软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(9):4153−4166 [doi: 10.13328/j.cnki.jos.006644]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种高效的 FDE 并行传播算法*

李 哲[1,2], 于哲舟[1,2], 李占山[1,2]

[1](吉林大学 计算机科学与技术学院, 吉林 长春 130012)

[2](符号计算与知识工程教育部重点实验室 (吉林大学), 吉林 长春 130012)

通信作者: 李占山, E-mail: lizs@jlu.edu.cn

摘 要: 约束规划 (constraint programming, CP) 是表示和求解组合问题的经典范式之一. 扩展约束 (extensional constraint) 或称表约束 (table constraint) 是约束规划中最为常见的约束类型. 绝大多数约束规划问题都可以用表约束表达. 在问题求解时, 相容性算法用于缩减搜索空间. 目前, 最为高效的表约束相容性算法是简单表约缩减 (simple table reduction, STR) 算法簇, 如 Compact-Table (CT) 和 STRbit 算法. 它们在搜索过程中维持广义弧相容 (generalized arc consistency, GAC). 此外, 完全成对相容性 (full pairwise consistency, fPWC) 是一种比 GAC 剪枝能力更强的相容性. 最为高效的维持 fPWC 算法是 PW-CT 算法. 多年来, 人们提出了多种表约束相容性算法来提高剪枝能力和执行效率. 因子分解编码 (factor-decomposition encoding, FDE) 通过对平凡问题重新编码. 它一定程度地扩大了问题模型, 使在新的问题上维持相对较弱的 GAC 等价于在原问题上维持 fPWC. 目前, FDE 的合适 STR 算法是 STRFDE 和 STR2, 而不是 CT. 这是由于 CT 算法可能产生内存溢出问题. 在维持相容性算法的过程中, 需要将迭代地调用各个约束执行其相容性算法过滤搜索空间, 这个过程称为约束传播. 动态提交方案是一个并行约束传播框架, 可以并行地调度约束执行传播算法. 它在大规模问题中, 改进效果尤为明显. 改进 STRFDE 和动态提交传播算法. 针对 FDE 提出了 PSTRFDE 算法. PSTRFDE 可以嵌入到动态提交方案中, 进一步提高了约束规划问题的求解效率. 大量的实验表明, PSTRFDE 与 CT 和 STRbit 相比, 可以减少内存占用; 与 STRFDE 和 STR2 相比, 可以提高算法的效率. 所作工作充分说明了 PSTRFDE 是 FDE 上最为高效的过滤算法.

关键词: 约束规划; 并行约束传播; 相容性算法; 简单表缩减算法

中图法分类号: TP18

中文引用格式: 李哲, 于哲舟, 李占山. 一种高效的FDE并行传播算法. 软件学报, 2023, 34(9): 4153–4166. http://www.jos.org.cn/1000-9825/6644.htm

英文引用格式: Li Z, Yu ZZ, Li ZS. Efficient Parallel Propagation Algorithm for FDE. Ruan Jian Xue Bao/Journal of Software, 2023, 34(9): 4153–4166 (in Chinese). http://www.jos.org.cn/1000-9825/6644.htm

Efficient Parallel Propagation Algorithm for FDE

LI Zhe[1,2], YU Zhe-Zhou[1,2], LI Zhan-Shan[1,2]

[1](College of Computer Science and Technology, Jilin University, Changchun 130012, China)

[2](Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education (Jilin University), Changchun 130012, China)

Abstract: Constraint programming (CP) is one of the classical paradigms for representing and solving combinatorial problems. Extensional constraints, also called table constraints, are the most common type of constraints in CP, and most CP problems can be expressed by table constraints. In the problem-solving process, consistency algorithms are used to reduce the search space, and the simple table reduction

* 基金项目: 国家自然科学基金 (61802056); 吉林省自然科学基金 (20180101043JC); 吉林省发展和改革委员会产业技术研究与开发项目 (2019C053-9); 中国科学院太空应用重点实验室开放基金 (LSU-KFJJ-2019-08)
收稿时间: 2021-01-31; 修改时间: 2021-09-27; 采用时间: 2021-12-30; jos 在线出版时间: 2022-12-28
CNKI 网络首发时间: 2022-12-29

(STR) algorithms are the most efficient consistency algorithms with table constraints, including Compact-Table (CT) and STRbit algorithms, both of which maintain the generalized arc consistency (GAC) during the search. In addition, the full pairwise consistency (fPWC) is stronger than GAC in the pruning capability, and the most efficient fPWC maintenance algorithm is the PW-CT algorithm. Over the years, many consistency algorithms with table constraints are proposed to improve the pruning capability and efficiency. Factor-decomposition encoding (FDE) recodes trivial problems, which enlarges the scale of the problem model to some extent, and as a result, maintaining a relatively weak GAC on a new model is equivalent to maintaining a strong fPWC on the original model. Currently, the appropriate STR algorithms for FDE are STRFDE and STR2 rather than CT as the CT algorithm may produce memory overflow. In the process of maintaining the consistency algorithm, it is necessary to call each constraint iteratively to perform its consistency algorithm to filter the search space. This process is called constraint propagation. The dynamic submission scheme is a parallel constraint propagation scheme, which can schedule constraint execution propagation algorithms in parallel, and it is particularly effective in large-scale problems. Therefore, this study proposes PSTRFDE for FDE by improving STRFDE and dynamic submission propagation algorithms. PSTRFDE can be embedded into the dynamic submission scheme to further improve the efficiency of constraint problem solving. Extensive experiments indicate that PSTRFDE can reduce the used memory compared with CT and STRbit, and compared with the results of STRFDE and STR2, the efficiency of PSTRFDE can be further improved. The research demonstrates that PSTRFDE is the most efficient filtering algorithm for FDE at present.

Key words: constraint programming (CP); parallel constraint propagation; consistency algorithms; simple table reduction (STR) algorithm

约束规划 (constraint programming, CP) 是人工智能领域的研究方向之一[1], 它已成功应用于时间规划[2,3]、调度[4]和配置[5]等多个领域. 约束规划主要用于表示和求解组合问题. 在问题表示方面, 约束规划提供了丰富的表示语言, 它采用不同类型的约束表示变量之间不同的限制关系, 最终它将组合问题建模成为约束网络 (constraint network, CN). 其中, 扩展约束 (extensional constraint) 或称表约束 (table constraint) 采用枚举的方法列出所有允许或禁止的值组合. 由于其强大的表示能力, 使得其得到最为广泛的研究. 在求解方面, 约束规划提供了一系列的算法求解约束网络, 包括: 相容性推理、问题分解、对称性等剪枝方法, 全局搜索算法、局部搜索算法和实例化变量、值排序启发式方法等. 回溯搜索是求解约束网络的完备方案, 通常, 搜索算法在搜索树的每个节点上对每个约束调用其相容性算法以过滤搜索空间, 称为约束传播. 目前针对表约束的相容性是 (广义) 弧相容 ((generalized) arc consistency, (G)AC), 主流的过滤算法为简单表缩减 (simple table reduction, STR)[6]及改进算法, 本文称为简单表缩减算法簇 (STRs). 其经典算法有 STR2[7]、STR3[8]、STRbit[9]、Compact-Table (CT)[10]等. 除 STRs 外, 还有基于多值决策图 (multi-valued decision diagram, MDD) 的 GAC 算法: MDDc[11]、MDD4[12]、Compact-MDD[13]和 bs-MVDs[14]等.

