 (Strongly Connected Component, SCC)。

图 $G = (V, E)$ 的一个匹配指一组不相交的边集 $M \subseteq E$, 即 M 中的任意两条边不共享顶点。如果 M 包含了所有属于 $S \subseteq V$ 的顶点, 我们称匹配 M 覆盖了 S 。如果 M 没有覆盖顶点 $v \in V$, 那么 v 就被称为 M -free, 又称自由节点; 如果 M 覆盖了顶点 $v \in V$, 我们称 v 为匹配节点。匹配 M 的大小是 $|M|$, 若最大匹配包含了图中所有的顶点, 那么这个匹配被称为完全匹配。

定义 2.8 (最大图匹配). 给定一个无向图 $G = (V, E)$, 其中 V 是顶点集, E 是边集。最大图匹配问题要求找出图中一个最大的匹配。

设 M 是图 $G = (V, E)$ 中的一个匹配。如果一条路径 P 的长度是奇数, 它的两端没有被 M 覆盖, 并且它的边交替出现在 M 的外部 and 内部, 那么我们称 P 为 M -增广路径。如果回路 P 的边交替出现在 M 的外部 and 内部, 那么我们称 P 为 M -交替路径。在 M -增广路径上, 我们可以交换在 M 中和不在 M 中的边,

得到的 M' 仍然是一个匹配，并且其中的 $|M'| = |M| + 1$ 。我们可以得到如下的定理，证明如下。

证明. 如果 M' 是一个比 M 大的匹配，考虑图 $G' = (V, M \cup M')$ 。在 G' 中，每个顶点最多连接两条边。因此， G' 的每个组件要么是一个回路，要么是一条路径（可能长度为零）。由于 $|M'| > |M|$ ，因此至少有一个组件包含的 M' 的边比 M 的边多。因为所有的回路都包含偶数条边，所以这个组件必须是一个 M 增广路径。如果 M' 是一个比 M 大的匹配，考虑图 $G' = (V, M \cup M')$ 。在 G' 中，每个顶点最多连接两条边。因此， G' 的每个组件要么是一个回路，要么是一条路径（可能长度为零）。由于 $|M'| > |M|$ ，因此至少有一个组件包含的 M' 的边比 M 的边多。因为所有的回路都包含偶数条边，所以这个组件必须是一个 M 增广路径。 \square

定理 2.1. 设 $G = (V, E)$ 是一个图， M 是 G 中的一个匹配。那么 M 要么是最大匹配，要么存在一个 M -增广路径。

因此，我们可以通过在 G 中迭代计算 M -增广路径并扩展 M 的方式来求一个图最大匹配，它是求解最大图匹配的一类经典方法，步骤如下。

设 $G = (U, W, E)$ 是一个二部图， M 是 G 的一个匹配。通过将 M 中的所有边从 W 指向 U ，并将所有其他边从 U 指向 W ，构造有向二部图 $G_M = (U, W, A)$ ，即：

$$A = \{(w, u) | uw \in M, u \in U, w \in W\} \cup \{(u, w) | uw \in E \setminus M, u \in U, w \in W\}$$

然后，在 G_M 中从 U 中的一个自由顶点开始并在 W 中的一个自由顶点结束的每一条有向路径对应于 G 中的一个 M -增广路径。通过选择 $|U| \leq |W|$ ，我们最多需要找到 $|U|$ 条这样的路径。由于每条路径可以通过广度优先搜索在最多 $O(|A|)$ 时间内被识别，因此这个算法的时间复杂度是 $O(|U||A|)$ 。

在^[27]中提出了其改进算法，可以在 $O(|V|^{1/2}|A|)$ 时间内运行，其中 $V = U \cup W$ 。其思想为，与其反复沿着单个 M -增广路径增强 M ，不如反复同时沿着一组不相交的 M -增广路径增强 M 。这样的路径集合可以再次在 $O(|A|)$ 时间内找到。可以证明，在 $|V|^{1/2}$ 次迭代之后，通过对路径长度的推理，最多可能还有 $O(|V|^{1/2})$ 次迭代，这导致总的时间复杂度为 $O(|V|^{1/2}|A|)$ 。而在一个普通的图 $G = (V, E)$ （不一定是二部图）中，可以在 $O(|V||E|)$ 时间内^[28] 或者 $O(|V|^{1/2}|E|)$ 时间内^[29] 计算出最大的匹配。

2.2.2 AllDifferent 约束和二部图匹配

我们首先展示 AllDifferent 约束的解和二部图中的匹配的等价性。为此，我们将 AllDifferent 约束涉及的变量使用二部图进行表示，称为值图。它的定义如下，注意，定理中的匹配 M 覆盖了 X ，因此是最大匹配。此外，我们给出了一个例子来描述这一转换过程。

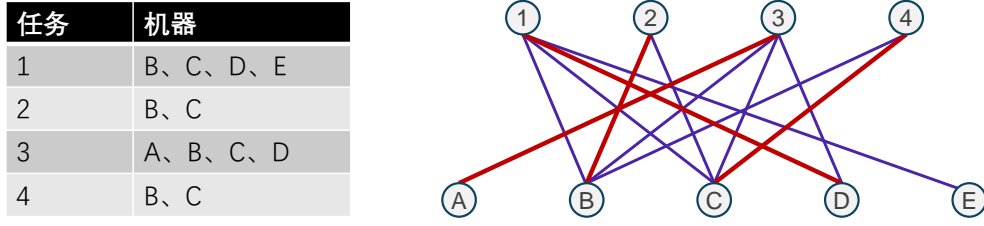


图 2.1 AllDifferent 约束和其二部图表示。

Figure 2.1 AllDifferent constraint and its bipartite graph representation.

定义 2.9 (值图). 设 X 是一组变量集合, 则二部图 $G = (X, D(X), E)$, 其中 $E = \{xd | d \in D(x), x \in X\}$, 被称为 X 的值图。

定理 2.2. 设 $X = x_1, x_2, \dots, x_n$ 是一系列变量, G 是 X 的值图。那么 $(d_1, \dots, d_n) \in \text{AllDifferent}(x_1, \dots, x_n)$ 当且仅当 $M = \{x_1d_1, \dots, x_nd_n\}$ 是 G 中的一个匹配。

证明. 在 M 中的边 x_id_i (对于某个 $i \in \{1, \dots, n\}$) 对应于赋值 $x_i = d_i$ 。由于 M 中的边不共享顶点, 所以对于所有 $i \neq j$, 都有 $x_i \neq x_j$ 。□

例 2.1. 假设有四个任务 (1, 2, 3 和 4) 要被分配给五台机器 (A, B, C, D 和 E)。每台机器最多只能分配一个任务, 但并非每个任务都能分配给每台机器。图2.1左中展示了可能的组合, 例如, 任务 2 可以分配给机器 B 和 C。由于任务必须分配给不同的机器, 我们可以使用 AllDifferent 约束表述该问题, 从而将问题建模为 CSP。具体地, 我们引入一个变量 x_i , 用于表示任务 $i = 1, \dots, 4$, 其赋值代表任务 i 被分配到的机器。变量的初始值域为图2.1左表示的任务和机器之间可能的组合, 下式是构造的 CSP 约束:

$$\text{AllDifferent}(x_1, x_2, x_3, x_4).$$

其中, $x_1 \in \{B, C, D, E\}, x_2 \in \{B, C\}, x_3 \in \{A, B, C, D\}, x_4 \in \{B, C\}$ 。设 $X = x_1, \dots, x_n$, 其值图如图2.1右所示。值图中的红粗线表示覆盖 X 的一个匹配, 它对应于 CSP 的解, 即 $x_1 = D, x_2 = B, x_3 = A$ 和 $x_4 = C$ 。

最后, 我们要介绍 Hall 的婚配定理^[30], 它是用于推导 AllDifferent 约束过滤算法的图论中的一个经典定理, 旨在描述一个二部图存在完全匹配的 necessary 和充分条件。这个定理的直观解释是, 如果每个女孩 (A 集合) 都有足够多的男孩 (B 集合) 可以选择, 那么就有可能为每个女孩找到一个男孩作为配对, 形成一个完全匹配, 反之亦然。接下来, 我们给出 AllDifferent 约束语境下, 该定理的形式化描述, 证明略。

定理 2.3. 约束条件 $AllDifferent(x_1, \dots, x_n)$ 有解, 当且仅当对于所有的 $K \subseteq \{x_1, \dots, x_n\}$, 下式成立:

$$|K| \leq |D(K)|$$

例 2.2. 考虑如下 CSP:

$$AllDifferent(x_1, x_2, x_3, x_4).$$

其中, $x_1 \in \{2, 3\}, x_2 \in \{2, 3\}, x_3 \in \{1, 2, 3\}, x_4 \in \{1, 2, 3\}$ 。可见, 对于任何 $K \subseteq \{x_1, x_2, x_3, x_4\}$ 且 $|K| \leq 3$, 都有 $|K| \leq |D(K)|$ 。然而, 对于 $K = \{x_1, x_2, x_3, x_4\}$, 有 $|K| > |D(K)|$ 。由定理 2.3 可知, 这个 CSP 没有解。

2.2.3 AllDifferent 约束的过滤算法

前面, 我们介绍了局部一致性的概念, 我们以弧一致性为例介绍 AllDifferent 约束如何应用局部一致性, 我们首先利用二元分解将 AllDifferent 约束分解成一组二元约束, 其定义如下。

定义 2.10 (二元分解). 设 C 是变量 x_1, \dots, x_n 上的一个约束。 C 的二元分解指在 x_1, \dots, x_n 的变量对上的一组最小的二元约束 $C_{dec} = \{C_1, \dots, C_k\}$ (其中 k 为整数且 $k > 0$), 使得 C 的解集等于 $\bigcap_{i=1}^k C_i$ 的解集。

关于 $AllDifferent(x_1, x_2, \dots, x_n)$ 的二元分解是

$$\bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}. \quad (2.1)$$

建立二元分解上的弧一致性的过滤算法很简单: 每当一个变量的域只包含一个值时, 这个值就从在 AllDifferent 约束中出现的其他变量的值域中被移除。这个过程一直重复, 直到没有更多的变化发生或者一个域变为空。通过这个算法, 我们可以建立 AllDifferent 约束上的局部一致性, 或者证明它是不一致的。换句话说, 当一个值域包含多个元素时, 我们就不能推理出更多的信息了。

这个算法的一个缺点是, 需要 $\frac{1}{2}(n^2 - n)$ 个不等约束来表示一个 n 元的 AllDifferent 约束, 从而这种方法的最坏情况时间复杂度是 $O(n^2)$ 。另一个更重要的缺点是信息的丢失。当二元约束集合被弧一致化时, 一次只比较两个变量。而当 AllDifferent 约束被超弧一致化时, 所有的变量都被同时考虑, 这允许更强的局部一致性, 下面我们给出一个简单的例子。

例 2.3. 设一个 CSP 包含三个变量 x_1, x_2, x_3 , 其中 $x_1 \in \{2, 3\}, x_2 \in \{2, 3\}, x_3 \in \{1, 2, 3\}$ 。对于约束 $AllDifferent(x_1, x_2, \dots, x_n)$, 在不采用二元分解时, 通过超弧一致性我们可以删减 x_3 值域中的 2 和 3; 而对该 AllDifferent 约束使用二元分解时, 由于约束被分解为多个二元约束, 我们无法推导出任何信息。

前面, 我们提到了 AllDifferent 约束和二部图的等价性, 以及求解二部图最大匹配的方法。接下来, 我们介绍一种利用二部图实现的 GAC 算法, 它基于最大

匹配和强连通分量,称为 *Région* 算法^[16]。算法思路是构造残差图表示 AllDifferent 约束,并通过在图上作图匹配删减冗余边,实现在约束上的全局一致性。

后文需要用到前面定义的 CSP 的一些概念,为了方便后文叙述,我们额外引入几个新定义。约束 c 的变量集是它所限制的有序变量集,记作 $X(c) = \{x_1, x_2, \dots, x_r\}$ 。 c 的值域是 $X(c)$ 的值域的并集,记作 $D(c) = \bigcup_{x \in X(c)} D(x)$ 。此外,我们使用 $B_c(a)$ 来表示 $X(c)$ 中值域包含 a 的变量集(即, $B_c(a) = \{x | x \in X(c), a \in D(x)\}$)。在 AllDifferent 约束表示的二部图中,冗余边指在任何最大匹配中都不出现的边。允许边则表示属于一些但不一定所有最大匹配的边,冗余边和允许边是互补的。

一个节点是允许的,当且仅当对于一个任意的最大匹配 M ,它可以通过一个从自由节点开始的偶数交替路径到达。交替路径中的变量节点集被记为 Γ ,其值节点集被记为 A 。关于允许边有如下定理,证明略。

定理 2.4. 一个 AllDifferent 约束是 GAC 的,当且仅当它的值图的每个边都是允许边,也即属于在 $B(c)$ 中覆盖 $X(c)$ 的一些匹配。

定理 2.5 (Berge 定理). 一个边是允许的,当且仅当,对于一个任意的最大匹配 M ,它属于从自由节点开始的偶数增广路径或者偶数交替路径。

基于上述定理, *Région* 提出了执行 GAC 的第一个 AllDifferent 约束算法,通过从值图中移除冗余边来实现 GAC。该算法首先计算一组 AllDifferent 约束的值图的最大匹配,通过引入新的节点和将边改为有向边,构造残差有向图如下。

定义 2.11 (残差有向图). 残余有向图被定义为 $R = \langle V_R, E_R \rangle$, 其中 $V_R = X(c) \cup D(c) \cup \{t\}$, t 是与 $D(c)$ 连接的一个新引入的汇节点, $E_R = E_M \cup E_U \cup E_{t_1} \cup E_{t_2}$ 。具体来说,每个边 $e \in E_M \cup E_U$ 表示约束 c 中的一个变量-值对。匹配边 E_M 将变量连接到它们各自的匹配值: $E_M = \{x \rightarrow a | x \in X(c), a = M(x)\}$ 。未匹配的边 E_U 将值连接到它们各自的未匹配变量: $E_U = \{a \rightarrow x | a \in D(c), x \in B_c(a) \setminus \{M(a)\}\}$ 。 E_{t_1} 中的边将匹配值节点连接到 t : $E_{t_1} = \{a \rightarrow t | M(a) \in X(c)\}$ 。 E_{t_2} 中的边将 t 连接到未匹配的值节点(自由节点): $E_{t_2} = \{t \rightarrow a | M(a) \notin X(c)\}$ 。

通过残差有向图,我们可以发现,通过引入汇节点 t ,从自由节点开始的偶数增广路径被扩展为偶数的交替路径,从而允许边表现为残差图上的交替路径。此外,我们很容易发现,交替路径上的顶点之间的其他边也可以扩展成一个交替路径,因此也属于允许边。从而得到如下定理:

定理 2.6. 一个非匹配边是允许的,当且仅当其顶点处于同一个强连通分量中。

因此,算法会计算残差有向图的强连通分量(SCC),这是算法最耗时的部分,通常通过 Tarjan 算法来解决。之后会删除两个端点处于不同强连通分量的非匹配边。图2.2展示了一个简单的值图转换为残差图的例子,在右图中每个框表示一个 SCC,而紫红色的线段就是应当被删除的冗余边。

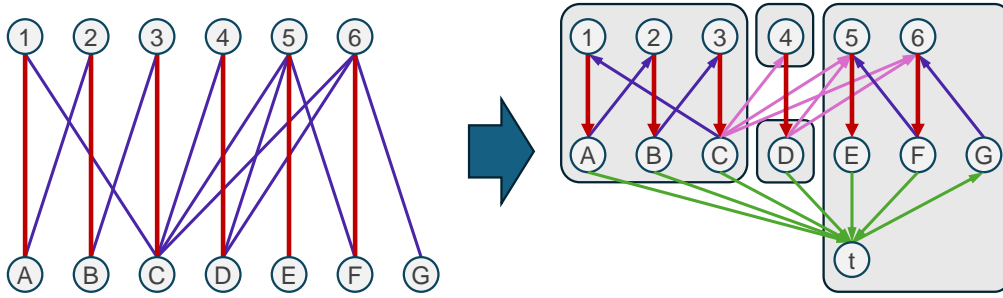


图 2.2 值图与残差有向图的转换。

Figure 2.2 Value graph to residual directed graph transformation.

