arXiv:2304.12610v1 [cs.DS] 25 Apr 2023

基于回溯缩减的流图快速连续子图匹配

Rongjian Yang,数据科学学院，复旦大学，中国Zhijie Zhang, 数据科学学院，复旦大学，中国Weiguo Zheng, 数据科学学院，复旦大学，中国Jeffrey Xu Yu,香港中文大学系统工程与工程管理系，中国

随着许多实际应用中的图随时间不断演变，流图日益受到学术界和工业界的关注。连续子图匹配（简称 CSM）旨在报告此类流图中查询图的增量匹配。回答 CSM 涉及两个主要步骤，即候选维护和增量匹配生成。在连续子图匹配过程中，增量匹配生成对搜索空间进行回溯，构成了总开销的主要部分。然而，以往大多数方法都着重开发高效的候选维护技术，而增量匹配生成尽管对 CSM 至关重要，却未得到足够关注。为尽可能降低总体开销，本文提出两种减少回溯的技术。我们提出了一种高性价比的索引 CaLiG，可实现更严格的候选维护，从而缩小回溯的搜索空间。此外，我们还开发了一种新颖的增量匹配范式 KSS，将查询顶点分解为条件核顶点和壳顶点。利用核顶点的匹配，无需任何回溯，只需连接壳顶点的候选项即可立即生成增量匹配。得益于回溯次数的减少，CSM 的运行时间显著缩短。在真实图上的大量实验表明，我们的方法比最先进的算法快数个数量级。

CCS 概念： • **信息系统 → 流管理**。

附加关键词和短语：子图匹配、流图、回溯缩减

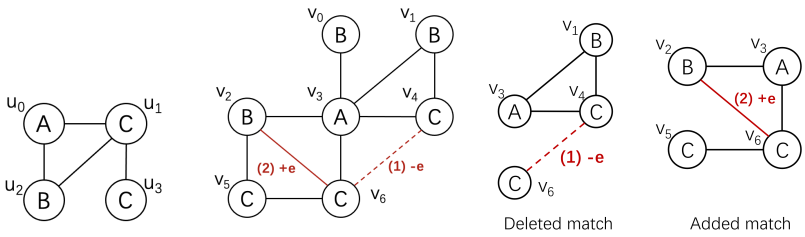
ACM 引用格式：
Rongjian Yang、Zhijie Zhang、Weiguo Zheng 和 Jeffrey Xu Yu。2023。基于回溯约简的流图快速连续子图匹配。*ACM* 数据管理会议录 1，1，第15篇文章（2023年5月），26页。https://doi.org/10.1145/3588695

1 引言

图被广泛用于表示各种对象之间的复杂关系，涵盖社交网络、知识图谱、道路网络以及电力网络。在大多数现实世界的应用中，图会随着时间的推移，通过添加顶点/边或删除顶点/边而不断演化，这种图称为流图。例如，在社交网络中，新用户注册和关闭账户分别可以视为图中的顶点添加和删除；创建和

作者地址：Rongjian Yang，复旦大学数据科学学院，中国上海，rjyang20@fudan.edu.cn；Zhijie Zhang，复旦大学数据科学学院，中国上海，zhijiezhang18@fudan.edu.cn；Weiguo Zheng，复旦大学数据科学学院，中国上海，zhengweiguo@fudan.edu.cn；Jeffrey Xu Yu，香港中文大学系统工程与工程管理系，中国香港，yu@se. cuhk.edu.hk。

允许出于个人或课堂使用目的，免费制作本作品的全部或部分数字或纸质副本，但前提是不得为营利或商业利益制作或分发副本，并且副本必须包含本声明及首页上的完整引文。对于本作品中由作者以外的其他人拥有版权的组成部分，必须尊重其版权。注明出处后，可以进行摘要。除上述用途外，如需复制、再版、发布到服务器或重新分发至列表，必须事先获得具体许可和/或支付费用。请通过 permissions@acm.org 申请许可。© 2023 版权所有归所有者/作者所有。出版权已授权给 ACM。2836-6573/2023/5-ART15 $15.00
https://doi.org/10.1145/3588695



（a）查询图 （b）流图 （c）增量匹配

图1。CSM 的运行示例。

取消用户之间的互动（例如关注）分别对应于边的添加和删除。此类社交网络是一种典型的流图。如何高效管理和查询流图正日益受到关注 [2, 12, 17, 25, 26, 31, 33, 38]。

作为一项重要任务，在流图上监测感兴趣的模式 Q，需要执行连续子图匹配（简称 CSM），以报告增量匹配，即由于图更新而 Q 增加或减少的子图匹配。如图1中的运行示例所示，删除边（v_4，v_6）后，一个匹配被删除；添加边（v_2，v_6）后，另一个匹配被添加。