除了广义弧相容, 高阶相容性 (higher-order consistencies) 也引发了学者们的研究兴趣, 如关系相容 (relation consistency), 最大限定成对相容 (max-restricted pairwise consistency, MaxRPWC)[15], 成对相容 (pairwise consistency, PWC) 和完全成对相容 (full pairwise consistency, fPWC). 其中, fPWC 能同时满足 GAC 与 PWC, 因此具有更强的删值能力, 我们称 fPWC 的相容性强于 GAC. 这意味着, 如果一个问题只满足 GAC 或 PWC 的要求, 它不一定满足 fPWC. 到目前为止, 实现高阶相容性的方法有两种: 一种方法是在传播过程中直接维护某种高阶相容性, 这类算法包括 max-restricted pairwise consistency (maxRPWC)[16]、maxRPWC+[17]、eSTR[18]、PW-AC[19]和 PW-CT[20], 其中最为有效的算法是 PW-CT. 另一种方法是通过某种编码方式, 将原约束网络变型, 使得到原网络上执行较强的相容性等价于在派生网络上行较弱的相容性. 这些编码方案包括 k-交叉编码 (k-interleaved encoding, kIL)[21], 因子编码 (factor encoding, FE)[22]和因子分解编码 (factor-decomposition encoding, FDE)[23]. 其中, FDE 是目前较为有效的编码方法, 在该约束网络上保持 GAC 相当于在原约束网络上保持 fPWC. 其默认的算法是相对较弱的相容算法: STR2. 而采用效率更高的 CT 或 STRbit 可能会带来内存溢出的问题. 为解决该问题, 王等人提出了 STRFDE 算法[24], 它借鉴了 CT 和 STRbit 的优点, 采用分而治之的思想, 为约束网络编码过程中原始和生成出的约束设计不同的过滤算法. 这使得 STRFDE 算法在保持高效的同时的减少了内存消耗.

并行约束规划可以大致分为以下几方面: 并行传播算子和并行传播、搜索空间分割、组合算法、分布式 CSP 及问题分解等[25]. 几十年来, 由于数据同步的问题较为复杂, 有关并行相容性和并行化传播机制的研究较少. 我们在 2017 年提出一种基于 GPU 的弧相容算法该算法改进了 AC4 算法[26], 用于解决二元约束网络. pf_{all} 算法[27]用于

求解二元约束问题, 它采用进程作为执行载体, 并发地运行多个相容性 (如 GAC 和其他相容性算法). 若发生删值或不相容事件, 则将这些信息共享给其他进程. 在不增加额外时间开销的情况下, 获得了更强的剪枝能力. Rolf 和 Kuchcinski 提出了一种带阻塞性质的并行传播方案[28,29]. 并行传播 (推理) 与回溯搜索交织在一起, 推理过程是通过唤醒执行传播算子的线程来完成相容性检查. 李等人改进了这种方案, 提出 PSTR 算法[30], 具体提出静态提交 $PSTR^{ss}$ 和动态提交 $PSTR^{ds}$ 两种约束传播方案. 陈等人改进了静态提交方案[31], 进一步地提高了约束传播的效率. 以上并行传播的实验展示了一个令人兴奋的趋势: 当约束网络规模越大时, 并行传播的加速效果越明显. 恰好经 FDE 编码后, 原约束网络规模明显增加, 采用并行传播方案加速 FDE 模型的传播效率, 最终提升其求解效率, 这是本文工作的一个动机.

基于并行传播模式, 本文改进了 STRFDE 算法, 提出并行算法——PSTRFDE: 我们分别为 FDE 的平凡约束和附加约束提出 $PSTRFDE^{ori}$ 和 $PSTRFDE^{add}$ 算法, 并改进动态提交约束传播方案使其与 PSTRFDE 结合, 提高了 FDE 约束传播效率. 我们在主要的 CP 问题实例集上测试了并行算法, 实验表明, PSTRFDE 在过滤能力和运行时间之间取得了很好的平衡, 极大地提高了 FDE 的求解效率, 足以成为 FDE 上默认的约束传播方案.

1 背景知识

约束网络是一个三元组 $P=(X, D, C)$, 其中 $X=\{x_1, x_2, \ldots, x_n\}$ 是 n 个变量的集合; D 是变量论域的全集, 即 $D=\cup_{x\in X} D(x)$, $D(x)$ 是变量 x 的论域, 设 d 为论域集合的模, 即 $d=|D|$; $C=\{c_1, c_2, \ldots, c_e\}$ 是 e 个约束的集合. 每个约束 $c\in C$ 包含变量辖域 $scp(c)$ 与元组集合含一个允许每个变量取值组合的集合 $rel(c)$, 称为约束的元组集. 其中, 辖域 $scp(c)=\{x_1, x_2, ..., x_r\}$ 是约束限制的变量组成的集合; 相应的, 由所有限制变量 x 的约束组成的集合表示为 $srb(x)=\{c|c\in C, x\in scp(c)\}$; 元组集 $rel(c)\subset\{D(x_1)\times D(x_2)\times\ldots\times D(x_r)\}$ 存储有效元组, 可以通过索引记录元组 τ 在变量 x_i 的取值, 即 $\tau[x_i]=a_i$, 一个元组 $\tau\in rel(c)$, $\tau=(a_1, a_2, ..., a_r)$ 是有效的, 当且仅当对于所有的变量值 $a_i\in\tau$, 都是有效的 $a_i\in D(x_i)$, 则 τ 称为变量值 (a_i, x_i) 在约束 c 上的一个支持 (support). 其中可以通过索引记录元组 τ 在变量 x_i 的取值, 即 $\tau[x_i]=a_i$. 若 $\tau_i=rel(c_i)$, $\tau_j=rel(c_j)$, 且两个约束的相交变量所对应的变量值都相同, 即, 对于所有的 $x\in scp(c_i)\cap scp(c_j)$, 使得 $\tau_i[x]=\tau_j[x_j]$, 那么我们称 τ_i 和 τ_j 互称为 PW 支持 (PW-support). 我们称 c_i 和 c_j 平凡的相交, 当且仅当 $|scp(c_i)\cap scp(c_j)|>1$.

定义 1. (广义) 弧相容算法 ((G)AC). 给定约束网络 $P=(X, D, C)$, 其中,

- 对于变量 x_i 一个值 $a\in D(x_i)$ 是关于约束 $c\in C$ (广义) 弧相容的, 当且仅当存在一个满足 c 的有效元组 τ 使得 $\tau[x_i]=a$, 这样的元组 τ 被称为 c 上关于 (x_i, a) 的一个支持.
- 对于变量 x_i 是 (广义) 弧相容的, 当且仅当所有在其论域 $D(x_i)$ 的值都是 (广义) 弧相容的.
- 对于约束网络 P 是 (广义) 弧相容的, 当且仅当 P 中所有的变量都是 (广义) 弧相容的.

该定义在二元约束网络上称为弧相容, 而在多元约束网络上则泛化为 (广义) 弧相容. 若约束网络 P 满足 (广义) 弧相容, 则称 P 是 (G)AC 相容的 ((G)AC-consistent), 否则是 (G)AC 不相容的 ((G)AC-inconsistent).

定义 2. 成对相容 (pairwise consistency, PWC).

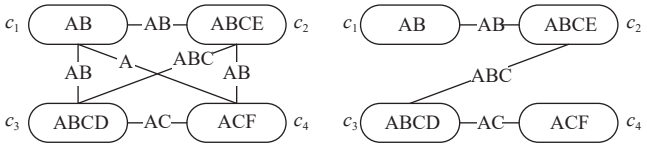
- 约束 c 的一个元组 τ 是成对相容的 (PWC), 当且仅当对于任意其他约束 c_j 至少存在一个元组 $\tau_j\in rel(c_j)$ 是 τ 的 PW 支持.
- 一个约束 c 是成对相容的, 当且仅当所有 $rel(c)$ 中的元组都是成对相容的.
- 约束网络 P 是成对相容的, 当且仅当所有的约束都是成对相容的.

定义 3. 完全成对相容 (full pairwise consistency, fPWC). 约束网络是完全成对相容的 (fPWC), 当且仅当它是 GAC 且 PWC.