近几年，有许多基于该算法的改进工作，主要思路是减少或避免 SCC 的计算。例如：SCC-分割^[31]，它只计算在回溯搜索过程中其变量的域已经改变的 SCC，从而减少了计算 SCCs 所需的变量范围；匹配优化^[32]，它证明了冗余边可以分为两类，其中一类可以直接删除，无需计算相应的 SCCs；早期检测^[33]，它避免了计算那些删除后不会分割当前 SCCs 的无关紧要的边；在文献^[34]中，证明了二部图中的所有冗余边都指向某些交错环，从而对值进行筛选；在文献^[35]中，发现 GAC 算法只关心图模型的一个节点是否在 SCC 中，而不是它属于哪个 SCC，从而使用比特位提高算法的性能。

2.3 局部搜索算法现状

局部搜索是相对于完备搜索来说的，它是一种用于解决优化问题的常用启发式策略，而几乎所有问题都可以看作是一类优化问题，例如组合优化问题。局部搜索的基本思想是从一个初始状态开始，然后在解的邻域中寻找更好的状态，通过一系列的移动来改进当前解。一些基本的概念的定义如下。

定义 2.12 (状态). 状态是解空间中的一个元素，表示问题的一个可能解。

定义 2.13 (移动). 移动是从当前状态转移到邻域中的另一个状态的操作。

定义 2.14 (打分函数). 打分函数（评价函数）是一个将解空间中的每个状态映射到实数的函数，用于评估每个状态的优劣，即解的质量。

定义 2.15 (邻域). 邻域是一个状态执行一次移动后的所有可达状态的集合。

为了实施局部搜索算法，首先需要明确候选解的邻域结构，即确定哪些候选解可以被视为邻居。一旦邻域结构被定义，问题的解空间就可以被视为一个图形结构。因此，局部搜索实际是在图论问题上求解。以下是其求解流程：

1. 设 $S = \{s_1, s_2, \dots, s_n\}$ 为所有候选解构成的解空间，也即状态空间。
2. 定义初始状态 $s_0 \in S$ ，一般由构造的初始化函数得到。

3. 定义邻域函数 $N : S \rightarrow \mathcal{P}(S)$ ，将每个状态映射到其邻域的状态集合。
4. 定义目标函数 $f : S \rightarrow \mathbb{R}$ ，用于评估每个状态的好坏，即解的质量。
5. 设置停止条件，当满足某个特定条件时终止算法。常见的停止条件如达到最大迭代次数、最大时间限制等。
6. 由打分函数（可能多个）构成移动规则，指导从当前状态的邻域中选择下一个状态。一般选择使得目标函数变好最多的状态。
7. 从初始状态 s_0 开始，按照移动规则在邻域中选择下一个状态，更新当前状态。重复这个过程，直到达到最优解或满足其他停止条件。

若将被满足的约束数量作为最小化目标，则 CSP 可以被看作组合优化问题，通过局部搜索，我们寻找使得 CSP 中一致约束最多的赋值。我们以一个简单的 CSP 为例，介绍上面提到的概念和算法流程。

例 2.4. 假设我们有一个简单的约束满足问题（CSP），其中有三个变量 $X = \{x_1, x_2, x_3\}$ ，每个变量的取值范围是 $\{1, 2, 3\}$ 。我们的目标是找到一组赋值 $A = \{a_1, a_2, a_3\}$ ，使得所有的约束 $C = \{c_1, c_2, c_3\}$ 都被满足。在这个问题中，状态就是变量的一组赋值，比如 $s = \{2, 1, 3\}$ 。移动就是改变一个变量的赋值，比如从状态 $\{2, 1, 3\}$ 移动到状态 $\{2, 2, 3\}$ 。目标函数就是被满足的约束数量，我们希望最小化未被满足的约束数量，也就是最大化被满足的约束数量。邻域就是通过一次移动可以达到的所有状态，比如从状态 $\{2, 1, 3\}$ 可以移动到的状态有 $\{1, 1, 3\}$ 、 $\{3, 1, 3\}$ 、 $\{2, 2, 3\}$ 、 $\{2, 3, 3\}$ 、 $\{2, 1, 1\}$ 和 $\{2, 1, 2\}$ 。我们从一个初始赋值开始，通过移动来最小化未满足约束的数量，直到得到可满足解。

近几年，局部搜索算法发展出了很多策略，这些策略可以根据问题的特性和需求进行组合，以提高局部搜索算法的效率和效果。下面是一些例子：

- **禁忌搜索**^[36]：这种策略旨在避免在已经访问过的区域中陷入循环。在禁忌搜索中，会维护一个禁忌表，记录一些在近期内已经访问过的状态。当我们从当前状态的邻域中选择下一个状态时，禁忌表中的状态将被排除在外。

- **重启策略**：重启策略能够有效避免搜索过程陷入局部最优解。若如果搜索过程在一段时间内没有找到更好的解，算法会随机选择一个初始状态重新开始搜索。这样可以使搜索过程有机会探索解空间的其他区域。

- **迭代局部搜索**^[37]：这种策略在没有达到局部最优时，持续执行迭代改进算法。当达到局部最优时，执行随机游走算法，即从当前的局部最优解出发，进行一次或多次随机的移动，以跳出当前的局部最优区域。策略的目标是搜索一系列的局部最优解，然后返回最好的一个作为结果。

- **解池技术**：解池技术在搜索过程中不仅保存当前最优解，还保存一些具有代表性的非最优解，这些解构成了一个解池。通常他和重启策略一起使用，在重启时，算法从解池中选择一个解执行随机游走得到初始状态，重新开始搜索。

在近十年来，已经有不少工作采用局部搜索来对 CSP 问题进行求解，并帮助解决了许多此前未解决的难题，下面我们列举一些经典及最新的工作。

在文献^[38]中, 作者设计了一种新的启发式算法, 它利用了问题在约束和变量方面的结构, 可以比全局代价函数更精确地指导搜索进行优化 (例如, 违反约束的数量)。该工作在拉丁方、N 皇后、全间隔等问题上进行了实验, 并且取得了较好的实验结果。在文献^[39]中, 作者设计了将拉丁方问题转化为图的方法, 从而将问题归约到图着色问题上, 并使用遗传算法, 对给定补全拉丁方实例进行求解, 取得了较好的求解效率。在文献^[40]中, 作者设计了针对数独问题的蚁群算法, 并尝试了多种邻域的定义, 算法可以在 25x25 阶的困难数独实例上取得高达 95% 的求解成功率。在文献^[41]中, 作者设计了针对性的解池技术和子代价函数, 并通过高效的化简规则对 CSP 进行了化简, 其在拉丁方问题上, 可以求解绝大部分 70 阶的拉丁方补全问题。

2.4 本章小结

本章节介绍了 CSP、AllDifferent 约束的形式化定义以及主流的求解它们的算法。在第一节中, 我们介绍了 CSP 的求解现状, 主要介绍了 CP 求解器, 以及其他可用于求解的技术, 如 SAT、SMT、ILP 求解器等。在第二节中, 我们介绍了 AllDifferent 约束的求解现状, 主要介绍了它和图匹配的关系, 以及两种主流的针对该约束的过滤算法。在第三节中, 我们介绍了局部搜索算法的概念和求解流程, 以及相关的各类技术。

第3章 AllDiff-LS 的基础组件和算法框架

本章首先对本文的研究问题进行详细介绍，即如何设计一个高效的求解算法来求解一个包含 AllDifferent 约束的约束满足问题。基于以上研究问题，本章介绍了本文提出的适用于 AllDifferent 约束求解的局部搜索算法。首先，本章节讨论了使用什么数据结构对约束进行表示，以及如何对约束进行建模。具体地，算法采用了一个包含两种类型的顶点和边的异构图来表示 AllDifferent 约束，不是单独处理每个 allDifferent 约束，而是将它们视为一个整体。其次，算法采用两个低复杂度的顶点简化规则对图进行化简，以减少不必要的搜索空间。接下来，算法针对构造的图设计求解算法，提出了分两步选择操作的方法，以及相应的打破平局和禁忌策略。最后，我们给出算法的整体流程图，并展示了包含 AllDifferent 约束的 CSP 是如何被求解的。

3.1 问题介绍和求解思路

本文主要研究以 AllDifferent 约束为主的 CSP 问题，这是一类特殊的 CSP，其中所有的全局约束由 AllDifferent 约束组成，但允许出现其他类型的二元约束，使得 CSP 约束的全部或大部由 AllDifferent 约束组成。在这类问题中，一个变量或包含它的变量表达式将受到多个 AllDifferent 约束的共同限制，问题的目标是寻找使得所有约束满足的变量赋值。这类问题的一个主要特点是，由于 AllDifferent 约束的全局性，解空间的大小会随着问题规模的增大而指数级增长。因此，寻找有效的求解策略和算法以提高求解效率，是研究这类问题的一个重要方向。

使用 AllDifferent 约束可以编码很多经典的 CSP，例如数独、n 皇后、All-Interval 和各类拉丁方问题。以数独问题为例，问题中每一行、每一列以及每一个宫（3x3 的小格子）的数字都需要满足 AllDifferent 约束，即这些位置上的数字都需要是 1 到 9 且互不相同。而除此之外，数独问题没有其他的约束条件。

设 $P = (X, D, C)$ 是一个约束满足问题，其中 C 是约束集。定义 $AD(c)$ 成立，当且仅当 c 是一个 AllDifferent 约束。我们取 $C_A = \{c | c \in C, AD(c)\}$ 为问题中所有 AllDifferent 约束组成的集合，本章节后续部分将主要研究如何获得在该约束集上的可满足赋值。不失一般性的，我们在本章节的最后会介绍如何将其他二元约束整合到求解的框架中，从而得到整个 CSP 的可满足赋值。

考虑到 AllDifferent 约束强大的约束力，现有的求解器在解决约束数量和长度不断增加的各种 CSP 问题时，表现出了性能下降的趋势。我们注意到在 AllDifferent 约束中，出现的大部分变量都是用来表示变量表达式的，同一个变量不可避免地会出现在多个约束中，这导致约束之间存在强烈的关联性。此前，关于 AllDifferent 约束求解的算法分为两类，一类是借助过滤算法对约束化简后再对问题进行回溯求解，第二类则是通过提前满足 AllDifferent 约束的方式将约

束作为隐含约束处理。这两种策略都是将问题中的 AllDifferent 约束分别进行处理，而没有考虑约束之间也存在约束关系。此外，随着约束数量的增多，第二类策略无法处理绝大多数 AllDifferent 约束，使得策略的有效性会大的折扣。

基于以上观察，在本文中，我们并没有单独对待每一个 AllDifferent 约束，而是将它们作为一个整体来处理。我们采用了包含两种类型顶点和边的异构图来表示 AllDifferent 约束，称为 AllDifferent 约束图 (ACG)。构造这个图的动机来源于针对 AllDifferent 约束的二元分解，我们在此基础上将变量和变量表达式的概念进行了区分，从而使得此框架可以应用于更为广泛的弧一致性算法。

3.2 AllDifferent 约束转化为图及其优化

设 $P = (X, D, C)$ 是一个约束满足问题， C_A 是其 AllDifferent 约束组成的集合。约为了表示该问题，我们引入一个异构图 G ，称之为 AllDifferent 约束图 (AllDifferent Constraint Graph, ACG)。为了方便后续描述，约束中涉及的基本单位是变量表达式，其中，变量表达式可以是包含加减乘除等操作的复杂表达式，也可以是单位表达式。例如，在约束条件 $AllDifferent(x_1, x_2 + x_3)$ 中， x_1 和 $x_2 + x_3$ 都是表达式。异构图的形式化表示如下。

定义 3.1 (AllDifferent 约束图). 异构图被定义为 $G = (V, E)$ ，其中 $V = X \cup NV$ ，表示顶点集合，其中 NV 指出现在 AllDifferent 约束中的变量表达式集合。 $E = E_p \cup E_x$ 表示边集合，它由两种类型的边表示。具体来说，一种类型是变量表达式之间的边，我们称为表达边，用 E_p 表示，如果两个变量表达式 p 和 q 出现在同一个 AllDifferent 约束中，我们表示为 $(p, q) \in E_p$ 。另一种类型是变量和变量表达式之间的边，用 E_x 表示，如果一个变量表达式 p 包含一个变量 x ，则有 $(p, x) \in E_x$ 。

AllDifferent 约束图的思想是，将全局约束用二元分解分解为多组二元约束的并集，再针对二元约束构成的二元关系以及变量表达式和变量之间的二元关系，构造图结构。和使用残差图对 AllDifferent 约束进行表示不同，我们没有引入变量-值这一关系。通过隐去值域这一信息，ACG 得以表示多组 AllDifferent 约束，从而将所有约束使用一张图进行表示，以施展更强的弧一致性算法。下面，我们通过一个简单的例子来介绍 ACG。

例 3.1. 给定四个变量 x_1, x_2, x_3, x_4 ，其有限值域分别为 $\{1, 2, 3\}$ ， $\{1, 2\}$ ， $\{2, 3\}$ 和 $\{1, 2, 3\}$ 。如果我们要求 x_1, x_2 和 $x_3 - 1$ 的赋值不能相同，并且要求 $x_1 + x_2$ 和 $x_3 + x_4$ 的赋值不能相同，那么可以如下建模此问题：

$$AllDifferent(x_1, x_2, x_3 - 1), \quad (3.1)$$

$$AllDifferent(x_1 + x_2, x_3 + x_4), \quad (3.2)$$

容易看出，该问题的一个解是 $\mathcal{A} = (3, 1, 3, 2)$ 。它的 ACG 如图3.1所示。

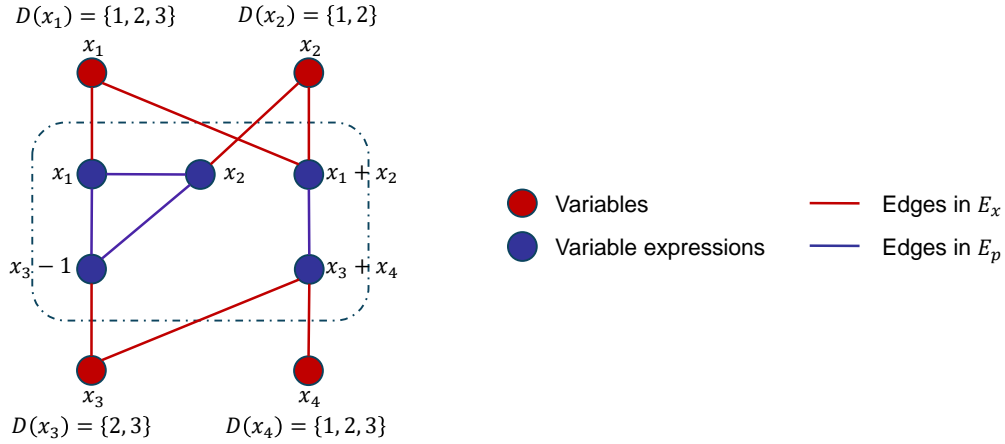


图 3.1 例子中约束转化得到约束图。

Figure 3.1 The constraints in the example are transformed into a constraint graph.

