arXiv:2602.03588v1 [cs.CL] 3 Feb 2026

控制流图上的高效部分约束满足问题算法

Xuran Cai 和 Amir Goharshady

牛津大学 英国牛津 xuran.cai@cs.ox.ac.uk,
amir.goharshady@cs.ox.ac.uk

摘要。 在本文中，我们研究程序控制流图（CFG）上的部分约束满足问题（PCSP）。PCSP 是广为人知的约束满足问题（CSP）的一种推广。在 CSP 框架中，我们定义一组变量、一组约束以及一个包含每个变量所有可能取值的有限域 D。目标是为每个变量赋予一个值，使所有约束都得到满足。在 CSP 的图变体中，我们考虑一个底层图，为图中的每个顶点设置一个对应变量，并为每条边设置一个或多个对应约束。在部分约束满足问题中，我们允许以指定成本违反某些约束，目标是找到总成本最小的解。许多经典的编译器优化任务都可以表述为控制流图上的部分约束满足问题。例如，寄存器分配、生命周期最优推测性部分冗余消除（LOSPRE）以及银行选择指令的最优放置。另一方面，众所周知，结构化程序的控制流图具有稀疏性，并且可以通过多种方式进行分解。在本文中，我们采用 [7] 所引入的串并循环（SPL）分解。我们的主要贡献是提出一种适用于 SPL 图上的部分约束满足问题的通用算法，其时间复杂度为 $O(|G| \cdot |D|^6)$，其中 $|G|$ 表示控制流图的规模。请注意，对于任意固定的有限域 D，这将得到一个线性时间解。我们的算法可以视为此前用于寄存器分配和 LOSPRE 的基于SPL的方法的推广与统一。此外，我们针对另一项经典的部分约束满足问题任务——最优银行选择——给出了实验结果，其运行时间比此前的最佳方法快四倍。

关键词： 结构化程序 · 编译器优化 · 控制流图 · 图分解 · 部分约束满足问题。

1 引言

约束满足问题和部分约束满足问题。 约束满足问题（CSP）为不同领域的各种任务提供了一个表达能力强大的框架，在编译器优化领域尤为如此。约束满足问题涉及为变量确定取值，同时遵循规定这些取值允许组合方式的约束 [25]。许多常见的图相关问题，例如图着色问题，都可以表述为

约束满足问题。然而，在某些情况下，寻找满足所有约束的完整解可能不可行或不切实际。在这种情况下，我们可以寻求部分解，具体而言，就是满足尽可能多的约束，或者为每个未满足的约束赋予成本并使总成本最小化；这便属于部分约束满足问题的研究范畴（PSCP） [30]。

部分约束满足问题有广泛的应用。一个能优雅地纳入部分约束满足问题框架的典型例子是最大可满足性问题 [31]。此外，各种编译器优化任务，尤其是与图论相关的任务，也可以表示为部分约束满足问题。例子包括寄存器分配 [7], 生命周期最优推测性部分冗余消除 [6], 以及存储体选择指令的最优放置 [32]。

即使定义域大小为 3，部分约束满足问题的 NP 难性也可通过从图的 3着色问题进行简单归约来证明。在 [31], 中，通过从最大可满足性问题进行归约，证明了即使所有定义域的大小都限制为二，部分约束满足问题仍是 NP困难。计算实验也在实际场景中支持了这些发现 [31]。

高效部分约束满足问题算法。 尽管约束满足问题和部分约束满足问题都是 NP 困难问题，但当底层原图是树或具有树状结构时，存在高效的多项式时间算法 [18,24,35]。即使不存在这类结构，认识到问题可以被视为包含一棵树或属于树状结构的一部分，仍然大有裨益，并能带来有效的启发式算法 [18,19]。通过 [30], 利用树分解，部分约束满足问题算法取得了重大进展。这种方法提供了一种基于图的树宽的高效参数化算法。树宽是衡量树状程度的指标。直观地说，树宽为 t 的图可以分解为若干较小的顶点集，每个顶点集至多包含 $t+1$ 个顶点，并且这些顶点集彼此相连，形成树状结构。

控制流图及其稀疏性。 程序的控制流图（控制流图）定义为一个图，其中每个顶点表示程序中的一条语句，若两条对应语句可以相继执行，则两个顶点之间存在一条有向边。某些应用采用略为粗粒度的控制流图定义，其中每个顶点对应一组程序语句的基本块。在这两种情形下，已充分确立的是，现实世界程序的控制流图是稀疏的，往往类似于树，并且可以分解为若干顶点集合，每个集合最多包含8个以树状方式相互连接的顶点。更形式化地说，结构化无 goto 程序的控制流图的树宽至多为 7 [40]。这一显著结果已被应用于多个领域，包括程序分析、编译器优化和模型检测，其中较小的树宽性质使得能够为 μ-演算模型检测 [36], 数据流分析 [11,27], 马尔可夫决策过程（马尔可夫决策过程） [15,3,1], 可达性分析 [37,9],代数程序分析 [16,14,10],寄存器分配 [33,4], 缓存管理 [2,12],以及等式饱和 [26]设计更快的算法。

[40] 中提出的原始树宽界适用于 Pascal 和 C 程序，但后续研究已将这一结果扩展到其他编程语言，包括 Ada [5], 、Java [29,13], 和 Solidity [8], ，以及路径分解 [17]。然而，也有一些负面发现表明，有界树宽并不总是有助于验证 [22]。

在寄存器分配领域，近期研究 [7] 引入了一种名为串并循环（SPL）的新分解概念，它准确刻画了结构化无 goto 程序的控制流图集合，并对其稀疏性进行了形式化。我们沿用这一分解概念，但表明它的潜力远超 [7]作者的设想，并且可用于解决范围显著广于 [7]所研究的寄存器分配任务的一类问题。

我们的贡献。本文聚焦于结构化无 goto 程序的控制流图上的部分约束满足问题（PCSP）这一一般问题族。我们的方法是 [7] 和 [6], 的推广，因此将寄存器分配和 LOSPRE 作为特例涵盖在内。我们为 PCSP 开发了一种线性时间算法。具体而言，我们利用控制流图及其 SPL 分解的稀疏性，设计了一种运行时间为 $O(n \cdot |D|^6)$的算法，其中n 是程序中的行数， D 是变量的定义域。与 [30], 中的方法不同，我们不依赖树分解，因此无需考虑树宽等额外参数。相反，我们使用 SPL 分解 [7]。这种选择使算法显著简化，因为 SPL 分解能够准确表示控制流图的集合，让我们无需在比必要范围更广的图集上求解问题。此外，SPL 分解还考虑了控制流图中边的方向。

在实验方面，我们将通用解决方案应用于存储体选择指令的最优放置问题，并将其与 [32] 提出的先进方法进行比较。算法的简洁性带来了实际优势。我们在 Small Device C 编译器（SDCC）上进行了大量实验，该编译器基于 [20,21], 并经过了高度优化。与 [32] 相比，我们的算法显著提升了性能。

