arXiv:2302.02644v1 [cs.DS] 6 Feb 2023

简单析取时间问题的模型与算法

CARLO S. SARTORI、Pieter Smet 和 Greet Vanden Berghe，鲁汶大学，计算机科学系，比利时

简单时间问题是一类强大的模型，能够描述事件之间的时间关系，这些关系出现在物流、机器人规划和管理系统等许多现实应用中。经典的简单时间问题规定每个事件只能有一个释放日期和一个截止日期。本文重点研究事件可以有任意多个释放日期和截止日期的情况。然而，这类问题有过多种不同称谓。为简化并规范术语，我们引入“简单析取时间问题”这一名称。我们提供三个数学模型，分别使用约束规划和线性规划来描述该问题。为高效求解简单析取时间问题，我们受以往研究启发设计了两种新算法；这两种算法均利用问题的结构，大幅降低空间复杂度。此外，我们实现了文献中的算法，并首次通过广泛的实验，对求解简单析取时间问题的方法进行了深入的实证研究。我们的分析与结论可为未来研究者和从业者在新的应用中处理类似时间约束问题提供指导。所有结果、源代码和实例均已公开，供公众获取，以进一步助力未来研究。

CCS 概念：• **计算方法** → **时间推理**；规划与调度；• **计算数学** → **图算法**。

补充关键词：简单时间问题、析取项、一致性检查、最短路径、实证分析

1 引言

简单时间问题（STP）提供了一种形式化结构，用于描述事件之间可能存在的时间关系。这些时间关系广泛存在于各种现实问题中 [12] ，包括优先关系、最大经过时间，以及每个事件各自的一个释放日期和截止日期。STP 的一大优势是，可以通过标准的最短路径方法在多项式时间内求解 [8]。然而，研究人员和从业者在使用 STP 建模时仍受到相当大的限制。即使是每个事件具有多个释放日期和截止日期这样看似简单的特性，也无法用 STP 表示。

相比之下，析取时间问题（DTP）提供了一个更为广泛的框架来描述时间关系。然而，这种表达能力的代价是，它们通常表示的是 NP 完全问题 [28]。简单析取时间问题（S DTP）是一类原始的 DTP，它对 STP 进行了推广，同时保留了高效的多项式时间求解方法。SDTP 扩展了 STP，使每个事件可以有多个互不重叠的释放日期和截止日期。

在学术文献中，SDTPs 有许多不同的称呼：星类问题、零一扩展简单时间问题和 t_2 DTPs。虽然很难确切知道为何同一个问题会有如此多的术语，但可以推测，不同研究者可能从不同角度接触并研究了 SDTPs。尽管这种术语差异的具体原因并非本文关注的重点，但显然，它造成了极低的效率。研究人员和从业者都受到一项繁琐任务的拖累：当有关 SDTPs 的研究成果以不同名称归档时，他们必须先找出已有的相关知识

作者地址：Carlo S.Sartori，carlo.sartori@cs.kuleuven.be；Pieter Smet，pieter.smet@cs.kuleuven.be；Greet Vanden Berghe，greet.vandenberghe@cs.kuleuven.be，鲁汶大学，计算机科学系，De Smet兄弟街1号，比利时根特，邮编9000。

和研究。此外，即使找到了有关 SDTPs 的现有文献，这些论文也主要是理论性的，既未提供方法的实际实现，也没有提供实证见解，说明这些算法在不同情境下的表现有何差异。

我们在尝试比较物流问题中调度具有多个时间窗的任务的不同方法时，确实遇到了上述情况，例如带同步的车辆路径问题 [11, 25], 带转运的取送货问题 [20], 带换乘的需求响应式运输问题 [18] 以及具有相互依赖路线的卡车司机排班 [26]。SDTP 非常适合用于这些问题中的调度。然而，例如，若要通过局部搜索启发式算法求解这些物流问题，就需要极其高效的 SDTP 方法。鉴于文献不仅难以查阅，而且缺乏实证结果，我们需要在实际应用 SDTP 之前：(i) 找出现有方法，(ii) 实现这些方法，以及 (iii) 评估它们在不同情况下的优势和局限性。

因此，本文有四个目标。首先，我们提出一套标准命名法来指称 SDTP，以便未来的研究人员能够用同一个名称指代同一个问题。其次，我们考察现有方法，并开发新的算法来求解 SDTP；这些算法不仅能够降低理论上的渐近最坏情况时间和空间复杂度，还能确保在实践中表现良好。第三，我们对所有技术进行实证研究，并提供相应的开源实现，同时公开我们的实例，以避免重复劳动[1]。第四，也是最后一点，我们希望本文能为有意在工作中应用 SDTP 的研究人员和从业者奠定基础，使他们能够基于我们的研究和实现，轻松实现自己的目标。

2 简单析取时间问题

首先，我们对简单时间问题进行形式化定义，这有助于介绍简单析取时间问题。

定义 *2.1*（简单时间问题 *[8]*）。简单时态网络（STN）表示为$N = (T, C)$，其中 T是变量或时间点的集合，C是关联 T中变量的二元约束集合。时间点 $i \in T$具有闭域 $[l_i, u_i]$，$l_i, u_i \in \mathbb{R}$。C中的约束是简单时序约束，以元组$(i, j, w_{ij}) \in C,\ i, j \in T,\ w_{ij} \in \mathbb{R}$给出，对应于方程 1：

$$s_i - s_j \leq w_{ij} \tag{1}$$

其中，$s_i, s_j \in \mathbb{R}$ 表示分配给时间点 i和 j的解值。

STP 涉及判断其对应的 STN 是否一致。网络 N一致，当且仅当可以推导出可行调度或解 s，使得分配给每个 $i \in T$的时间 s_i均满足 C 中的所有约束以及每个时间点的域。

Dechter 等人 [8] 证明，STN 可以用距离图表示，其中时间点T 是节点，而约束 C 是连接这些节点的弧。首先，我们将一个特殊时间点 α与域 $[0, 0]$ 关联，用以表示时间范围 $s_\alpha = 0$的起点。这个固定点可用于将时间点 T 上的一元约束（例如定义域边界）表示为简单时序约束。例如，$i \in T$的边界 $[l_i, u_i]$可以表示为：

$$s_\alpha - s_i \leq -l_i$$
$$s_i - s_\alpha \leq u_i$$

[1]完整代码仓库将在稍后开放。

然后，我们可以为 STN $N=(T,C)$关联两个距离图：正向图 $G_D=(V,A_D)$和反向图 $G_R=(V,A_R)$。对于这两个图 $V=T\cup\{\alpha\}$。弧集 $A_D=C\cup\{(\alpha,i,-l_i),(i,\alpha,u_i)\ \forall i\in T\}$，其中元素 $(i,j,w_{ij})\in A_D$表示一条从节点 i到j、权重为 w_{ij}的弧。同时， A_R与 A_D相同，只是每条弧的方向都反转了：$(i,j,w_{ij})\in A_D$即 $(j,i,w_{ij})\in A_R$。

确定一个简单时序网络的一致性，进而可归结为验证 G_D 或 G_R中是否存在负成本环。如果不存在负成本环，那么从节点 $\alpha\in V$到其他每个节点 $i\in V\backslash\{\alpha\}$的最短路径距离 $\tau_{\alpha i}$就能给出一个可行调度。在G_D上计算时， $s_i=-\tau_{\alpha i}$给出最早可行调度。而在 G_R上计算时， $s_i=\tau_{\alpha i}$给出最晚可行调度。最早可行调度可定义为解 s：对于 SDTP 的任意其他可行解 s'，均有 $s_i\le s_i'$， $\forall i\in T$。类似地，对于最晚可行调度，有$s_i\ge s_i'$， $\forall i\in T$。

有许多算法可用于快速检测距离图 [3]中的负成本环。其中最简单的一种是 Bellman-Ford [7]。事实上，这是大多数用于求解 SDTP 的方法所采用的算法。在本文余下部分中，我们所说的 Bellman-Ford 始终指其作为标号修正算法、并使用先进先出队列 [3]的实现。

STP 只能容纳域为单一区间的时间点。为了处理时间点可能被赋予多个析取区间之一中的值的问题，需要一种表达能力更强的模型。现在让我们转向本文的主要问题：简单析取时间问题。

定义*2.2*（简单析取时间问题）。简单析取时间网络（SDTN）记为 $N=(T,C)$，其中 T 是时间点集合， $C=C_1\cup C_2$ 是这些时间点上的约束。根据 Kumar 引入的分类， [15], 中 C_1 里的约束属于类型1，而 C_2 里的约束属于类型2。

（类型1）简单时序约束（$i,\ j,\ w_{ij}$） $\in C_1$， $i,\ j\in T$表示等式 n 1

（类型2）简单析取约束（$i,\ D_i$） $\in C_2$，其中 $i\in T$和 D_i是区间或域的列表，记为 $[l_i^c,u_i^c]\in D_i$, $l_i^c,\ u_i^c\in\mathbb{R}$，表示析取

$$\bigvee_{[l_i^c,u_i^c]\in D_i}(l_i^c\le s_i\le u_i^c)$$

请注意， C_2包含一元约束（$|D_i|=1$）。 $T_D\subseteq T$表示满足 $|D_i|>1$的所有时间点的集合。

为了求解 SDTP，我们需要确定析取式差分约束时间网络 N是否一致。与 STP 类似，当且仅当能够导出一个满足约束 C_1 和 C_2的可行解 s时，N才是一致的。

我们假设 D_i中的域按升序排列 [6, 15]。令 $K=\max_{(i,D_i)\in C_2}|D_i|$表示任一给定时间点所对应的域的最大数量，令$\omega=\sum_{(i,D_i)\in C_2}|D_i|$表示实例中的域总数。进一步地，令 $L(D_i)$和 $U(D_i)$分别表示 $i\in T$所有域中的下界值和上界值。 i的全局边界 由 $[L(D_i)\ U(D_i)]$给出，使得 s_i必须属于此边界。然而，这些界内的某些值仍可能不可行。换言之，时间点的域并非连续的。

类型2约束的存在意味着我们无法直接通过最短路径求解该问题。不过，我们可以利用时间点的全局边界重新定义图 G_D 和G_R，得到 $A_D=C_1\cup\{(\alpha,\ i,\ -L(D_i)),\ (i,\ \alpha,\ U(D_i))\ \forall(i,\ D_i)\in C_2\}$以及 A_R（A_D，其中所有弧的方向均已反转）。这些图可用于计算 SDTP 的下界和上界解，并以与 STP 相同的方式使用最短路径算法。如果在考虑这些全局边界时，G_D 或 G_R 中存在负环，则相应的 SDTP 实例必定不可行。

为每个时间点（i, D_i） $\in C_2$选定一个域 $d_i \in D_i$后，SDTP 就会简化为 STP。事实上，文献中一些专用算法正是利用这一问题结构来求解 SDTP 实例。第3节将对此作进一步讨论。

2.1 相关问题与分类

正如本文引言中所述，文献中曾以多种名称指称 SDTP。Schwalb和Dechter [27] 将其称为星类问题，因为每个时间点有多个域，从而与时间的起点 α产生关联，其形状类似一颗星。Kumar [14] 将该问题称为零一扩展简单时间问题，其中时间点域的子区间对应一个权重：若区间不可行，则权重为 0；若区间可行，则权重为 1。因此，当所选区间的权重之和为 $|C_2$时，便构造出了一个 SDTP 解。与此同时，Kumar [15] 提出了受限析取时间问题（RDTP），其中包含类型 1、2 和 3[2]的约束。因此，当不存在 3 型约束时，SDTP 就是 RDTP 的一个特例。Comin 和 Rizzi [6] 也将 SDTP 称为 t_2DTP，并将其视为仅包含类型 1 和 2 约束的 DTP。

为简化并统一术语体系，我们决定引入名称 简单析取时序问题 ，命名思路与 STP [8] 相同。下方的图1展示了 SDTP 在更广泛的 DTP 体系中的位置。

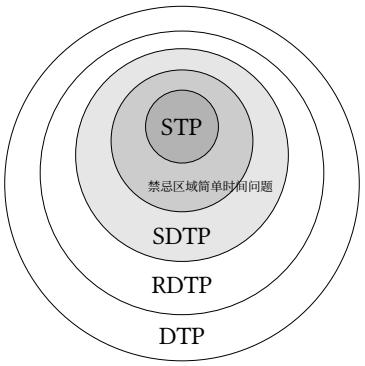


图1. 集合图中的 DTP 分类。

SDTP 推广了 STP，因为后者可以表示为一种 SDTP，其中 $|D_i| = 1, \quad \forall\, i \in T$。SDTP 还推广了禁忌区域简单时间问题，该问题包含瞬时事件和固定持续时间的过程（STPTE）[16]。这类问题与 SDTP 不同，因为 STPTE 定义的是不允许安排任何时间点的公共区间，而不是为每个时间点分别定义区间。这清楚地表明 SDTP 推广了 STPTE。然而，如果禁忌区域 STP 中的过程持续时间可以在某个区间内变化，那么 SDTP 就无法推广这类问题，因为这需要 3 型约束 [16]。最后，RDTP 推广了上述所有问题，而 DTP 又进一步推广了 RDTP。图1中的灰色区域表示本文所讨论的模型与算法所针对的问题。

另一个与 SDTP 相关的问题是时变简单时间问题 [22]。乍看之下，这种联系可能并不明显；在时变 STP 中，类型 1

[2]3 型约束考虑两个不同的时间点 $i, j \in T$, $i \neq j$并通过一个恰好包含两项的析取将它们关联起来，形式为 $(l'_i \leq s_i \leq u'_i) \vee (l'_j \leq s_j \leq u'_j)$，其中 $l'_i, u'_i, l'_j, u'_j \in \mathbb{R}$表示时间点 i或 j的界。本文不处理这类约束；感兴趣的读者可参阅 [6, 15]，了解更多信息。