对于得到的 ACG，我们需要根据 ACG 中二元分解得到的二元约束，对变量的值域进行化简。由于二元约束涉及的变量表达式可能包含多个变量或其他常数，直接套用二元约束的弧一致性算法的效率可能不太理想。因此，我们根据变量的值域设计了两个化简规则。其核心思想是，对于一个域大小为 1 的变量 x (例如， $\{i\}$)，我们可以直接将值 i 分配给 x ，并实施基于二元约束的全局弧一致性检测。同时，在图中，将 x 的变量表达式邻居中相应的变量更改为值 i ，并删除变量顶点 x 及其相关边。需要注意的是，虽然这些规则以减少 ACG 中的顶点为主，并不保证整组或单个 AllDifferent 约束的全局弧一致性。

我们先给出一些基础定义。在 ACG 中，我们用 $N(x)$ 表示变量 x 的邻居，用 $XN(p)$ 和 $PN(p)$ 分别表示变量表达式 p 的变量邻居和表达式邻居。在 ACG 中，我们用 $N(x)$ 表示变量 x 的邻居，用 $XN(p)$ 和 $PN(p)$ 分别表示变量表达式 p 的变量邻居和表达式邻居。在一个 CSP 中，设 $Y = x_{i_1}, x_{i_2}, \dots, x_{i_m}$ 是一串有限的变量，其中 $m > 0$ 。我们称一个元组 $\mathcal{A}(Y) = (d_{i_1}, d_{i_2}, \dots, d_{i_m}) \in D(x_{i_1}) \times D(x_{i_2}) \times \dots \times D(x_{i_m})$ 为 Y 的赋值，并且我们使用 $\mathcal{A}(x)$ 来表示在赋值 \mathcal{A} 下赋给变量 x 的值。通常，如果一个关于 X 的赋值给出了 CSP 中的每个变量的赋值，那么它被称为完整赋值；否则，它就是部分赋值。我们定义赋值操作如下：

定义 3.2 (赋值操作). 给定一个变量 x 和一个值 i ，如果我们将值 i 赋给 x ，那么就意味着在每个 $p \in N(x)$ 中将对应的变量改为值 i ，并在 ACG 中移除顶点 x 及其相关的边。

显然，对于值域大小为 1 的变量 x (例如， $\{i\}$)，我们可以直接将值 i 赋给 x 。而对于线性单变量表达式 p (也即， $XN(p) = \{x\}$)， p 的值域与 x 的值域一一对应。在 AllDifferent 约束中，这种变量表达式很常见。基于此，我们给出如下两条化简规则，这两条规则可以移除变量的值域中不满足一致性的赋值。

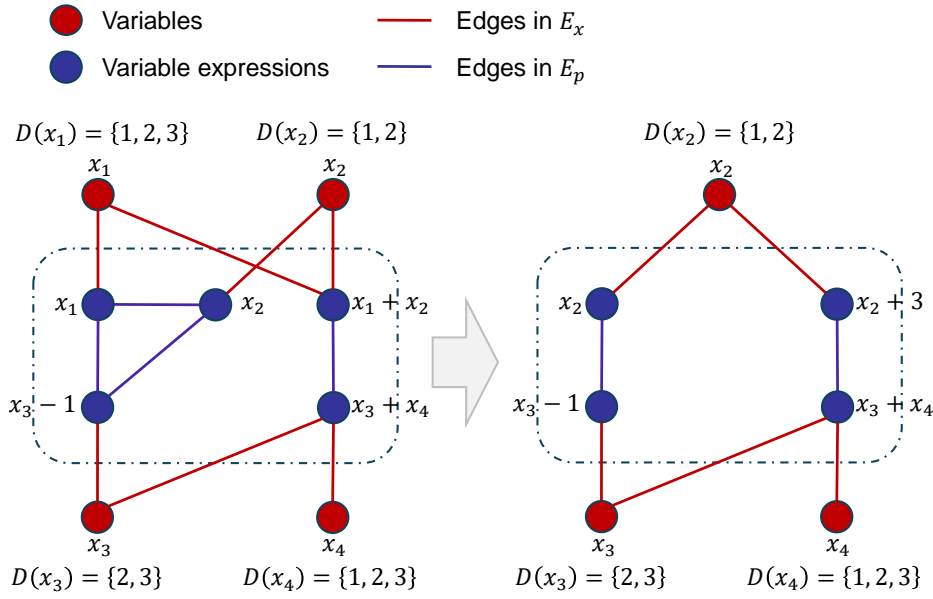


图 3.2 例子中约束图的化简。

Figure 3.2 Simplification of the constraint graph in the example.

- **规则 1.** 给定一个约束 c , 设 $N(c)$ 为 c 中包含的表达式, $D(c)$ 为 $\forall p \in N(c)$ 的可能赋值集的并集。对于一个单变量表达式 $p \in N(c)$ 和一个值 $i \in D(c)$, 如果 $|N(c)| = |D(c)|$, 并且 p 是 c 中唯一可以取值 i 的表达式, 那么将值 i 赋给 x 。
- **规则 2.** 给定一个表达式 p , 若 $|XN(p)|$ 变为 0, 意味着 p 变成一个常数 (例如, i)。对于它的一个单变量表达式邻居 $q \in PN(x)$ (例如, $XN(q) = \{x\}$), 如果 i 是 q 的可能赋值, 那么从 x 的域中移除使得 $q = i$ 的值。如果 p 的所有邻居都是单变量表达式, 那么移除顶点 p 及其相关的边。

我们对 ACG 反复应用两个规则, 从而实施赋值操作以减少 ACG 中的顶点和边。直到 ACG 中不能再移除任何顶点时, 我们得到最终的 ACG。同样的, 我们以一个简单的例子介绍化简的过程。

例 3.2. 对于例3.1中构造的 ACG, 考虑约束 c_1 , 其中 $N(c_1) = \{x_1, x_2, x_3 - 1\}$ 和 $D(c_1) = \{1, 2, 3\}$ 。我们可以通过应用规则 1 将值 3 赋给 x_1 , 赋值操作会将变量表达式 $x_1 + x_2$ 修正为 $x_2 + 3$, 变成单变量表达式, 同时让表达式 x_1 变为一个常数 3, 此外变量顶点 x_1 和以它为端点的边也会被删除掉。然后, 由于表示常数 3 的结点的所有邻居都是单变量结点, 我们可以应用规则 2 删除这个常数顶点和相关的边。ACG 的化简过程如图 3.2 所示。

对于剩余的变量, 他们的值域在后续算法中将不再发生改变。我们后面要用到的局部搜索算法需要从一个初始的完整赋值开始, 这里我们选择了随机构造完整赋值的方法。具体来说, 对于剩余的变量, 我们从每个变量的域中随机选择一个值作为其赋值。Initialization 函数在算法 1 中给出, 第 9 行是关键步骤。

算法 1 *Initialization function***输入:** A CSP $\mathcal{P} = (X, D, C)$ **输出:** An ACG \mathcal{G} and an complete assignment \mathcal{A}

```

1: Build an ACG  $\mathcal{G} = (V, E)$  for  $\mathcal{P}$ , where  $V = X \cup NV$  and  $E = E_p \cup E_x$ ;
2:  $SimpSet \leftarrow \{x \mid x \in X, |D(x)| = 1\}$ ;
3: repeat
4:   Select a variable  $x$  from  $SimpSet$  and let  $i$  be the unique value in  $D(x)$ ;
5:   for all  $v$  in  $N(x)$  do
6:     Assign  $x$  contained in  $v$  to  $i$ ;
7:   end for
8:   Remove  $x$  from  $X$ ;
9:   Simplify  $D$  and  $\mathcal{G}$  according to Rule 1 and Rule 2;
10:  // Both  $V$  and  $E$  in  $\mathcal{G}$  may potentially be simplified.
11:   $SimpSet \leftarrow \{x \mid x \in X, |D(x)| = 1\}$ ;
12: until  $SimpSet$  is empty
13: // Initialize  $\mathcal{A}$  of  $X$  randomly;
14: for all  $x$  in  $X$  do
15:   Select a random value  $i$  from  $D(x)$ ;
16:    $\mathcal{A}(x) \leftarrow i$ ;
17: end for
18: return  $\mathcal{G}, \mathcal{A}$ ;

```

除此之外,我们还可以使用针对 AllDifferent 约束的 GAC 算法对约束进行过滤,在过滤后再将约束转化为 ACG。这种做法对值域的限制较强,保证了每个 AllDifferent 约束的超弧一致性,但也存在计算复杂度较高等问题。

3.3 分步选择操作策略

在局部搜索中,一个移动(又称操作)意味着对赋值进行轻微的修改,它定义了候选解的邻域。一般来说,移动的复杂性应尽可能地保持低,同时确保它能够充分探索当前解的邻域。在启发式的局部搜索框架 CBLS (Constraint-Based Local Search, 基于约束的局部搜索)中,会先满足部分 AllDifferent 约束,之后通过交换约束中两个变量的赋值的操作实现对当前状态的修改,这里提到的交换操作称为 *swap*,就是一种移动。

相对的,本文采用的是一个简单但有效的操作符 *mov*,它通过将一个变量顶点的值修改为其值域中的其他值的方式来修改当前的完全赋值,上面提到的交换操作相当于对交换的两个变量分别作了一次修改赋值的操作 *mov*。这样做的动机是 *mov* 相较于 *swap* 对约束修改的粒度更细,使得其邻域范围更广,能够实现

更快的收敛速度，实际上，实验也表明了 *swap* 移动在求解包含复杂 AllDifferent 约束的 CSP 时存在困难。下面是 *mov* 的形式化定义。

定义 3.3. 给定一个变量顶点 x 和一个值 i ，将 x 的值从 $\mathcal{A}(x)$ 改变为 i 的操作符定义为 $mov(x, i)$ 。

对于每个 $(p, q) \in E_p$ ，如果 $\mathcal{A}(p) = \mathcal{A}(q)$ ，我们称 (p, q) 为冲突边。对于一个 ACG \mathcal{G} ，一个完整赋值 \mathcal{A} 的代价，表示为 $cost(\mathcal{G}, \mathcal{A})$ ，是在 \mathcal{A} 下的冲突边的总数。我们采用 ACG 中冲突边的改进数量，作为 *mov* 的度量，因此一个 *mov* 的打分函数可以表示为

$$score(mov) = cost(\mathcal{G}, \mathcal{A}) - cost(\mathcal{G}, \mathcal{A}'), \quad (3.3)$$

其中 \mathcal{A}' 是通过在 \mathcal{A} 上应用 *mov* 获得的。我们使用 $cost(x, i)$ 来表示将值 i 赋给 x 的代价，即当 $\mathcal{A}(x) = i$ 时，以 $\forall y \in N(x)$ 为终点的冲突边的总数。由于 $mov(x, i)$ 只影响那些以 x 的邻居为终点的边，所以 $mov(x, i)$ 的冲突分数可以由 x 的改变代价来表示，定义如下：

定义 3.4. 给定一个 ACG \mathcal{G} ，操作符 $mov(x, i)$ 的冲突分数定义为

$$score(mov) = cost(x, \mathcal{A}(x)) - cost(x, i). \quad (3.4)$$

通常我们倾向于通过反复选择最合适的 *mov*，不断降低 \mathcal{G} 在 \mathcal{A} 下的冲突边数，也即 $cost(\mathcal{G}, \mathcal{A})$ ，直到找到一个解。一个最常见的贪心选择准则是选择使得 $score(mov)$ 最大的 *mov*，这个方法叫做爬山法，它是很多启发式算法的基础。

在选择操作符时，需要确保它不会陷入循环，并且能在考虑约束条件的满足情况下逃离局部最优。给定一个变量 x ，我们用 $mc(x) = \min(\{cost(x, i) \mid i \in D(x)/\mathcal{A}(x)\})$ 表示 x 在除当前赋值外的所有可能赋值下的最小 $cost$ 。基于此，我们定义一个变量集 $X' = \{x \mid cost(x, \mathcal{A}(x)) \geq mc(x), x \in X\}$ ，称为候选变量集，它拥有一个重要的性质：对于 X' 中的每个变量，都有至少一个 *mov* 保证冲突边的数量不会增加。考虑到 *mov* 包含两部分：变量和值，我们提出以下两步选择策略：

- **步骤 1.** 从 X' 中选择在 $\mathcal{A}(x)$ 下具有最高 $cost$ 的变量 x ；
- **步骤 2.** 从 $D(x)$ 中选择使所选 x 的 $cost$ 最小的值 i 。

在依次选择变量 x 和值 i 后，我们已经确定了要执行的 $mov(x, i)$ 。虽然两步选择可能会错过一些更贪婪的 *mov*（在第一步未被选择的变量可能有更好的赋值），但它可以大大降低选择 *mov* 的时间复杂度。设 $|D|$ 表示每个变量的平均值域的大小，那么直接的 *mov* 选择方法需要 $|X| \cdot |D|$ 的复杂度，而我们的两步 *mov* 选择只需要 $|X| + |D|$ 的复杂度，从而降低了最坏情况下的时间成本。此外，候选变量集 X' 确保了所选变量的下限质量。

然而，局部搜索算法经常会陷入局部最优，这意味着 X' 是空的。在这种情况下，我们定义了一个新的候选变量集 X'' ，它表示当前存在冲突的变量集，称

为冲突变量集。这个变量集的动机来自于改变没有冲突的变量的赋值不会减少冲突的总数。 X'' 不能保证变量的质量，所以在这种情况下，我们直接选择 *score* 最大的 *mov*，*mov* 中的 x 只能从 X'' 中选择。为了和两步选择策略有所区分，我们称这种选择方式为直接选择策略。

3.4 禁忌策略

局部搜索算法使用禁忌策略^[42] 来避免陷入循环。鉴于需要通过两步来选择操作，我们提出了一个禁忌策略组合，包括以下策略。

- **策略 1.** 在一个变量表达式改变其值后，它将被禁忌，直到出现一个以它为端点的新的冲突边。此外，如果一个变量的所有邻居都被禁忌，那么该变量也将被禁忌；
- **策略 2.** 在一个变量 x 被赋予值 i 后，接下来的 tt 次迭代中，操作 $mov(x, i)$ 将被禁忌，其中 tt 是一个被称为禁忌期限的参数；
- **策略 3.** 如果选择的 *mov* 被禁忌的同时其变量也被禁忌，那么算法在接下来的 β 步中切换到直接选择策略。

在算法的求解流程中，策略 1 在变量选择阶段实施。策略 3 确保算法在两步选择策略的性能下降时，能够快速切换到直接选择策略。在直接选择中，只使用策略 2。接下来，我将具体介绍这三个策略的动机和细节。

策略 1 可以看作是一种格局检测策略^[43] 的变体，它最初是为了解决最小顶点覆盖问题中局部搜索算法的循环问题而被设计出来的，并在此后应用在许多图论问题上。格局检测策略的核心目标是在向当前候选解添加点时考虑该点的邻域。在本文中，我们用 S_x 表示关于变量的禁忌表，它初始为空。我们在选择一个操作 $mov(x, i)$ 执行时，会同步地将变量 x 放入 S_x 中，在禁忌表中的变量在第一阶段选择变量时不会被选中，除非 $S_x = X'$ ，也即所有变量都在禁忌表中。基于此，我们定义一个变量的格局如下：