组织。 第 2 节介绍 SPL 分解的形式化定义，沿用 [7] 的方法。第 3 节阐述我们的主要贡献：一种针对控制流图上一般部分约束满足问题、利用 SPL 分解的线性时间算法。随后，第 4 节介绍其在存储体选择中的应用示例。最后，第 5 节报告实验结果。

2 SPL 分解

我们将在 [7] 提出的一种分解方法基础上构建算法。这种分解方法称为 SPL（串并循环），是在串并图的基础上扩展而来，并增加了一种循环操作。 [7] 证明，一个图是结构化程序的控制流图，当且仅当它具有 SPL 分解。

结构化程序 ***[40]***。如果一个程序可以通过以下语法生成，我们就称它为结构化程序：

$$\begin{aligned} P := \epsilon \mid \texttt{break} \mid \texttt{continue} \mid P; P \\ \mid \texttt{if } \varphi \texttt{ then } P \texttt{ else } P \texttt{ fi} \mid \texttt{while } \varphi \texttt{ do } P \texttt{ od}. \end{aligned} \tag{1}$$

此处， ϵ 是任何不影响控制流的原子操作，例如对变量进行赋值。定义 for 和

switch 等其他结构作为语法糖也很容易。详情请参见 [40] 。如果由上述语法生成的程序中，每个 break 和 continue 语句都出现在 while 循环体内，我们就称该程序为 闭合的 。

SPL 图 *[7]*。 SPL 图 $G = (V, E, S, T, B, C)$ 是一个有向图 (V, E)，具有四个不同的特殊节点 $S, T, B, C \in V$，分别称为起始、终止、中断 和 继续 节点，由以下语法生成：

$$G := A_\epsilon \mid A_{\texttt{break}} \mid A_{\texttt{continue}} \mid G \otimes G \mid G \oplus G \mid G^{\circledast} \tag{2}$$

现在我们解释该语法中的原子图和操作。

原子 ***SPL*** 图。 共有三种不同的原子 SPL 图： A_ϵ、 $A_{\texttt{break}}$ 和 $A_{\texttt{continue}}$， 分别对应于程序 ϵ，中断和继续， 。它们都只包含这四个特殊节点，并且只有一条边，如图 1 所示。

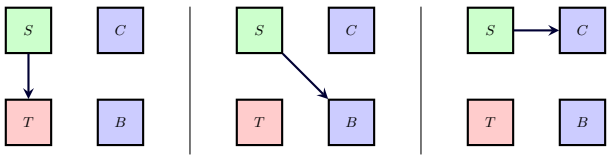


图 1：原子 SPL 图： A_ϵ （左）、 $A_{\texttt{break}}$ （中）和 $A_{\texttt{continue}}$ （右） [7]。

SPL 操作。 SPL 定义了三种操作。设 $G_1 = (V_1, E_1, S_1, T_1, B_1, C_1)$ 和 $G_2 = (V_2, E_2, S_2, T_2, B_2, C_2)$ 为两个互不相交的 SPL 图。那么，通过以下操作得到的图也是 SPL 图。

1. 串联操作。 $G_1 \otimes G_2$ 通过对 G_1 与 G_2 取并集，并合并顶点对 $M =$ （T_1， S_2）、 $B =$ （B_1， B_2）和 $C =$ （C_1， C_2）而生成。 $G_1 \otimes G_2$ 的特异顶点为（S_1， T_2， B， C）。容易验证串联操作满足结合律。图 2 展示了串联操作的两个示例。直观地说，若 G_1 是程序 P_1 的控制流图，且 G_2 是 P_2 的控制流图，则 $G_1 \otimes G_2$ 是程序 P_1的控制流图； P_2.

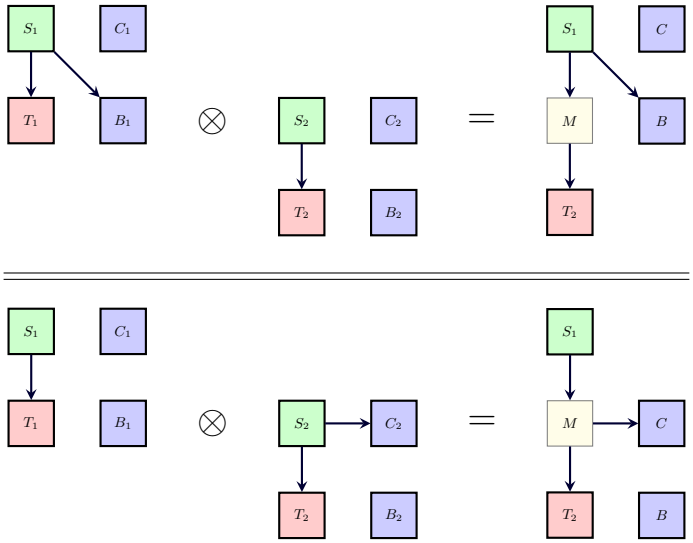


图 2：串联操作 $\otimes$ 的两个示例，取自 [7]。

2. 并行运算。 $G_1 \oplus G_2$ 通过对 G_1 与 G_2 取并集，并合并顶点对 $S = (S_1, S_2)$、$T = (T_1, T_2)$、$B = (B_1, B_2)$ 和$C = (C_1, C_2)$ 而生成。 $G_1 \otimes G_2$ 的特殊顶点元组为（S，T，B，C）。图 3 展示了该运算的一个示例。非正式地说，若 G_1 和 G_2 分别是程序 P_1 与 P_2，的控制流图，则 $G_1 \oplus G_2$ 是程序 if φ then P_1 else P_2 fi 的控制流图。

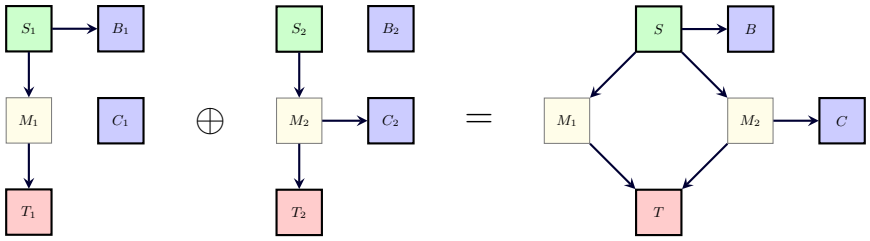


图 3：并行运算 $\oplus$ [7] 的一个示例。

3. 循环操作。 $G_1^{\circledast}$ 通过向G_1 添加四个新顶点 S、T、B、C，再添加以下边生成：(S, S_1)、(S, T)、(T_1, S)、(C_1, S) 和 (B_1, T)。 $G_1^{\circledast}$的特殊顶点元组为(S, T, B, C)。图 4 展示了循环操作的一个示例。直观地说，如果 G_1 是程序P_1的控制流图，那么 $G_1^{\circledast}$就是以 P_1 为循环体的循环的控制流图，即 while φ do P_1 od。