定义 4. 对偶图 (dual graph). 约束网络 P 的对偶图是一个顶点表示约束的图, 连接两个顶点的边表示两个点对应约束辖域变量的非空交集.

例 1: 如图 1(a) 所示, 对偶图的顶点表示约束网络的约束, 边表示两个约束的公共变量, 每条边的权值表示相同的变量. 图 1(b) 所示的极小对偶图是对偶图的简化. 若两个顶点之间最终存在一条路径, 使得路径中的每个顶

点都出现相同的变量 (例如, c_1 和 c_4 之间的边是一条冗余的边), 那么两个顶点之间的一条边是冗余的[21−23]. 通过消除对偶图的所有冗余边, 形成极小对偶图 (minimal dual graph).



(a) Dual graph (b) Minimal dual graph of (a)

图 1 对偶图和极小对偶图

一个对偶图可以有多个极小对偶图. 对偶图及其极小对偶图具有相同的表示能力, 因此极小对偶图的选择对正确性没有影响. 以往的研究工作表明, 使用极小对偶图比使用对偶图[20]的效率有很大的提高. 因此, 本文实现的所有 fPWC 算法都是基于极小对偶图的.

1.1 因子分解编码 FDE

因子分解编码是对约束网络 P 的一种变型, 它将约束之间公共变量的码表重新编码为新的变量和约束, 是为因子变量和附加约束, 原约束和原变量称为平凡约束和平凡变量, 平凡变量中的平凡约束被其对应的因子变量取代, 从而生成新的约束网络 P'. 故原约束网络经 FDE 后生成平凡变量、平凡约束、因子变量和附加约束. 为了更好地说明本文算法, 下面给出一些定义.

定义 5. 平凡变量 (ordinary variable). FDE 后定义域不变的变量称为平凡变量.

定义 6. 因子变量 (factor variable). 将约束之间公共变量的码表重新编码形成的新变量称为因子变量.

定义 7. 平凡约束 (ordinary constraint). 用相应的因子变量代替一些平凡变量的约束称为平凡约束.

定义 8. 附加约束 (additional constraint). 由因子变量及其相关平凡变量组成的新约束为附加约束.

例 2: 考虑如图 2(a) 所示的约束网络 P=(X, D, C), 其中 X = {x, y, u, v, w}, $D(x)$ = $D(y)$ = {0, 1}, $D(u)$ = $D(v)$ = $D(w)$ = {0}, C = {c_1, c_2, c_3}. 图 2(b) 展示了 FDE 编码构成的新约束网络 P'=(X', D', C'), 其中平凡变量为 x, y, u, v, w, 因子变量为 f, $D(f)$={0, 1, 2, 3}; 新约束集合为 C' = {c'_1, c'_2, c'_3, c_A}, 其中平凡约束为 c'_1, c'_2, c'_3, 附加约束为 c_A.

c_1

x	y	u
0	0	0
0	1	0
1	1	0

c_2

x	y	v
0	0	0
0	1	0
1	0	0

c_3

x	y	w
0	0	0
1	0	0
1	1	0

(a) Original CN

c'_1

u	f
0	0
0	1
0	2

c'_2

v	f
0	0
0	1
0	3

c'_3

w	f
0	0
0	3
0	2

c_A

x	y	f
0	0	0
0	1	1
1	1	2
1	0	3

(b) FDE of (a)

图 2 平凡约束网络及其 FDE

在对约束网络 P 进行 FDE 后, 在新生成的网络上维持 GAC 相当于在原网络上维持 fPWC. 一般而言, 维持 GAC 的时间效率成本和剪枝能力相对较低. 也就是说, 经 FDE 后我们可以在新约束网络以低时间成本维持一种强剪枝能力的相容性. 可以发现, 因子变量论域的模明显大于平凡变量, 且附加约束中因子变量值是不重复的. 根据以上这些新的特点, 使得采用分而治之的思想处理不同种类的变量和约束是合理的. 在搜索过程中, 每个约束都是相对独立的, 因为它们有自己的特点. 它们由相同的变量连接在一起. 我们只需要将变量的变化传播到相应的约束条件. 从而可以根据约束的特点选择不同的约束处理方法来求解实例, 提高求解效率.

1.2 约束传播调度方案

对于主流面向对象的约束求解器, 约束会以传播算子类 (class propagator) 的形式建模. 传播算子与其调度程序的接口是 propagate() 方法, 这是一个虚方法 (abstract method) 所有传播算子的子类都需要重写该方法. 传播调度程序通过调用该方法执行传播算子. 传播调度方案是一个迭代地执行多个传播算子 (约束) 以减少变量域的过程.

图 3 显示了 3 个约束传播方案. 第 1 种是经典的串行传播方案 (如图 3(a) 所示), 它被广泛用于 CP 求解器中. 它使用队列或堆来串行地执行传播算子. 在文献 [30] 中, 我们改进了 Rolf 的并行传播方案方法[28,29], 提出了两种并行传播方案: 静态和动态提交 (如图 3(b) 和图 3(c) 所示). 这两种方案都使用线程池作为调度工具, 在不改变当前计算设备, 利用主流多核 CPU 加速了约束传播的效率. 如图 3(a) 所示, 约束传播的传统串行调度方案维护传播队列 Q 来存储修改过的变量. Q 的底层数据结构可以是数组也可以是堆. 调度程序迭代地从 Q 中弹出并调用所有修改过的约束 c (即 c 的任意所辖变量 $x \in scp(c)$ 的论域发生了修改) 的 propagate() 方法. 该方法根据 c 所剩的有效元组来删减变量值, 并且检测不相容, 再将修改过的元素重新放入队列 Q 中.

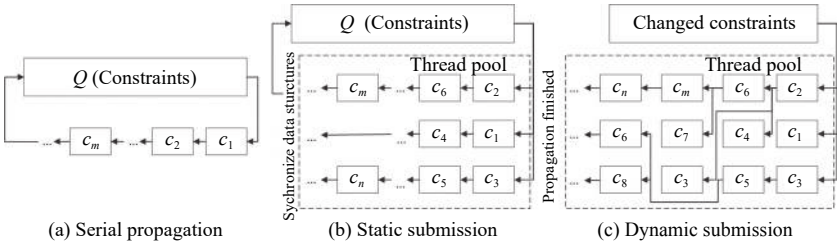


图 3 多种传播方案示意图

在设计并行传播算法时, 为了确保数据的正确性并减少设计的难度, PSTR 在一个传播算子内按顺序在内部执行, 多个传播算子被安排并行执行. 在并行传播的过程中, 传播算子 (约束) c 的变量 x 的论域删减事件会先缓存到变量快照 (snapshot) 中即 $\Sigma c(x)$, 并在该传播算子完成时提交给相应的变量的全局论域 $D(x)$ 中, 即删除事务并行执行.

静态提交方案 (见图 3(b)) 引入了线程栅栏, 传播算子会立即提交到线程池. 已完成的传播算子进入线程栅栏等待其他传播工作完成. 当所有传播算子到达线程栅栏时, 即抵达“synchronize data structures”处, 调度程序函数总结传播算子的执行信息, 然后选择需要进一步传播的传播算子重新提交到线程池中执行. 动态提交方案 (见图 3(c)) 删除了线程栅栏, 因此传播算子可以动态地将其他的传播算子提交到线程池中. 算法 1 是动态提交算法的伪代码, 其上层的搜索算法的调用该方法并传入修改后的变量集合 X_{evt} 以开启并行约束传播. *pool* 是一个全局线程池, 线程池中的工作线程数称为线程池的容量, 也称为线程池的并行度. 算法 1 中提到的线程池有两种常见的方法如下.