定义 3.5 (格局). 对于 ACG 中的一个变量顶点 x ，其格局定义为其所有邻居的状态向量 F_v 。其中，一个变量表达式顶点的状态表示为 $f_p \in \{0, 1\}$ ，具体地：

$$f_p = \begin{cases} 1, & \text{若出现新的以 } p \text{ 为端点的冲突边;} \\ 0, & \text{其他情况.} \end{cases}$$

如果我们把变量表达式的格局定义为其与其变量表达式邻居构成的边，那么变量的格局可以视为变量表达式格局构成的格局，形成嵌套关系。一个变量的格局发生改变，当且仅当它所关联的变量表达式的格局发生改变。在 S_x 中的变量的格局改变时，我们会从 S_x 中移除该变量。这种策略是合理且直观的，因为它避免了重复选取同一个变量导致的循环情况。

然而，在实际求解过程中我们发现，仅采用策略一时仍然会发生循环。考虑如下场景：两个相关联的变量对应的 *mov* 会互相改变对方的格局，在不限

mov 的情况下，一个变量在被禁忌后会被另一个变量对应的 *mov* 激活，导致两个 *mov* 被无限循环地选择。基于此，我们构造了关于变量值域的禁忌表 S_m ，采用传统的 tabu 策略维护 S_m 。该策略基于以下思路：通过禁止对最近的变化进行反转，避免局部搜索马上返回刚刚访问过的状态进而导致循环。在算法求解时，我们在选择一个 *mov* 时，会优先选择没有被禁忌的变量，并优先从变量的值域中选择没有被禁忌的赋值。

在应用策略 1 和策略 2 时，我们需要注意到，在执行一个操作后，该操作在被禁忌的同时可能无法从禁忌表中激活任何变量。因此，禁忌搜索可能会出现如下情况：在两步选择策略中，会出现所有变量都被禁忌的情况。这种情况往往发生在局部搜索趋于稳定的后期。

我们考虑一种极端情况，在两步选择策略中，若所有 X' 中的变量都被禁忌，在步骤一中我们将不得不从禁忌表里选择一个变量 x ，而恰好变量 x 的所有赋值也都被禁忌，导致在步骤二中我们还需要从禁忌表中选择一个被禁忌的赋值 i 。在这种情况下，我们会得到一个被双重禁忌的操作 $mov(x, i)$ 。一个更合理的策略是，直接切换到直接选择策略，通过更贪心的选择策略寻找当前状态的邻域下最合适的操作。因此，我们设计了策略 3，它实际上是一种快速转换机制，作为前面提到的根据冲突变量集 X'' 切换的补充。

3.5 打破平局策略

另一方面，由于 AllDifferent 约束的性质，在直接选择 *mov* 时，经常有许多最大 *score* 的 *mov* 不唯一的情况。常用的打破平局策略有随机选择策略、子打分函数策略等。其中，随机选择策略会在所有具有相同优化目标值的解决方案中随机选择一个；而子打分函数策略除了主要的打分函数外，还会设计子打分函数。当且仅当通过主打分函数得到的最优操作有多个时，我们会使用子打分函数进行评分和排序，以此来打破平局。

在本文中，我们提出了一个新的 *nscore* 作为 *score* 的补充，它利用了变量的邻域信息。如果两个变量的邻居之间存在冲突边，我们就说这两个变量处于冲突中。我们使用 $n(x, i)$ 来表示当值 i 被赋予 x 时，与 x 冲突的变量的数量。

定义 3.6. 给定一个 ACG \mathcal{G} ，一个 $mov(x, i)$ 的 *nscore* 定义为

$$nscore(mov) = n(x, \mathcal{A}(x)) - n(x, i). \quad (3.5)$$

与 *score* 相比，*nscore* 关注的是变量之间的冲突，由于在 ACG 中的侧重点不同，一个 *mov* 在两个打分机制下不会出现高度相似的现象，从而保证打破平局的有效性。我们通过一个例子介绍该打分函数如何起作用。

例 3.3. 考虑图 3.2 右图所示的 ACG 示例，并假设它当前在赋值 $\mathcal{A} = (1, 2, 2)$ 下。那么 $score(mov(x_2, 2)) = score(mov(x_3, 3)) = 2$ ，而 $nscore(mov(x_2, 2)) = 2$ 和 $nscore(mov(x_3, 3)) = 1$ 。因此，我们选择执行 $mov(x_2, 2)$ 。

算法 2 *SelectMove* function

输入: A complete assignment \mathcal{A} , a control variable $mode$ and candidate variable sets X', X''

输出: A variable x and a value i

```

1: if  $mode = 0$  then
2:   Select a variable  $x$  from  $X'$  based on  $cost(x, \mathcal{A}(x))$  and Strategy 1;
3:   Select a value  $i$  from  $D(x)$  based on  $cost(x, i)$  and Strategy 2;
4:   if  $x$  is Tabu and  $mov(x, i)$  is Tabu then
5:      $mode \leftarrow \beta$ ; // Strategy 3
6:     Initialize  $X''$  according to  $\mathcal{A}$ ;
7:   end if
8: else
9:   Select  $x$  from  $X''$  and  $i$  based on  $score(mov)$  and  $nscore(mov)$ ;
10:   $mode \leftarrow mode - 1$ ;
11: end if
12: return  $x, i$ 

```

我们得到最终的评价准则如下：首先通过打分函数 $score$ 寻找评分最高的 mov ，在 mov 不唯一时，我们使用更细粒度的子打分函数 $nscore$ 对 mov 进行评价，优先选择 $nscore$ 最大的 mov 打破平局，若 mov 仍然不唯一，我们通过随机选择的方式打破进一步的平局。在本节的最后，我们给出最终的 *SelectMove* 函数，如算法 2 所示。它最初采用两步选择策略，当 X' 为空时，接下来的 β 步将使用直接选择策略，其中 β 是用于控制步长的常数。

3.6 局部搜索算法框架

在构造了状态、打分函数、邻域以及评价准则后，我们得到了构造一个局部搜索算法所需的基本框架。基于前几节提到的定义和策略，我们设计了一个用于解决包含 AllDifferent 约束问题的 CSP 的局部搜索算法，称为 AllDiff-LS。算法的主体部分如图 3.3 所示，算法采用了迭代局部搜索的框架，主要由三个部分组成，即两步选择策略、直接选择策略和重启策略，本章中已经介绍了前两个部分，重启策略部分将在下一章进行介绍。

我们在算法 3 中给出了我们方法的伪代码描述。AllDiff-LS 算法的输入是一个仅包含 AllDifferent 约束的 CSP $\mathcal{P} = (X, D, C)$ 以及一个时间限制 T ，输出则是一个关于输入的 CSP 的完整赋值 \mathcal{A}_b 。首先，我们使用前述的初始化算法 1 随机生成了一个初始的完整赋值 \mathcal{A} 。接下来，算法以迭代的方式运行，包含一个外循环和一个内循环。注意，内循环由最大迭代次数 $maxIter$ （第 24 行）控制，而外循环由截止时间 T （第 29 行）控制。

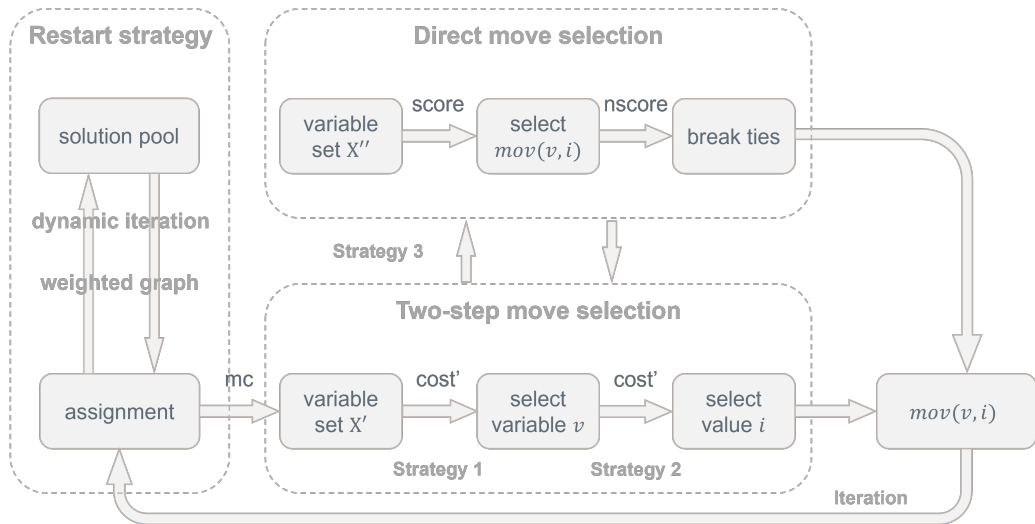


图 3.3 Local Search 的求解流程。

Figure 3.3 The solving process of Local Search.

在内循环（第 13-23 行）中，算法搜索具有最小代价的完整赋值。算法首先通过 *SelectMove* 函数获取一个操作 *mov*，然后实施 *mov*，同时更新两个禁忌列表 S_x 和 S_m ，以及有冲突的顶点。每个内循环结束后，下一轮的初始完整赋值和最大迭代次数将通过 *SelectSolution*（第 27-28 行，下一章介绍）获得。如果在给定的时间限制 T 内找到了一个没有冲突的完整赋值，那么返回该赋值（第 25 行）；否则，返回一个空集（第 30 行）。

算法 3 AllDiff-LS algorithm**输入:** A CSP $\mathcal{P} = (X, D, C)$ and time limit T **输出:** A complete assignment \mathcal{A}_b

```

1: Initialize  $\mathcal{G}, \mathcal{A}$ ;
2:  $\mathcal{AC} \leftarrow []$ ;
3:  $maxIter \leftarrow \alpha$ ;
4: repeat
5:   Initialize  $mc(x)$  for  $\forall x \in X$ ;
6:   Initialize  $X'$  according to  $mc(x)$  and  $\mathcal{A}$ ;
7:    $mode \leftarrow 0$ ;
8:    $\mathcal{A}_b \leftarrow \mathcal{A}$ ;
9:   if  $X' = []$  then
10:    Initialize  $X''$  according to  $\mathcal{A}$ ,  $mode \leftarrow \beta$ ;
11:   end if
12:   repeat
13:      $x, i \leftarrow SelectMove(\mathcal{A}, mode, X', X'')$ ;
14:      $\mathcal{A}(x) \leftarrow i$ ;
15:     Update  $mc(x)$ , update  $X'$  or  $X''$ ;
16:     if  $mode > 0$  and  $X' = \emptyset$  then
17:       Initialize  $X''$  according to  $\mathcal{A}$ ;
18:        $mode \leftarrow \beta$ ;
19:     end if
20:     if  $cost(\mathcal{G}, \mathcal{A}) \leq cost(\mathcal{G}, \mathcal{A}_b)$  then
21:        $\mathcal{A}_b \leftarrow \mathcal{A}$ ;
22:     end if
23:      $iter \leftarrow iter + 1$ ;
24:   until  $iter \geq maxIter$ 
25:   if  $cost(\mathcal{G}, \mathcal{A}_b) = 0$  then return  $\mathcal{A}_b$ 
26:   end if
27:    $\mathcal{A} \leftarrow SelectSolution(\mathcal{G}, \mathcal{A}_b, \mathcal{AC})$ ;
28:    $maxIter \leftarrow \mathcal{A}.step$ ;
29: until  $T$  is reached
30: return  $\emptyset$ 

```

3.7 本章小结

本章节将 CSP 转化为了图结构，并设计了一个用于求解此类问题的局部搜索算法，具体如下。在第一节中，我们给出了要研究的问题，以 AllDifferent 约束为主的 CSP 的形式化定义。在第二节中，我们将 CSP 中包含的 AllDifferent 约束转化为约束图，并设计了两个化简规则对约束进行化简。在第三节中，我们介绍了分两步选择操作的策略，作为直接选择操作策略的补充。在第四节中，我们设计了局部搜索的禁忌搜索框架。在第五节中，我们设计了包含打破平局策略的操作评价准则。在第六节中，我们将前面提到的策略整合在了一起，实现了局部搜索算法：AllDiff-LS，并介绍了算法的流程细节。

第4章 针对 AllDifferent 约束的加权和重启策略

本章节主要介绍用于局部搜索算法 AllDiff-LS 的重启策略，它主要介绍了基于解池构造的两个策略：图加权和动态迭代。首先，我们介绍了我们拟采用的解池技术，用来存储每轮局部搜索过程中得到的最优解。之后，我们介绍了如何对 AllDifferent 约束图加权，以及权值如何起作用。此外，我们将解池中的候选解绑定了新的信息，用来指导后续局部搜索的迭代次数。最后，我们还介绍了结合重启策略后的局部搜索算法的实现。

4.1 解池技术

在上一章，我们介绍了 AllDiff-LS 的算法框架，它基于迭代局部搜索的框架，其中迭代局部搜索是一个将局部搜索作为组件的元启发式方法。在迭代局部搜索流程中，会迭代执行局部搜索改进当前最优解，并在迭代一定次数还没有找到改进时重启，以生成新的解来扩展搜索空间。生成解的过程一般是通过施加扰动实现的，并且该算法会重复执行，直到达到算法设置的终止条件。引入扰动，有助于算法跳出当前的局部最优解。

在 AllDiff-LS 算法流程中，重启时我们会从局部最优解中抽取新的解，用于后续搜索，这是通过从一组选定的解生成重启点来实现的。这组解，我们称为解池 (solution pool)^[44]，它维护了搜索过程中得到的当前的最优解，也就是说，ACG 在解池 \mathcal{AC} 中的每个赋值下的冲突边的数量是相同的，并且当前已经是最小的。在搜索的初始阶段，解池被初始化为空并且拥有一个最大池尺寸限制。搜索过程中解池的维护机制如下：

- 当搜索在数次迭代后**没有找到改进**时，如果此时的局部最优解还没有在解池中，就将其添加到解池中，此时解池中的每个解都拥有相同的代价。
- 当搜索在数次迭代后**找到改进**时，为了保持最好的解，算法会直接更新解池，清空当前解池的同时将最新的局部最优解加入解池中。

我们会额外记录解池中每个候选解被访问的次数，如果此解池的大小超过了池尺寸限制，就会从解池中删除被访问最多的候选解，这样做的动机源于我们需要探索那些较少被访问的候选解，以探索更广的搜索空间。接下来，我们会介绍局部最优解的进入准则和候选解的生成准则，以通过解池存储和生成用于后续搜索的优良解。

解的准入策略。一个解 \mathcal{A}_1 在被加入到解池之前，需要先判断这个解在解池中是否已经存在。也就是说，我们需要保证对于解池中的每个候选解 \mathcal{A}_2 ，都有 $\mathcal{A}_1 \neq \mathcal{A}_2$ ，这个过程的复杂度是较高的。为了简化和候选解比较的复杂度，我们在保存局部最优解 \mathcal{A} 的同时，保存该解的冲突边集合。我们采用了文献^[41]的思路，引入相似解的概念如下。