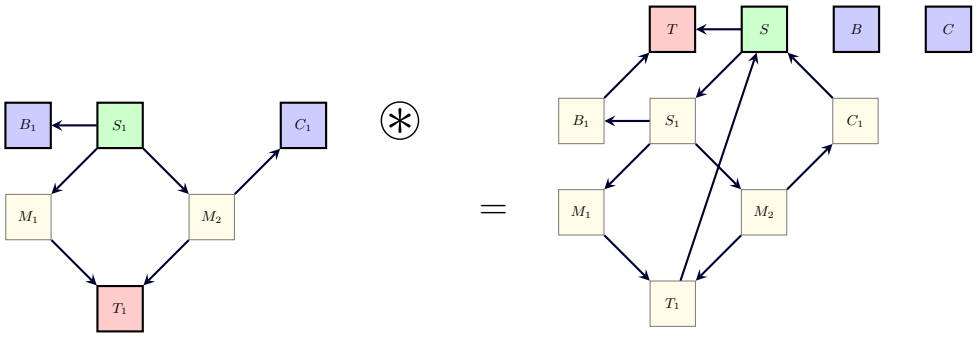


图 4：循环操作的一个示例 $^{\circledast}$ [7]。

如果顶点 B 和 C.没有入边，我们就称 SPL 图 $G = (V, E, S, T, B, C)$是闭合的

SPL 作为控制流图。 根据上述结构化程序和 SPL 图的定义，我们得到如下同态，它将每个结构化程序映射到其控制流图。此外，该同态保持闭合性，即闭合程序会映射为闭合图。当且仅当一个图是某个程序的控制流图时，它才是 SPL 图 [7]。

$$\mathsf{cfg}(\epsilon) = A_\epsilon \; \mathsf{cfg}(\texttt{break}) = A_{\texttt{break}} \; \mathsf{cfg}(\texttt{continue}) = A_{\texttt{continue}}$$

$$\mathsf{cfg}(P_1; P_2) = \mathsf{cfg}(P_1) \otimes \mathsf{cfg}(P_2)$$

$$\mathsf{cfg}(\texttt{if } \varphi \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ fi}) = \mathsf{cfg}(P_1) \oplus \mathsf{cfg}(P_2)$$

$$\mathsf{cfg}(\texttt{while } \varphi \texttt{ do } P_1 \texttt{ od}) = \mathsf{cfg}(P_1)^{\circledast}$$

SPL 分解。 给定一个闭合程序 P， 我们可以首先根据

语法 (1) 对其进行解析，以生成一棵解析树。随后，将上述同态应用于这棵解析树，我们可以根据 (2) 推导出其控制流图的解析树。我们将根据 (2) 得到的控制流图的解析树称为 ***SPL*** 分解 。很容易验证，这一过程在线性时间内完成。示例见图 5。

3 我们的算法

约束满足问题。 著名的约束满足问题（CSP）框架 [25]定义了一个元组 $\langle V,\ D,\ C\rangle$，其中 V是有限的变量集合， D是所有 $v \in V$的定义域集合， C 是约束集合[1]。求解 CSP 的方法是找到一个变量赋值，使所有约束都得到满足。应用于图时，我们将每个节点视为一个变量，将每条边视为一个约束，并规定约束只存在于相邻

[1] 我们的算法可以轻松扩展，以支持变量具有不同的定义域。

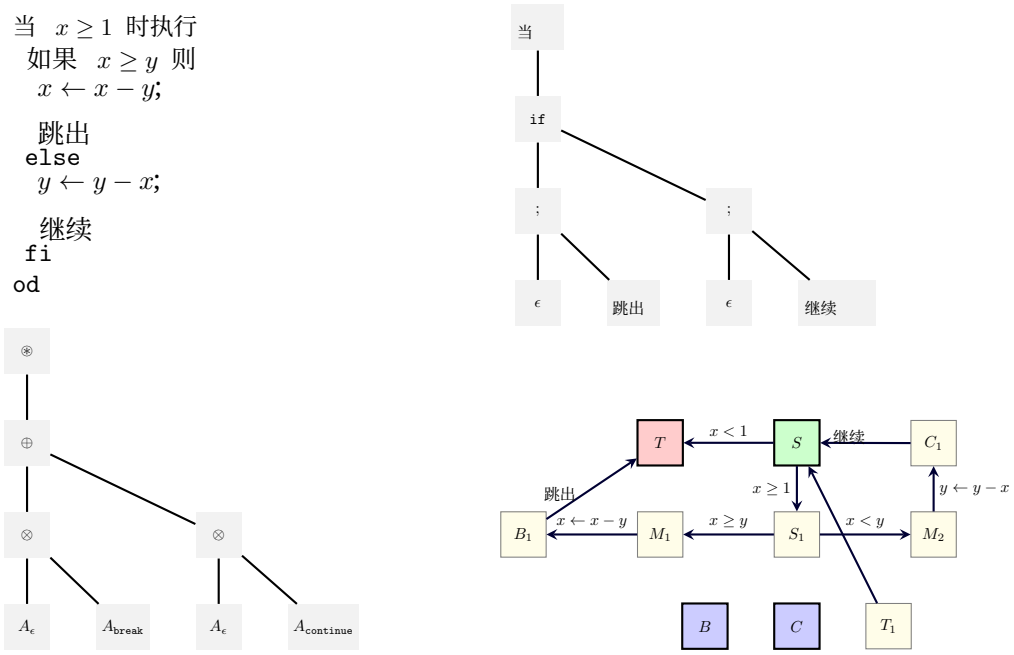


图 5：一个程序 P（左上）、其解析树（右上）、 $G = \text{cfg}(P)$ 对应的解析树（左下）以及图 $G = \text{cfg}(P)$（右下） [7]。图中的边根据程序中的语句标记。

节点。例如，图着色问题可以表述为约束满足问题。考虑图 6，给定三种颜色，我们希望为图着色，使任意两个相邻节点的颜色都不同。我们可以将每个节点视为一个变量，并由每条边施加一个约束，要求相邻节点的颜色不同，那么这个图着色问题就是一个典型的约束满足问题。众所周知，即使将定义域集合限制为仅有三种颜色，图着色问题仍是 NP困难，这意味着约束满足问题也是 NP困难。

部分约束满足问题。在基于图的部分约束满足问题（部分约束满足问题）中，[25], 我们允许以指定成本违反某些约束，目标是找到使该成本最小化的解。为定义成本，我们引入代价函数 $c(e, b_0, b_1)$，其中 e 是边，b_0 和 b_1 是分配给由该边连接的两个节点的值。如果 b_0 和 b_1 不违反约束，则成本为 0；否则，将赋予一个正成本。我们的目标是找到：