- *pool.submit*(c): 将传播算子 c 提交到线程池 *pool* 中.
- *pool.awaitQuiescence*(): 等待线程池 *pool* 的所有线程完成工作.

布尔字段 consistent 初始设置为 true (第 1 行), 当检测到不相容发生时, 传播算子都会将其设置为 false, 整个约束传播过程会停止, 向上层求解算法返回不相容. 搜索算法需要回溯到失败之前的状态重新开启新的搜索方向. 若 consistent 为 true, 它将待传播的传播算子提交到池中, 等待并发调度执行. 为了确保并行算法的正确性, 每个并行传播算子一次只能被一个线程执行, 该算法为每个传播算子维护一个原子计数器 *numReq*, 以记录其他传播算子提交的执行请求. 若主流编程语言通常会提供原子类型来解决数据竞争的问题. 当多个线程借助它不阻塞地访问同一数据, 并保证访问的正确性. 对于原子整型 a 的常用方法如下.

- *a.set**(b): 原子地将 a 设置为给定值 b.
- *a.Try&Set*(*expected*, *updated*): 若 *a*=*expected*, 则原子地将 a 设置为给定值 *update* 并返回 true, 否则不改变 a 的值并返回 false.
- *a.get&Inc*(): 原子地将 a 的值加 1 并返回原始值.

算法 1. Dynamic Submission Propagation.

Data: X_{evt}: set of modified variables; *pool*: global thread pool.

1. consistent ← true

```
2. foreach x ∈ X_evt do
3.     foreach propagator c ∈ srb(x) do
4.         if ¬consistent then
5.             return
6.         if c.numReq.get&Inc*() = 0 then
7.             pool.submit(c)
8. pool.awaitQuiescence()
```

在传播算子 c 被提交到线程池 *pool* 之前 (第 7 行), 需要原子地检查 c 是否已在 *pool* 中, 即调用 *c.numReq.get&Inc**() 方法 (第 6 行), 如果原始值等于 0, 这意味着 c 在此之前没有被提交到 *pool*, 此时可以直接提交. 否则, 这表明 c 已经有其他传播请求, 不需要再次提交 c. 该方法确保 c 一次只能由一个线程执行.

2 并行传播算子

本节我们首先介绍动态并行提交约束传播方案. 进而基于此方案, 为 FDE 网络的附加约束提出其并行传播算法——PSTRFDE^{add}, 为平凡约束网络提出相应的算法——PSTRFDE^{ori}, 并在最后分析整个算法的时空复杂度.

2.1 并行传播的动态提交方案

除了并行化调度算法之外, 传统串行传播算子也需要改造为并行传播算子, 以确保算法正确性. 因此, 我们需要向传播算子添加 3 个公共附加方法 (如算法 2 所示), 注意这 3 个方法是后面所提出的 PSTRFDE^{add} 和 PSTRFDE^{ori} 共用的方法. 其中, 方法 threadEntrance() 是整个传播算子的入口, 当线程池调度传播算子时调用该方法. threadEntrance() 函数的主体是一个循环, 它确保连续过滤变量的快照, 直到 *numReq* 成功设置为 0 为止. 传播算子的 *numReq* 设置为 1 表示线程开始处理这个传播算子. propagate() 是各种过滤算法的核心函数. 这里值得注意的是, 对于串行算法, 它直接过滤全局论域 D; 而对于并行算法, 这个函数过滤缓存在传播算子中的快照 Σ. 执行过滤算法后, 将调用 submitSnapshot() 方法. 它将变量快照 Σ 提交给全局域 D. 如果全局域在提交过程中发生变化, 传播算子需要更新 last 快照 Σ' (第 14 行), last 快照用于记录上一次传播时论域的快照. 并将该变量添加到 Y_{evt} 中 (第 15 行). submitPropagators()方法是动态提交方案的核心功能, 传播算子调用方法向线程池 *pool* 中提交其他传播算子. 在提交之前, 我们需要检查传播算子的 *numReq* (第 20 行), 以避免重复提交.

算法 2. Additional methods for PSTRFDE propagators.

```
Data: Y_evt: set of domain changed Variables; pool: global thread pool; numReq: atomic integer; Σ: array of local
intermediate domains; Σ′: array of last intermediate domains
1.  Method ThreadEntrance() do
2.  repeat
3.      numReq.set*(1)
4.      propagate()
5.      submitSnapshot()
6.      submitPropagators()
7.  until numReq.TryAndSet*(1, 0) ∨ ¬globalConsistent
8.  Method propagate() do
9.      ... /* same as PSTRFDE propagators */
10. Method submitSnapshot() do
11.  for all x ∈ scp do
```

```
12.      atomic submit Σ(x) to its global domain D(x)
13.      if D(x) = Σ(x) then
             /* update last snapshot Σ′(x) for next propagation */
14.        Σ′(x) ← Σ(x)
15.        Y_evt ← Y_evt ∪ x
16.    Method submitPropagators() do
17.    for all x ∈ Y_evt and globalConsistent do
18.      for all propagator c ∈ srb(x) do
19.        if c ≠ this then
20.          if c.numReq.getAndInc*() = 0 then
21.            pool.submit(c)
```

2.2 附加约束的并行算子 PSTRFDE^{add}

基于经典的 CT 算法、STRFDE^{add} 及并行算法 PSTRs, 我们首先为附加约束提出并行过滤算法 PSTRFDE^{add} (见算法 3), 其省略部分与 CT 算法相同, 并采用动态提交方案进行约束传播. 因此 CT 和 PSTRs 算法的一些字段 (数据结构) 被保留了下来.

数据结构 RSparseBitSet: CT、PW-CT、STRFDE 等一系列主流算法所使用的数据结构, 它结合了稀疏集 (sparse set) 便于回溯而位集 (bit set) 压缩存储的优点. 存储数据规模较大且需要回溯的字段, 本算法用其来表示元组有效性的字段 *currTab* 和因子变量的论域.

变量的快照 Σ 和 last 快照 Σ': 当前传播算子 c 缓存的变量 x 的快照为 $\Sigma_c(x)$, 算法 3 中简写为 $\Sigma(x)$, 用于缓存变量 x 论域的修改情况. 在并行传播过程中, 变量 x 可能在不同的传播算子中存储不同的快照信息, 在该传播算子执行结束之后, 它们都会提交到全局论域中. 而 last 快照 Σ'记录在上一次传播时的变量快照, 在传播开始时通过与当前快照的比较确定哪些变量论域发生了改变. 记录 Σ'的目的是减少冗余传播.

与 CT 算法相似, PSTRFDE^{add} 同样使用了 *scp*、S_{val}、S_{sup}、*supports* 和 *residue* 字段. 其中, *scp* 为附加约束的限制的变量集合, S_{val} 记录修改的变量集合, S_{sup} 记录未赋值的变量集合, 二者用于缩小过滤元组集和论域的迭代范围; *supports*[x][a] 采用 bit set 为每个平凡变量值 (x, a) 记录该值在元组集上的支持. *residue*[x][a] 为每个平凡变量值 (x, a) 记录其在 *supports*[x][a] 剩余有效支持的索引. 而对于因子变量 μ, 其论域大小$|D(\mu)|$等于当前约束的元组大小$|rel(c)|$, 若像 CT 那样为其每个变量值 a 建立 *support*[μ][a] 数据字段可能会导致内存溢出. 为了解决这个问题, 我们不记录因子变量 μ 的 *support* 信息, 只对平凡变量构造 *support*. 因此, *supports* 和 *residue* 的大小为$|scp|-1$, 这是因为本算法没有为因子变量建立以上两个字段, 否则可能因其过大的论域, 引发内存溢出.

算法 3. class PSTRFDE^{add} Propagator.