约束中的权重 f并非常数，而是依赖于分配给时间点的值的函数 $f(s_i, s_j)$。当 $f(s_i, s_j)$ 是定义在全局边界$[L(D_j)\ U(D_j)]$,上的分段线性偏函数时，就可以将 SDTP 表述为时变 STP。在这种情况下，该函数定义于 α与每个 $j \in T$之间，即 $f(s_\alpha, s_j)$。函数 $f(s_\alpha, s_j)$的各个分段表示时间点 j的域。文献中此前尚未确立这种关系，原因之一可能是 Pralet 和 Verfaillie [22]最初更关注全函数，因为在他们的应用中，每个时间点都只有一个域。偏函数对算法开发的影响将在第3节进一步讨论。为了清晰呈现时序推理文献中常被放在一起讨论的问题之间的关系，即那些涉及析取项的问题，我们没有在图1中纳入时变 STP。尽管如此，我们所建立的联系对于计算 SDTP 的解具有重要意义。

2.2 约束规划模型

约束规划（CP）工具广泛应用于规划与调度领域。因此，值得考虑在实践中约束规划是否适合求解 SDTP。相应的约束规划模型为：

$$s_i - s_j \le w_{ij}, \quad \forall\ (i, j, w_{ij}) \in C_1 \tag{2}$$

$$\bigvee_{[l_i^k, u_i^k] \in D_i} (l_i^k \le s_i \le u_i^k), \quad \forall\ (i, D_i) \in C_2 \tag{3}$$

这实际上与定义2.2中的方程组相同。这非常方便，因为经典的 SDTP 定义与其约束规划形式之间本质上存在一一对应关系。为简单起见，我们将模型 (2)-(3) 称为 CP。

简化的约束规划形式可以写成如下形式：

$$s_i - s_j \le w_{ij}, \quad \forall\ (i, j, w_{ij}) \in C_1 \tag{4}$$

$$L(D_i) \le s_i \le U(D_i), \quad \forall\ (i, D_i) \in C_2 \tag{5}$$

$$s_i \notin \Phi_i, \quad \forall\ i \in T \tag{6}$$

约束 (4) 与方程 1 相同，而约束 (5) 则对时间点的全局边界进行建模。约束 (6) 是兼容性约束，用于替代析取约束 (3)。集合 Φ_i 包含属于区间 $[L(D_i)\ U(D_i)]$的 $i \in T$所有不可行赋值的枚举。换句话说：$\Phi_i = \{u_i^1+1,\ u_i^1+2,\ \ldots,\ l_i^2-1,\ u_i^2+1,\ \ldots,\ l_i^k-1\}\ k = |D_i|$。只有在额外假设$s_i \in \mathbb{Z}$的情况下，才能添加这些约束。然而，鉴于典型的约束规划工具仅处理整数变量，这一假设在实践中未必会造成限制。我们将由 (4)-(6) 定义的形式称为简化约束规划 （SCP）。

2.3 整数线性规划模型

整数线性规划（ILP）也常用于规划与调度领域，这促使我们也将 SDTP 表述为整数线性规划形式。首先，我们定义二元决策变量x_i^c：当解值 s_i属于域 $[l_i^c, u_i^c] \in D_i,\ (i,\ D_i) \in C_2$ 时，其取值为 1；否则取值为 0。SDTP 的相应整数线性规划模型如下：

$$s_i - s_j \leq w_{ij}, \quad \forall\, (i, j, w_{ij}) \in C_1 \tag{7}$$

$$l_i^c - M_i^L(1 - x_i^c) \leq s_i, \quad \forall\, (i, D_i) \in C_2,\ [l_i^c, u_i^c] \in D_i \tag{8}$$

$$s_i \leq u_i^c + M_i^U(1 - x_i^c), \quad \forall\, (i, D_i) \in C_2,\ [l_i^c, u_i^c] \in D_i \tag{9}$$

$$\sum_{[l_i^c, u_i^c] \in D_i} x_i^c = 1, \quad \forall\, (i, D_i) \in C_2 \tag{10}$$

$$x_i^c \in \{0, 1\}, \quad \forall\, (i, D_i) \in C_2,\ [l_i^c, u_i^c] \in D_i \tag{11}$$

约束（7）指的是简单时序约束（方程 1）。同时，约束（8）-（9）限制分配给解 s 的值，使其属于变量 x_i^c定义的有效界。请注意，约束（8）-（9）是 big-M 约束。对于每个 $(i, D_i) \in C_2$，可通过如下设置收紧这些约束：

$$M_i^L = \max_{[l_i^c, u_i^c] \in D_i} l_i^c - L(D_i)$$

$$M_i^U = U(D_i) - \min_{[l_i^c, u_i^c] \in D_i} u_i^c$$

约束 (10) 确保每个时间点 $i \in T$ 恰好选择一个域。最后，约束 (11) 限制 x_i^c变量只能取二进制值。请注意，SDTP 是一个可行性问题，这也解释了为什么该整数线性规划中没有目标函数。

这三种模型（整数线性规划、约束规划和 SCP）均可借助 IBM 的 CPLEX 等先进求解器来求解 SDTP。不过，这些求解器通常价格昂贵。此外，特定方法能够提供有关预期运行时间的保证，例如渐近多项式最坏情况时间复杂度。在下一节中，我们将介绍多种算法，这些算法在实践中可用于快速求解 SDTP。

3 算法

针对 SDTP，已有多种专用算法被提出。本节介绍的所有算法都将用于我们的计算实验。值得注意的是，所有算法都保证具有多项式渐近最坏情况时间复杂度。算法按发表时间的先后顺序介绍。在某些情况下，我们对算法进行了调整，以确保它们能够在实践中高效实现。因此，我们尽可能提供详细的实现细节。对于所有缺少细节的情况，我们建议有兴趣的读者查阅我们的代码，以作进一步了解。

我们假设每种算法都以一个 SDTP 实例作为输入，该实例包含网络 $N = (T, C)$ 及其关联图 G_D 和 G_R。有些算法还需要输入其他结构，我们会在必要时针对各个方法详细说明。所有算法都会返回一个解向量 s。当 SDTP 实例可行时，每个条目 s_i 都包含分配给 $i \in T$ 的时间；该时间与其他条目相结合后，可使解可行（网络 N 一致）。当 SDTP 实例不可行时，则返回 $s = \emptyset$。

3.1 上界-下界紧化

Schwalb和Dechter [27] 提出了上界-下界紧化 (ULT)算法，以处理析取时序问题中的一般析取项。ULT的初衷是收紧析取约束并简化析取时序问题实例。然而，Schwalb和Dechter [27] 首次证明，可以通过ULT算法在多项式时间内求解SDTP。

ULT算法处理两个变量之间的约束，并将其表示为区间。因此，第一步是定义集合 $H = \{(i, j) \ \forall (i, j, w_{ij}) \in C_1\} \cup \{(\alpha, i) \ \forall (i, D_i) \in C_2\}$。进一步假设$(i, j) \in H \implies (j, i) \notin H$。对于$(i, j) \in H$，边界集 B_{ij}定义为： $i \neq \alpha, B_{ij} = \{[-w'_{ji}, w'_{ij}]\}$，其中若$(i, j, w_{ij}) \in C_1$，则 $w'_{ij} = w_{ij}$；否则 $w'_{ij} = +\infty$；若$(j, i, w_{ji}) \in C_1$，则$w'_{ji} = w_{ji}$；否则 $w'_{ji} = +\infty$。同时，对于$(\alpha, i) \in H$，我们通过 $B_{\alpha i} = D_i$将 i与时间起点 α关联起来：$(i, D_i) \in C_2$。我们将分别用记号 $L(B_{ij})$和 $U(B_{ij})$表示 B_{ij}中的下界和上界。

算法 1 概述了 ULT 的工作方式。首先，在第 1 行初始化距离矩阵 δ。算法的主循环位于第 2–8 行。在第 3–4 行，根据每对 $(i, j) \in H$当前的界 B，更新距离矩阵 δ中的一些条目。随后，使用 Floyd–Warshall算法 [7]，以 δ中的当前值作为弧权重（第 5 行），通过计算全点对最短路径（APSPs）来更新矩阵 δ。第 6 行利用 δ中刚计算出的值创建临时边界集 B'。请注意，在实际实现中，我们不会创建 B'，因为需要时可以直接用矩阵δ 替代它（例如第 7 行）。第 7 行计算 B' 与 B 的交集。这里，我们遵循 Schwalb和Dechter [27] 提出的 ∩ 操作定义：该操作返回一个区间集合，其中的值同时为 B' 和 B所允许。只要 B中的界发生变化（由 Change操作表示），且没有界为空或不可行，ULT 就会持续迭代。通过在第 6–7 行后维护正确的标志，第 8 行的所有检查都可以在 $O(1)$时间内完成。同样，第 3–4 行可以在第 7 行的 ∩ 操作期间完成，而不会增加渐近最坏情况时间复杂度。第 9–11 行准备要返回的解 s。如果实例可行，则第 10 行将最早可行调度赋给 s；否则返回$\emptyset$。

算法 1ULT

1: $\delta_{ij} \leftarrow +\infty, \ \forall i, j \in T \cup \{\alpha\}$
2: **执行**
3: $\quad \delta_{ij} \leftarrow U(B_{ij}) \ \ \forall (i, \ j) \in H$ ▷ 使用新的界更新当前 δ 条目
4: $\quad \delta_{ji} \leftarrow -L(B_{ij}) \ \ \forall (i, \ j) \in H$
5: $\quad$ Floyd–Warshall算法(δ) ▷ 更新距离矩阵 δ
6: $\quad B'_{ij} \leftarrow \{[-\delta_{ji}, \delta_{ij}]\}, \ \forall (i, \ j) \in H$
7: $\quad B \leftarrow B \cap B'$ ▷ 收紧边界
8: **当**变化(B) **且**($B_{ij} \neq \emptyset$ **和** $L(B'_{ij}) \leq U(B'_{ij}) \ \ \forall (i, \ j) \in H$)
9: $s \leftarrow \emptyset$
10: **如果** $B_{ij} \neq \emptyset$ **和** $L(B'_{ij}) \leq U(B'_{ij}) \ \ \forall (i, \ j) \in H$ **则** $s_i \leftarrow L(B_{\alpha i}) \ \ \forall i \in T$
11: **返回** s

ULT 的渐近最坏情况时间复杂度为 $O(|T|^3|C|K + |C|^2K^2)$ [27],，而其空间复杂度为 $O(|T|^2)$，这是由距离矩阵 δ 导致的。尽管 ULT 的计算复杂度看起来很高，但它是一种多项式时间算法。此外，Schwalb和Dechter [27] 指出，即使问题实例在时间点之间的约束中包含多个析取项i, $j \in T$，因此并非 SDTP 实例，ULT 也可能成功消除足够多的析取项，将问题简化为 SDTP。在这种情况下，ULT 保证能够精确求解该问题。在我们的研究中，只有这一算法能够实现这种简化。

3.2 Kumar 算法

Kumar [14] 提出了一种多项式时间算法，用于求解零一扩展简单时间问题，它们本质上对应于 SDTP。算法 2 给出了*Kumar* 算法（KA算法）的伪代码概要。第 1 行通过在图 G_R 上计算全点对最短路径来构造距离矩阵 δ。此

步骤可以检测不可行情况，例如由 C_1 约束和全局边界构成的环中存在负环；在这种情况下，返回 $\delta = \emptyset$。

矩阵 δ 可以通过以下方式计算：(i) Floyd–Warshall算法，(ii) 重复调用 Bellman-Ford，或 (iii) Johnson算法 [7]。Kumar [14] 未指定计算 δ 时应采用哪种方法，因此我们将考虑选项 (ii) 和 (iii)。由于选项 (i) 在我们的初步实验中整体表现较差，因此不予考虑。

算法 2KA算法

1: $\delta \leftarrow$计算距离矩阵(G_R)
2: **如果** $\delta = \emptyset$，**则返回** $\emptyset$
3: $G_C \leftarrow$CreateConflictGraph(δ, C_2) ▷图 $G_C = (E, A_C)$
4: **如果** $G_C = \emptyset$ **则返回** $\emptyset$
5: $G_B \leftarrow$ 创建二分图(G_C) ▷图 $G_B = (E, E', A_B)$，其中 E' 是 E的副本
6: $G_F \leftarrow$SolveMaxFlow(G_B) ▷从源点 θ_1 到汇点 θ_2，其中 G_F对应于残量图
7: $S \leftarrow \{(\theta_1, e_i^c) : e_i^c \notin R(G_F, \theta_1)\} \cup \{(e_j^{k\prime}, \theta_2) : e_j^{k\prime} \in R(G_F, \theta_1)\}$ ▷最小割于 G_F
8: $S' \leftarrow \{e_i^c : (\theta_1, e_i^c) \in S \vee (e_i^{c\prime}, \theta_2) \in S\}$ ▷ G_C的顶点覆盖
9: $S'' \leftarrow E \backslash S'$
10: **如果** $|S''| \neq |T|$，**则返回** $\emptyset$
11: UpdateGraph(G_R, S'')
12: $s \leftarrow$ Bellman-Ford(G_R, α) ▷求解 STP
13: **返回** s