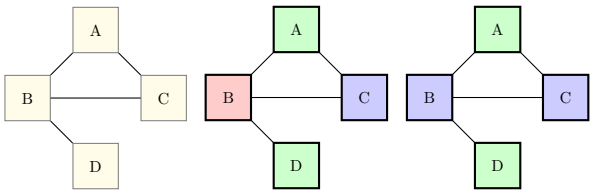
$$\min_A \sum_{e \in E} c(e, A(v_0), A(v_1))$$

其中 A: $V \to D$ 是一个赋值，将每个节点映射到一个域元素。

图 6 中使用两种颜色的示例。与约束满足问题（CSP）类似，我们将每个节点视为一个变量，将每种颜色视为一个取值。我们将代价函数定义如下：

$$c(e, b_0, b_1) = \begin{cases} 0 & \text{if } b_0 \neq b_1 \\ 1 & \text{if } b_0 = b_1 \end{cases}$$

在这种情况下，最小成本为 1，因为连接节点 B 和 C 的边产生了 1 的成本，而所有其他边都不会增加成本。可以直接验证，这种配置构成了部分约束满足问题（PCSP）的一种有效解，因为不可能仅用两种颜色为图形着色并满足这些约束。如果我们为这些约束赋予无限成本，该问题就会简化为约束满足问题（CSP）。因此，PCSP 也是 NP困难问题。



(a) 图　(b) 三种颜色　(c) 两种颜色

图 6：图着色

我们的设定。我们考虑一个 PCSP 实例，其底层图是结构化无 goto 程序的控制流图。我们的目标是利用控制流图的 SPL 分解来获得一种高效算法。正如下文将进一步看到的，这是因为程序分析和编译器优化中的许多经典问题

都是在控制流图上定义的，并且是 PCSP 的特殊情形。

我们基于 ***SPL*** 的算法。我们的算法在 SPL 分解上采用自底向上的动态规划。注意，SPL 分解中的每个节点 u 都对应一个 SPL 子图 $G_u = (V_u, E_u, S_u, T_u, B_u, C_u)$，该子图$G$ 要么是一个原子 SPL 图（当 u 是叶节点时），要么由对 u. 的子节点所对应的图应用某种 SPL 操作得到。令$\Gamma_u = \{S_u, T_u, B_u, C_u\}$ 为 G_u 的特殊顶点集合。令 $X : \Gamma_u \to D$ 为 Γ_u中顶点的赋值。我们定义一个动态规划变量 $\mathtt{dp}[u, X]$. 。我们的目标是计算这个动态规划值，使得

$$\mathtt{dp}[u, X] = \min_{A} \sum_{e \in E_u} c(e, A(v_0), A(v_1))$$

其中，最小值取遍 G_u 中顶点的所有赋值 A，这些赋值在特殊顶点上与 X 一致。换句话说，对于所有 $v \in \Gamma_u$, $A(v) = X(v)$。下面我们将说明算法如何在 SPL 分解的每种节点上计算这些 $\mathtt{dp}[\cdot, \cdot]$ 值:

1. 原子节点：如果 G_u 是一个原子 SPL 图，那么 G_u 中唯一的顶点就是四个特殊顶点。因此，我们必须有 $A = X$. 我们的算法设置

$$\mathtt{dp}[u, X] = \sum_{e \in E_u} c(e, X(v_0), X(v_1))$$

2. 串联节点：假设 $G_u = G_v \otimes G_w$，其中 v 和 w 是 SPL 分解中 u 的子节点。令 X_i 表示 Γ_i 中特殊顶点/变量针对 $i \in \{u, v, w\}$. 的赋值。我们称 X_u 和 X_w 兼容，并记作 $X_u \leftrightharpoons X_w$，如果满足以下条件:
   - $X_u(S_u) = X_v(S_v)$;
   - $X_u(B_u) = X_v(B_v)$;
   - $X_u(C_u) = X_v(C_v)$.

   It turns out that 兼容意味着 当赋予相同的顶点/变量时，赋值 X_u 和 X_v 返回相同的值。

   现在考虑 X_u 和 X_w。我们称 X_u 和 X_w 兼容，并记作 $X_u \leftrightharpoons X_w$，如果满足以下条件:
   - $X_u(T_u) = X_w(T_w)$;
   - $X_u(B_u) = X_w(B_w)$;
   - $X_u(C_u) = X_w(C_w)$.

   The intuition is 与前一种情况相同，只是现在我们有 $T_u = T_w$. 最后，我们称 X_v 和 X_w 兼容，并记作 $X_v \leftrightharpoons X_w$，如果
   - $X_v(T_v) = X_w(S_w)$.

   This is, because v and w 因为 S_w 是控制流图中的同一个顶点。

   根据上述定义，我们的算法只需设置

$$\mathtt{dp}[u, X_u] = \min_{\substack{X_u \leftrightharpoons X_v \\ X_u \leftrightharpoons X_w \\ X_v \leftrightharpoons X_w}} \mathtt{dp}[v, X_v] + \mathtt{dp}[w, X_w].$$

这是因为， G_u 中的每条边都出现在 G_v 或 G_w 中，但不会同时出现在两者中。因此，这些边的成本就是它们在两个子图中的成本之和。

3. 并行节点：我们可以用与串联节点相同的方式处理并行节点，即在两个子节点上寻找兼容赋值。更准确地说，令 $G_u = G_v \oplus G_w$. 兼容条件现改为简单相等。这是因为并行运算会合并 G_v 与 G_w. 中各自的起始、终止、中断和继续顶点。在某些情况下，可能会引入重复边，例如，图 v 与图 w 都有连接 S 顶点与 B 顶点的边。这两条边表示图 u中的同一条边，因此我们只需计算一次成本。我们创建一个集合 E' ，其中包含所有此类边。

采用与前一情形相同的论证，我们的算法设定

$$\mathtt{dp}[u, X_u] = \min_{\substack{X_u \leftrightharpoons X_v \\ X_u \leftrightharpoons X_w \\ X_v \leftrightharpoons X_w}} \mathtt{dp}[v, X_u] + \mathtt{dp}[w, X_u] - \sum_{e \in E'} c(e).$$

4. 循环节点：最后，我们应处理 $G_u = G_v^{\circledast}$. 这一情形。根据构造，与 G_v，相比，图 G_u 具有四个新顶点

$$\Gamma_u = \{S_u, T_u, B_u, C_u\}$$

以及三条新边

$$E_{\text{new}} = \{(S_u, S_v), (S_u, T_u), (T_v, S_u))\}.$$

已知 X_u 和 X_v ，便足以计算新边的成本。我们将基于 X_u 和 X_v 的新边 e 的成本记为 $c(e)$。我们的算法设置

$$\mathtt{dp}[u, X_u] = \min \mathtt{dp}[v, X_v] + \sum_{e \in E_{\text{new}}} c(e).$$

计算出所有 $\mathtt{dp}[\cdot, \cdot]$ 值后，我们的解，即可能的最小成本，为 $\min_X \mathtt{dp}[r, X]$ ，其中 r 是 SPL 分解的根节点， $X : \Gamma_r \to D$ 是根 SPL 图中特殊顶点的赋值。每个节点的确切赋值都可以在动态规划过程中轻松跟踪和更新。