Data: *scp*: array of ordinary variables; *currTab*: *RSparseBitSet*; S_{val}, S_{sup}: array of variables; Σ, Σ':snapshot and last snapshot for the domain of scope variables; μ: factor variable of this constraint

```
1.    Method initial() do
2.      remove all elements in S_val and S_sup
3.      for all x ∈ scp and consistent do
          /* update S_val according to snapshot Σ and Σ′ */
4.        Σ(x) ← x.getBitDom()
5.        if Σ′(x) ≠ Σ(x) then
6.          Σ′(x) ← Σ(x)
7.          S_val ← S_val ∪ {x}
```

```
8.    S_sup ← {x ∈ scp : |D(x)| > 1}
9.  Method updateTable() do
10.  if μ.isChanged() then
11.    currTab.intersectWords(μ.getBitDom())
12.  for all x ∈ S_val do
13.    ... /* same as CT, update table according to Σ[x] for PSTRFDE^add*/
14.    if inconsistent is detected then
15.      globalConsistent ← false /* Notify other propagators to stop running */
16.      return false
17.    else
18.      return true
19.  Method filterDomain() do
20.    ... /* same as CT, filter Σ(x) for STRFDE^add */
21.  Method propagate() do
22.  initial()
23.  if ¬ updateTable() then
24.    return false
25.  filterDomain()
26.  if currTable.isChanged() then
27.    μ.removeValues(currTab.getWords())
```

与 CT 算法相近, initial() 方法将初始化变量集 S_{sup} 和 S_{val}. 此外, 它还更新快照 Σ 和 Σ'来缓存全局域 *D*.updateTable() 方法使用约束中限制的变量来更新当前的 *currTable*. 由于因子变量 μ 的论域 $D(\mu)$ 较大, PSTRFDE 算法采用 *RSparseBitSet* 类表示 $D(\mu)$, 如果 $D(\mu)$ 已被修改 (第 10 行), 那么我们需要使用它来更新 *currTable*. 这是通过调用第 11 行的 intersectWords() 方法来实现的. 最后, 根据平凡变量更新 *currTable*.filterDomain() 方法来过滤平凡变量的快照. 如果快照为空, 则会发生回溯. propagate() 方法是 PSTRFDE^{add} 算法的入口. 在 25 行调用 filterDomain() 方法可以修改平凡变量, 但此时因子变量 μ 的论域没有被过滤, 若 *currTable* 发生改变, μ.removeValues() 方法用于过滤因子变量 μ 的论域 (27 行). 执行完 propagate() 后, 它将返回到算法 2 的第 4 行. 余下的并行传播过程已在算法 2 提到.

2.3 平凡约束的并行传播算子

对于平凡约束, 变量集 *scp* 是由若干平凡变量与若干因子变量组成的, 所以不能再使用处理附加约束的方法. 为了解决这个问题, 我们在算法 4 提出了一种基于 STRbit 和 STRFDE 的并行传播算子——PSTRFDE^{ori}.

算法 4. class PSTRFDE^{ori} Propagator

```
Data: scp: array of variables; del: del; bitSup: bitSup; val: val; res: residues
1.  Method deleteInvalidTuple() do
2.    for all x ∈ scp do
3.      Σ(x) ← x.getBitDom()
4.      update del[x] according Σ and Σ'
5.      for all a ∈ del[x] do:
6.        for i ← bitSup[x, a].size down to 0 do
7.          θ ← bitSup[x, a][i].ts
```

```
8.              u ← bitSup[x, a][i].mask & val[θ]
9.              if u ≠ 0 then
10.                 val[θ] ← (¬u) & val[θ]
11.         del[x].clear()
12.     Method searchSupport() do
13.     for all x ∈ scp and globalConsistent = true do
14.       for all a ∈ D[x] do
15.         now ← res[x, a]
16.         θ ← bitSup[x, a][now].ts
17.         if bitSup[x, a][now].mask & val[θ] = 0 then
18.           now ← bitSup[x, a].size − 1
19.           θ ← bitSup[x, a][now].ts
20.           repeat
21.             now ← now − 1
22.             if now = −1 then
23.               Σ(x) ← Σ(x) \ {a} /* remove a from Σ(x) */
24.               if Σ(x) = ∅ then
                      /* Notify other propagators to stop running */
25.                   globalConsisntent ← false
26.                   return false
27.               Break
28.               θ ← bitSup[x, a][now].ts
29.               until bitSup[x, a][now].mask & val[θ] = 0
30.               if now ≠ −1 then
31.                 res[x, a] ← now
32.     Method propagate() do
33.     deleteInvalidTuple()
34.     searchSupport()
```

算法 4 给出了并行传播算子的伪代码 PSTRFDE^{ori}. 如第 2.2 节所言, 如果平凡约束的变量集 *scp* 中含有多个因子变量, 则不能使用 CT 算法的 *support* 字段, 否则会产生内存溢出和情况. PSTRFDE^{ori} 不再使用 *last* 和 *restoreL* 这两个字段, 其余字段与 STRbit 基本相同, 算法 4 删除它们是由于其维护信息是高度复杂的且不重要的. 在 PSTRFDE^{ori} 中, 由于 *bitSup* 只存储非零的位支持, 每个值 (x, a) 的 *bitSup* 长度可能不同, 这是在算法初始化时确定的. 对于因子变量, 它的 *bitSup* 的长度非常短, 迭代 *bitSup* 比平凡变量快. 在该算法中, 我们使用 *residues* 来储存 (x, a) 的支持的索引, 这将减少为 (x, a) 寻找支持的时间.

deleteInvalidTuple() 方法首先获取快照并获取当前已被删除的值集合 *del*. 然后, 它使用 *del* 和 *bitSup* 来更新当前表 *val* (第 5–11 行). 如果 *u* 不等于 0 (第 9 行), 意味着当前表中有一些支持值 (x, a) 的无效元组. 因此, 算法 4 在第 10 行更新当前表. 方法 searchSupport() 用于查找对每个变量值的支持. 如果不支持 (第 22 行), 则从快照中删除该值. 如果论域为空, 就会发生回溯. propagate() 方法是 PSTRFDE^{ori} 的入口.

基于伪代码中使用的数据结构, 下面给出了 PSTRFDE 的空间复杂度和时间复杂度的证明. 为了计算复杂度, 我们假设约束网络 P 中的约束数量为 e, 约束中涉及的变量数量为 $r = r_o + r_f$, 其中 r_o 和 r_f 分别为平凡变量和因子变量的数量. 平凡变量论域的大小为 d_o, 因子变量论域的大小为 d_f. 约束中的元组数目为 t, 计算机上一个 word 的

大小为 w (例如 $w = 64$).

2.4 复杂度讨论

命题 1. PSTRFDE^{add} 的最差时间空间复杂度是 $O(r_o d_o t/w)$.

证明: PSTRFDE^{add} 使用 *currTab*, *supports*, *residues*, Σ 和 Σ'这 5 种数据结构. 它们的复杂度分别为 $O(3t/w)$, $O(r_o d_o t/w)$, $O(r_o d_o)$, $O((r_o d_o + d_f)/w)$ 与 $O((r_o d_o + d_f)/w)$. S_{val} 和 S_{sup} 的复杂度是 $O(r_o)$, 总计 $O(3t/w + r_o d_o t/w + 2r_o + r_o d_o + 2(r_o d_o + d_f)/w)$ 即 $O(r_o d_o t/w)$. 证毕.

CT 中最大的数据结构为 *supports* 的复杂度为 $O(r_o d_o t/w + d_f t/w)$, 所以在附加约束条件中 CT 的空间复杂度为 $O(r_o d_o t/w + r_f d_f t/w)$. 大多数问题的 d_f 比 d_o 大得多, 因此使用 CT 来解决 FDE 问题可能会导致内存溢出问题.

命题 2. PSTRFDE^{ori} 的最差时间空间复杂度为 $O(r_o d_o t/w + r_f d_f t/w)$.