第 3 行根据 SDTP 中的域创建冲突图 $G_C = (E, A_C)$。首先，定义区间集 E为 $E = \{e_i^c : (i, D_i) \in C_2, [l_i^c, u_i^c] \in D_i\}$。因此，每个元素 $e_i^c \in E$恰好表示一个时间点的域。如果某个域 $[l_i^c, u_i^c]$ 会产生大小为 1 的冲突，则它在E 中没有对应元素，也就是说，以下条件成立：

$$\delta_{i\alpha} + u_i^c < 0 \ \vee \ \delta_{\alpha i} - l_i^c < 0$$

创建顶点集 E后，即可定义弧集 A_C。弧 $(e_i^c, e_j^k) \in A_C$表示两个时间点域 $[l_i^c, u_i^c]$和 $[l_j^k, u_j^k]$之间的大小为 2 的冲突。此类冲突发生于：

$$u_i^c + \delta_{ij} - l_j^k < 0$$

请注意，大小为 2 的冲突也定义在同一时间点 $i \in T$ 的域之间。由于 $\delta_{ii} = 0$ 和 $u_i^c < l_i^{c+1}$，因此总是存在冲突 $(e_i^c, e_i^{c+1}) \in A_C$（回想一下第 2 节，域按升序排列）。

过程 CreateConflictGraph 返回图 G_C或 $\emptyset$。每当某个时间点 $i \in T$的所有域都产生大小为 1 的冲突时，就会返回后者。在这种情况下，没有与 i 关联的域会被纳入 E，这意味着该 SDTP 实例不可行。构建图 G_C后，第 5 行通过将每个元素 $e_i^c \in E$ 复制到$e_i^{c\prime} \in E'$，创建二分图 $G_B = (E, E', A_B)$。对于每个 $(e_i^c, e_j^k) \in A_C$，我们创建一条弧 $(e_i^c, e_j^{k\prime}) \in A_B$。$A_B$ 中的所有弧都将 E 的元素连接到 E' 的元素。

第 6 行将 G_B 上的最大二分匹配问题转化为最大流问题（max-flow），并生成残量图 G_F [7]。为了以最大流的形式求解该问题，我们向 G_B引入一个源节点 θ_1和一个汇节点 θ_2。将弧 $(\theta_1, e_i^c) \ \forall e_i^c \in E$和弧 $(e_j^{k\prime}, \theta_2) \ \forall e_j^{k\prime} \in E'$与 A_B中的所有弧一并加入图中。此外，所有弧的容量均设为单位容量。然后，只需求解从 θ_1到 θ_2的最大流，即可生成 G_F。

最小割 S根据最大流最小割定理在 G_F中计算得出（第 7 行）。$R(G_F, \theta_1)$表示从源点 θ_1在 G_F中可到达的节点集合（即存在一条残余容量为正的路径）。第 8 行将 S中的节点副本合并，以创建 S'，它是

G_C 将其视为无向图时的顶点覆盖。由于 S' 是顶点覆盖，如果我们取 E 中不属于 S' 的所有元素来创建集合 S''（第 9 行），那么 S'' 中任意两个元素之间都不会存在冲突。换言之，S'' 中的所有域都可以属于一个可行的 SDTP 解。

如果 $|S''| = |T|$，那么每个时间点都恰好分配有一个域，即 $S_i'' = e_i^c, \forall\, i \in T$。如果 $|S''| < |T|$，则该实例不可行（第 10 行）。第 11 行继续使用关于各时间点选定域的信息 S'' 更新图 G_R：

$$(\alpha, i, w_{\alpha i}) \in A_R \implies w_{\alpha i} = U(S_i'')$$
$$(i, \alpha, w_{i\alpha}) \in A_R \implies w_{i\alpha} = -L(S_i'')$$

其中 $U(S_i'') = u_i^c$，且 $L(S'') = l_i^c$。由于 SDTP 现已简化为一个可行的 STP，因此使用标准 Bellman-Ford 算法计算最终解 s（第 12 行）。

请注意，在我们的实现中，我们不会显式创建图 G_C 和 G_B。相反，我们直接创建最大流图 G_F。SolveMaxFlow 也会修改此图以生成残量图。采用这种方法，我们既缩短了 KA算法 的执行时间，也减少了它所需的内存量。不过，从概念上讲，逐步说明每个图的创建过程，更容易解释 KA算法 的工作原理。

Kumar [14] 没有给出 KA算法的渐近最坏情况时间复杂度，而是认为 KA算法在多项式时间内运行，因为每一步都可以在多项式时间内完成。因此，为了完整起见，我们现在将明确分析 KA算法的时间复杂度。决定 KA算法复杂度的三个算法组件位于第 1、3 和 6 行。算法的其他部分所需时间都不会超过这三个主要组件中的任何一个。

如果使用重复的 Bellman-Ford 计算，第 1 行所需时间为 $O(|T|^2|C|)$；如果使用 Johnson算法 计算，并且 Dijkstra 算法 [7] 采用斐波那契堆 [9]实现，则所需时间为 $O(|T||C| + |T|^2 \log |T|)$。然而，由于指针操作会导致较差的缓存局部性和性能 [4, 17]，斐波那契堆在实践中通常效率不高。因此，我们在实现中选择使用序列堆 [24]，这会将渐近最坏情况时间复杂度提高到 $O(|T||C| \log |T|)$，但能改善算法在实际环境中的性能。第 3 行的复杂度为 $O(\omega^2)$，因为我们需要检查 E 中的每一对区间，以及 $|E| = O(\omega)$。第 6 行求解一个最大流问题。尽管有许多算法可以求解最大流 [10],，我们选择使用 Dinic 算法；当应用于最大二分匹配的图时，其复杂度为 $O(\omega^{\frac{5}{2}})$。我们观察到，最大流并不是 KA算法中的瓶颈。实际上，第 1 行和第 3 行是最耗时的步骤（完整讨论见第 5 节）。

在本文余下部分中，我们将采用重复运行 Bellman-Ford 的 KA 版本称为 KAB，将采用 Johnson算法 的版本称为 KAJ。泛指同时涵盖 KAB 和 KAJ 的算法时，我们将称其为 KA。KAB 的复杂度为$O(|T|^2|C| + \omega^{\frac{5}{2}})$，而 KAJ 的复杂度为 $O(|T||C| \log |T| + \omega^{\frac{5}{2}})$。同时，由于距离矩阵 δ和图 G_F，KA 的空间复杂度为$O(|T|^2 + \omega^2)$。

3.3 Comin-Rizzi 算法

Comin 和 Rizzi [6] 提出了渐近速度更快的算法来求解 SDTP 和 RDTP，使他们的方法成为这两个问题当前最先进的算法。对于 SDTP，他们提出了一种类似于用于全点对最短路径的 Johnson算法的算法。该方法首先使用 Bellman-Ford 检测负环，随后迭代使用 Dijkstra算法修正图中不含负环时的计算结果。然而，直到现在，还没有针对该方法开展实验研究。

用于 SDTP 的 *Comin-Rizzi* 算法 （CRA）详见算法 3。CRA 首先仅考虑 s^0 约束，计算初始最早可行解 C_1。在我们的实现中，我们通过使用第 2 节中定义的全局边界来部分考虑 C_2 约束，并在 G_D 中进行计算。对 s^0 的计算随后要么得出最早可能的解，要么证明该解不存在，原因是：（i）由 C_1 约束形成了负环；或（ii）无法在满足全局界限 $[L(D_i)\ U(D_i)]$的同时，为至少一个时间点 $i \in T$指定时间 s_i。

如果 $s^0 \neq \emptyset$，那么 CRA 会进入主循环。首先，将当前解 s_i^0不属于任一域 D_i 的每个时间点 $i \in T$ 添加到需要修正的赋值列表 F 中。当 F中仍有元素时，重复执行以下步骤（第 5–12 行）。从 F 中移除一个时间点 i（第 6 行）。首次从 F中移除 i时，我们会计算距离矩阵 δ中从 i到包含仅有 C_1 约束的底层图 $G_R^{1\prime}$中所有其他节点的条目 δ_i（第 7–9 行）。在该图中，每条弧 $(i,\ j,\ w_{ij}) \in A_R$的权重w_{ij}会被修改为 $w'_{ij} = w_{ij} + s_j^0 - s_i^0$。Comin 和 Rizzi [6] 证明了 $G_R^{1\prime}$不可能包含负环，因为始终有 $w'_{ij} \geq 0$成立。因此，可以使用 Dijkstra 算法而非 Bellman-Ford 来计算距离 δ_i，从而大幅提升 CRA 的性能。每个条目 δ_i 只计算一次，因为在 CRA 的执行过程中 $G_R^{1\prime}$保持不变。

算法 3CRA

```
1:  s^0 ← Bellman-Ford(G_D, α)                          ▷ 使用 SDTP 全局边界求解 STP
2: 如果 s^0 = ∅ 则返回 ∅
3:  s ← s^0
4:  F ← {i : (i, D_i) ∈ C_2 ∧ s_i ∉ D_i}                ▷ 所有时间点的集合 i ∈ T 其赋值 s_i 不可行
5: 当 F ≠ ∅ 且 s ≠ ∅ 且 s_i ≤ U(D_i) ∀(i, D_i) ∈ C_2 时
6:      i ← 弹出(F)
7:      如果 δ_i 尚未计算，则
8:          δ_i ← Dijkstra 算法(G_D^{1\prime}, i)          ▷ 对 δ进行惰性计算
9:      结束 if
10:     UpdateAssignments(s, s^0, i, δ_i)
11:     F ← {i : (i, D_i) ∈ C_2 ∧ s_i ∉ D_i}
12: 结束 while
13: 返回 s
```

对于第 6 行中从 F 取出的每个 i，我们通过 UpdateAssignments 过程（第 10 行）将赋值更新为 s_i。首先，该过程执行以下操作

$$s_i \leftarrow \lambda(s_i, D_i) \ : \ (i, D_i) \in C_2$$

其中，$\lambda(s_i,\ D_i)$ 是一个函数，它返回满足 $s_i < l_i^c$ 的第一个域（按升序排列）中的值 l_i^c $[l_i^c, u_i^c] \in D_i$；如果不存在这样的域，则返回 $\perp$。每当 $\lambda(s_i,\ D_i) = \perp$ 时，CRA 就会停止计算，因为这证明该实例不可行。在这种情况下，UpdateAs-signments 将 $s = \emptyset$ 设为相应值。或者，如果 $\lambda(s_i,\ D_i) \neq \perp$，那么新的赋值 s_i 可能会导致其他时间点分配发生变化，因为 s_i 必然已经增大。为了正确传播这些变化，Comin 和 Rizzi [6] 引入了以下更新规则

$$\rho_{ij} \leftarrow \delta_{ij} + (s_j - s_j^0) - (s_i - s_i^0), \qquad \forall\ j \in P(G_D^1, i)$$

$$s_j \leftarrow s_j + \max(0, \lambda(s_i, D_i) - s_i - \rho_{ij}), \qquad \forall\ j \in P(G_D^1, i)$$

其中，$P(G_D^1,\ i)$ 表示所有可从 i 在 G_R^1 中到达的节点 $j \in V$ 的集合。换言之：在 G_D^1 中存在一条从 i 到 j 的路径。这些更新规则对每个 $j \in P(G_D^1,\ i)$ 可在 $O(1)$ 时间内应用，或总共在 $O(|T|)$ 时间内应用。

将赋值固定为 i 以及可能的其他时间点后，CRA 会构造一个新列表 F（第 11 行）。一旦 $F=\emptyset$，s 中的赋值便是可行的，并对应最早可行解。随后在第 13 行返回该赋值。对于不可行实例，则返回 $s=\emptyset$。

使用斐波那契堆进行 Dijkstra 算法计算时，CRA 的渐近最坏情况时间复杂度为 $O(|T||C|+|T|^2\log|T|+|T|\omega)$。改用序列堆时，渐近复杂度会增加到$O(|T||C|\log|T|+|T|\omega)$，但实测性能显著提升 [24]。无论采用哪种堆实现，由于距离矩阵 δ，CRA 的空间复杂度为 $O(|T|^2)$。

在 CRA 的原始描述中，Comin 和 Rizzi [6] 预先计算了距离矩阵 δ，然后才开始算法 3 的主循环。在我们的实现中，由于只有在绝对必要时才会计算 δ 中的条目（第 7–9 行），因此我们将此计算描述为惰性计算。尽管两种方法的渐近最坏情况时间复杂度相同，但在实践中，惰性计算的性能显著更好，因为它避免了许多不必要的计算。此外，我们还将第 11 行中列表 F 的创建并入 UpdateAssignments 过程。每当时间点 $j\in T$ 的赋值s_j 发生变化时，我们都会检查是否应将 j 加入 F或从中移除。这样便无需在主循环的每次迭代中（第 5–12 行）重建列表 F，从而加快计算。

3.4 缩减上下界收紧

缩减上下界收紧 (RULT)方法是 ULT 的一种加速方法，专门针对 SDTP。利用 SDTP 的结构，可以很容易地从 ULT 推导出 RULT。回想一下，在 ULT 中，我们必须使用 Floyd–Warshall算法计算全点对最短路径，其复杂度为 $O(|T|^3)$，因为 Schwalb和Dechter [27] 假设输入是一般的析取时序问题，即两个时间点之间的每个约束可能包含多个析取项 $i,\ j\in T$。

然而，SDTP 的结构特点是，T中的时间点之间仅包含简单时序约束。因此，只需计算两次单源最短路径：第一次确定每个时间点的最早可行分配，第二次确定每个时间点的最晚可行分配。这样，每个时间点都会对应一个区间，用于表示其赋值可能更紧的全局边界。与 ULT 类似，我们可以利用这个全局边界，在每次迭代中缩减 C_2 析取项，从而减少析取项的数量。

算法4概述了 RULT。首先，使用每个时间点的域初始化边界集 B（第 1-2 行）。与 ULT 不同，我们只需为每个 $i\in T$维护边界，而不必为每个约束维护边界。只要 B 发生变化且界仍然可行，主循环（第 3-11 行）就会持续运行。每次迭代中，使用 UpdateGraph 修改图 G_D，将与 α相连的弧的权重替换为：

$$(\alpha,i,w_{\alpha i})\in A_D \implies w_{\alpha i}=-L(B_i)$$
$$(i,\alpha,w_{i\alpha})\in A_D \implies w_{i\alpha}=U(B_i)$$