定义 4.1. 两个解相似，当且仅当这两个解具有相同的冲突边。

从而，我们在判断一个解是否需要加入到解池中时，仅需要保证它和池中的任何解不相似即可。仅在出现相似解的情况下，我们才会比较两者是否相同，在不同时将池中的候选解更新为新的解，否则则抛弃这个解。在搜索的后期，冲突边的数量会非常少，由于判断相似仅需要比较解的冲突边，所以这极大地减小了比较的时间开销。

解的生成策略。一个解 \mathcal{A}_1 在被选中作为下一轮搜索的初始解之后，需要对它进行扰动以逃离局部最优。过去的研究表明，用于扰动的方法对性能有显著影响。扰动的方法有很多，例如，交换几对随机选定的相邻变量的赋值；删除几组随机选定的变量的赋值，使用随机或启发式的方法填充他们的赋值；以及随机游走策略等。而修改的范围也需要进行考量，Tasgetiren 等人^[45]发现，通过将一个或两个随机选择的移动插入到随机选定的位置，性能相当好。在本文中，我们通过随机修改变量赋值的方法来扰动解。

关于扰动的程度，我们将选定的扰动次数与当前解的冲突边个数，以及当前解被选中的频率挂钩。我们定义扰动次数为对选定解使用指定扰动技术扰动的次数。具体地，我们令基础的扰动次数为冲突边个数乘以一个系数，并在这个解被多次选中时，相应地增加系数的权重以得到最终的扰动次数。我们发现搜索的前期可能存在搜索不充分的情况，导致冲突边数量较多，进而导致扰动程度过大的问题。因此，在实际实验中，我们仅在冲突边的数量低于一定阈值时才对初始解施加扰动。

4.2 约束加权图和动态迭代策略

在局部搜索的过程中，加权策略是常用的一种调整搜索方向和搜索步长的方法。常见的加权策略有动态权重策略、自适应权重策略等。比如模拟退火算法中的温度参数，就是一种动态权重。在搜索初期，为了增加搜索的随机性和全局搜索能力，权重设置得较大；随着搜索的进行，为了逐渐收敛到全局最优解，权重逐渐减小。

为了避免找到重复的赋值，我们在重启部分提出了一种基于解池技术的加权策略，对 ACG 中的边集进行加权。具体地，对于 ACG 中的每个边 $e \in E_p$ ，我们使用 $e.w$ 作为 e 的权重，初始设置为 1，由此得到的 ACG，我们称为加权 AllDifferent 约束图，简称 WACG (Weighted ACG)。关于如何加权，我们采用了 PAWS 策略的带概率版本，具体地，边集的权值按照如下方式更新：当一个新的赋值被添加到 \mathcal{AC} 时，每个在赋值下的冲突边的权重以概率 θ 增加一。

由于解池中仅保存当前最优解，因此我们将 WACG 的权值与解池进行绑定。当解池 \mathcal{AC} 被重置时（即，找到了一个更小的 $cost$ 赋值），加权图中所有的边 $e \in E$ 的 $e.w$ 也被重置为 1。这样做的动机是找到新解意味着我们跳出了一个局部最优解，因此，之前关于边集的权值更新对于新的搜索空间可能起不到好的效

算法 4 *SelectSolution* function

输入: An ACG $\mathcal{G}(V, E)$, a complete assignment C and a solution pool \mathcal{AC}

输出: A complete assignment \mathcal{A}'

```

1:  $\mathcal{A}.step \leftarrow \alpha$ ;
2: if  $\mathcal{AC} = \emptyset$  or  $cost(\mathcal{G}, \mathcal{A}) < \mathcal{AC}.cost$  then
3:   Reset  $e.w$  for  $\forall e \in E_p$ ;
4:    $\mathcal{AC} \leftarrow \{\mathcal{A}\}$ ;
5:    $\mathcal{AC}.cost \leftarrow cost(\mathcal{G}, \mathcal{A})$ ;
6:   Update  $e.w$  ( $e \in E_p$ ) based on weight strategy;
7: else if  $cost(\mathcal{G}, \mathcal{A}) = \mathcal{AC}.cost$  then
8:   if  $\mathcal{A} \neq \mathcal{A}'$  for  $\forall \mathcal{A}' \in \mathcal{AC}$  then
9:      $\mathcal{AC} \leftarrow \mathcal{AC} \cup \{\mathcal{A}\}$ ;
10:    Update  $e.w$  ( $e \in E_p$ ) based on weight strategy;
11:   else
12:      $\mathcal{A}'.step \leftarrow \mathcal{A}'.step + \gamma * \alpha$ ;
13:   end if
14: end if
15: Update  $\mathcal{AC}$  based on its size and weight strategy;
16: Select a  $\mathcal{A}'$  from  $\mathcal{AC}$  randomly;
17: Disturb  $\mathcal{A}'$  based on disturbance strategy;
18: return  $\mathcal{A}'$ 

```

果, 甚至影响搜索的改进方向。

对边进行加权后, 我们得到加权代价, 即当 $\mathcal{A}(x) = i$ 时, 以 $\forall y \in N(x)$ 为终点的冲突边的总权重, 其计算如下。在得到加权代价后, 我们在两步选择策略的变量和值选择过程中使用 $cost'$ 代替 $cost$, 而算法的其他部分仍然使用 $cost$ 。

定义 4.2. 给定一个 ACG \mathcal{G} , 变量 x 赋值 i 的加权代价定义为

$$cost'(x, i) = \sum_{p \in N(x)} \sum_{q \in CN(p)} (p, q).w, \quad (4.1)$$

其中 $CN(p) = \{q \mid q \in PN(p)\}$ 。

我们最后介绍的一个策略是动态迭代策略。考虑我们前一节中提到的扰动策略, 我们提到了前期搜索不充分的问题, 除了调整扰动的程度外, 这种情况可以通过将静态的迭代次数设为动态次数来改善。

此外, 考虑一个候选解被选中的次数较多的情况, 除了表明此候选解在解池中存在时间较长 (通过更新相似解解决), 还说明以该候选解为初始解的搜索过程始终无法得到一个有效的解。其原因是搜索空间探索不充分, 除了通过施加更

加强烈的扰动外，我们还可以通过增加迭代次数的方式，对该解处于的搜索空间进行充分探索。基于以上两点，我们设计了用于充分探索的动态迭代策略如下，具体实现在介绍伪代码时介绍。

- 在选中候选解后，下一轮迭代的次数随着候选解的冲突边数量的增多而适当增加。
- 在一轮搜索没有提升（即当前最优解即初始解）时，下一轮以该解为初始解的内循环的迭代次数会适当增加。

我们在算法 4 中显示了选择初始完整赋值的伪代码，AllDiff-LS 算法会通过 *SelectSolution* 获得一个赋值 \mathcal{A} 和其相应的最大迭代次数 $\mathcal{A}.step$ 。在有新的拥有更少冲突边的赋值 \mathcal{A} 出现时，我们重置 WACG 的边权，并将 \mathcal{AC} 重置为 $\{\mathcal{A}\}$ 。我们用 $\mathcal{AC}.cost$ 表示解池中的赋值的代价。如果上一轮得到的赋值与 \mathcal{AC} 中的赋值代价相同且是新的赋值，我们将其放入 \mathcal{AC} 。当 \mathcal{A} 被插入到 \mathcal{AC} 时，我们要做两件事：首先， $\mathcal{A}.step$ 被初始化为 α ，其中 α 是一个手动设置的超参数，并在大循环迭代的过程中逐步减少；其次，更新 WACG，令 \mathcal{A} 的冲突边的权重以概率 θ 增加一。在更新 \mathcal{AC} 后，我们从中随机选择一个赋值 \mathcal{A} 进行扰动后作为下一轮内循环的初始解，并将 $\mathcal{A}.step$ 作为其最大迭代次数。如果一个以 \mathcal{A} 为初始赋值的内循环没有找到更好的解， $\mathcal{A}.step$ 就会增加 $\gamma * \alpha$ 并在其过大时将该赋值从 \mathcal{AC} 中删掉，其中 γ 和 α 是常数。

4.3 AllDiff-LS 工具设计


通过前面的叙述，我们能够得到一个求解仅由 AllDifferent 约束构成的 CSP 的算法。这一小节，我先介绍如何将其他的二元约束引入 AllDiff-LS 求解算法的框架，再简单介绍基于该算法实现的求解工具。

ACG 是通过 AllDifferent 约束进行二元分解构造的。因此，在这个框架下，我们可以引入其他类型的二元约束，只需构造类似的边类型即可。具体来说，我们可以考虑等于和不等偏序这两种二元约束。等式约束要求两个变量的值必须相等，而不等偏序约束则要求一个变量的值必须大于或小于另一个变量的值。

对于等式关系，我们将等式关系涉及的两个变量表达式顶点合并，并合并两个变量表达式涉及的边，得到新的 ACG。此外，计算打分函数时，需要额外计算合并的变量表达式顶点内部的冲突。通过调整顶点内部冲突和冲突边之间的权重，我们可以将等式约束引入到 AllDiff-LS 的框架之中。实际上，ACG 的构造过程中已经存在边集的合并。例如，对于约束 $AllDifferent(x_1, x_2, x_3)$ 和 $AllDifferent(x_1, x_2, x_4)$ ，通过二元分解得到的二元约束中， x_1 和 x_2 之间的不相等关系出现了两次，但由于该关系重复，所以它们在 ACG 中表现为一条边。

对于不等偏序关系的引入则比较自然，仅需新引入两个边类型，分别表示大于号和小于号，并修改针对这两类符号的冲突边定义。例如，对于约束 $x_1 < x_2$ ，我们引入边 (x_1, x_2) ，并在 $x_1 \geq x_2$ 时将该边定义为冲突边。

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		



1	6	2	8	5	7	4	9	3
5	3	4	1	2	9	6	7	8
7	8	9	6	4	3	5	2	1
4	7	5	3	1	2	9	8	6
9	1	3	5	8	6	7	4	2
6	2	8	7	9	4	1	3	5
3	5	6	4	7	8	2	1	9
2	4	1	9	3	5	8	6	7
8	9	7	2	6	1	3	5	4

图 4.1 名为 “AI Escargot” 的数独实例。

Figure 4.1 A Sudoku instance called "AI Escargot".

接下来介绍工具的实现环境和实验参数。求解工具由 C++ 实现，并用 g++ 编译，选项为 ‘-O3’。运行环境在一台配备有 Intel Xeon Platinum 8153 和 2048G 内存的服务器上进行，操作系统为 Centos 7.7.1908。AllDiff-LS 是用 C++ 实现的，AllDiff-LS 有五个参数： α 用于初始最大迭代次数， β 用于模式切换阈值， tt 用于禁忌方案， γ 用于增加赋值步骤，以及 θ 用于权重策略。参数根据^[46]和初步实验进行调整，设置如下： $\alpha = 100,000$ ， $\beta = 100$ ， $tt = rand(10) + 0.6 \times cost(\mathcal{G}, \mathcal{A})$ ， $\gamma = 5$ 和 $\theta = 0.25$ 。参数 β 被固定为 100，因为经验表明这个值可以产生良好的性能：在算法优化过程的后期，我们观察到，算法会频繁切换到直接移动选择策略，因此将参数 β 配置为一个相对较大的常数是较为合理的。

工具的输入是由两部分组成，一个 CSP，由整数变量、值域和 AllDifferent 约束组成，以及一个求解时间。工具可以对大数独、全间隔、N 皇后问题和互正交拉丁方问题进行求解。对于符合输入规格的输入文件，在给定时间内求解器会输出一个符合约束的可行解，或超时求解返回 unknown。以数独为例，我们给出一个著名的 9 阶数独 “AI Escargot”，如图 4.1 所示，算法可以在小于 0.01 秒内找到该问题的解。这是一个简单的小问题实例，在实验部分我们主要针对大规模实例进行比较。

4.4 本章小结

在这一章中，介绍了算法的补充策略，解池技术及其维护。具体地，在第一节中，我们介绍了解池技术的基本概念，以及分别通过解近似和扰动策略构造的解的准入和生成策略。在第二节中，我们介绍了基于解池技术的加权和动态迭代策略，具体地，算法构造了新的 WACG，并根据求解情况动态地调整后续求解的迭代次数。在第三节中，我们通过构造新的冲突类型，将二元约束引入到求解的框架中，并介绍了实现的求解工具的具体细节。

第5章 实验设计及结果分析

本章节针对基于 AllDiff-LS 算法实现的求解工具设计了实验，并对实验结果进行了分析。章节主要包含三部分：第一部分介绍实验安排，比如采用的数据集、比较的方法等；第二部分介绍算法在多个实验基准上同其他方法的性能比较；第三部分对实验中提到的各种策略的有效性进行消融实验。

5.1 实验设置

实验样例：在本章节，我们选择了四个经典问题作为基准：数独，N 皇后，全间隔和二正交拉丁方 (2-MOLS) 问题^[47]。它们都可以使用 AllDifferent 约束进行建模，并且这些问题的约束生成的 ACG 具有独特的特征，使这些问题极具代表性。下面，我给出这些问题的编码的具体细节。

定义 5.1 (数独). 一个数独 (Sudoku) 问题实例 S^n 可以用一个 $n^2 \times n^2$ 的网格表示，其中填充着 1 到 n^2 范围内的数字，这里 n 被称为问题的阶数。网格被划分为 n^2 个大小为 $n \times n$ 的子网格。通常一些单元格已经被固定为特定的值。问题要求每单元格中的数字满足每行、每列和每个子网格中每个数字恰好出现一次。

在数独问题中，所有变量表达式都是单变量表达式，并出现在多个 AllDifferent 约束中。以下为其编码：

$$\begin{aligned} & \text{AllDifferent}(x_{i,1}, x_{i,2}, \dots, x_{i,n^2}) \text{ for } 1 \leq i \leq n^2, \\ & \text{AllDifferent}(x_{1,j}, x_{2,j}, \dots, x_{n^2,j}) \text{ for } 1 \leq j \leq n^2, \\ & \text{AllDifferent}(x_{i*n+u}, x_{j*n+v} \mid \text{ for } 1 \leq u, v \leq n) \text{ for } 0 \leq i, j \leq n-1, \\ & x_{i,j} \in \{1, 2, \dots, n^2\} \text{ for } 1 \leq i, j \leq n^2. \end{aligned}$$

定义 5.2 (N 皇后). 一个 N 皇后 (*N-queens*) 问题实例 Q^n 是将 n 个皇后放置在一个 $n \times n$ 的棋盘上，使得没有皇后互相攻击（任何两个皇后都不能处于同一行、同一列或同一对角线上），这里 n 被称为问题的阶数。

在 N 皇后问题中，一个变量可以有多个变量表达式，但是每个变量表达式只存在于一个 AllDifferent 约束中。建模这个问题的一种方法是为每一行引入一个整数变量 x_i ，其中 $i = 1, 2, \dots, n$ ，它的取值范围是从列 1 到 n 。这意味着在每一行 i 中，皇后被放置在第 x_i 列上。以下为其编码：

$$\begin{aligned} & \text{AllDifferent}(x_1, x_2, \dots, x_n), \\ & \text{AllDifferent}(x_1 - 1, x_2 - 2, \dots, x_n - n), \\ & \text{AllDifferent}(x_1 + 1, x_2 + 2, \dots, x_n + n), \\ & x_i \in \{1, 2, \dots, n\} \text{ for } 1 \leq i \leq n. \end{aligned}$$

定义 5.3 (全间隔). 一个全间隔 (*All-Interval*) 问题实例, 表示为 \mathcal{A}^n , 涉及将 n 个不同的元素排列成一个序列, 使得任意两个元素之间的绝对差形成一个包含所有可能整数从 1 到 $n-1$ 的集合。 n 的值被称为问题的阶数。