现在我们来分析算法每个步骤的时间复杂度：

– 原子节点： 由于原子图中只有四个顶点，且每个顶点都有 $|D|$ 个可能的取值，因此时间复杂度为 $O(|D|^4)$。为了稍微优化运行时间，我们可以只考虑连通顶点，并忽略非连通顶点。根据定义，非连通顶点不受任何约束。由于原子节点中最多有两个连通顶点，因此时间复杂度为 $O(|D|^2)$。– 串联节点： 计算串联节点时，需要考虑总共八个特殊顶点，但由于我们还需要

考虑兼容性，其中有三对顶点的取值必定相同。因此，时间复杂度为 $O(|D|^5)$，因为我们只需要考虑五个独立变量的取值。– 并行节点： 分析与串联节点类似，不同之处在于有四对顶点的取值必须相同。因此，时间复杂度为 $O(|D|^4)$。– 循环节点： 这也与前面的情况类似，但不会合并任何顶点，也没有顶点需要取值相同。因此，时间复杂度为 $O(|D|^8)$。不过，我们知道，经过循环操作后， C_u 和 B_u 都不与其他顶点连通。因此，我们可以暂时忽略它们，将时间复杂度降至 $O(|D|^6)$。

因此，SPL 分解中每个节点的动态规划计算可以在 $O(|D|^6)$内完成。 此外，SPL 分解的渐近规模与控制流图相同。因此，我们的算法总体运行时间为 $O(|G| \cdot |D|^6)$。

在讨论存储体选择问题的布局之前，我们先介绍另外两个利用了 SPL 分解的应用 [6,7]如何作为部分约束满足问题的实例来处理。

寄存器分配。 这里我们展示寄存器分配 [7] 是部分约束满足问题的一个特例。假设给定一个程序 P ，其控制流图为$G = \text{cfg}(P) = (V, E, S, T, B, C)$。设 $[r] = \{0, 1, \ldots, r-1\}$ 为可用寄存器集合， $\mathbb{V}$ 为程序变量集合。每个变量 $v \in \mathbb{V}$ 都有一个生命周期 $\text{lt}(v)$，它是 G. 的一个连通子图。由于生命周期可以通过简单的数据流分析计算得到，我们不失一般性地假设它们是算法的输入。对于 G的顶点 v 或边 e ，我们用 $L(v)$或 $L(e)$表示在该顶点/边处存活的变量集合。赋值 是一个函数 $f\colon \mathbb{V} \to [r] \cup \{\bot\}$ ，它将每个变量映射到一个寄存器或 $\bot$. 。后者表示该变量发生溢出。若一个赋值不会将生命周期相交的两个变量映射到同一个寄存器，则该赋值有效。我们用 F. 表示所有有效赋值的集合。需要切换寄存器的值时会产生成本，目标是找到成本最低的赋值。

在这种情况下，分配给每个节点的“值”是一个活跃变量的分配。假设每个节点上最多有 V 个活跃变量，且有 r 个寄存器，那么我们可以创建一个包含 V 个节点的完整推理图，并尝试用 r 种颜色为它们着色。因此，域的大小为

$$\binom{V}{r} \cdot r! + \binom{V}{r-1} \cdot (r-1)! + \cdots + \binom{V}{0} \cdot 0! \in O(r \cdot V^r).$$

因此，使用我们的算法，时间复杂度为 $O(|G| \cdot r^6 \cdot V^{6 \cdot r})$。

LOSPRE。LOSPRE [6] 是部分约束满足问题的另一个例子。这是一种现代的冗余消除技术，只计算一次重复表达式，并将结果存储在模板变量中。每当需要使用该表达式时，我们可以直接使用模板变量，而不必重新计算。对于控制流图 $G = \{V, E\}$，我们可以将该问题定义如下：

–使用集：考虑一个表达式 e. 。我们将 e 的使用集 U 定义为控制流图中计算表达式 e 的所有节点的集合。

– 生命集合：我们的目标是在若干个位置预先计算表达式 e，将结果保存到临时变量 temp 中，然后在 $U.$ 的每个节点中用 temp 代替 e。我们用 L 表示变量 temp 的生命周期，并称之为生命集合。

– 失效集合：我们称控制流图中的节点 v 会使 e 失效，如果 v 处的语句改变了 $e.$ 的值。例如，如果 e =a+b，那么语句 a = 0 会使 $e.$ 失效。我们用 $I.$ 表示所有失效节点构成的集合。这些节点在 LOSPRE 中起着关键作用，因为它们会迫使我们通过重新计算 $e.$ 来更新保存在 temp 中的值。我们假设入口节点和出口节点均为失效节点，因为 LOSPRE 是一种过程内分析，不掌握当前函数执行之前或之后程序的执行情况。

– 计算集合：给定上文中的集合 U，L 和 I，我们必须确保临时变量 temp 的值在 $U \cup L.$ 的每个节点处都正确。因此，对于控制流图中的每条边（x，y）$\in E$，若 $x \notin L$ 且 $y \in U \cup L$，我们就必须在 x 和 $y.$ 之间插入计算 temp $= e$。类似地，如果 $x \in I$，那么执行 x后存储在 temp 中的值就会失效，因此需要在 x 和 $y.$ 之间插入相同的计算。形式化地说，必须将计算 temp $= e$ 插入控制流图的以下边集合中：

$$C(U, L, I) = \{(x, y) \in E \mid x \notin L \setminus I \ \wedge \ y \in U \cup L\}.$$

这一次，上述过程涉及两类成本：（i）向 C(*U*、 *L*、 *I*) 中的边注入计算；以及（ii）在 $L.$ 的每个节点保留一个额外变量 temp。这些成本取决于编译器所追求的目标。例如，旨在最小化代码大小的编译器会重点关注（i）。另一方面，如果我们的目标是缓解寄存器压力，就会希望最小化（ii）。LOSPRE 是一个具有较强表达能力的框架，其中这些成本由两个函数建模

$$c : E \to K$$

and

$$l : V \to K.$$

其中，K 是一个带有加法运算的全序集合，c 是一个函数，用于将每条边映射到在该边添加 e 计算的成本；l 同样是一个函数，用于将 CFG 的每个顶点映射到在该顶点保持临时变量temp存活的成本。我们的目标是找到一个生命集合 L，使总成本最小

$$\textsc{Cost}(G, U, I, L, c, l) = \sum_{e \in C(U,L,I)} c(e) + \sum_{v \in L} l(v).$$

要将这个问题视为部分约束满足问题（PCSP），只需稍作修改，因为边的成本与 PCSP 中的定义相同，而节点的成本可以轻松计入总成本。在这种情况下，若每个节点属于使用集、生命集合或失效集合，则为其赋予相应的“值”。由于每个集合都有 2 种不同的情况，因此域大小为 $2^3 = 8$。因此，应用我们的 PCSP 算法时，可以在 $O(|G| \cdot 8^6) = O(|G|)$ 内完成；在这种情况下，得益于