对 G_R也执行相同的过程，但从 α出发的弧取上界 $U(B_i)$，而入弧取下界 $-L(B_i)$。在 RULT 执行期间，值 $L(B_i)$ 单调不减，而 $U(B_i)$ 单调不增。因此，更新图会收紧每个时间点 $i\in T$的全局边界B_i。

第 6—7 行在更新后的图上计算最早可行调度 p 和最晚可行调度 q。如果 $p=\emptyset$ 或 $q=\emptyset$，则该实例不可行，因为即使放宽所有时间点的全局边界，仍然存在负环（第 8 行）。否则，第 9 行构造集合 B'，第 10 行计算 B和 B'的交集。操作 $\cap$与 ULT 中使用的操作相同，其定义见 Schwalb和Dechter [27]。最后，第 12—14 行准备要返回的解 s。如果边界在

算法4RULT

```
1:  B_i ← {[−∞, +∞]}  ∀ i ∈ T
2:  B_i ← D_i,  ∀ (i, D_i) ∈ C_2
3: 执行
4:      UpdateGraph(G_D, B)                          ▷ 更新与 α相连的弧权重
5:      G_R UpdateGraph(G_R, B)
6:      p ← Bellman-Ford(G_D, α)                     ▷ 最早可行分配
7:      q ← Bellman-Ford(G_R, α)                     ▷ 最晚可行分配
8:      如果 p = ∅ 或 q = ∅，则返回 ∅
9:      B'_i ← {[−p_i, q_i]}  ∀ i ∈ T
10:     B ← B ∩ B'                                   ▷ 收紧边界
11: 当 Change(B) 且 L(B_i) ≤ U(B_i)  ∀ i ∈ T
12:  s ← ∅
13: 如果 L(B_i) ≤ U(B_i)  ∀ i ∈ T，则 s_i ← L(B_i)  ∀ i ∈ T
14: 返回 s
```

B 可行，则第 13 行为每个时间点分配其最早可行值。如果需要最新可行解，则可以在第 13 行将 $U(B_i)$分配给 s_i。

RULT 的正确性可直接由 ULT [27] 的正确性推出，同时还需考虑到 C_1 约束是固定的，且只需考虑 C_2中的区间。RULT 的渐近最坏情况时间复杂度与 ULT 类似。考虑到使用 Bellman-Ford 而非 Floyd–Warshall算法进行最短路径计算所带来的效率提升，RULT 的复杂度为：$O(|T|^2|C|K + |T|^2|K|^2)$。由于只需额外分配大小为 $|T|$ 的向量，RULT 的空间复杂度降低至 $O(|T|)$。请注意，在我们的实现中，并未显式维护边界集 B。

3.5 使用域检查的 Bellman-Ford

迄今为止所述的所有算法都在执行过程中使用过 Bellman-Ford。这并不奇怪，因为 Bellman-Ford 可以相当高效地实现，用于检测负环 [3], ，而这是求解 STP、SDTP 和 RDTP 时的一项核心任务。因此，考虑对原始算法进行变体以求解 SDTP，似乎顺理成章。下面定义带域检查的 *Bellman-Ford*（BFDC），它对 Bellman-Ford 做了少量修改，以处理最短路径计算中不可行值之间的间隙。我们的方法借鉴了先前关于时间问题的研究 [2, 6, 22]。

算法 5 描述了完整的 BFDC 过程，其主要在图 G_D 上运行。第 1–5 行负责初始化辅助变量，包括距离数组 τ、路径长度数组π（用于计算从 α 到 $i \in V$ 的最短路径所包含的节点数）、域索引数组 z（用于保存每个时间点 $i \in T$当前的域索引 z_i），以及 Bellman-Ford 中使用的先进先出队列 Q。初始化完成后，主循环开始（第 6–20 行）。在第 7 行，从队列中移除一个元素 i，并在第 8 行检查其域。DomainCheck过程的详细内容见算法 6。如果 DomainCheck 能证明该 SDTP 实例不可行，则将其设为 $\tau = \emptyset$。否则，该过程会视情况更新对 τ、 π 和 z 的赋值。算法继续执行至第 9 行；如果此时尚未证明该实例不可行，则松弛从 $i \in V$出发的所有出弧，并传播最短路径（此处松弛一词沿用 Cormen 等人 [7] 的术语）。

对于 i 在 A_D 中的每条出弧（回顾图 $G_D = (V, A_D)$），第 10 行会检查是否应更新至与 j 相邻的 i 的当前最短路径；如果需要，算法还会更新 π_j，并可能更新队列 Q。在第 14 行，如果到 j 的路径形成一个环，或者该路径返回到 α，则判定该实例不可行。如果 j 尚未在队列 Q 中，我们就在第 15 行将其加入

算法 5BFDC

```
1:  τ_i ← +∞,  ∀ i ∈ T ∪ {α}
2:  π_i ← 0, ∀ i ∈ T ∪ {α}
3:  z_i ← 1, ∀ i ∈ T ∪ {α}
4:  Q ← Push(Q, α)
5:  τ_α ← 0
6:  当 Q ≠ ∅ 且 τ ≠ ∅ 时执行
7:      i ←弹出(Q)
8:      域检查(i, τ, z, π)                                   ▷ 算法 6
9:      如果 τ ≠ ∅ 则
10:         对于(i, j, w_ij) ∈ A_D执行                       ▷ Bellman-Ford 的标准松弛阶段
11:             如果 τ_j > τ_i + w_ij 则
12:                 τ_j ← τ_i + w_ij
13:                 π_j ← π_i + 1
14:                 如果 π_j ≥ |T| 或 j = α 则 τ ← ∅          ▷ 检查负环
15:                 如果 j ∉ Q 则 Q ← 推入(Q, j)
16:             结束如果
17:             如果 τ = ∅ 则跳出
18:         结束循环
19:     结束条件判断
20: 结束循环
21: 如果 τ ≠ ∅ 则 s_i ← −τ_i, ∀ i ∈ V否则 s ← ∅
22: 返回 s
```

（不允许出现重复元素）。当实例已被证明不可行时，第 17 行会中止 for 循环（第 10–18 行）。

只要 Q 中仍有元素且 τ 不等于 $\emptyset$，主循环就会继续运行。一旦其中一个条件不成立，算法 5 就会继续执行第 21 行。如果实例可行，则使用存储在 τ 中的最短路径值创建赋值 s；否则返回 $\emptyset$。

DomainCheck过程（算法 6）体现了 Bellman-Ford 与 BFDC 之间的主要区别。第 1 行会检查当前赋给 i 的时间点 $s_i = -\tau_i$ 是否属于索引为 z_i 的当前域。如果赋值的时间不超过域的上界 $u_i^{z_i}$，DomainCheck 就会直接终止。否则，算法会按递增顺序搜索第一个包含 $s_i = -\tau_i$ 的域（第 2 至 8 行）。找到域后，第 4 至 5 行会相应地更新 τ_i 和 π_i 的赋值。如果 $s_i = -\tau_i$ 超出 D_i 中的所有域，那么我们就证明了该实例不可行（第 9 行）。

引理 3.1. 算法 *5* 正确无误，并返回以下结果之一：*(i)* 最早可行解；或 *(ii)* 不存在可行解的证明。

证明。首先注意，在 BFDC 中，τ 始终单调不增。这意味着 SDTP 解$s = -\tau$ 单调不减。在每个松弛阶段，BFDC 都会为 G_D 中的一个子集赋予最短路径，因此会为一部分时间点赋予最早可行值。每当域检查阶段因 $-\tau_i > u_i^{z_i}$而必须增大对 z_i 的赋值时，它会赋予最早可行的域，并使 τ_i 减小，或保持 τ_i 不变（算法 6 第 4–5 行）。值 τ_i单调不增，因此减小对 z_i 的赋值永远不会得到可行解。因此，z 在 BFDC 中也单调不减，这意味着域赋值是一种无回溯搜索。

牢记这些事实后，我们现在可以说明 BFDC 结束时存在两种可能。如果 $\tau \neq \emptyset$那么 $s = -\tau$ 就是该 SDTP 实例的最早可行解。这是因为 τ包含了 G_D 中从 α到每个其他节点 $i \in V$的最短路径，而这是通过使用域的最小可行赋值 z实现的。当 $\tau = \emptyset$ 时，我们要么已经用尽了

将赋值 z_i 分配给时间点 $i \in T$，这意味着 BFDC 已经耗尽了 i（$z_i > |D_i|$）的域；要么由 C_1 约束构成的负成本环已在松弛阶段检测到（算法 5 第 14 行）。□

算法 6域检查

输入： 时间点 i，距离数组 τ，域索引数组 z，路径长度数组 π

如果 $-\tau_i > u_i^{z_i}$ **then** ▷域 $[l_i^{z_i}, u_i^{z_i}] \in D_i$
  对于 $z_i = z_i + 1$，**直到** $|D_i|$，**执行**
    如果 $-\tau_i \leq u_i^{z_i}$**那么**
      $\tau_i \leftarrow \min\{\tau_i, -l_i^{z_i}\}$
      如果 $\tau_i = -l_i^{z_i}$**那么** $\pi_i \leftarrow 1$
      跳出
    结束如果
  结束循环
  若 $-\tau_i > U$（D_i）**则** $\tau \leftarrow \emptyset$ ▷没有任何域能够容纳当前 τ_i 赋值
结束 if

可以证明，引理 3.1 对逆序情形同样成立：即生成最新可行解。为此，需按降序对域进行排序，并在图 G_R 上而非 G_D 上进行计算。这需要对算法 6 做少量修改，以适应域的逆序排列，而算法运行方式的一般推理保持不变。若最新可行解 s 存在，则可通过 $s = \tau$ 直接获取。现在我们来讨论 BFDC 的渐近最坏情况时间复杂度，该复杂度由引理 3.2 给出。

引理 3.2. *BFDC* 会在与 O（$|T||C| + |T|\omega$）成正比的迭代次数内停止。

证明。首先，注意到在同一图 G_D 上，Bellman-Ford 的复杂度为 $O(|T||C|)$。加入域检查不会改变队列 Q 的大小，因此总体迭代次数仍与标准 Bellman-Ford 相同。区别在于每次迭代各自的计算开销。

使用先进先出队列的 Bellman-Ford 至多有 $O(|V|)$ 个阶段。在每个阶段中，一个节点从 Q 中提取的次数至多为一次 [1]。换句话说，算法 5 第 7 至 11 行中的操作在每个阶段中执行的次数至多为 $|V|$。这些操作的复杂度相当于 $O(\text{OutDeg}(i) + |D_i|)$，其中 $\text{OutDeg}(i)$ 表示集合 A_D 中以 i 为源点的弧的数量。因此，每个阶段的复杂度为 $O(\sum_{i \in V} \text{OutDeg}(i) + |D_i|)$，相当于 $O(|C| + \omega)$。总的来说，复杂度为 $O(|V||C| + |V|\omega)$；由于 $|V| = |T|$，求解 SDTP 时该复杂度相当于 $O(|T||C| + |T|\omega)$。□

BFDC 的空间复杂度为 $O(|T|)$。BFDC 中使用的辅助数组 τ、s、π、z 和队列 Q 都需要额外的空间，其大小与 $|T|$ 成正比。

3.6 渐近最坏情况复杂度

现在，我们来评估所有算法的理论复杂度，并对实证结果应有的预期得出一些初步结论。表 1 根据我们的实现，列出了每种算法的渐近最坏情况时间复杂度和空间复杂度。鉴于整数线性规划、约束规划和 SCP 模型通常借助通用黑盒求解器求解，我们决定不在分析中纳入它们的理论复杂度。

就最坏情况时间复杂度而言，BFDC 显然优于所有其他方法。CRA 是第二快的方法。与此同时，由于 KA 和 RULT 的复杂度项各不相同，且彼此可能占主导地位，因此很难对二者进行排名。请注意，$\omega = O(|T|K)$，因此无论何时

表 1. 各算法的渐近最坏情况复杂度。

算法	时间复杂度	空间复杂度
ULT	$O(\|T\|^3\|C\|K + \|C\|^2K^2)$	$O(\|T\|^2)$
KAB	$O(\|T\|^2\|C\| + \omega^{\frac{5}{2}})$	$O(\|T\|^2 + \omega^2)$
KAJ	$O(\|T\|\|C\|\log\ \|T\| + \omega^{\frac{5}{2}})$	$O(\|T\|^2 + \omega^2)$
CRA	$O(\|T\|\|C\|\log\ \|T\| + \|T\|\omega)$	$O(\|T\|^2)$
RULT	$O(\|T\|^2\|C\|K + \|T\|^2K^2)$	$O(\|T\|)$
BFDC	$O(\|T\|\|C\| + \|T\|\omega)$	$O(\|T\|)$

$\omega \leq (|T|K)^{\frac{4}{5}}$KA 中的第二项（最大流）复杂度始终不会比 RULT 的第二项慢。在这种情况下，我们可以将比较范围限制在涉及最短路径的第一项上。显然，在这方面，KAB 和 KAJ 的渐近速度都快于 RULT。然而，当 $\omega \approx |T|K$ 由于最大流阶段的影响，KA 的时间复杂度低于 RULT。正如前文所述，我们还可以看到，在 KA 中使用 Johnson算法降低了其时间复杂度，使 KAJ 更接近 CRA。最后，表 1 中 ULT 的速度最慢，主要是因为它大量使用了 Floyd–Warshall算法。