在全间隔问题中, 仅存在两个 AllDifferent 约束, 变量和变量表达式之间是多对多的关系, 并且一个变量可能在一个约束中的两个变量表达式中出现。问题建模的形式化表示如下:

$$\begin{aligned} & \text{AllDifferent}(x_1, x_2, \dots, x_n), \\ & \text{AllDifferent}(|x_1 - x_2|, |x_2 - x_3|, \dots, |x_{n-1} - x_n|), \\ & x_i \in \{1, 2, \dots, n\} \text{ for } 1 \leq i \leq n. \end{aligned}$$

定义 5.4 (互正交拉丁方). 互正交拉丁方 (*MOLS*) 是由数学家欧拉提出的一个极具挑战性的组合问题。拉丁方阵可以看作没有子网格约束的数独问题。给定两个相同阶数 n 的拉丁方阵, 如果它们在每个对应位置的元素组合是唯一的, 那么它们是互相正交的, 表示为 *2-MOLS*(n)。

在互正交拉丁方问题中, 在采用 AllDifferent 约束进行建模的情况下, 一个变量存在于多个变量表达式中, 而且一个单一的变量表达式中包含多个变量, 这些变量也出现在多个 AllDifferent 约束中。因此, MOLS 的 AllDifferent 约束图比前三种情况更为复杂。以下为其一种编码方式:

$$\begin{aligned} & \text{AllDifferent}(x_{i,1}, x_{i,2}, \dots, x_{i,n}) \text{ for } 1 \leq i \leq n, \\ & \text{AllDifferent}(x_{1,j}, x_{2,j}, \dots, x_{n,j}) \text{ for } 1 \leq j \leq n, \\ & \text{AllDifferent}(y_{i,1}, y_{i,2}, \dots, y_{i,n}) \text{ for } 1 \leq i \leq n, \\ & \text{AllDifferent}(y_{1,j}, y_{2,j}, \dots, y_{n,j}) \text{ for } 1 \leq j \leq n, \\ & \text{AllDifferent}(x_{i,j} * n + y_{i,j} \mid \text{ for } 1 \leq i, j \leq n), \\ & x_{i,j}, y_{i,j} \in \{1, 2, \dots, n\} \text{ for } 1 \leq i, j \leq n. \end{aligned}$$

虽然存在一些人工制作的实例^[33,48], 但这些实例的规模较小, 相对容易解决。因此, 我们主要生成大规模随机实例进行实验。使用^[49]和^[50]中提到的方法, 我们生成了从 4 到 9 阶的数独实例 (即, 从 16×16 到 81×81 的大小)。我们为每个固定单元格比例生成了 100 个实例, 步长为 10%, 从 0% 到 90%, 总共生成了 6000 个单独的实例, 并使用 ‘ \mathcal{S} - n - r ’ 来表示阶数为 n 、固定单元格比例为 r 的数独实例族。对于 N 皇后问题, 我们比较了从 500 到 6000 阶的十二个实例 (步长为 500), 记为 ‘ \mathcal{Q} - n ’。对于全间隔问题, 我们比较了从 12 到 26 阶的八个实例 (步长为 2), 记为 ‘ \mathcal{A} - n ’。对于 2-MOLS 问题, 由于其难度级别, 我们只比较了阶数为 3、4、5、7、8 和 9 的六个实例 (已经证明 2-MOLS(6), 也称为 36 官问题, 没有解), 记为 ‘ \mathcal{L} - n ’。

比较方法: 我们将 AllDiff-LS 与一个启发式求解器 Yuck¹, 以及两个完备求解器 ILOG CPLEX Optimizer (版本 20.10, 简称 CPLEX) 和 Choco (版本 4.10.13)^[51] 进行了比较, 所有这些都支持 ‘AllDifferent’ 约束。Yuck 采用了^[52] 的想法, 并在 2022 MiniZinc 挑战赛的局部搜索赛道中获得了冠军, 而 CPLEX 和 Choco 分别是知名的商业和开源求解器, 它们都支持各种化简策略。此外, 对于数独, 我们还比较了两个专用的启发式算法, 来自^[40] 的蚁群算法 (ACS) 和来自^[50] 的迭代局部搜索算法 (ILS), 其中 ACS 被调整为接受大于 5 的阶数。在 ACS 算法中, 作者将约束传播和蚁群算法相结合, 通过 Minimum Remaining Values Heuristic 搜索求解空间。而在 ILS 算法中, 算法会在初始化时满足部分约束, 并通过交换变量赋值的方式进行局部搜索。

实验安排及指标: 后续所有实验都在一台配备有 Intel Xeon Platinum 8153 (2.00GHz) 和 2048G RAM 的服务器上进行, 系统为 Centos 7.7.1908。每个算法在一个实例上的执行时间限制 T 为 1000 秒。启发式算法在每个实例上运行十次, 其中支持随机种子 (AllDiff-LS, ACS) 的方法将使用不同的随机种子 (从 1 到 10)。其他算法在每个实例上只运行一次。对于每个算法, 我们使用 R 来表示一个实例族的成功运行百分比, 并使用 $time$ (以秒为单位) 来表示其平均成功时间。当 R 为 0 时, 其 $time$ 被记录为 ‘-’。

实验结果展示: 根据上文提到的求解方法和求解样例, 后续我们将比较这些求解器的求解性能, 以及通过一系列消融实验来证明我们设计的算法框架中各个策略的有效性, 具体指标为求解的成功率、比例和开销。

5.2 AllDiff-LS 求解 AllDifferent 约束的能力

根据每个问题的特点, 我们选择了具有代表性的基线方法进行比较, 具体地, 我们将问题实例分为了两类, 第一类是数独, 它是一类包含固定点的 CSP, 因此在搜索之前需要进行化简, 而其他三类无固定点的 CSP 属于第二类。

关于数独, 由于已经有专门的启发式算法性能优于一般的启发式求解器, 我们没有展示 Yuck 的实验结果, 其整体实验结果如表 5.1 所示。可以观察到, 在填充率超过 70% 后, 数独的难度急剧下降, 所有算法都可以通过推理规则轻易找到解决方案。其中, ACS 算法和 ACP-LS 算法的效率优于其他算法, 这证明了 ACP-LS 减少规则的有效性。

当填充比例在 40% 和 60% 之间时, 其求解难度最大。随着阶数的增加, 我们选择的算法中没有一个算法能保证 100% 的成功率。尤其是当阶数高于 5 时, 这些算法的求解性能开始恶化, 出现无法解决的情况。而 AllDiff-LS 的性能下降较慢, 直到最困难的 9 阶, 它仍然可以在 10 秒内解决除填充比例在 40%-60% 范围外的每一个实例。

¹<https://github.com/informarte/yuck>

表 5.1 AllDiff-LS 和其他最先进的基线方法在数独实例上的结果。

Table 5.1 Results of AllDiff-LS and other state-of-the-art baseline methods on Sudoku.

Instance	ACS		ILS		CPLEX		Choco		AllDiff-LS		Instance	ACS		ILS		CPLEX		Choco		AllDiff-LS	
family	R(%)	time	R(%)	time	R(%)	time	R(%)	time	R(%)	time	family	R(%)	time	R(%)	time	R(%)	time	R(%)	time	R(%)	time
S-4-0	100	0.04	100	0.08	100	18.50	100	1.11	100	<0.01	S-7-0	94.1	550.37	100	51.60	0	—	100	2.85	100	0.15
S-4-10	100	0.03	100	0.10	100	18.18	100	1.09	100	<0.01	S-7-10	92.2	559.82	29.2	544.69	1	814.81	90	58.82	100	0.15
S-4-20	100	0.03	100	0.15	100	17.15	100	1.09	100	<0.01	S-7-20	24.9	665.38	0	—	0	—	68	163.23	100	0.18
S-4-30	100	0.03	100	0.71	100	16.48	100	1.08	100	<0.01	S-7-30	0.3	798.53	0	—	0	—	4	226.72	100	0.24
S-4-40	100	0.02	100	1.52	100	16.79	100	1.08	100	<0.01	S-7-40	0	—	0	—	0	—	0	—	100	0.63
S-4-50	100	<0.01	100	0.04	100	14.01	100	1.03	100	<0.01	S-7-50	0	—	0	—	0	—	0	—	98.4	100.53
S-4-60	100	<0.01	100	<0.01	100	9.59	100	1.01	100	<0.01	S-7-60	99.2	8.92	68.3	271.46	49	619.98	100	1.48	100	0.06
S-4-70	100	<0.01	100	<0.01	100	7.40	100	1.02	100	<0.01	S-7-70	100	<0.01	100	3.14	100	30.86	100	1.32	100	<0.01
S-4-80	100	<0.01	100	<0.01	100	5.96	100	0.99	100	<0.01	S-7-80	100	<0.01	100	0.52	100	17.94	100	1.29	100	<0.01
S-4-90	100	<0.01	100	<0.01	100	5.39	100	1.05	100	<0.01	S-7-90	100	<0.01	100	0.20	100	9.60	100	1.27	100	<0.01
S-5-0	100	0.75	100	0.47	100	34.93	100	1.24	100	0.01	S-8-0	0	—	100	238.08	0	—	0	—	100	0.45
S-5-10	100	1.18	100	0.72	100	31.87	100	1.55	100	0.01	S-8-10	0	—	0	—	0	—	6	106.90	100	0.56
S-5-20	100	2.25	100	2.27	100	29.28	100	1.26	100	0.01	S-8-20	0	—	0	—	0	—	0	—	100	0.66
S-5-30	100	3.93	61.2	17.20	100	35.53	100	1.60	100	0.01	S-8-30	0	—	0	—	0	—	0	—	100	0.95
S-5-40	98.7	9.31	68.9	47.15	100	130.53	100	2.58	100	0.01	S-8-40	0	—	0	—	0	—	0	—	100	3.19
S-5-50	96.4	2.99	41.2	13.67	100	68.06	100	1.80	100	0.25	S-8-50	0	—	0	—	0	—	0	—	79.4	168.75
S-5-60	100	<0.01	100	0.08	100	15.73	100	1.01	100	<0.01	S-8-60	0	—	0	—	0	—	0	—	62	258.15
S-5-70	100	<0.01	100	0.02	100	10.23	100	1.04	100	<0.01	S-8-70	100	<0.01	100	18.24	100	90.17	100	1.28	100	<0.01
S-5-80	100	<0.01	100	<0.01	100	6.63	100	0.98	100	<0.01	S-8-80	100	<0.01	100	1.62	100	33.06	100	1.24	100	<0.01
S-5-90	100	<0.01	100	<0.01	100	5.58	100	1.03	100	<0.01	S-8-90	100	<0.01	100	0.47	100	5.90	100	1.22	100	<0.01
S-6-0	100	42.61	100	238.08	100	260.11	100	2.09	100	0.03	S-9-0	0	—	5	784.62	0	—	0	—	100	1.74
S-6-10	100	45.56	100	28.75	100	159.55	99	1.89	100	0.04	S-9-10	0	—	0	—	0	—	0	—	100	2.03
S-6-20	100	67.08	97.1	240.98	100	264.67	99	5.16	100	0.04	S-9-20	0	—	0	—	0	—	0	—	100	3.50
S-6-30	100	175.69	20.7	574.28	37	610.38	95	78.67	100	0.06	S-9-30	0	—	0	—	0	—	0	—	100	7.60
S-6-40	5.4	320.93	3.9	687.09	0	—	21	207.36	100	0.14	S-9-40	0	—	0	—	0	—	0	—	100	23.47
S-6-50	0	—	0	—	0	—	0	—	99.2	45.76	S-9-50	0	—	0	—	0	—	0	—	40.7	629.96
S-6-60	100	<0.01	100	4.49	100	40.45	100	1.24	100	<0.01	S-9-60	0	—	0	—	0	—	0	—	0	—
S-6-70	100	<0.01	100	0.54	100	16.78	100	1.20	100	<0.01	S-9-70	100	<0.01	100	109.72	100	114.35	100	1.64	100	<0.01
S-6-80	100	<0.01	100	0.16	100	11.56	100	1.19	100	<0.01	S-9-80	100	<0.01	100	5.19	100	39.13	100	1.59	100	<0.01
S-6-90	100	<0.01	100	0.08	100	8.80	100	1.17	100	<0.01	S-9-90	100	<0.01	100	1.18	100	10.92	100	1.57	100	<0.01

表 5.2 AllDiff-LS 和其他最先进的基准方法在 N 皇后，全间隔和 2-MOLS 问题上的结果。

Table 5.2 Results of AllDiff-LS and other state-of-the-art baseline methods on N-queens, All-interval and 2-MOLS.

Instance CPLEX Choco							Yuck							AllDiff-LS							Instance CPLEX Choco							Yuck							AllDiff-LS						
family	time	time	time	R(%)	time	R(%)	family	time	time	time	R(%)	time	R(%)	family	time	time	time	R(%)	time	R(%)	family	time	time	time	R(%)	time	R(%)	family	time	time	time	R(%)	time	R(%)							
Q-500	8.33	—	896.07	100	0.09	100	Q-2500	436.40	—	—	0	14.98	100	Q-4500	—	—	—	0	80.13	100	Q-5000	—	—	—	0	119.18	100	Q-5500	—	—	—	0	135.37	100							
Q-1000	69.93	—	—	0	1.08	100	Q-3000	624.28	—	—	0	28.03	100	Q-6000	—	—	—	0	167.59	100	Q-6000	—	—	—	0	167.59	100	Q-6000	—	—	—	0	167.59	100							
Q-1500	138.22	—	—	0	2.94	100	Q-3500	837.99	—	—	0	39.70	100	Q-4000	—	—	—	0	60.98	100	Q-4000	—	—	—	0	60.98	100	Q-4000	—	—	—	0	60.98	100							
Q-2000	272.83	—	—	0	8.48	100	Q-4000	—	—	—	0	60.98	100	Q-4000	—	—	—	0	60.98	100	Q-4000	—	—	—	0	60.98	100	Q-4000	—	—	—	0	60.98	100							
A-10	0.52	0.54	1.16	100	<0.01	100	A-16	2.86	174.36	6.84	100	0.01	100	A-22	217.25	—	565.35	100	6.01	100	A-22	217.25	—	565.35	100	6.01	100	A-22	217.25	—	565.35	100	6.01	100							
A-12	0.56	1.51	1.58	100	<0.01	100	A-18	3.24	276.35	52.44	100	0.23	100	A-24	164.58	—	—	0	42.31	100	A-24	164.58	—	—	0	42.31	100	A-24	164.58	—	—	0	42.31	100							
A-14	0.62	19.24	2.56	100	<0.01	100	A-20	76.51	—	64.10	100	1.13	100	A-26	—	—	—	0	214.75	90	A-26	—	—	—	0	214.75	90	A-26	—	—	—	0	214.75	90							
L-3	1.80	2.62	3.31	100	<0.01	100	L-5	3.20	1.22	29.27	100	<0.01	100	L-8	—	—	—	0	345.53	90	L-8	—	—	—	0	345.53	90	L-8	—	—	—	0	345.53	90							
L-4	2.16	0.95	2.71	100	<0.01	100	L-7	14.38	—	—	0	3.86	100	L-9	—	—	—	0	—	0	L-9	—	—	—	0	—	0	L-9	—	—	—	0	—	0							

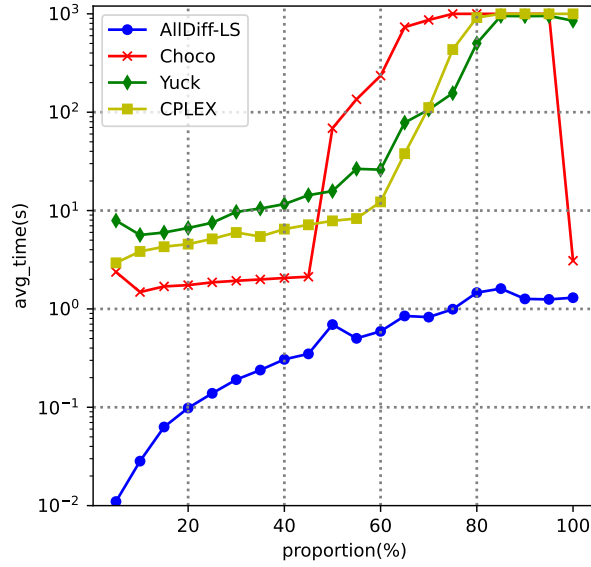


图 5.1 四种方法在给定基准测试上的结果。当 avg_time 为 1000s 时，意味着超时。

Figure 5.1 Results of methods on given benchmarks. when avg_time is 1000s, it means timeout.