证明: PSTRFDE^{ori} 使用 *bitSup*, *val*, *del*, Σ 和 Σ'. *val* 和 *del* 的空间复杂度分别为 $O(t/w)$ 和 $O(r_o d_o + r_f d_f)$. 对于 *bitSup*, 由于因子变量与平凡变量存在显著差异, 我们在此单独计算. 因子变量的论域较大, 因此 *bitSup*[x][a] 的长度远小于 t/w. 虽然因子变量最坏的空间复杂度为 $O(r_f d_f t/w)$, 但还远远达不到. 对于平凡变量, 最差的空间复杂度是 $O(r_o d_o t/w)$. 因此, 总空间复杂度为 $O(r_o d_o t/w + r_f d_f t/w + r_o d_o + r_f d_f + t + 2(r_o d_o + r_f d_f)/w) = O(r_o d_o t/w + r_f d_f t/w)$. 证毕.

可以发现, PSTRFDE^{ori} 的空间复杂度与 CT 在附加约束条件下相同. 但实际上, $d_f t/w$ 是一个固定值. 附加约束的 CT 算法的空间复杂度要大于它, 而 PSTRFDE^{ori} 的空间复杂度远不能达到, 这已经被实验证明. 与 STRbit 相比, PSTRFDE^{ori} 有取消了两个极耗空间的数据结构 *last* 和 *restoreL*. 其中, *last* 的空间复杂度为 $O(r_o d_o + r_f d_f)$, 但 *last* 将被 restoreL 存储在搜索树的每一层. 总体而言, PSTRFDE^{ori} 的实际空间复杂度小于 CT 和 STRbit.

命题 3. PSTRFDE 的动态提交方案的最坏时间复杂度为 $O(erd(r_o d_o + r_f d_f)tS/wp)$.

证明: 在文献 [30] 中, 我们给出动态提交方案的时间复杂度为 $O(erdTS/p)$, 其中 T 为底层传播算子的时间复杂度, 我们算法的传播算子是 PSTRFDE^{ori} 和 PSTRFDE^{add}. 其中 PSTRFDE^{ori} 时间复杂度更高. 所以整个算法的时间复杂度是 $O(erd(r_o d_o + r_f d_f)tS/wp)$. 证毕.

3 实验结果

我们从 XCSP3 的在线测试实例数据库 (http://xcsp.org/series) 中选择了 11 组多元约束满足问题作为测试用例集, 分别为 aim 系列、dubois、renault 系列、ModelRB 系列 (rand 系列) 和旅行商系列 (TSP 系列), 总实例数超过 300 个. 我们实现并比较了 7 组 STR 类算法: CT、PW-CT、CT+FDE、STRbit+FDE、STR2+FDE、STRFDE 和 PSTRFDE. 其中 CT 算法是最先进的 GAC 算法, 其他算法相当于在原问题上执行 fPWC 相容性. PSTRFDE 算法的并行度 (即线程池大小) 在“@”后给出, 如 PSTRFDE@2 表示该算法在并行度为 2 的线程池下运行. 引导回溯搜索的赋值启发式则采用 dom/ddeg 变量排序启发式[32]和最小值排序启发式. 所有算法均基于 Scala 2.12 和 Java 11 编写的求解器上实现, 所有的代码均已公开 (https://github.com/cpresearchers/ScalaCP/tree/wzlz/src/main/scala/cpscala/TSolver/Model/Constraint/DSPFDECons-traint). 整个实验在 6 核心 12 线程, 主频为 3.20 GHz 的英特尔酷睿 i7 处理器上运行, 操作系统为 Windows 10, 内存为 8 GB. 时间限制设置为 1 200 s, 没有在限制时间内求解的实例称为超时实例. 我们剔除了那些在所有算法上都超时的测试用例.

表 1 展示了相关算 STR 法求解不同实例集的实验结果. 对于每组实例集, 我们给出了实例个数 (#=), 以及其以 s 为单位的平均求解时间 (cpu) 和搜索节点数 (#n), 其中超时例的搜索节点数不计入#n, 超时比率 (TO) 实例和平均内存使用 (MEM), 单位为 MB. 发生内存溢出的实例集被标记为“MO”. 每组实例集最优和次优的数据分别用粗体和下划线表示. 表格的最后一行表示每种算法解出总共的实例总数.

我们可以观察到, CT 在 2 组实例上是最快的, 但是算法对于某些 SAT 实例 (如 aim-200 和 dubois), 它们表约束的元组数较少, 会消耗更多的时间. STRFDE 在 1 组实例上是最快的; PSTRFDE 在 5 组实例列是最快的. 此外, 如果只比较 fPWC 算法 (即排除 CT 算法), PSTRFDE 算法在 8 组实例集上领先. 通过对数据的分析, CT 检测当前约束网络是否满足 GAC 所需的时间非常短, 而其他算法检查 fPWC 所需要的时间相对较长. fPWC 算法通常比

GAC 算法效率低. 这是因为在某些情况下维护 fPWC 将执行额外的检查以删除更多元组. 与 PSTRFDE 相比, PW-CT 在某些 SAT 实例中具有优势. 也就是说, 当约束含有较少的元组时 (即|*rel*(*c*)|相对较小), PW-CT 的效率相对较高. 但 PW-CT 在约束涉及大量元组时效率低于 STRFDE 和 PSTRFDE. 因此, 当约束所包含元组数较多时 (如 rand 系列), PSTRFDE 的效率远远优于 PW-CT. 与 CT+FDE 和 STRbit+FDE 相比, PSTRFDE 的效率得到了明显地提高. 但是, 前两种算法的内存占用都太大. 就像 CT 和 STRbit 比 STR2 快一样, PSTRFDE 也比 STR2 有了更大幅度的改进. 总体而言, PSTRFDE 算法的效率是所有 fPWC 算法中最好的.

表 1　相容性过滤算法的求解结果比较

series		CT	PW-CT	FDE						
				CT	STRbit	STR2	STRFDE	PSTRFDE @2	PSTRFDE @4	PSTRFDE @6
aim-50 #=24	cpu	0.04	**0.014**	0.069	0.077	0.068	0.069	0.118	0.136	0.161
	#n	3316	98	3391	3391	3391	3391	3391	3391	3391
	MEM	0.44	5.24	1.98	5.50	0.83	1.17	1.61	1.61	1.61
aim-100 #=24	cpu	113.083	**5.899**	118.612	122.007	120.61	118.028	130.371	133.538	137.531
	#TO (%)	8.33	0	8.33	8.33	8.33	8.33	8.33	8.33	8.33
	MEM	1.41	19.32	6.40	18.42	2.64	3.64	4.01	4.01	4.01
aim-200 #=16	cpu	788.941	**361.227**	546.262	546.768	544.895	545.118	682.798	682.929	683.322
	#TO (%)	81	38	56.25	56.25	56.25	56.25	56.25	56.25	56.25
	MEM	4.11	51.72	12.69	35.95	5.53	8.40	8.90	8.90	8.90
dubois #=13	cpu	718.362	591.833	549.997	562.407	542.834	**53 417**	673.638	713.195	730.943
	#TO (%)	46.15	38.46	38.46	38.46	30.77	30.77	46.15	46.15	46.15
	MEM	0.74	5.31	0.77	1.70	0.46	0.73	1.21	1.21	1.21
renault #=2	cpu	**0.015**	4.001	1.072	0.072	0.072	0.061	0.038 6	0.026	0.026 2
	#n	101	101	101	101	101	101	101	101	101
	MEM	5.17	234.93	1 450.96	307.66	4.67	26.67	35.72	35.72	35.72
modiRenault #=50	cpu	523.617	99.464	68.209	72.066	72.088	66.782	52.34	**52.339**	52.34
	#TO (%)	43	8	4	6	6	4	4	4	4
	MEM	5.75	266.06	1 455.73	378.53	5.71	29.17	38.65	38.65	38.65
rand-3-20 #=50	cpu	24.722	673.287	73.965	7418	142.138	59.276	32.321	23.233	**21.61**
	#TO (%)	0	26	0	0	0	0	0	0	0
	MEM	1.84	38.35	19.53	25.37	1.05	6.88	8.20	8.20	8.20
rand-3-20-fcd #=50	cpu	12.167	421.324	35.546	37.216	67.944	29.708	16.614	12.005	**11.164**
	#TO (%)	0	8	0	0	0	0	0	0	0
	MEM	1.84	38.35	19.53	25.37	1.05	6.88	8.20	8.20	8.20
rand-8-20 #=20	cpu	**3.86**	1 184.849	MO	17.099	17.128	16.614	10.577	7.652	7.101
	#TO (%)	0	95	MO	0	0	0	0	0	0
	MEM	10.56	184.47	MO	280.94	7.50	128.02	136.77	136.77	136.77
rand-10-20 #=20	cpu	0.031	0.662	0.132	0.017	0.024	0.01	0.007	0.006	**0.005**
	#n	830	0	0	0	0	0	0	0	0
	MEM	0.77	16.75	648.98	75.82	1.34	14.9	26.81	26.81	26.81
TSP20~25 #=30	cpu	2419	MO	26.478	36.135	41.696	31.165	17.894	14.571	**14.51**
	#n	52 932.67	MO	52 932.67	52 932.67	52 932.67	52 932.67	52 932.67	52 932.67	52 932.67
	MEM	59.73	MO	59.73	42.9	2.20	24.24	29.81	29.81	29.81
#solved		261	226	284	284	285	**287**	285	285	285