表 1 中记录的时间复杂度说明了解决 SDTP 所面临的挑战。尽管它们与最短路径问题密切相关，但负环和析取域的存在要求采用更复杂、更精细的技术。特别是，我们希望强调文献中大多数既有技术的空间复杂度有所增加。只有 RULT 和 BFDC 能够使用线性空间解决 SDTP。尽管鉴于计算资源的可用性，这一点看似并不重要，但二次内存开销在实践中很快就可能变得难以承受。鉴于 SDTP 似乎是其他更复杂问题的子问题，而这些问题本身也需要占用相应的内存，这通常会带来问题。我们将在第 5 节稍后讨论内存使用的影响。

最后要说明的是第 2 节中建立的 SDTP 与时变简单时间问题之间的关系。Pralet 和 Verfaillie [22] 展示了如何在时间$O(|T||C|)$内求解时变简单时间问题。然而，尽管这两个问题存在关联，表 1 显示，在最坏情况下，求解 SDTP 所需的渐近时间比求解时变简单时间问题更长。这主要是由于时间点域不连续，导致简单时序差分问题的解中的某些赋值不可行，因此需要额外的步骤来修正赋值并重新检查可行性。当每个时间点至多只有一个域时（$K \leq 1$），则无需进行此类修正。

4 实验

实验在一台运行 Ubuntu 20.04 LTS 的计算机上进行，该计算机配备两颗主频为 2.60GHz 的 Intel Xeon E5-2660v3 处理器，共有 160 GB RAM、5 MB L2 缓存和 50 MB L3 缓存。为避免对实验产生负面影响，始终禁用了 Intel 的超线程技术。所有算法均使用 C++ 实现，并使用 GNU GCC 9.3 和优化标志 -O3 编译。整数线性规划、约束规划和 SCP 模型均使用 CPLEX 12.9 的 C++ API 实现。各方法在执行期间均只允许使用一个线程。

我们的实验主要侧重于测量实际计算时间。为获得准确的时间测量结果，我们使用 C++ 的 std::steady_clock 测量 CPU 时间。为了尽可能公平地比较输入表示方式差异显著的方法，我们决定仅记录求解实例所需的计算时间。这意味着我们的结果不包含输入所需的时间信息，

也不包含某些算法将数据转换为更适合格式时所进行的输出或预处理所需的时间。同样，构建整数线性规划、约束规划和 SCP 模型所需的时间也未计入相应结果。

虽然我们明白，依据计算时间评估方法并非总是理想的 [13, 19], ，但要为基本操作和组成部分差异如此之大的算法找到一个统一的评估指标并不容易。此外，Moret 和 Shapiro [21] 指出，对于易解问题，算法的运行时间通常是合理的指标。

由于我们提出了首个用于评估求解 SDTP 的算法的实验研究，因此引入了四个数据集，用以评估各种方法及其实现的性能。这四个数据集在问题结构上各不相同，尤其是底层距离图有所差异。实例分为最短路径实例、负环实例、车辆路径实例 和超大实例。所有实例仅包含整数值。因此，无需作任何改动即可纳入所有方法，包括约束规划和 SCP。我们将首先详细说明生成各实例集的流程，然后介绍实验所得的计算结果。

4.1 最短路径实例

实例由用于最短路径（SP）问题的图生成器创建。通过设置 $T = V$、 $C_1 = A$，并根据图 G 中的最短路径推导出时间点域的 C_2 约束，可将图 $G = (V, A)$ 转换为析取式差分约束时间网络 $N = (T, C)$。对于一个 SDTP 实例，我们定义以下参数：时间点数量 $|T|$、类型 1 约束数量 $|C_1|$、包含多个域的元素数量 $|T_D|$，以及每个 $i \in T_D$ 的域数量 $K > 1$，其中 $|D_i| = K$。根据图 G 或约束 C_2 的生成方式，SP 实例分为四组。下面概述各组的生成过程（更多详情见附录 A）。

(1) Rand：使用 SPLib [4] 中介绍的 Sprand 生成图 G。节点和弧均随机生成。根据从虚拟节点到每个 $i \in V$ 的最短路径生成约束 C_2。(2) Grid：使用 SPLib [4] 中同样介绍的 Spgrid 生成图 G。节点按网格形式生成，包含 X 层，每层有 Y 个节点。弧连接同一层内的节点以及后续层中的节点。C_2 约束的生成方式与 Rand 相同。(3) Seq：使用专门设计的生成器 Spseq 生成图 G。节点以类似 Sprand 的方式随机生成。创建一条连接所有节点且包含 $|V| - 1$ 条弧的路径，所有弧的权重均为 $w_{ij} = 1$。随后随机生成其余 $|A| - |V| + 1$ 条权重较大的弧。由此得到一条已知的最短路径，某些方法可能难以找到该路径。C_2 约束的生成方式与 Rand 和 Grid 相同。(4) Late：使用 Sprand 或 Spseq 生成图 G。生成 C_2 约束时，确保至少 60% 的最早可行解 s_i 属于相应时间点的最后一个域。

对于这四个数据集中的每一个，我们还创建了四个子集，以评估哪些实例关键特征对算法性能的影响最大。对于每个子集，我们固定一个 SDTP 的三个参数，然后改变第四个参数。

4.1.1 Nodes 数据集。 数据集 Nodes 的时间点数量 $|T|$ 在 {100,200,. . . ,12800,25600} 范围内变化。其他参数固定为 $|C_1| = 6 \cdot |T|$、 $|T_D| = 0.8 \cdot |T|$ 和 $K = 10$。对于数据集 (1)–(4) 与时间点数量 $|T|$ 的每种组合（以下简称一种配置），均生成了五个实例：三个可行实例和两个不可行实例。例如，对于配置 (Rand, Nodes, $|T| = 100$)，生成了五个实例。

图2给出了 Nodes 子集的结果。每张图均根据每个数据集的时间点数量，报告各方法的平均计算时间。所报告的数值是20次运行的平均值，以减小因求解 SDTP [21] 所需计算时间较短而产生的异常值影响。此外，为各方法设置了两秒的时间限制。如果某方法对给定规模的所有实例均超时，则为清晰起见省略这些结果。这解释了部分图中曲线不完整的原因。不过，如果对于某些实例规模，某方法至少能够求解其中一个实例（五个实例中的一个），我们仍会报告包含潜在超时的平均值。KAJ 很少优于 KAB。基于这些结果，我们决定仅展示 KAB 的结果。在实验中，如有必要，我们将对这两种方法之间的具体差异进行说明。

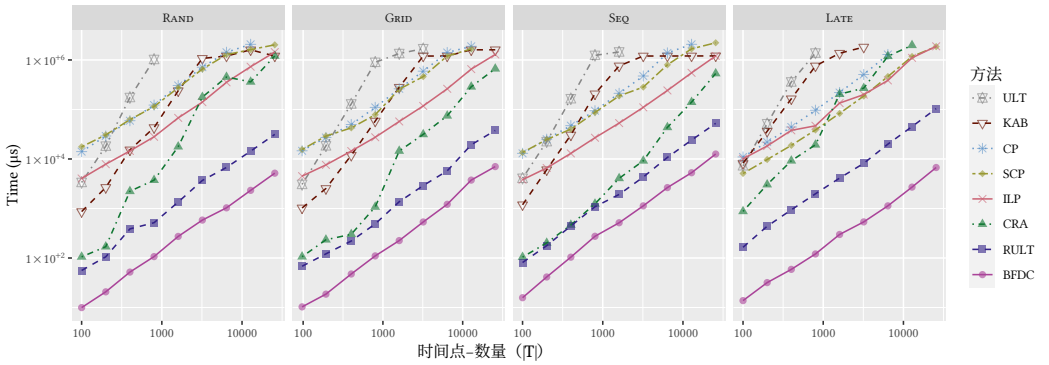


图2. Nodes 子集的平均计算时间（微秒，μs）。 x轴和 y轴均采用对数尺度。

图2中各图的算法总体性能表现相当一致。我们可以清楚地看到，图的上方有一簇曲线，包括 ULT、CP、SCP、ILP 和 KAB。图的下方也很容易辨认出由 RULT 和 BFDC 构成的另一簇曲线。与此同时，CRA 位于这两簇之间，通常在较小实例中处于底部，并随着实例规模增大而逐渐上升至顶部。这些差异并不令人意外，因为顶部簇中的大多数方法都比底部簇中的方法（包括 CRA）更通用。虽然这些差异在意料之中，但它们是否也适用于不同场景，仍有待回答。

ULT 在各图中表现出最快的增长，且很少能求解包含$|T| > 1000$的实例。这种表现很容易解释：每次迭代都要使用 Floyd–Warshall算法，因此时间复杂度为 $\Theta(|T|^3)$。CP 和 SCP 的执行时间相对稳定，其中 SCP 在 Rand、Seq 和 Late 中能够求解规模稍大的实例。对于 Late 实例，SCP 在所有场景中的表现都优于 CP。这表明，CP 求解器 CPLEX 能够从 SCP 的模型结构中获益。与此同时，在 Rand、Grid 和 SEQ 中，ILP 是最上方方法簇中速度最快的方法；而对于 Late 实例，其表现与 SCP 相近。

可以观察到，对于小型实例（$|T| \leq 800$），KAB 仅在最上方的簇中优于其他方法。原因在于，计算全点对最短路径需要耗费大量时间，面对更大的实例时很快就会变得难以承受。对于 Late 实例，KAB 无法求解满足 $|T| > 3200$ 的实例。这是因为 Late 实例往往具有更大的区间集 E，这会直接影响冲突图的构建（算法 2 第 3 行），进而限制 KA 的执行时间。

CP、SCP、ILP、ULT 和 KAB 计算结果所需的时间始终超过 1 毫秒。与此同时，CRA 在所有情况下的耗时都低于或等于这一阈值，并且增长迅速，通常会达到 1 秒的阈值。尽管如此，我们可以看到，CRA 的表现通常优于 ILP。即使 CRA 更慢，差异也并不显著。但 Late 实例是例外：对于满足 $|T| = 25600$ 的全部五个实例，CRA 均超时。在这些情况下，大量时间点与较晚的可行调度相结合，使 CRA 使用 Dijkstra 算法计算距离矩阵中更多条目，导致该方法产生巨大的计算开销（算法 3 第 8 行）。RULT 和 BFDC 的耗时始终低于 100 毫秒，且速度快于 CRA。尽管从理论上看，RULT 的最坏情况时间复杂度比 CRA 更慢，但仍然如此。在底部簇中，我们没有观察到这两种方法中的任何一种超时，这使它们在图 2中的曲线较低。这两种算法的相对位置也始终非常稳定：RULT 仅略慢于 BFDC。

4.1.2 密度数据集。第二个子集是密度子集，其中我们改变约束的数量C_1，范围为 {20,25,. . ., 85,90}%，该数值表示为基图中最大约束数量（即最大弧数）的百分比。其他参数固定如下： $|T| = 1008$,$|T_D| = 0.8 \cdot |T|$和 $K = 10$。与节点子集类似，每种配置生成五个实例。图3展示了随着密度增加而变化的平均计算时间。

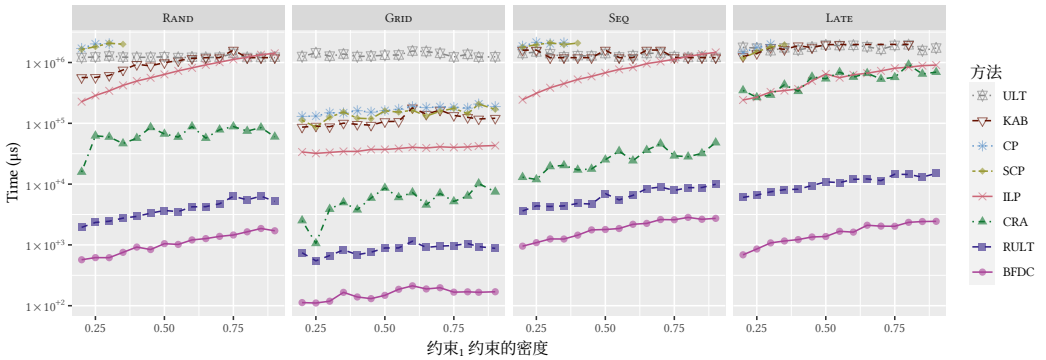


图3. 密度子集的平均计算时间。 y轴采用对数刻度 e.