域大小为常数，我们可以开发出一种线性算法，而无需考虑任何参数。

4 存储体选择指令的最优放置

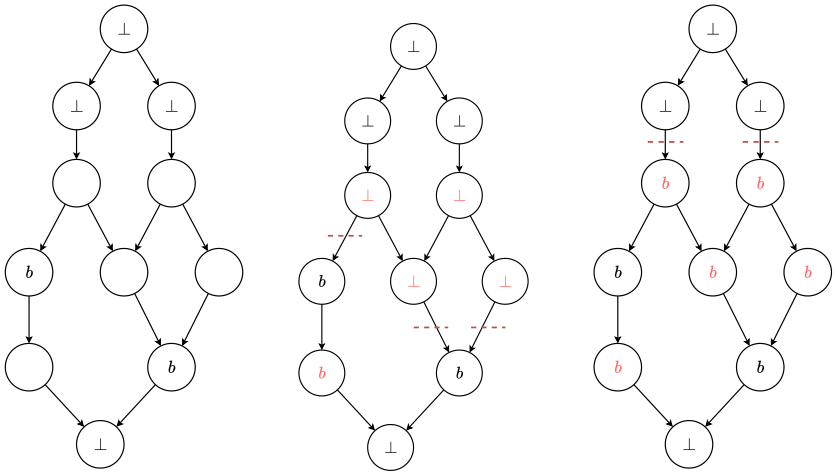
如上一节所述，寄存器分配和 LOSPRE 都是 PCSP 的特殊情形；此前，它们均通过树分解和 SPL 分解求解。本节将介绍 PCSP 在控制流图上的另一种经典应用，即存储体选择指令的最优放置。据我们所知，目前还没有利用 SPL 分解解决这一问题的方法。

动机。 分区存储器架构在 8 位和 16 位微控制器中十分常见。在这些系统中，逻辑地址空间的一部分充当通向更大物理地址空间的窗口。物理地址空间中可映射到该窗口的各个片段称为存储体。系统提供了一种机制，用于确定物理地址空间的哪一部分在窗口中可见，通常通过存储体选择指令实现。

通常由程序员（例如通过嵌入式 C 中的命名地址空间）或编译器（例如使用装箱启发式算法）将变量分配到特定存储体，以尽量减少随机存取存储器的使用量。有些方法会将变量在存储体中的放置与存储体选择指令的插入结合起来。然而，在嵌入式系统中，可用于代码的空间通常比可用于数据的空间更充裕。因此，存储在分银行存储器中的变量往往较大，所以优先将变量高效地装入各存储体，再处理代码大小和执行速度等其他因素，更为有利。因此，变量在存储器中的放置通常发生在编译过程的较早阶段，而存储体切换指令则在之后插入。

域。 以下银行选择问题的形式化定义改编自 [32]。令 D 为存储体域，其中包含一个特殊符号 $\perp \in D$，表示当前选定的存储体未知。程序可建模为控制流图 $G = (V, E)$，其中 $E \subseteq V^2$，每个节点 $v \in V$都可以分配一个来自 D的存储体。有些节点已预先分配存储体，因为在这些节点上必须启用特定的存储体。

示例。 图 7 展示了一个包含存储体选择指令的程序示例。该程序有两个预先指定的顶点，在这些顶点处应激活存储体 b。在程序的其他位置，由于没有访问任何存储体的操作，因此对所选存储体没有限制。一种简单的临时方法是在需要激活存储体之前插入存储体选择指令。图 7 (b) 展示了这种做法，需要插入三条存储体选择指令。假设我们的目标是尽量减少存储体选择指令的数量。那么，可以像图 7 (c) 所示，在程序开头插入这些指令。这样只需两条指令，且得到的是最优解。



(a) 程序 (b) 临时方法 (c) 最优

图 7：示例实例

代价函数。 在经典的最优存储体选择问题表述中，会在控制流图 $G = (V, E)$上定义一个代价函数。这是一个函数c：$E \times D \times D \to \mathbb{R}$，它根据分配给边 $e \in E$ 两个端点的存储体，为每条边分配一个成本。换句话说，如果在边 e 之前激活的是 b_0，而经过该边后激活的是 b_1，那么就会产生成本 $c(e, b_0, b_1)$。此类代价函数可以根据不同的优化准则来设计。用于优化代码大小的一种典型代价函数如下：

- $c(e, b, b) = c(e, b, \bot) = 0$ 因为无需插入任何指令；- $c(e, b_0, b_1) = c_1 > 0$ 当 $b_0 \neq b_1 \neq \bot$ 是来自已跳转的条件分支的一条边时，因为拆分这样的边会产生额外的无条件跳转指令。– 对于所有其他情况，$c(e, b_0, b_1) = c_0 > 0$ 对于 $b_0 \neq b_1 \neq \bot$，条件是 $c_0 < c_1$。

目标。 最优存储体选择的目标是，为控制流图的顶点分配存储体，使边的总成本最小化。换言之，我们旨在计算

$$\min_{A} \sum_{e \in E} c(e, A(v_0), A(v_1))$$

其中 $A: V \to D$ 将每个节点映射到一个域。我们注意到，该问题是NP困难的，即使当代价函数被简化为 $c(e, b_0, b_1) = 0$ 对于 $b_0 = b_1$或 $b_1 = \bot$，以及 $c(e, b_0, b_1) = 1$ 在所有其他情况下 [32]。

我们的算法。 容易验证，存储体选择指令的最优插入是PCSP 图问题的一个实例。因此，我们可以直接应用第3节中提出的通用解法，并在 $O(|G| \cdot |D|^6)$ 时间内求解，其中 $|D|$ 是可能的存储体数量。

5 实验结果

如第 3 节所述，寄存器分配和 LOSPRE 此前都已通过 SPL 分解求解 [6,7]。事实上， [6,7] 提供的算法可以视为我们通用算法的特例，其中定义域和代价函数固定为第 3 节中的设定。因此，我们直接沿用了它们的实验结果。本节将进一步给出第 4 节最优存储体选择问题的实验结果；据我们所知，此前尚未在 SPL 分解的背景下研究过该问题。此外，我们还将所提出的方法与用于 NP 难问题的经典算法进行比较，包括可满足性问题和整数线性规划。

实现。 我们使用 C++ 实现了存储体选择算法，并将其集成到 Small Device C 编译器（SDCC） [20,21] 中。我们将以免费开源软件的形式发布该实现，并将其贡献至公有领域，相关链接将在本文最终版本中提供。为保持匿名，该链接已在评审阶段移除。

基线。 我们将本算法的运行时间与 [32], 的基于树宽的参数化方法进行了比较，该方法是当前存储体选择领域的最先进方法。该方法的渐近运行时间为 $O(|G| \cdot \mathrm{tw}(G) \cdot |D|^{\mathtt{tw}(G)+1})$，其中tw($G$)是 G的树宽, 理论上最高可达 7 [40]。我们未考虑其他先前方法，因为它们要么运行时间对程序规模呈指数依赖 [38,39] ，要么无法保证结果的质量或最优性 [34]。