值得注意的是, PW-CT 的节点数量通常比其他算法要少. 这是因为 PW-CT 使用在平凡的约束网络上执行了更强的相容性——fPWC, 强相容性因其剪枝能力较强, 从而其搜索的节点较少. 但它也执行了额外的 PWC 检查, 增加了时间开销. 若 PWC 相对 GAC 而言并未显著地移除更多的元组, 则 PW-CT 是低效的. 这很好地解释了为什么 PW-CT 在随机实例集 (rand 实例集) 上花费了很长时间. 后 7 种算法的搜索节点数在某些实例集中 (如 aim50

和 TSP20~25) 是相同的, 这是由于它们都是在 FDE 模型上执行 GAC 的算法, 剪枝能力相同, 这意味着它们在不超时的情况下求解这些实例时搜索相同的搜索树. 实际上, 在单位时间内, PSTRFDE 能比 STR2 搜索更多的节点. 另外, 对于一些无解的随机实例 (如 rand-10-20), fPWC 算法能够在预处理阶段直接推理出该问题无解, 因此这些 fPWC 算法的搜索节点数都为 0, 该实例集上执行 fPWC 要比 GAC 快得多. 对于可解的随机实例 (如 rand-8-20), 其每个约束均包含大于 10 000 个元组. 执行 PWC 将花费很长时间来检查这些元组是否为 PWC. 因此, 在这种实例集上执行 GAC 比 PWC 更有效. 通过比较 fPWC 算法的内存使用量, 我们发现 STR2 的内存使用量最低, 但其求解效率远低于 STRFDE 和 PSTRFDE. 尽管 CT 算法是 GAC 中最好的算法, 但它在 FDE 的很多实例上都存在内存不足的问题. 通过对所有算法进行比较, PSTRFDE 算法在求解效率和内存使用之间取得了最佳的平衡.

为了使结果更加明显, 图 4 显示了算法的总体性能, x 轴为求解时间, y 轴为该时间上解出实例的个数, 即单位时间内求解实例总数, 可见处于上方的折线所对应的算法其求解效率更高. 我们可以观察到无论在何种并行度下, PSTRFDE 算法都具有最高的求解效率, 而串行的 STRFDE 的效率接近 CT, 在某些地方甚至优于 CT, 甚至解出更多的实例. 其他串行 fPWC 算法的效率不如 STRFDE 和 CT 算法. 通过观察, 我们注意到串行算法的折线在并行算法的折线之下. 这主要是由于 FDE 模型在原有模型的基础上增加了新的变量和约束, 增大了整个问题的规模. 因此, 并行算法在大规模问题上的性能较好. 文献 [30] 提出将并行度设置为 5 已经可以获得良好的加速. 这里我们观察到将并行度设置为 4 和 6 的折线非常接近, 且都达到了最高的效率, 这也印证了该文献中的观点. 总之, PSTRFDE 可以稳定地提高约束传播的算法效率, 是 FDE 模型上求解效率最高的过滤算法.

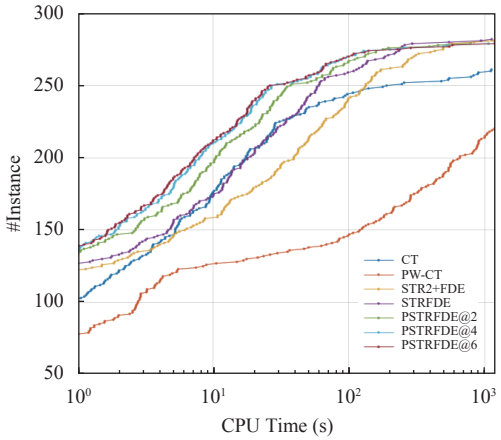


图 4 PSTRFDE 算法与其他算法求解效率的比较

4 总 结

在本文中, 我们提出了一种并行传播算法 PSTRFDE 以提高求解 FDE 中约束传播的效率. 对于 FDE 不同的约束类型, PSTRFDE 采用相适应的方法建模, 改进它们的传播算子, 以提高过滤效率. 进而我们尝试将 PSTRFDE 传播算子嵌入到动态传播方案中, 以进一步提高传播效率. 实验表明, PSTRFDE 减少了时间和内存消耗, 是求解 FDE 的理想方法. 我们的工作充分说明了 PSTRFDE 是目前最先进过滤传播算法的有力竞争者. 未来我们将在两个方向深入开展相关研究工作: 一是提高并行动态提交方案的算法效率, 主要任务是进一步降低变量论域快照的内存占用, 以求解出更大规模的问题实例; 二是提出比 FDE 更为高效的编码方案, 使得我们的并行传播方法能够得到更为广泛的应用.

References:

[1] Lecoutre C. Constraint Networks: Techniques and Algorithms. London: John Wiley & Sons, 2009. 27–38. [doi: 10.1002/9780470611821]

[2] Schutt A, Feydy T, Stuckey PJ. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In: Proc. of the

10th Int'l Conf. on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Yorktown Heights: Springer, 2013. 234–250. [doi: 10.1007/978-3-642-38171-3_16]

[3] Schutt A, Stuckey PJ. Explaining producer/consumer constraints. In: Proc. of the 22nd Int'l Conf. on Principles and Practice of Constraint Programming. Toulouse: Springer, 2016. 438–454. [doi: 10.1007/978-3-319-44953-1_28]

[4] Nouri HE, Driss OB, Ghédira K. A classification schema for the job shop scheduling problem with transportation resources: State-of-the-art review. In: Silhavy R, Senkerik R, Oplatkova ZK, Silhavy P, Prokopova Z, eds. Artificial Intelligence Perspectives in Intelligent Systems. Cham: Springer, 2016. 1–11. [doi: 10.1007/978-3-319-33625-1_1]

[5] Hotz L, Felfernig A, Günter A, Tiihonen J. A short history of configuration technologies. In: Felfernig A, Hotz L, Bagley C, Tiihonen J, eds. Knowledge-based Configuration: From Research to Business Cases. Amsterdam: Morgan Kaufmann, 2014. 9–19. [doi: 10.1016/B978-0-12-415817-7.00002-5]