一方面，由于 Floyd–Warshall算法阶段使 ULT 始终处于最慢方法之列，因此 ULT 的性能几乎不受密度变化的影响。另一方面，ULT 总是能够为每种配置至少求解一个实例，但并非所有方法都能做到这一点。除 Grid实例外，当密度超过 40% 时，约束规划和 SCP 在求解问题时都会遇到困难。对于 Grid实例，KAB 的表现远胜 ULT；但在其他所有情况下，KAB 的表现通常与 ULT 相近。KAB 的速度也从未慢于约束规划或 SCP。从表现最好的方法来看，整数线性规划始终是性能最佳的方法。

这里我们注意到，对于 Rand、网格和 Seq 实例，CRA 的表现远胜于 ULT、约束规划、SCP、整数线性规划和 KAB。它仍属于表现中等的算法，但其性能受网络密度影响不大。然而，对于 Late 实例，CRA 的表现与整数线性规划相当。与 Nodes 数据集类似，这是由 Dijkstra 计算带来的开销所致。不过，对于密度实例，网络的连接更为紧密，因此其他节点发生变化时，会影响更多时间点。这也意味着需要进行更多赋值更新。

RULT 和 BFDC 的表现也几乎不受网络密度影响。它们仍是速度最快的算法，且未观察到超时。

4.1.3 NumDisj 数据集。在第三个子集中，我们将每个时间点 $i \in T_D$的域数 K 设为 5、10、20、……、90、100。其他参数固定如下： $|T| = 2000$、 $|C_1| = 12000$和$|T_D| = 1600$。图 4 展示了参数 K对应的平均计算时间。

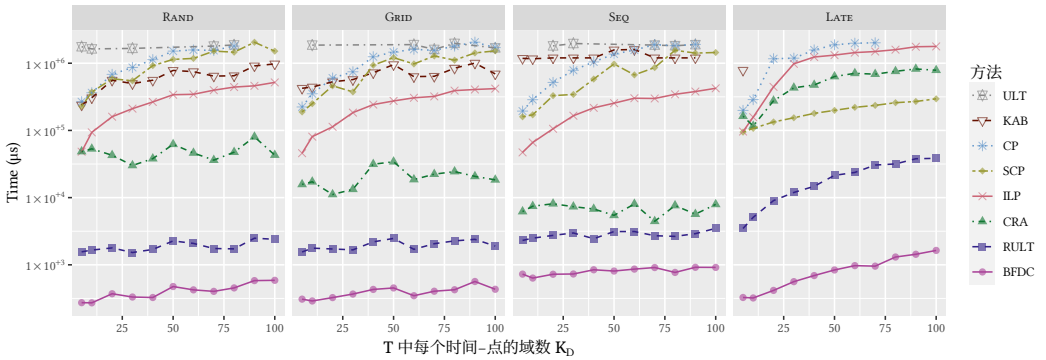


图 4. NumDisj 子集的平均计算时间。 y轴以对数刻度表示 le.

在这些实验中，我们仍然可以区分出之前的三类方法，但它们各自的表现如今有所不同。ULT 遭遇的超时情况要多得多，而且在任何配置下通常都无法求解单个实例。不过，这似乎与参数 K 无关，更可能是由某些未在这些实验中捕捉到的特定实例特征所致。CP 和 SCP 可以求解大多数实例规模。此外，随着 K 的值增加，与 CP 相比，SCP 速度更快且能求解更多实例。这种差异在 Late 数据集上尤为明显：SCP 不仅胜过 CP，在所有情况下也优于 CRA 和 ILP。尽管 ILP 在 Late 实例上的表现较慢，但在其他所有配置中，它都胜过 ULT、CP、SCP 和 KAB。

KAB 在求解 Late 实例时也遇到了同样的困难，只有其中最小的实例得以求解$K = 5$。这并不令人意外，因为 ω 与 K 直接相关（回顾第 2 节）。KAJ 的表现略优于 KAB，能够求解带有 $K \leq 90$的 Seq 实例。在最顶部的集群中，CRA 再次优于所有方法；但对于 Late 实例，SCP 的速度更快。对于 Seq 实例，我们注意到这些 Dijkstra 计算的作用，因为它们帮助 CRA 轻松找到实例构造过程中使用的隐藏最短路径。这使得执行时间大幅缩短。RULT 和 BFDC 仍然是速度最快的方法。然而，与 BFDC 相比，随着 Late 实例中时间点域数量的增加，RULT 明显受到更大程度的影响。这一观察结果与它们的渐近最坏情况时间复杂度一致。

图 4 中的图表表明，通过 CPLEX 求解的方法（CP、SCP 和整数线性规划）受 K增加的影响最大。这可能是由于域数量增加时所创建的约束数量增多，以及 SCP 模型中集合 Φ_i 的基数增加所致。

*4.1.4 VarDisj*数据集。SP 实例的第四个也是最后一个子集是 VarDisj。该子集改变 T_D 的大小，范围为 {10,20,. . . ,90,100}%，即占具有多个域的时间点总数 $|T|$ 的百分比。其他参数固定如下：$|T| = 2000$, $|C_1| = 6 \cdot |T|$ 和 $K = 10$。图 5 展示了按集合 T_D 的大小统计的计算运行时间结果。

图 5 展示了该实例子集的结果与之前的子集有显著差异。首先，对于 Rand、Grid 和 Seq 实例，各种方法如今在图的不同区域中分布得更加分散

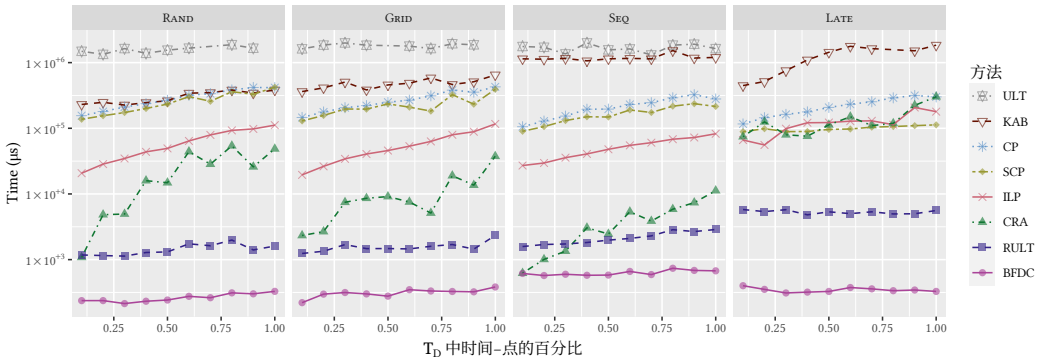


图 5. VarDisj 子集的平均计算时间。 y 轴采用对数尺度表示。

。ULT 再次成为最慢的方法，凸显了它在计算全点对最短路径方面的困难。KAB 在 Rand 数据集上可与 CP 和 SCP 竞争，但在 Grid 和 Seq 数据集上明显慢于这些方法。与 NumDisj 类似，KAJ 在 Seq 实例上的表现略优于 KAB，但仍不足以胜过 CP 或 SCP。事实上，这两种方法始终优于 KAB 和 ULT。与此同时，ILP 比前四种方法更快，表现一如既往地较为稳定，是求解 SDTP 的最佳通用算法。随着具有多个域的时间点数量增加，CRA 显然是受影响最大的方法。尽管如此，在前三个数据集上，CRA 始终比 ILP 快；对于较小的 Seq 实例，其表现甚至优于 RULT。最后，RULT 和 BFDC 仍然是最快的方法，并且随着集合 T_D 大小的增加，它们的性能几乎没有明显变化。

考虑到 Late 实例（可以说是最难求解的实例）时，情况便有所不同。ULT 无法求解该集合中的任何一个实例。KAB 看起来是速度最慢的方法。与此同时，CP、SCP、ILP 和 CRA 的表现都优于 KAB。CP 显然是这四种方法中最慢的。ILP 和 CRA 的表现有一定波动，但总体而言，其增长趋势远不如另外三个集合（Rand、Grid 和 Seq）明显。SCP 在所有实验中的表现都非常稳定；对于较大的 T_D 集合，其性能优于另外三种算法，尽管优势并不算大。除 Seq 外，在几乎所有数据集中，RULT 都比 BFDC 慢一个数量级。不过，RULT 和 BFDC 的性能都没有显著波动。

4.2 负环实例

负环实例采用 Cherkassky 等人 [3] 提出的同名筛选器。该筛选器通过在底层基图中引入负环，应用于第 4.1 节的实例。我们仅考虑应用筛选器前可行的实例，使其恰好因负环而变得不可行。与 Cherkassky 等人 [3], 类似，我们考虑四类负环：（Nc02）一个含三条弧的环；（Nc03）$\lfloor\sqrt{|T|}\rfloor$ 个环，每个环含三条弧；（Nc04）$\lfloor\sqrt[3]{|T|}\rfloor$ 个环，每个环含 $\lfloor\sqrt{|T|}\rfloor$ 条弧；以及（Nc05）一个哈密顿环。

对于这四类中的每一类，我们将 $|T|$ 的取值范围设为 {100,200,. . . ,12800,25600}，同时固定参数 $|C_1| = 6\cdot|T|, |T_d| = 0.8\cdot|T|, K = 10$。对于每种配置，我们生成三个实例。图 6 给出了各类负环的计算结果，并报告了每种方法在 20 次运行中证明不可行所需的平均时间。

首先可以观察到，在环数量更少且规模更小的情况下（Nc02 和 Nc03），ULT 反而更难求解实例。对于所有情况，CP 都无法求解规模最大的实例，

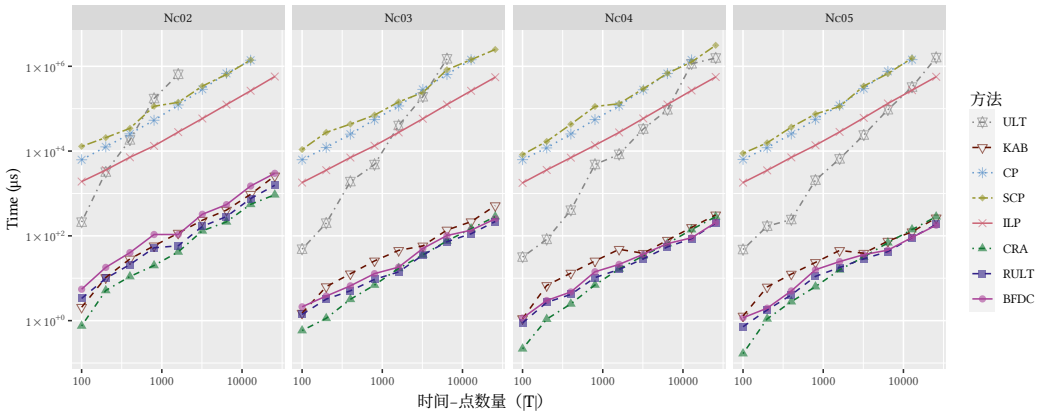


图 6. Negcycle 数据集的平均计算时间。 x轴和 y轴均采用对数尺度表示。

而当存在多个环时（Nc03 和 Nc04），SCP 能够证明大型实例不可行。与此同时，尽管存在环，ILP 仍展现出非常稳定的性能。

我们首次注意到，KAB 位于图表底部的最快算法之列。这并不令人意外，因为对于负环实例，KAB 能够在计算距离矩阵 δ时检测到该环（算法 2 的第 1 行）。同样是首次，BFDC 相较于 CRA、RULT 和 KAB，在 Nc02 上始终表现出最低的执行时间。对于同一数据集，CRA 始终是最快的方法，因为它还能够在使用标准 Bellman-Ford 的第一阶段检测到负环（算法 3 的第 1 行）。这展示了 BFDC 中域检查所带来的开销（算法 5 的第 8 行）。

然而，在 Nc03-Nc05 数据集中，位于底部集群中的方法更难区分。KAB 通常看起来是最慢的，CRA 最快，而 BFDC 和 RULT 则处于两者之间。不过，对于大多数用途而言，负环实例中 KAB、RULT、BFDC 和 CRA 之间的差异可以忽略不计。

4.3 车辆路径实例

我们从满足多重同步约束 [11]的车辆路径问题（VRPMS）的解中提取车辆路径实例（Vrp）。这些 VRPMS 实例包含多条路径，需要为其分配出发时间和服务时间，同时满足路径之间的同步约束以及最大路线时长约束。有关这些实例具体如何生成的更多信息，请参阅附录 A。

Vrp 实例的主要区别在于时间点的数量，范围从 10 到 1300。对于每种实例规模，我们同样创建五个实例：三个可行实例和两个不可行实例。图 7 根据实例中的时间点数量，展示了各方法的计算时间，所列数值为 20 次运行的平均值。结果按实例的可行性分组，因为实例是否可行会导致性能出现一些差异。

不难发现，ULT 能够轻松证明实例不可行，但在为可行实例生成解时却遇到困难。事实上，对于可行实例，ULT 无法求解满足 $|T| > 1000$ 的实例。CP 和 SCP 的表现各不相同，其中 SCP 的平均速度更快，尤其是在不可行实例上。KAB 的表现始终与 CP 相近。其性能

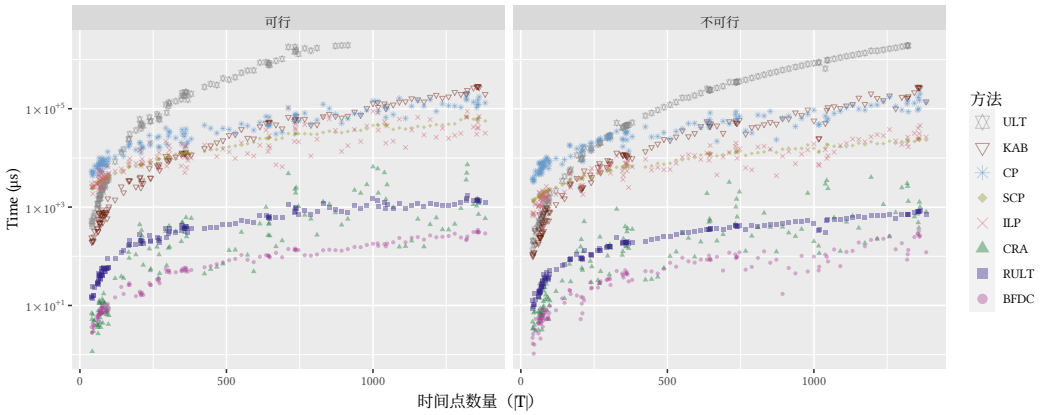


图 7. Vrp 数据集的平均计算时间。 y 轴采用对数尺度。

ILP 在可行实例上的表现差异显著，而在不可行实例上的差异则相对较小。很难断定总体而言 ILP 是否比 SCP 更快，不过可以清楚地看出，这两种方法都比 ULT、CP 和 KAB 更快。

CRA、RULT 和 BFDC 再次集中在图表底部，表明它们是速度最快的方法。RULT 是三者中最慢的，不过 CRA 的表现差异较大，在某些情况下速度更慢。可以得出，BFDC 是速度最快的方法；但与之前一样，CRA 在某些情况下确实快于 BFDC。总体而言，速度从慢到快的顺序为：RULT、CRA 和 BFDC。