此外，AllDiff-LS 算法可以保证除了阶数为 9 的实例外，其它的成功率超过 60%。特别是，在 11 个实例族上，ACS、ILS 和 CPLEX 无法解决任何实例，而 AllDiff-LS 在其中 8 个上可以达到 100% 的成功率，其余三个的成功率则分别是 98.4%、79.4% 和 40.7%。

对于 N 皇后、全间隔和 2-MOLS 问题，我们将 AllDiff-LS 的性能与 Yuck、CPLEX 和 Choco 进行了比较。由于每个实例族只包含一个实例，我们只显示启发式算法 (AllDiff-LS, Yuck) 的解决成功率。代表这三种问题的 ACGs 不需要被简化，但它们的结构更复杂，使得局部搜索有更高的求解压力。根据表 5.2，可以看出，除 \mathcal{L} -9 外，AllDiff-LS 可以解决几乎所有给定的实例。对于一些困难的实例，如 \mathcal{A} -26 和 \mathcal{L} -8，AllDiff-LS 也可以达到 90% 的解决成功率。而对于其他算法，CPLEX 的表现最好，但仍然无法求解大部分 N 皇后问题，以及一些高阶的全间隔问题和 2-MOLS 问题。因此在这三类实例上，相较于其他三个求解器，AllDiff-LS 的效果仍然是最好的。

我们还比较了 AllDifferent 约束数量对解决难度的影响。具体来说，我们选择了难度适中的实例族 \mathcal{S} -7-0，并通过随机删除实例族的约束，按顺序得到了 20 个新的约束逐渐减少的实例族（累计 2000 个实例）（约束从原始的 5% 到 100% 不等）。结果如图 5.1 所示。由于一些实例不再是数独，我们使用 Yuck 代替 ACS 和 ILS。横坐标表示约束逐渐增加的实例族，纵坐标表示每个实例族的平均时间消耗。结果表明，随着约束数量的增加，问题的解决难度在一定比例内会急剧增加，与其他方法相比，AllDiff-LS 受影响最小。

最后，我们给出额外的三个求解器：OR-Tools、LocalSolver 和 Kissat，出于类型原因，我们把他们的求解结果放在最后介绍。其中，Google 的 OR-Tools 是一套开源的软件库，它包含了丰富的解决方案，可以用于求解 CSP、LP、MIP

表 5.3 OR-Tools、LocalSolver 和 SAT 求解器 Kissat 在数独实例上的结果。

Table 5.3 Results of OR-Tools, LocalSolver and SAT Solver Kissat on Sudoku.

Instance	OR-Tools	Kissat	LocalSolver	Instance	OR-Tools	Kissat	LocalSolver	Instance	OR-Tools	Kissat	LocalSolver						
family	R(%)	time	R(%)	time	time	family	R(%)	time	R(%)	time	time						
S-4-0	100	1.06	100	0.03	0.79	S-6-0	100	101.77	100	3.51	12.71	S-8-0	100	230.82	100	5.05	—
S-4-10	100	0.81	100	0.04	0.84	S-6-10	100	70.36	100	12.90	19.32	S-8-10	98	396.82	0	—	—
S-4-20	100	0.67	100	0.04	0.55	S-6-20	100	15.78	100	12.38	95.83	S-8-20	99	295.43	0	—	—
S-4-30	100	0.58	100	0.03	0.51	S-6-30	100	7.45	100	11.45	307.31	S-8-30	100	229.50	0	—	—
S-4-40	100	0.51	100	0.03	0.47	S-6-40	100	15.71	100	54.29	—	S-8-40	10	—	0	—	—
S-4-50	100	0.03	100	0.03	0.22	S-6-50	80	255.22	100	475.99	—	S-8-50	0	—	0	—	—
S-5-0	100	8.89	100	0.33	3.69	S-7-0	100	386.07	100	88.60	49.62	S-9-0	0	—	0	—	—
S-5-10	100	2.64	100	0.36	4.56	S-7-10	100	213.12	95	264.54	83.90	S-9-10	0	—	0	—	—
S-5-20	100	1.53	100	0.65	9.36	S-7-20	100	216.22	83	307.86	605.231	S-9-20	0	—	0	—	—
S-5-30	100	1.26	100	0.72	35.66	S-7-30	100	143.73	42	289.48	—	S-9-30	0	—	0	—	—
S-5-40	100	1.08	100	1.40	65.53	S-7-40	76	—	0	—	—	S-9-40	0	—	0	—	—
S-5-50	100	0.53	100	0.52	5.03	S-7-50	0	—	0	—	—	S-9-50	0	—	0	—	—

表 5.4 OR-Tools 和 LocalSolver 在其他实例上的结果。

Table 5.4 Results of OR-Tools and LocalSolver on other benchmarks.

Instance	OR-Tools	LocalSolver	Instance	OR-Tools	LocalSolver	Instance	OR-Tools	LocalSolver	Instance	OR-Tools	LocalSolver
family	time	time	family	time	time	family	time	time	family	time	time
Q-500	105.29	212.35	Q-2000	—	—	Q-3500	—	—	Q-5000	—	—
Q-1000	559.75	—	Q-2500	—	—	Q-4000	—	—	Q-5500	—	—
Q-1500	—	—	Q-3000	—	—	Q-4500	—	—	Q-6000	—	—
A-12	0.94	0.14	A-16	0.60	0.20	A-20	0.73	0.22	A-24	0.93	0.25
A-14	0.56	0.14	A-18	0.66	0.22	A-22	0.88	0.23	A-26	1.05	0.24
L-3	0.89	0.15	L-5	0.51	0.20	L-8	13.41	—			
L-4	0.37	0.16	L-7	6.37	17.55	L-9	63.69	—			

等各类优化问题，是比较新的开源求解工具，并且支持多线程求解；LocalSolver 是一款基于局部搜索的优化求解器，专门设计用来解决各种复杂的组合优化问题，它是一款商业求解器，同样支持多线程求解；Kissat 是一款高效的 SAT 求解器，它的一个显著特点是它的求解策略旨在利用现代多核处理器的并行计算能力，以提高求解效率。

由于 LocalSolver 不支持并行运行多个实例，对于数独的每个实例族，我们仅运行其中随机的一个实例，在表中我们不给出求解比例，仅提供单个实例的求解时间作为参考。同时，由于除数独外的例子都包含变量表达式，使用 SAT 对其编码比较困难（需要借助加法器），所以我们仅给出数独的 Kissat 求解结果。这些求解器的多线程求解结果如表 5.3 和表 5.4 所示，其中数独实例仍然舍弃了容易求解的部分样例族。可以观察到，OR-Tools 的求解能力最好，并且能够求解 AllDiff-LS 无法求解的实例 L-9，但在除此之外的其他实例上仍然不如 AllDiff-LS。而将 CSP 编码成 SAT 问题再使用 Kissat 求解的方法，相较于 OR-Tools 求解器在低阶数独上效果较好，但在高阶例子上表现较差。

5.3 AllDiff-LS 涉及的策略的有效性

我们首先研究两个简化规则的有效性。由于 N-queens 和 2-MOLS 的 ACG 无法被简化，我们在数独实例上进行了实验，其中推理规则的效果是非琐碎的（去除了填充比例小于 40% 的实例）。结果如表 5.5 所示，其中 *avg* 表示每个实例族变量数量的平均简化比例。可以看出，使用两个简化规则比只使用 rule1 或 rule2 更好，并且在填充率高的情况下，大部分变量都可以被简化。

表 5.5 在给定数独实例上使用不同化简规则对应的变量简化比率。

Table 5.5 The variable simplification ratio of different simplification rules on given Sudoku.

Instance family	Rule1+2 <i>avg</i> (%)	Rule1 <i>avg</i> (%)	Rule2 <i>avg</i> (%)	Instance family	Rule1+2 <i>avg</i> (%)	Rule1 <i>avg</i> (%)	Rule2 <i>avg</i> (%)	Instance family	Rule1+2 <i>avg</i> (%)	Rule1 <i>avg</i> (%)	Rule2 <i>avg</i> (%)
S-4-40	9.44	8.22	1.18	S-6-40	0.43	0.43	0.01	S-8-40	0.02	0.02	0.00
S-4-50	79.34	70.41	27.20	S-6-50	4.28	3.97	0.21	S-8-50	0.48	0.47	0.01
S-4-60	94.91	94.90	93.95	S-6-60	96.54	96.54	10.63	S-8-60	7.10	6.42	0.41
S-4-70	95.29	95.29	95.28	S-6-70	98.93	98.93	98.93	S-8-70	99.12	99.12	99.09
S-4-80	91.06	91.06	91.06	S-6-80	98.10	98.10	98.09	S-8-80	99.40	99.40	99.40
S-4-90	82.96	82.96	82.96	S-6-90	93.78	93.78	93.78	S-8-90	97.22	97.22	97.22
S-5-40	1.53	1.42	0.09	S-7-40	0.09	0.09	0.00	S-9-40	0.01	0.01	0.00
S-5-50	20.92	14.69	2.25	S-7-50	1.35	1.31	0.04	S-9-50	0.16	0.16	0.00
S-5-60	95.98	95.92	83.16	S-7-60	43.80	18.48	1.63	S-9-60	3.05	2.92	0.10
S-5-70	97.65	97.65	97.64	S-7-70	99.11	99.11	99.10	S-9-70	99.31	99.30	15.45
S-5-80	95.54	95.54	95.54	S-7-80	98.90	98.90	98.90	S-9-80	99.56	99.56	99.56
S-5-90	89.42	89.42	89.42	S-7-90	95.95	95.95	95.95	S-9-90	98.28	98.28	98.28

此外，我们需要对文章中提到的主要策略进行消融实验，我们将实验分为了两大组，一组用来研究移动选择方法和重启策略的影响，而另一组用于展示了禁忌策略 1、禁忌策略 3 和打破平局技术对算法解决效率的影响。

第一组实验结果如图 5.2 所示，每个点代表一个随机种子实验。为了使结果更清晰，对于数独实验，我们选择了五个困难的实例族（S-6-50、S-7-50、S-8-50、S-8-60 和 S-9-50），一个点代表十个随机种子实验的结果，其中运行时间是它们运行时间的平均值。其中，AllDiff-LS-DM 用直接移动选择方法替换了我们的移动选择；在 AllDiff-LS-FL 中，使用固定数量的内循环迭代替代动态迭代，并从算法 4 中移除权重策略。可见，在数独问题上，AllDiff-LS-DM 的性能在几乎所有实例中都比 AllDiff-LS 弱，从运行时间的比较中可以观察到移动选择策略的有效性。此外，AllDiff-LS-FL 在耗时的数独实例中表现出劣势。在其他三种类型的问题上，AllDiff-LS 对 AllDiff-LS-DM 的优势仍然存在，但由于一些实例的规模较小，AllDiff-LS 对 AllDiff-LS-FL 的效率提升不明显。

在图 5.3 中，我们展示了其他策略对数独实例的提升效果。具体的，相较于原 AllDiff-LS 算法，在 AllDiff-LS-BT 中我将策略 1 替换为了传统的 tabu 策略，在 AllDiff-LS-SW 中我删除了策略 3，而在 AllDiff-LS-TA 中我删除了打破平局的策略。AllDiff-LS-TA 算法在一些较难的例子上会出现循环，因此求解会超时。

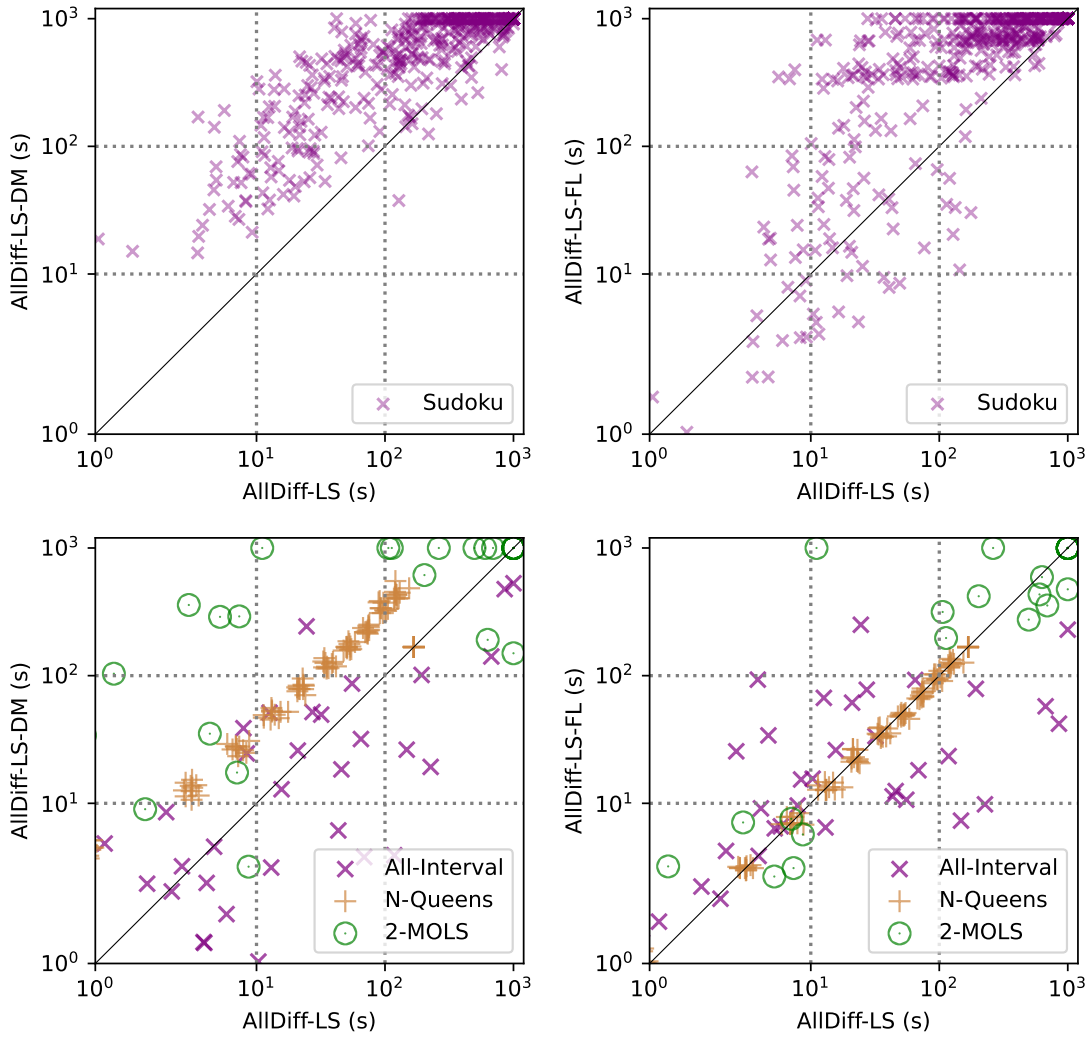


图 5.2 在四类实例上 AllDiff-LS (横坐标) 和其他两个版本 (纵坐标) 的平均运行时间比较。
 Figure 5.2 Comparison of the average running time of AllDiff-LS (abscissa) and the other two versions (ordinate) on four kinds of benchmarks.