CSM 可用于广泛的应用场景，例如推荐系统 [6, 14, 39],欺诈检测 [34, 37], 和网络安全 [9, 10],等。

例如，在电子商务平台中，环模式可能是虚假交易的有力 संकेत，其中用户（买方或卖方）的账户被表示为顶点，在线交易（如支付活动）则表示为动态边。借助 CSM，可以检测出可疑交易，从而生成实时警报并触发及时行动。

1.1 现有方法

借助成熟的子图匹配算法 [3, 4, 16], ，一种针对 CSM 的朴素方法是在图更新前后分别枚举匹配。然而，直接应用子图匹配会产生难以处理的开销 [19, 36], ，因为这种方法没有充分利用动态图更新的特征。考虑到更新边包含在所需的匹配中（如果存在），无索引算法（例如 IncIsoMat [11] 和 Graphflow [23], ）通过扩展添加边或删除边来查找发生变化的匹配，而不会生成不必要的匹配。由于没有利用更新前的任何信息，即使不存在任何匹配，这类算法也可能在探索数据图时浪费过多时间。

为实时捕获增量匹配，近期有更多研究致力于开发索引技术来维护中间结果 [10, 24, 32]。SJ-Tree [10] 定义了一棵左深分解树，其中每个节点都维护一组部分匹配。由于匹配数量可能呈指数级增长，SJ-Tree 因而面临难以承受的存储开销。TurboFlux [24] 和 SymBi [32] 并不维护部分匹配，而是利用辅助数据结构和精心设计的过滤规则，深入研究候选生成与维护。TurboFlux [24] 为 Q 生成一棵生成树 T_Q，并引入一种名为数据中心图（DCG）的辅助数据结构，用于维护 G 中的一条边是否与 T_Q 中的一条边匹配。它采用自底向上方法进行过滤，且只考虑 T_Q 中的边。对于图更新，TurboFlux 会更新 DCG 中边的状态，以确定任意两条边是否能够匹配，以及 G 中的每条边是否能够包含在 Q的一个匹配中。SymBi [32]

表 1. 候选维护所占比例（%）

方法	Deezer	电子邮件	GitHub	Last.fm	Skitter	Twitch
TurboFlux	0.13	0.093	0.072	0.19	0.091	0.091
SymBi	0.306	0.380	0.072	0.971	0.476	0.003
CaLiG	1.402	2.652	0.913	1.623	1.112	0.235

采用由 Q 构建的有向无环图（DAG），并检查前向或后向邻居顶点，从而覆盖 TurboFlux 忽略的非树边。SymBi 在 DAG 上采用自顶向下和自底向上的动态规划。只有通过自顶向下过滤的顶点才需要进行自底向上过滤，而只有通过两种过滤的顶点才是有效候选项。TurboFlux 和 SymBi 都强调索引的简洁性和更新效率，尽管维护候选项的时间开销通常只占总体开销的一小部分（如第 1.2 节所述）。在搜索阶段，TurboFlux 采用回溯算法，并使用固定的匹配顺序，该顺序取决于查询路径的估计规模。SymBi 则根据查询顶点的可扩展候选项的估计规模，选择动态匹配顺序，并可在回溯过程中自适应调整。它们只关注匹配顺序，而没有改进回溯框架。

1.2 CSM 的瓶颈

在上述所有方法中，基于索引的方法通常比其他方法表现更好。为了进一步加速连续子图匹配，需要回答一个关键问题：提升效率性能的瓶颈是什么？如上所述，现有方法主要关注如何通过使用简洁的索引来快速维护候选顶点。令人惊讶的是，尽管增量匹配生成在回答 *CSM* 中发挥着主导作用，但这一更为重要且关键的问题迄今尚未受到足够关注。具体而言，查找增量匹配涉及两个主要步骤，即候选维护（包括候选生成和索引更新）以及增量匹配生成。一旦获得由图更新导致的新候选顶点，就会进行增量匹配生成，以查找 Q增加或减少的匹配。由于子图同构具有 NP-困难性，增量匹配生成占据了总体开销的主要部分。表 1 报告了 SNAP [28], 中 6 个真实图上不同算法的候选维护时间开销占总时间开销的比例，其中 CaLiG 是我们提出的方法。对于每个数据图，结果取 50 个随机生成查询的平均值。可以观察到，在大多数图上，TurboFlux 和 SymBi 的候选维护时间开销均低于 0.5%。