最后请注意，Vrp 实例的底层距离图都非常稀疏。除连接到时间范围起点 α的顶点外，每个顶点最多有两条出弧和两条入弧。此外，这些图几乎没有环，只包含数量极少的会形成环的弧。虽然我们在求解 Vrp 数据集时没有利用这一结构，但这无疑是一个值得未来研究探索的方向。

4.4 超大实例

尽管前面三个数据集各有不同特点，但它们都未能涵盖时间点数量非常庞大的情形。这类实例对于真正验证方法的可扩展性至关重要，因为某些方法可能在小规模问题上表现出优势，但会在实例规模大幅增长时遭遇困难。为了验证此前分析在这类实例上的适用程度，我们生成了五个超大（Vl）问题，它们均为可行问题。表 2 按以下特征介绍了这些实例：基准 生成方法、时间点数量、 C_1 约束数量、每个时间点的最大域数以及域的总数 ω。基础 Tsp 表示图 G 取自旅行商问题库（ TSPLib ） [23] 中的 pla85900 实例，该实例包含 85900 个节点。对于 Vl-1 实例，我们从 pla85900 中随机选取一个节点子集。与此同时，Vl-3、Vl-4 和 Vl-5 实例则使用第 4.1 节中概述的 SP 程序生成。

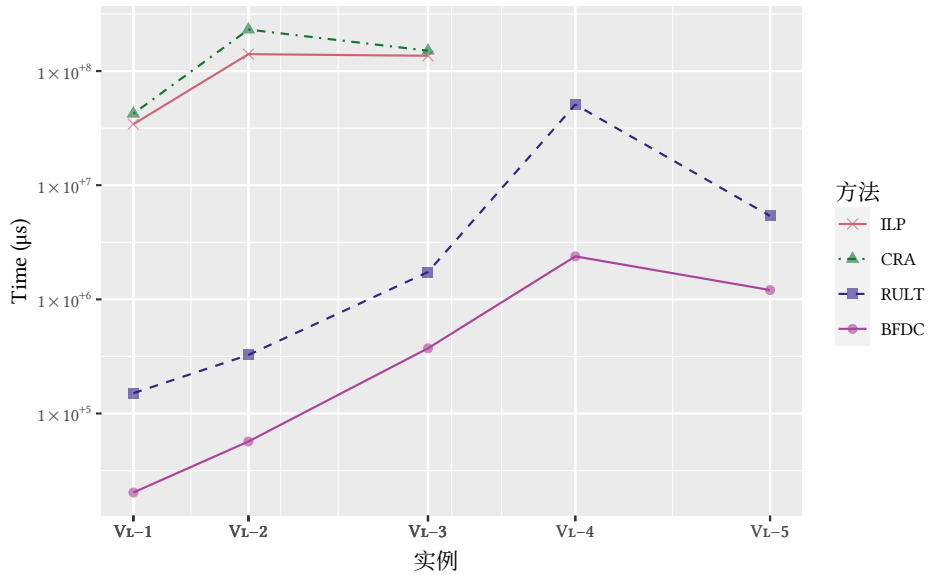
图 8 展示了部分方法在 Vl 实例上运行 10 次的平均执行时间。我们选择仅测试表现最佳的方法：整数线性规划、CRA、RULT 和 BFDC。每次运行中，这些方法的时间限制均为一小时。

即使在表现最佳的四种方法中，显然 CRA 和整数线性规划在可求解的实例规模方面也存在局限。事实上，它们无法在

表 2. 超大实例。

实例	Base	$\lvert T\rvert$	$\lvert C_1\rvert$	K	ω
Vl-1	Tsp	50 000	500 000	20	905 000
Vl-2	Tsp	85 900	859 000	60	4 647 190
Vl-3	Seq	200 000	2 000 000	100	16 040 000
Vl-4	Late	400 000	4 000 000	180	57 680 000
Vl-5	Rand	1 000 000	10 000 000	500	400 200 000

一小时内求解超过 Vl-3 的实例。虽然难以确定整数线性规划求解器出现这种表现的确切原因，但 CRA 的原因在于，大规模实例需要进行更多 Dijkstra 计算（尤其是 Vl-4）。计算全点对最短路径不仅耗时，而且会导致内存局部性显著变差。这会影响缓存使用，从而降低 CRA 相较于 RULT 的性能，尽管后者在理论上的最坏情况时间复杂度更高。图 8 还表明，Late 实例比其他实例难得多。即使 Late 实例的时间点数量不到 Rand 实例的一半，这一点依然成立。



Vl 数据集的平均计算时间。

5 讨论

Johnson [13] 指出，实验论文中报告的许多内容，描述的是算法的实现，而非算法本身这一数学对象。一方面，我们的研究在某种程度上符合这一趋势。尽管这是实证研究众所周知的局限，我们仍希望通过提供代码，让感兴趣的读者能够检查甚至改进这些实现，从而减轻这一问题。另一方面，本文记录的一些结果与实现无关。例如：与所有其他方法相比，RULT 和 BFDC 都实现了更低的空间复杂度。

现在我们来对这项计算研究进行全面分析。表 3 总结了第 4 节实验的结果，但不包括 Vl 实例。最大时间（毫秒）、平均时间（毫秒）和时间标准差（毫秒）各列报告了每种方法记录到的执行时间的最大值、平均值和标准差，单位为毫秒。总时间（秒）列报告了每种方法解决所有实例所需的总时间，包括可能发生的超时。最后，超时（%）列给出了方法发生超时的运行次数所占的百分比。

在考察表 3 时，必须考虑到 ULT、约束规划、SCP 和整数线性规划都比 CRA、RULT 和 BFDC 更通用。因此，后者算法在几乎所有情况下都优于前者，这并不令人意外。不过，我们的实验表明，在求解析取时序问题时，各算法的性能存在显著差异，而且有些结论乍看之下可能有悖直觉。例如，尽管 ULT 和 KA算法具有多项式最坏情况渐近时间复杂度，但在求解析取时序问题时，这些算法的整体表现并不理想。与此同时，整数线性规划在一个乍看之下似乎更适合采用约束规划求解的问题上表现良好。

表 3. 结果汇总。

方法	最大时间（毫秒）	平均时间（毫秒）	平均时间（毫秒）	总时间（秒）	超时率（%）
ILP	1947	350	462	7058	2.28
CP	2017	1062	985	21406	33.39
SCP	2002	875	954	17633	25.15
ULT	1990	1476	812	29758	66.76
KAB	2000	868	873	17491	28.02
KAJ	2001	1065	1276	21478	30.26
CRA	1895	138	317	2772	1.19
RULT	152	6	13	115	0.00
BFDC	28	1	2	18	0.00

对 KAB 和 KAJ 的实现进行性能分析后发现，$\approx 90\%$ 的执行时间始终花在构建距离矩阵 δ 和计算冲突上（算法 2 的第 1–3 行）。观察发现，最大流计算（算法 2 的第 6 行）耗费的总时间不到 5%。这表明，瓶颈在于全点对最短路径和冲突的计算，而非理论上较慢的最大流步骤。

同样，对 CRA 的实现进行性能分析后发现，Dijkstra 计算最多占处理时间的 95%。这一观察结果也涵盖了我们对距离矩阵的惰性求值实现。如果按照 Comin 和 Rizzi [6], 最初所述的方式预先计算完整矩阵，那么 Dijkstra 算法所占的时间比例可能会更加极端。在许多情况下，无法在规定的时间限制内预先计算距离矩阵。此外，由于内存不足，对于大型实例，预先计算距离矩阵根本不可行。尽管存在这些缺点，CRA 的一个优势是，当初始解已可行时，部分实例可在其第一阶段（Bellman-Ford）快速求解，因此无需对时间点分配 s_i 进行修正。

表 3 中详细列出的结果进一步证实了第 4 节中的观察结果。总体而言，RULT 和 BFDC 的表现最佳，其计算时间比所有其他方法短两到三个数量级。在我们的实验中，这两种算法也都没有出现超时记录。这种性能可以用两个因素来解释。首先，RULT 和 BFDC 都专注于计算单源最短路径，而 ULT、KA算法和 CRA 在求解全点对最短路径时会消耗更多计算资源。其次，作为第一个因素的直接结果，

RULT 和 BFDC 的空间复杂度均为线性，只使用大小为 $|T|$ 的一维数组，从而改善了缓存局部性并提高了整体效率。事实上，这两种方法几乎可以完全在缓存中求解某些实例，而其他方法的二次空间复杂度使这种情况极不容易发生。

图 9 展示了 CRA、RULT 和 BFDC 在求解 Vl 实例时的缓存引用测量结果。我们重点关注这个数据集，因为它需要投入最多的算法工作。缓存引用事件的记录使用 Linux 内核 [29] 中的 perf_events 软件包完成。图上半部分的“*Full scale*”行展示了比较不同方法的行为有时会有多困难。因此，我们还在下方加入了“*Small scale*”图，这些图放大了“*Full scale*”图，以便进一步呈现各方法行为的细节。这些图清楚地表明，RULT 和 BFDC 都能通过更高效地利用缓存，大幅加快求解大型实例的速度。在所有实验中，RULT 和 BFDC 所需的缓存引用总数都少于 CRA 的缓存未命中数。

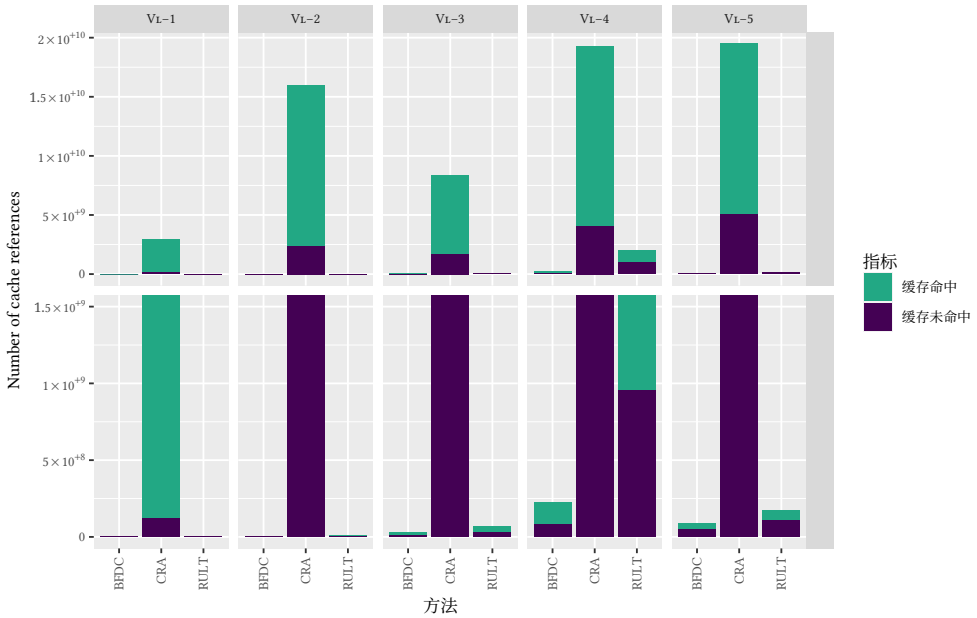


图 9. Vl 数据集中各方法的缓存引用数。

本文实现的方法在所产生的解的类型方面也有所不同。ULT、CRA、RULT 和 BFDC 会在执行结束时给出最早可行解。不过，ULT、RULT 和 BFDC 也可以返回最新可行解。Comin 和 Rizzi [6] 没有说明他们的方法是否能够返回最新可行解，不过，在图 G_R 上计算解并对操作（例如 UpdateAssignments）稍作修改时，似乎可以做到这一点。相比之下，CP、SCP、ILP 和 KA 均无法保证返回最早或最新可行解。通过为底层模型定义适当的目标函数，可以确保找到其中一种解，但这会对其性能造成多大影响尚不明确。例如，KA 需要求解弧容量任意的最大流，而非弧容量为单位值的最大流 [14]。

人们或许会倾向于认为，面对 SDTP 时应首选 BFDC 和 RULT，但这未必是我们所主张的结论。我们的建议要更为审慎一些。鉴于 BFDC 在单独处理 SDTP 时表现最佳，因此在评估相互依赖的车辆路径的可行性等问题时，它是最合乎情理的选择。然而，其他涉及 SDTP 的问题可能需要采用其他算法才能获得最佳性能。对于受限析取时序问题，Co min 和 Rizzi [6] 提出了一种利用 CRA 结构来降低时间复杂度的方法。从理论上说，也可以使用 BFDC，但这会提高该算法处理 RDTP 时的渐近最坏情况时间复杂度。同样，Kumar [14]指出，只需稍作修改，KA 就能用于求解这类 SDTP：为每个域指定任意偏好权重，并以找到使所选域权重总和最大的可行解为目标。在这种问题背景下，不清楚如何使用 BFDC 或 RULT。另一方面，我们通过实证表明，KA 即使在求解中等规模的实例时也会遇到困难。因此，值得进一步研究能更快求解这类带偏好的 SDTP 的方法。

6 结论

简单析取时间问题是简单时间问题的推广。它们在现实应用中用途广泛，通常作为子问题出现。应用领域包括机器人规划、员工排班、物流和管理系统。SDTP 还可用于分解方法，以求解更一般的时序约束满足问题。因此，研究人员和从业者除了了解求解 SDTP 的算法在理论上的时间界，也有必要了解其实际表现。遗憾的是，过去文献对这些方法在实践中的表现知之甚少。