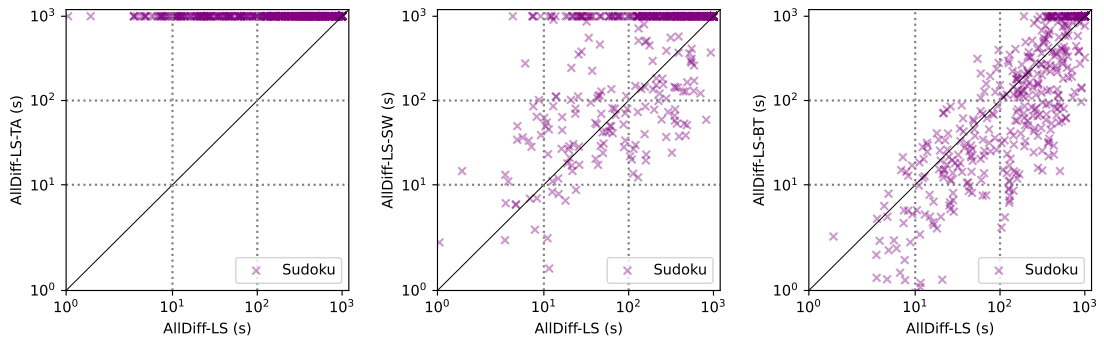


图 5.3 在数独上 AllDiff-LS (横坐标) 与其他三个版本 (纵坐标) 的平均运行时间比较。
 Figure 5.3 Comparison of the average running time of AllDiff-LS (abscissa) and the other three versions (ordinate) on Sudoku benchmarks.

5.4 本章小结

本章节通过设计实验，对基于 AllDiff-LS 算法实现的求解工具进行了深入的性能评估和策略分析。在第一节中，我们详细介绍了实验的安排，包括所使用的数据集和比较的方法。在第二节中，我们展示了算法在多个实验基准上与其他方法的性能比较结果，证明了其优越的性能。在第三节中，我们通过消融实验，对实验中提到的各种策略的有效性进行了验证。实验结果表明，基于 AllDiff-LS 算法实现的求解工具在处理各类问题时都展现出了强大的性能，证明了我们选择和实施的策略的有效性。

第6章 总结与展望

本章节主要对前面介绍的工作进行系统性的总结概括，归纳其中的主要贡献点，并对下一步的工作进行展望。

6.1 工作总结

包含 AllDifferent 约束的约束满足问题既重要又具有挑战性。虽然已有学者提出了一些算法来解决这些问题，但它们在大规模问题实例上的扩展性不佳。本工作主要提出了一种新的用于求解 AllDifferent 约束的局部搜索算法——AllDiff-LS，具体地，它由三部分组成：将 CSP 转化为局部搜索算法易处理的图结构，构造适用于给定图结构的局部搜索算法，以及设计用于该算法的重启策略。在三、四章中，我们对这三部分进行了详细的介绍。通过一系列实验，表明我们的算法在解决 AllDifferent 约束中的有效性，它可以在几分钟内解决大规模和复杂的问题实例。

对于 CSP 中的 AllDifferent 约束，我们构造了一个异构图称为 AllDifferent 约束图 (ACG)，它的思路是使用图结构对这些 AllDifferent 约束二元分解后得到的二元约束的并集进行表示。由于我们隐去了值域这个信息，一个 ACG 得以描述整组 AllDifferent 约束，从而可以施展更强的弧一致性算法。之后，我们定义了图上的赋值操作，以及基于它的化简规则，其核心思想是，根据变量和变量表达式顶点的度对图中的顶点和边进行删减。在对约束化简之后，我们通过随机赋值的方式获得最初的候选解。

我们将初始化后的 ACG 和候选解作为局部搜索的输入，通过定义状态、移动、评价准则等要素，设计了局部搜索算法。鉴于一个移动可以被分为两部分，我们设计了两步选择的移动策略作为直接选择策略的轮换策略。在此基础上，我们设计了一整套禁忌策略，它由三个小策略组成，前两个策略用于保证算法可以跳出循环，后一个策略则提供了选择操作的快速轮换机制，保证了算法的性能。同时，我们设计了打破对称的策略，借助变量表达式的邻域关系打破僵局。

此外，我们设计了基于解池技术的一整套重启策略。首先我们介绍了解池的维护机制，并提出了解的准入和生成策略，这两个策略分别用到了解近似的概念，和对解的扰动策略。此外，我们介绍了通过解池对 ACG 进行加权，从而减少对已探索区域的重复探索。最后，我们介绍了动态迭代的思想，在解加入解池和从解池中选择时，通过一定的策略增加下一轮迭代的最大迭代次数，这样做的动机是对复杂区域进行充分的探索，以寻找潜在的更优解。

基于上述思想，我们得到了求解 AllDifferent 约束的 AllDiff-LS 算法，通过将等式约束、偏序不等式约束引入到 ACG 中，我们可以处理比 AllDifferent 约束更广泛的 CSP 类型。比较实验和消融实验表明 AllDiff-LS 算法拥有更好的求解

能力，并且证明了算法中涉及各个策略的有效性。

6.2 下一步的工作

在未来，我们希望将局部搜索算法更加完善，让其适用于更广泛的约束满足问题上，实现一个更加通用的求解器，用来提升 CSP 求解的能力和效率。此外，将启发式策略和完备策略结合在一起对 CSP 进行求解，是一个有希望的改进方向。完备求解器一般依赖于传播和搜索，并因此具有较强的推理能力。一方面，在局部搜索之前或中途都可以使用约束传播对解空间进行化简；另一方面，完备搜索算法中关于变量序、分支等的启发式，同样可以应用于局部搜索中，做为指引。而其中一个思路是先在一个小的搜索窗口中使用完备的求解算法，如果无法求解则转为调用启发式求解方法。

参考文献

- [1] Van Beek P. Backtracking search algorithms [M]//Foundations of artificial intelligence: volume 2. Elsevier, 2006: 85-134.
- [2] Hoos H H, Tsang E. Local search methods [M]//Foundations of Artificial Intelligence: volume 2. Elsevier, 2006: 135-167.
- [3] Barták R. Theory and practice of constraint propagation [C]//Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control: volume 50. 2001.
- [4] Beame P, Kautz H, Sabharwal A. Understanding the power of clause learning [C]//IJCAI. Citeseer, 2003: 1194-1201.
- [5] Huang J, et al. The effect of restarts on the efficiency of clause learning. [C]//IJCAI: volume 7. 2007: 2318-2323.
- [6] Van Laarhoven P J, Aarts E H, van Laarhoven P J, et al. Simulated annealing [M]. Springer, 1987.
- [7] Mirjalili S, Mirjalili S. Genetic algorithm [J]. Evolutionary algorithms and neural networks: Theory and applications, 2019: 43-55.
- [8] Wang D, Tan D, Liu L. Particle swarm optimization algorithm: an overview [J]. Soft computing, 2018, 22: 387-408.
- [9] Biere A, Heule M, van Maaren H. Handbook of satisfiability: volume 185 [M]. IOS press, 2009.
- [10] Barrett C, Tinelli C. Satisfiability modulo theories [J]. Handbook of model checking, 2018: 305-343.
- [11] Dantzig G B. Linear programming [J]. Operations research, 2002, 50(1): 42-47.
- [12] Sorensson N, Een N. Minisat v1. 13-a sat solver with conflict-clause minimization [J]. SAT, 2005, 2005(53): 1-2.
- [13] De Moura L, Bjørner N. Z3: An efficient smt solver [C]//International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008: 337-340.
- [14] Barbosa H, Barrett C, Brain M, et al. cvc5: A versatile and industrial-strength smt solver [C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2022: 415-442.
- [15] Van Hoes W J. The alldifferent constraint: A survey [J]. arXiv preprint cs/0105015, 2001.
- [16] Régim J C. A filtering algorithm for constraints of difference in cps [C]//AAAI: volume 94. 1994: 362-367.
- [17] Ford L R, Fulkerson D R. A simple algorithm for finding maximal network flows and an application to the hitchcock problem [J]. Canadian journal of Mathematics, 1957, 9: 210-218.
- [18] Edmonds J, Karp R M. Theoretical improvements in algorithmic efficiency for network flow problems [J]. Journal of the ACM (JACM), 1972, 19(2): 248-264.
- [19] Michel L, Hentenryck P V. A constraint-based architecture for local search [C]//Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 2002: 83-100.
- [20] Simonis H. Sudoku as a constraint problem [C]//CP Workshop on modeling and reformulating Constraint Satisfaction Problems. 2005: 13-27.

- [21] Rivin I, Vardi I, Zimmermann P. The n-queens problem [J]. The American Mathematical Monthly, 1994, 101(7): 629-639.
- [22] Morris R, Starr D. The structure of all-interval series [J]. Journal of Music Theory, 1974, 18 (2): 364-389.
- [23] Mann H B. The construction of orthogonal latin squares [J]. The Annals of Mathematical Statistics, 1942, 13(4): 418-423.
- [24] Apt K R. The essence of constraint propagation [J]. Theoretical computer science, 1999, 221 (1-2): 179-210.
- [25] Apt K. Principles of constraint programming [M]. Cambridge university press, 2003.
- [26] Nethercote N, Stuckey P J, Becket R, et al. Minizinc: Towards a standard cp modelling language [C]//International Conference on Principles and Practice of Constraint Programming. Springer, 2007: 529-543.
- [27] Hopcroft J E, Karp R M. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs [J]. SIAM Journal on computing, 1973, 2(4): 225-231.
- [28] Edmonds J. Paths, trees, and flowers [J]. Canadian Journal of mathematics, 1965, 17: 449-467.
- [29] Micali S, Vazirani V V. An $O(V^2)$ algorithm for finding maximum matching in general graphs [C]//21st Annual symposium on foundations of computer science (Sfcs 1980). IEEE, 1980: 17-27.
- [30] Hall P. On representatives of subsets [J]. Classic Papers in Combinatorics, 1987: 58-62.
- [31] Gent I P, Miguel I, Nightingale P. Generalised arc consistency for the alldifferent constraint: An empirical survey [J]. Artificial Intelligence, 2008, 172(18): 1973-2000.
- [32] Zhang X, Li Q, Zhang W. A fast algorithm for generalized arc consistency of the alldifferent constraint. [C]//IJCAI. 2018: 1398-1403.
- [33] Zhang X, Gao J, Lv Y, et al. Early and efficient identification of useless constraint propagation for alldifferent constraints [C]//Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence. 2021: 1126-1133.
- [34] Li Z, Wang Y, Li Z. A bitwise gac algorithm for alldifferent constraints [C]//Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence. 2023: 1988-1995.
- [35] Zhen L, Li Z, Li Y, et al. Eliminating the computation of strongly connected components in generalized arc consistency algorithm for alldifferent constraint [C]//Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence. 2023: 2049-2057.
- [36] Glover F. Tabu search: A tutorial [J]. Interfaces, 1990, 20(4): 74-94.
- [37] Lourenço H R, Martin O C, Stützle T. Iterated local search [M]//Handbook of metaheuristics. Springer, 2003: 320-353.
- [38] Codognet P, Diaz D. Yet another local search method for constraint solving [C]//Stochastic Algorithms: Foundations and Applications: International Symposium, SAGA 2001 Berlin, Germany, December 13–14, 2001 Proceedings. Springer, 2001: 73-90.
- [39] Jin Y, Hao J. Solving the Latin square completion problem by memetic graph coloring [J/OL]. IEEE Transactions on Evolutionary Computation, 2019, 23(6): 1015-1028. <https://doi.org/10.1109/TEVC.2019.2899053>.
- [40] Lloyd H, Amos M. Solving Sudoku with ant colony optimization [J/OL]. IEEE Transactions on Games, 2020, 12(3): 302-311. <https://doi.org/10.1109/TG.2019.2942773>.
- [41] Pan S, Wang Y, Yin M. A fast local search algorithm for the Latin square completion problem [C/OL]//Thirty-Sixth AAAI Conference on Artificial Intelligence. 2022: 10327-10335. <https://ojs.aaai.org/index.php/AAAI/article/view/21274>.

- [42] Glover F, Laguna M. Tabu search [J]. Handbook of Combinatorial Optimization, 1998: 2093-2229.
- [43] Cai S, Su K, Sattar A. Local search with edge weighting and configuration checking heuristics for minimum vertex cover [J]. Artificial Intelligence, 2011, 175(9-10): 1672-1696.
- [44] Dong X, Chen P, Huang H, et al. A multi-restart iterated local search algorithm for the permutation flow shop problem minimizing total flow time [J]. Computers & Operations Research, 2013, 40(2): 627-632.
- [45] Tasgetiren M F, Liang Y C, Sevkli M, et al. A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem [J]. European journal of operational research, 2007, 177(3): 1930-1947.
- [46] Dorne R, Hao J K. Tabu search for graph coloring, t-colorings and set t-colorings [M]//Metaheuristics: Advances and trends in local search paradigms for optimization. Springer, 1999: 77-92.
- [47] Colbourn C J, Dinitz J H. Mutually orthogonal latin squares: a brief survey of constructions [J]. Journal of Statistical Planning and Inference, 2001, 95(1-2): 9-48.
- [48] Mantere T, Koljonen J. Solving and analyzing Sudokus with cultural algorithms [C]//Proceedings of the IEEE Congress on Evolutionary Computation (CEC). 2008: 4053-4060.
- [49] Lewis R. Metaheuristics can solve Sudoku puzzles [J/OL]. Journal of Heuristics, 2007, 13(4): 387-401. <https://doi.org/10.1007/s10732-007-9012-8>.
- [50] Musliu N, Winter F. A hybrid approach for the Sudoku problem: Using constraint programming in iterated local search [J/OL]. IEEE Intelligent Systems, 2017, 32(2): 52-62. <https://doi.org/10.1109/MIS.2017.29>.
- [51] Prud'homme C, Fages J G, Lorca X. Choco solver [J]. Website, March, 2019.
- [52] Björddal G, Monette J N, Flener P, et al. A constraint-based local search backend for minimizing [J]. Constraints, 2015, 20: 325-345.

作者简历及攻读学位期间发表的学术论文与研究成果

已发表（或正式接受）的学术论文

1. ASE 2023, New Ideas Track 共一作者
2. ISSTA 2023, Distinguished Paper 第二作者
3. PRICAI 2023 第二作者
4. ASE 2023, Tool Track 第三作者

投稿经历

1. AllDiff-LS: Solving Alldifferent Constraints with Efficient Local Search, AAAI 2023 过第一阶段，未中。