[6] Ullmann JR. Partition search for non-binary constraint satisfaction. Information Sciences, 2007, 177(18): 3639–3678. [doi: 10.1016/j.ins.2007.03.030]

[7] Lecoutre C. STR2: Optimized simple tabular reduction for table constraints. Constraints, 2011, 16(4): 341–371. [doi: 10.1007/s10601-011-9107-6]

[8] Lecoutre C, Likitvivatanavong C, Yap RHC. STR3: A path-optimal filtering algorithm for table constraints. Artificial Intelligence, 2015, 220: 1–27. [doi: 10.1016/j.artint.2014.12.002]

[9] Wang RW, Xia W, Yap RHC, Li ZS. Optimizing simple tabular reduction with a bitwise representation. In: Proc. of the 25th Int'l Joint Conf. on Artificial Intelligence. New York: IJCAI/AAAI Press, 2016. 787–793. [doi: 10.5555/3060621.3060731]

[10] Demeulenaere J, Hartert R, Lecoutre C, Perez G, Perron L, Régin JC, Schaus P. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In: Proc. of the 22nd Int'l Conf. on Principles and Practice of Constraint Programming. Toulouse: Springer, 2016. 207–223. [doi: 10.1007/978-3-319-44953-1_14]

[11] Cheng KCK, Yap RHC. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints, 2010, 15(2): 265–304. [doi: 10.1007/s10601-009-9087-y]

[12] Perez G, Régin JC. Improving GAC-4 for table and MDD constraints. In: Proc. of the 20th Int'l Conf. on Principles and Practice of Constraint Programming. Lyon: Springer, 2014. 606–621. [doi: 10.1007/978-3-319-10428-7_44]

[13] Verhaeghe H, Lecoutre C, Schaus P. Compact-MDD: Efficiently filtering (s)MDD constraints with reversible sparse bit-sets. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence. Stockholm: AAAI, 2018. 1383–1389. [doi: 10.5555/3304415.3304611]

[14] Verhaeghe H, Lecoutre C, Schaus P. Extending compact-diagram to basic smart multi-valued variable diagrams. In: Proc. of the 16th Int'l Conf. on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Thessaloniki: Springer, 2019. 581–598. [doi: 10.1007/978-3-030-19212-9_39]

[15] Janssen P, Jegou P, Nouguier B, Vilarem MC. A filtering process for general constraint-satisfaction problems: Achieving pairwise-consistency using an associated binary representation. In: Proc. of the 1989 IEEE Int'l Workshop on Tools for Artificial Intelligence. Fairfax: IEEE, 1989. 420–427. [doi: 10.1109/TAI.1989.65349]

[16] Bessiere C, Stergiou K, Walsh T. Domain filtering consistencies for non-binary constraints. Artificial Intelligence, 2008, 172(6-7): 800–822. [doi: 10.1016/j.artint.2007.10.016]

[17] Paparrizou A, Stergiou K. An efficient higher-order consistency algorithm for table constraints. In: Proc. of the 26th AAAI Conf. on Artificial Intelligence. Toronto: AAAI, 2012. 535–541. [doi: 10.5555/2900728.2900805]

[18] Lecoutre C, Paparrizou A, Stergiou K. Extending STR to a higher-order consistency. In: Proc. of the 27th AAAI Conf. on Artificial Intelligence. Bellevue: AAAI, 2013. 576–582. [doi: 10.5555/2891460.2891540]

[19] Samaras N, Stergiou K. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. Journal of Artificial Intelligence Research, 2005, 24: 641–684. [doi: 10.1613/jair.1776]

[20] Schneider A, Choueiry BY. PW-CT: Extending compact-table to enforce pairwise consistency on table constraints. In: Proc. of the 24th Int'l Conf. on Principles and Practice of Constraint Programming. Lille: Springer, 2018. 345–361. [doi: 10.1007/978-3-319-98334-9_23]

[21] Mairy JB, Deville Y, Lecoutre C. Domain k-wise consistency made as simple as generalized arc consistency. In: Proc. of the 11th Int'l Conf. on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Cork: Springer, 2014. 235–250. [doi: 10.1007/978-3-319-07046-9_17]

[22] Likitvivatanavong C, Xia W, Yap RHC. Higher-order consistencies through GAC on factor variables. In: Proc. of the 20th Int'l Conf. on Principles and Practice of Constraint Programming. Lyon: Springer, 2014. 497–513. [doi: 10.1007/978-3-319-10428-7_37]

[23] Likitvivatanavong C, Xia W, Yap RHC. Decomposition of the factor encoding for CSPs. In: Proc. of the 24th Int'l Conf. on Artificial Intelligence. Buenos Aires: AAAI. 2015. 353–359. [doi: 10.5555/2832249.2832298]

[24] Wang Z, Li Z, Li ZS. Optimizing simple tabular reduction algorithm for factor-decomposition encoding instances. Ruan Jian Xue Bao/Journal of Software, 2021, 32(11): 3530−3540 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6094.htm [doi: 10.13328/j.cnki.jos.006094]

[25] Hamadi Y, Sais L. Handbook of Parallel Constraint Reasoning. Cham: Springer, 2018. 337–345.

[26] Li Z, Li ZS, Li Y. A constraint network model and parallel arc consistency algorithms based on GPU. Journal of Computer Research and Development, 2017, 54(3): 514–528 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2017.20150912]

[27] Dasygenis M, Stergiou K. Methods for parallelizing constraint propagation through the use of strong local consistencies. Int'l Journal on Artificial Intelligence Tools, 2018, 27(4): 1860002. [doi: 10.1142/S0218213018600023]

[28] Rolf CC, Kuchcinski K. Parallel consistency in constraint programming. In: Proc. of the 2009 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications. Las Vegas: CSREA Press, 2009. 638−644.

[29] Rolf CC, Kuchcinski K. Combining parallel search and parallel consistency in constraint programming. In: Proc. of the 2010 TRICS Workshop at the Int'l Conf. on Principles and Practice of Constraint Programming. Cham: Springer, 2010. 38−52.

[30] Li Z, Yu ZZ, Wu P, Chen JN, Li ZS. A novel multi-thread parallel constraint propagation scheme. IEEE Access, 2019, 7: 167823–167835. [doi: 10.1109/ACCESS.2019.2951027]

[31] Chen JN, Li Z, Li ZS. Research on parallel propagation mode of table constraint based on multi-core CPU. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2769−2782 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5989.htm [doi: 10.13328/j.cnki.jos.005989]

[32] Bessière C, Régin JC. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In: Proc. of the 2nd Int'l Conf. on Principles and Practice of Constraint Programming. Cambridge: Springer, 1996. 61–75. [doi: 10.1007/3-540-61551-2_66]

附中文参考文献:

[24] 王震, 李哲, 李占山. 优化简单表缩减算法求解因子分解编码实例. 软件学报, 2021, 32(11): 3530−3540. http://www.jos.org.cn/1000-9825/6094.htm [doi: 10.13328/j.cnki.jos.006094]

[26] 李哲, 李占山, 李颖. 基于GPU的约束网络模型和并行弧相容算法. 计算机研究与发展, 2017, 54(3): 514–528. [doi: 10.7544/issn1000-1239.2017.20150912]

[31] 陈佳楠, 李哲, 李占山. 基于多核CPU的表约束并行传播模式研究. 软件学报, 2021, 32(9): 2769–2782. http://www.jos.org.cn/1000-9825/5989.htm [doi: 10.13328/j.cnki.jos.005989]

李哲(1990－), 男, 博士, 主要研究领域为约束规划, 并行计算.

李占山(1966－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为机器学习, 约束推理.

于哲舟(1961－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为计算智能, 嵌入式系统应用.