为弥合这一差距，拉近这些时间问题的理论与实验研究，我们针对求解 SDTP 的新算法和已有算法开展了一项大规模探索性实证研究。结果表明，理论上的最坏情况时间复杂度不一定能反映这些算法实际观测到的计算时间。此外，我们还表明，先前算法的二次空间复杂度伴随着若干缺点，无论其渐近时间复杂度如何，都会限制它们在实践中的应用。事实上，对于超大数据集，一些方法会因内存限制而无法求解一个原本简单的问题，即使是在现代计算机上运行也是如此。相比之下，空间复杂度较低但时间复杂度较高的算法，只需几秒钟便能求解超大实例。

我们希望本文的结果能够为未来的研究者提供有用的证据，帮助他们根据自身的应用做出明智的选择，确定最合适的算法，从而减少重复实现的工作。我们公开提供的代码也将有助于未来的研究检验我们的结论是否适用于其他数据集。最后，我们的实现还为简单析取时间问题及其特殊情形的新算法或加速技术提供了共同的基准测试平台。

未来研究方面，可以考虑针对受限析取时序问题 [15]开展类似的计算实验。可以通过添加 3 型约束，从本文介绍的实例中派生出新的实例。另一种选择是研究带偏好的 SDTP 变体算法，其中每个时间点的域都关联有偏好 [14]。例如，某个时间点可能更倾向于安排在上午执行，而不是晚上。还可以从现实应用中提取 SDTP 实例，并验证我们的研究结论对于其他图结构是否仍然成立，或者是否存在性能更好的方法。

致谢

本研究得到了鲁汶大学内部基金（IMP/20/021）以及比利时佛兰德政府通过人工智能研究计划 （AI）的资助。Luke Connolly（鲁汶大学）提供了编辑咨询。

参考文献

[1] Ravindra K. Ahuja、Thomas L. Magnanti 和 James B. Orlin。1993。《网络流：理论、算法与应用》。Prentice-Hall, Inc.[2] Amedeo Cesta 和 Angelo Oddi。1996。在简单时间问题中提高效率与灵活性。载于《第三届时间表示与推理国际研讨会论文集》（TIME'96）。IEEE，45–50。[3] Boris V. Cherkassky、Loukas Georgiadis、Andrew V. Goldberg、Robert E. Tarjan 和 Renato F. Werneck。2010。最短路径可行性算法：一项实验评估。《ACM 算法实验学杂志》14，文章 7（2010 年 1 月），37 页。[4] Boris V. Cherkassky、Andrew V. Goldberg 和 Tomasz Radzik。1996。最短路径算法：理论与实验评估。《数学规划》73，2（1996），129–174。[5] Jan Christiaens 和 Greet Vanden Berghe。2020。通过字符串移除引入松弛，解决车辆路径问题。《运输科学》54，2（2020），417–433。[6] Carlo Comin 和 Romeo Rizzi。2018。论受限析取时间问题：更快的算法与可处理性边界。载于《第 25 届时间表示与推理国际研讨会》（TIME 2018）（《莱布尼茨国际信息学会议录》（LIPIcs），第 120 卷），Natasha Alechina、Kjetil Nørvåg 和 Wojciech Penczek（编）。Schloss Dagstuhl–Leibniz 信息学中心，德国达格施图尔，10:1–10:20。[7] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein。2009。《算法导论》。麻省理工学院出版社。[8] Rina Dechter、Itay Meiri 和 Judea Pearl。1991。时间约束网络。《人工智能》49，1（1991），61–95。[9] Michael L. Fredman 和 Robert Endre Tarjan。1987。斐波那契堆及其在改进网络优化算法中的应用。《ACM 杂志》34，3（1987 年 7 月），596–615。[10] Andrew V. Goldberg 和 Robert E. Tarjan。1988。最大流问题的一种新方法。《ACM 杂志》35，4（1988），921–940。[11] Hossein Hojabri、Michel Gendreau、Jean-Yves Potvin 和 Louis-Martin Rousseau。2018。结合约束规划的大邻域搜索，用于具有同步约束的车辆路径问题。《计算机与运筹学》92（2018），87–97。[12] Luke Hunsberger 和 Roberto Posenato。2021。简单时间网络：时间表示与推理的实用基础。载于《第 28 届时间表示与推理国际研讨会》（TIME 2021）（《莱布尼茨国际信息学会议录》（LIPIcs），第 206 卷），Carlo Combi、Johann Eder 和 Mark Reynolds（编）。Schloss Dagstuhl–莱布尼茨信息学中心，德国达格施图尔，1:1–1:5。[13] David S. Johnson。2002。算法实验分析理论家的指南。《DIMACS 离散数学与理论计算机科学系列》59（2002）。[14] T.K. Satish Kumar。2004。一种求解带有分段常数域偏好函数的简单时间问题的多项式时间算法。载于人工智能促进协会会议论文集，67–72。[15] T.K. Satish Kumar。2005。论受限析取时间问题的可处理性。载于国际自动规划与调度会议论文集，110–119。[16] T.K. Satish Kumar、Marcello Cirillo 和 Sven Koenig。2013。带有禁忌区域的简单时间问题。载于第二十七届人工智能促进协会人工智能会议论文集。[17] Daniel H. Larkin、Siddhartha Sen 和 Robert E. Tarjan。2014。一项回归基础的优先队列实证研究。载于《2014 年第十六届算法工程与实验研讨会论文集》（ALENEX）。SIAM，61–72。[18] Renaud Masson、Fabien Lehuédé 和 Olivier Péton。2014。需求响应接送问题中的换乘。《计算机与运筹学》41（2014），12–23。[19] Catherine C McGeoch。1996。迈向算法模拟的实验方法。《INFORMS 计算机学报》8，1（1996），1–15。[20] Snežana Mitrović-Minić 和 Gilbert Laporte。2006。带时间窗和转运的取送货问题。《INFOR：信息系统与运筹学研究》44，3（2006），217–227。[21] Bernard ME. Moret 和 Henry D. Shapiro。2001。算法与实验：新方法论（及旧方法论）。《通用计算机科学杂志》7（2001），434–446。[22] Cédric Pralet 和 Gérard Verfaillie。2012。时变简单时间网络。载于《约束规划原理与实践国际会议论文集》。Springer，608–623。[23] Gerhard Reinelt。1991。TSPLIB——旅行商问题库。《ORSA 计算机学报》3，4（1991），376–384。[24] Peter Sanders。2001。适用于高速缓存存储器的快速优先队列。《ACM 算法实验学杂志》5（2001 年 12 月），25 页。

[25] Briseida Sarasola 和 Karl F. Doerner。2020。用于配送地点具有同步约束的车辆路径问题的自适应大邻域搜索。*Networks* 75，1（2020），64–85。[26] Carlo S. Sartori、Pieter Smet 和 Greet Vanden Berghe。2022。根据欧盟法规调度具有相互依赖路径的卡车司机。*European Journal of Operational Research* 298，1（2022），76–88。[27] Eddie Schwalb 和 Rina Dechter。1997。处理时间约束网络中的析取项。*Artificial Intelligence* 93，1（1997），29–61。[28] Kostas Stergiou 和 Manolis Koubarakis。2000。析取时间约束的回溯算法。*ArtificialIntelligence* 120，1（2000），81–117。[29] Vince Weaver。2013。非官方 Linux Perf Events 网页。在线网址：https://web.eece.maine.edu/ ~vweaver/projects/perf_events/。最后访问日期：2022-10-17。

A 实例生成

本附录补充说明了计算研究中使用的部分实例是如何生成的。我们将进一步详细介绍最短路径实例和车辆路径实例。

A.1 最短路径实例

下面将详细阐述第 4 节中针对每个 Sp 数据集给出的内容。

(1) Rand：使用 SPLib 中介绍的 Sprand 生成图 G。节点和弧均随机生成。通过计算从虚拟源节点 z 到每个 $i \in V$ 的最短路径 τ 来创建约束 C_2。向集合 T_D 中随机添加若干时间点，直至达到所需规模 $|T_D|$。对于每个 $j \in T_D$，从均匀分布 $\mathrm{U}[1, K]$ 中生成一个随机数 κ_j。随后将 j 的第 κ_j 个域定义为 $d_j^{\kappa_j} = [s_i^0 - \varphi_1, s_i^0 + \varphi_2]$，其中 $s_i^0 = -\tau_i$ 和 φ_1，$\varphi_2 \in \mathrm{U}[0, 2000]$。定义好 $d_j^{\kappa_j}$ 后，按如下方式生成其余域。首先，从 $[l_j^{\kappa_j}, u_j^{\kappa_j}]$ 开始反向构造域 $[l_j^1, u_j^1]$、……、$[l_j^{\kappa_j-1}, u_j^{\kappa_j-1}]$。然后通过 $u_j^{\kappa_j-1} = l_j^{\kappa_j} - a - 1$ 和 $l_j^{\kappa_j-1} = u_j^{\kappa_j-1} - b_j$ 定义域 $[l_j^{\kappa_j-1}, u_j^{\kappa_j-1}]$，其中 $a \in \mathrm{U}[0, 200]$ 且 $b_j = u_j^{\kappa_j} - l_j^{\kappa_j}$。如此递归重复，直到 $[l_j^1, u_j^1]$。类似地，对于最后几个域 $[l_j^{\kappa_j+1}, u_j^{\kappa_j+1}]$、……、$[l_j^K, u_j^K]$，通过设置 $l_j^{\kappa_j+1} = u_j^{\kappa_j} + a + 1$ 和 $u_j^{\kappa_j+1} = l_j^{\kappa_j+1} + b_j$ 生成 $[l_j^{\kappa_j+1}, u_j^{\kappa_j+1}]$。再次递归重复，直到创建出 j 的第 K 个域。我们只接受这样的实例：在 SDTP 的最早可行解 s 中，至少 60% 的条目 s_i 属于域 $[l_i^c, u_i^c]$，$c > 1$。这意味着，仅将第一个域赋给每个时间点并不能保证可行性。

(2) 网格：使用 Spgrid 生成图 G，该方法也在 SPLib [4]中介绍。节点以网格形式生成，包含 X 层，每层有 Y 个节点。弧连接同一层内的节点以及后续层中的节点。不同层的节点之间也可能添加额外的弧。与 Cherkassky 等人 [3], 类似，我们在所有使用 Spgrid 生成的图中固定 $Y = 16$，并根据需要调整 X，以创建包含 $|T| = 16 \cdot X$ 个时间点的 SDTP 实例。调整参数 Y 并未显著改变结果。C_2 约束的生成方式与 Rand 相同。

(3) Seq：使用专用生成器 Spseq 生成图 G。节点以随机方式生成，与 Sprand 类似。创建一条包含 $|V| - 1$ 条弧、连接所有节点的路径，每条弧的权重为 $w_{ij} = 1$。随后，以随机方式创建剩余的 $|A| - |V| + 1$ 条弧，其较大的权重从 $\mathrm{U}[500, 20000]$ 中均匀选取。这样便生成了一条已知的最短路径，而某些方法可能难以找到这条路径。C_2 约束的生成方式与 Rand 相同。

(4) Late：使用 Sprand 和 Spseq 生成图 G。不过，C_2 约束的生成方式有所不同。首先，按照 Rand 数据集中的方式生成初始域 $d_i^0 = [s_i^0 - \varphi_1, s_i^0 + \varphi_2]$

适用于所有 $i \in T$。随后，我们选择一个时间点子集纳入 T_D。对于每个 $j \in T_D$，我们设置 $D_j = \{d_j^0\}$，并按以下方式持续迭代。首先，随机选择一个时间点 $j \in T_D$，$|D_j| < K$。接着，我们为 j创建一个新域 $[l_j^c, u_j^c]$，其形式为 $l_j^c = -\tau_j + 5$和 $u_j^c = l_j^c + 1$。将新域追加到 D_j，并在满足 $u_j^{c-1} \geq l_j^c$时，将较早的域向后移。一旦完成，我们就更新弧（z，i，w_{zi}）在 G 中的权重，将其设为 $w_{zi} = -l_j^c$，并重新计算距离 τ。这实际上更新了解，并调整后续迭代中将生成的域。下一轮迭代接着选择另一个$j \in T_D$，并向 D_j追加一个新域，直到 $|D_j| = K \forall j \in T_D$。此过程使我们能够确保可行解属于每个时间点 $j \in T_D$中较晚 的域；这可能意味着，如果某些算法从每个时间点的第一个域开始逐步构建，或者必须收紧全局边界，那么它们就需要运行更长时间。我们只生成 Late 实例，其中最早可行解s 中的条目 s_i 至少有 60% 属于各自时间点的最后一个域。

A.2 车辆路径实例

车辆路径实例取自为多重同步约束车辆路径问题（VRPMS） [11]所提出的实例。我们使用字符串移除松弛诱导启发式 [5]求解 VRPMS 实例。每个新的可行 VRPMS 解都会保存为一张图，其中包含多条通过同步弧连接的节点链（路径）。加入时长弧以形成环，从而生成更具挑战性的实例。这些问题中，每个客户位置都对应一个时间窗，因此可以将其解释为简单时间问题 [26]。

为创建 SDTP 实例，我们将上述图转换为 SDTN 网络$N = (T, C)$。节点（VRPMS 中的客户）会转换为时间点，并连同车辆出发的仓库一并纳入。C_1 约束根据路径中各位置之间的行驶时间以及两条路径之间的相互依赖约束创建。C_1中还包含最大路线时长约束。对于 C_2，我们只需在域之间选定一个间隔大小 a，将 VRPMS 节点的时间窗拆分为K 个析取域。对于两个连续的区间 $[l_i^c, u_i^c]$ 和 $[l_i^{c+1}, u_i^{c+1}]$，我们设定 $l^{c+1} = u_i^c + a$。这样一来，我们既尽可能保留了 VRPMS 实例的原有特征，又能确保基于实际应用构建出完整的 SDTP 实例。