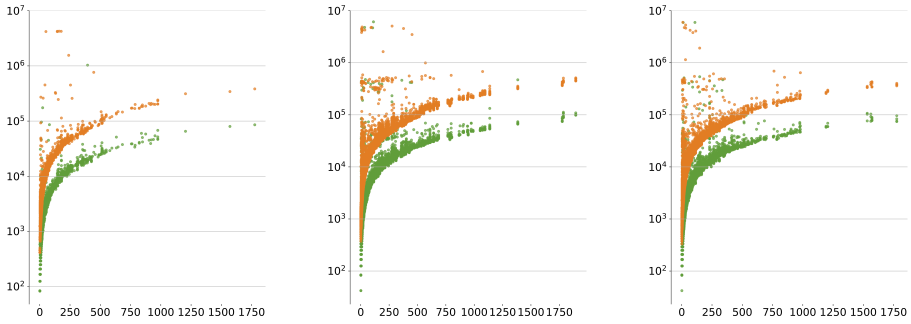
机器。 所有实验均在一台运行 Oracle Linux（ARM 64 位）的虚拟机上进行，该虚拟机配备 1 个 Apple M2 处理器核心和 4GB 随机存取存储器。

基准测试。 我们严格遵循 [32], 的实验设置，使用 SDCC 回归测试套件作为基准集。该套件共包含 HC08 的 13,463 个实例、Z80 的 179,611 个实例，以及 MCS51 的 84, 700 个实例。这些基准测试是预期在资源受限环境中运行的嵌入式程序。因此，重点强烈集中在代码尺寸优化上。于是，我们使用了以下标准代价函数：

- $c(e, b, b) = c(e, b, \bot) = 0$;- $c(e, b_0, b_1) = 1$ 当 e 是来自已跳转的条件分支的一条边时；– 对于所有其他情况， $c(e, b_0, b_1) = 1$。

我们只比较运行时间。由于我们的方法和 [32] 都能为存储体选择找到最优解，且输出结果一致，因此不进行输出比较。

运行时间。 图 8 对比了我们的算法与 [32]的算法在三种不同架构上的运行时间。对于 HC08，我们的算法平均耗时 1,998.3 纳秒，而基于树宽的方法平均耗时 8,558.8 纳秒。对于 Z80，我们的算法平均耗时 2,402.1 纳秒，而基于树宽的方法平均耗时 8,209.3 纳秒。对于 MCS51，我们的算法平均耗时 1, 834.3 纳秒，而基于树宽的方法平均耗时 6,268.3 纳秒。



(a) HC08 (b) Z80 (c) MCS51

图 8： [32] （橙色）的基于树宽的算法与我们的方法（绿色）的运行时间比较。 x 轴表示控制流图中的顶点数量， y 轴表示以纳秒为单位的时间。 y 轴采用对数尺度。

与 ***SAT*** 和 ***ILP*** 求解器的比较。由于最优存储体选择是 NP完全问题，我们将运行时间与通过 SAT 和整数线性规划求解 NP完全问题的最先进方法进行了比较。我们分别以自然方式用 SAT 和整数线性规划对该问题进行了编码。随后，我们在各项基准测试中使用了最先进的 SAT 求解器 Kissat [23],和工业级整数线性规划求解器 Gurobi [28] 。图 9 所示结果表明，我们的方法比整数线性规划方法快近 10 倍，而后者又比 SAT 求解器快约 100 倍。

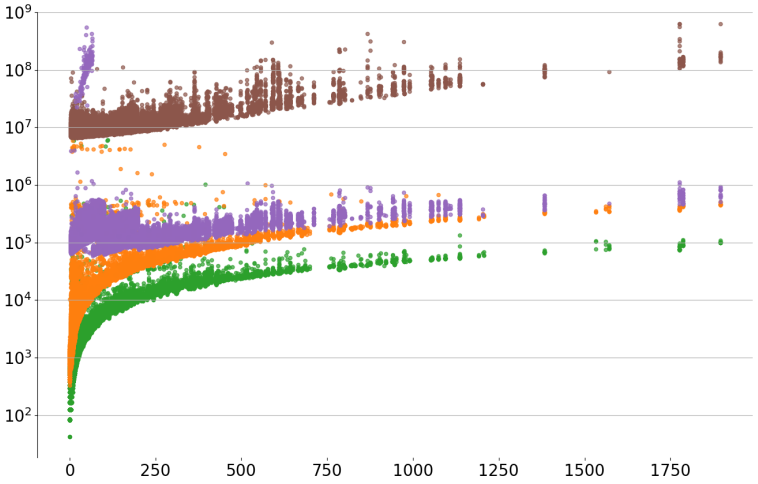


图 9： [32] 的基于树宽的算法（橙色）和我们的方法（绿色）与基于 SAT（棕色）和基于 ILP（紫色）的方法的运行时间比较。 x 轴表示控制流图中的顶点数， y 轴表示时间（纳秒）。 y 轴采用对数尺度。

讨论。总而言之，我们的方法比先前最先进的存储体选择方法快约四倍。这一提升相当显著，尤其考虑到 [32] 的基于树宽的方法已经过高度优化，并且已集成到成熟的 SDCC 编译器中。我们认为，速度提升主要是因为我们的算法运行时间不受树宽等参数的影响，而且在实践中，我们的 $O(n)$ 渐近运行时间分析中隐藏的常数因子很小。

6 结论

在本工作中，我们提出了一种针对控制流图上的图上部分约束满足问题的高效线性时间算法，该算法利用 SPL 分解 [7]。我们的方法适用于多种编译器优化任务，包括寄存器分配 [7], 生命周期最优推测性部分冗余消除（LOSPRE） [6], 和最优存储体选择。本文的主要贡献是一种能够解决所有图上部分约束满足问题的动态规划算法，无需针对不同的优化任务分别设计算法。我们的方法在存储体选择方面实现了显著的理论性能提升，实验表明，在实践中，其速度约为基于树宽的方法 [32]的四倍。

参考文献

1. Ahmadi, A., Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Safavi, R., Zikelic, Đ.：马尔可夫链核心计算的算法与困难性结果。载于：FSTTCS。第 29:1–29:20 页（2022） 2. Ahmadi, A., Daliri, M., Goharshady, A.K., Pavlogiannis, A.：面向缓存的数据放置高效近似算法。载于：PLDI。第 857–871 页（2022） 3. Asadi, A., Chatterjee, K., Goharshady, A.K., Mohammadi, K., Pavlogiannis, A.：针对低树宽马尔可夫链和马尔可夫决策过程定量分析的更快算法。载于：ATVA。第 12302 卷，第 253–270 页（2020） 4. Bodlaender, H.L., Gustedt, J., Telle, J.A.：固定数量寄存器的线性时间寄存器分配。载于：SODA。第 574–583 页（1998） 5. Burgstaller, B., Blieberger, J., Scholz, B.：Ada程序的树宽。载于：Ada-Europe。第 78–90 页（2004） 6. Cai, X., Goharshady, A.：无 goto 程序中更快的生命周期最优推测性部分冗余消除。载于：SETTA。第 382–398 页。Springer（2024） 7. Cai, X., Goharshady, A.K., Hitarth, S., Lam, C.K.：通过控制流图的文法分解实现更快的类Chaitin寄存器分配。载于：ASPLOS。第 463–477 页。ACM（2025）。 https://doi.org/https://doi.org/10.1145/3669940. 3707286 8. Chatterjee, K., Goharshady, A.K., Goharshady, E.K.：智能合约的树宽。载于：SAC。第 400–408 页。ACM（2019） 9. Chatterjee, K., Goharshady, A.K., Goyal, P., Ibsen-Jensen, R., Pavlogiannis, A.：针对常数树宽基本RSM的动态代数查询的更快算法。《ACM程序设计语言与系统汇刊》**41**(4), 23:1–23:46（2019） 10. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.：常数树宽组件并发系统中代数路径性质的算法。载于：POPL。第 733–747 页（2016） 11. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.：按需数据流分析的最优且完全并行算法。载于：ESOP。第 112–140 页（2020） 12. Chatterjee, K., Goharshady, A.K., Okati, N., Pavlogiannis, A.：数据打包的高效参数化算法。《ACM程序设计语言汇刊》**3**(POPL), 53:1–53:28（2019） 13. Chatterjee, K., Goharshady, A.K., Pavlogiannis, A.：Jtdec：用于 Soot 树分解的工具。载于：ATVA。第 10482 卷，第 59–66 页（2017） 14. Chatterjee, K., Ibsen-Jensen, R., Goharshady, A.K., Pavlogiannis, A.：常数树宽组件并发系统中代数路径性质的算法。《ACM程序设计语言与系统汇刊》**40**(3), 9:1–9:43（2018） 15. Chatterjee, K., Lacki, J.：低树宽马尔可夫决策过程的更快算法。载于：CAV。第 543–558 页（2013） 16. Conrado, G.K., Goharshady, A.K., Kochekov, K., Tsai, Y.C., Zaher, A.K.：利用控制流图和调用图的稀疏性，实现高效且按需的代数程序分析。《ACM程序设计语言汇刊》**7**(OOPSLA2), 1993–2022（2023） 17. Conrado, G.K., Goharshady, A.K., Lam, C.K.：控制流图的有界路径宽度。《ACM程序设计语言汇刊》**7**(OOPSLA2), 292–317（2023） 18. Dechter, R., Pearl, J.：约束满足问题的基于网络的启发式方法。《人工智能》**34**(1), 1–38（1987）。 https://doi.org/https://doi.org/10. 1016 6 19 0004 3702 87 90002 / - ( ) - . Dechter, R., Pearl, J.：约束处理的树聚类方案。载于：AAAI。第 150–154 页（1988） 20. Dutta, S.：编译器剖析：一种可重定向ANSI-C编译器。《Circuit Cellar》**121**(5)（2000）

21. Dutta, S., Drotos, D., Vigor, K., 等：Small Device C 编译器 (2003)，http: // sdcc.sourceforge.net/ 22. Ferrara, A., Pan, G., Vardi, M.Y.：验证中的树宽：局部与全局。载于：LPAR. 第 3835 卷，第 489–503 页 (2005) 23. for Formal Models, I., Verification：Kissat-public (2025)，https://fmv.jku.at/ kissat/ 24. Freuder, E.C.：k-树结构约束满足问题的复杂性。载于：AAAI. 第 4–9 页 (1990) 25. Freuder, E.C., Wallacex, R.J.：部分约束满足 **58**，21–70 (1992). https://doi.org/https://doi.org/10.1016/0004-3702(92)90004-H 26. Goharshady, A.K., Lam, C.K., Parreaux, L.：稀疏等式图的快速最优提取。载于：OOPSLA (2024) 27. Goharshady, A.K., Zaher, A.K.：利用树深度和树宽进行高效的过程间数据流分析。载于：VMCAI. 第 13881 卷，第 177–202 页 (2023) 28. Gurobi Optimization, L.：Gurobi (2008)，https://www.gurobi.com/ 29. Gustedt, J., Mæhle, O.A., Telle, J.A.：Java 程序的树宽。载于：ALENEX.《计算机科学讲义》，第 2409 卷，第 86–97 页。Springer (2002) 30. Koster, A.M.C.A., van Hoesel, S.P.M., Kolen, A.W.J.：利用树分解求解部分约束满足问题 **40**，170–180 (2002). https://doi. org/https://doi.org/10 10046 31 1002. /net. . Koster, A.M., Hoesel, S.P., Kolen, A.W.：部分约束满足问题：面与提升定理。《运筹学通讯》**23**(3)，89–97 (1998). https://doi.org/https://doi.org/10 1 32 1016 0167 6377 98 00043 . /S - ( ) - . Krause, P.K.：在多项式时间内实现存储体选择指令的最优放置 (2013 2463598 33 10 1145 2463596). https://doi.org/https://doi.org/ . / . . Krause, P.K.：在多项式时间内实现最优寄存器分配。载于：CC. 第 7791 卷，第 1–20 页 (2013) 34. Liu, T., Xue, C.J., Li, M.：分区存储器架构的联合变量划分与存储体选择指令优化。《ACM 嵌入式计算系统汇刊》**12** 2442126 35 2013 10 1145 2442116 ( ). https://doi.org/https://doi.org/ . / . . Mackworth, A.K., Freuder, E.C.：若干用于约束满足问题的多项式网络一致性算法的复杂性。《人工智能》**25**，65–74 4 36 1985 10 1016 0004 3702 85 90041 ( ). https://doi.org/https://doi.org/ . / - ( ) - . Obdrzálek, J.：树宽有界时的快速 μ演算模型检测。载于：CAV. 第 80–92 页 (2003) 37. Sankaranarayanan, S.：在树分解上通过消息传递进行可达性分析。载于：CAV. 第 604–628 页 (2020) 38. Scholz, B., Burgstaller, B., Xue, J.：面向分区存储器架构的存储体选择指令最小化。载于：CASES. 第 201–211 页。ACM (2006). https: //doi.org/https://doi.org/10 1176786 39 1145 1176760 . / . . Scholz, B., Burgstaller, B., Xue, J.：分区存储器架构中存储体选择指令的最小化放置。《ACM 嵌入式计算系统汇刊》**7** (2008). https://doi.org/https://doi.org/10 1331336 40 1145 1331331 . / . . Thorup, M.：所有结构化程序都有较小的树宽和良好的寄存器分配。《信息与计算》**142**，159–181 (1998). https://doi.org/https://doi. org/10.1006/inco.1997.2697