实际上，CSM 的索引旨在定位候选顶点，从而进一步降低增量匹配生成的成本。更好的索引不应只关注快速的候选生成，却以产生更多假阳性候选项为代价，进而降低增量匹配生成的效率。因此，在索引设计中，更新效率与候选准确性之间存在权衡。为了衡量索引的准确性，我们采用假阳性比例，即候选集中不属于任何已返回匹配的顶点所占的比例。假阳性越少，生成的候选项就越紧凑。遵循成本效益原则，我们发现 TurboFlux 和 SymBi 的索引还不够紧凑。多花费一点时间来获得更紧凑的候选项是值得的，这样可以显著加快后续的增量匹配生成。

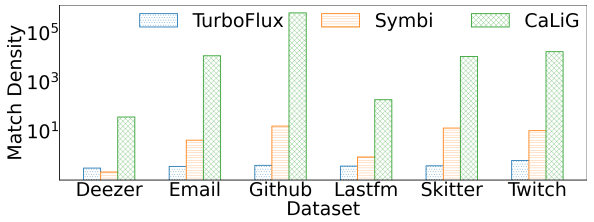


图 2. 在每个图的 50 个随机查询上，不同算法的平均匹配密度 h.

增量匹配生成的一种常见方法是在搜索空间中进行回溯，即每次将部分匹配扩展一个顶点，并在失败或最终成功时进行回溯。为了减少回溯，人们希望通过更少的回溯返回更多匹配，也就是说，一次回溯应当生成更多匹配。因此，我们将指标 匹配密度定义为一次回溯生成的平均匹配数量。匹配密度越高，表明算法能够通过更少的回溯找到全部匹配，通常运行时间也更短。图 2 展示了 TurboFlux、SymBi 和 CaLiG 的平均匹配密度。显然，TurboFlux 和 SymBi 的匹配密度都较低，一次回溯平均只能找到约 0.2~10 个匹配，这意味着大多数回溯都是无效的，仍有很大的改进空间。

1.3 我们的方法与贡献

如上文所述，大多数已有方法致力于候选维护，而不是直接降低匹配生成的成本。与这些方法不同，我们提出了两项技术，即一种新颖的索引结构 CaLiG（用于获取更紧的候选项）和一种强大的增量匹配范式（用于避免不必要的回溯），旨在将 CSM 的总体成本降至最低。

首先，我们提出了一种高性价比的结构，称为 候选照明图（简称 *CaLiG*），它是一种有向图，其中每个节点表示一对顶点的匹配对（u，v）。与已有方法相比，CaLiG 能提供更紧的候选空间，从而减少搜索空间和总回溯次数。所谓高性价比，是指为了实现更紧的候选维护，构建索引是值得的，因为这能显著加快后续的增量匹配生成，即使可能需要多花费一点时间。

此外，我们提出了一种高效的增量匹配范式——核与壳搜索（简称 *KSS*）。其基本原理是，有些顶点无需在反复回溯中逐一扩展。KSS 将查询分解为条件核顶点和壳顶点。首先通过回溯计算核顶点的部分匹配，随后无需任何回溯，只需联接壳顶点的候选项，即可立即生成完整增量匹配。我们证明，寻找最小条件核集是一个 NP难问题，因此提出了一种高效的贪心方法。为评估所提方法的效率，我们在真实图上进行了大量实验。实验结果证实了我们的方法具有显著优势。

总之，我们的贡献如下。

- 我们提出了一种经济高效的索引 CaLiG，其候选维护效果优于现有方法，从而减少了总回溯次数。
- 我们设计了一种新颖的子图匹配范式，称为核与壳搜索。该方法只需连接壳顶点的候选项即可生成增量匹配，从而减少无望的回溯。

表2。符号说明

符号说明	描述
Q 和 G	查询图和数据图
$u \in V_Q$ 和 $v \in V_G$	查询顶点和数据顶点，分别为
$e(v_i, v_j) \in E_G$	v_i 和 v_j 之间的边
$N_Q(u)$	图 Q 中 u 的邻居顶点
$N_G(v)$	图 G 中 v 的邻居顶点
d_Q 和 d_G	Q 和 G 的最大度，分别为
(u, v)-匹配对	一个匹配对由 u 和 v 组成
(u, v)-MP 状态	(u, v)-匹配对的点亮状态
$In_{CaLiG}(u, v)$	CaLiG 中 (u, v)-匹配对的入邻居
$Out_{CaLiG}(u, v)$	CaLiG 中 (u, v)-匹配对的出邻居
$BI(u, v)$	(u, v)-匹配对的二部图

- 我们形式化定义了最小条件核集问题，并证明了其NP-困难性。为了增强连续子图匹配，KSS 会针对不同的更新边动态选择最佳的条件核和壳集合。
- 在真实图上进行的大量实验表明，我们提出的方法在性能上远超最先进算法数个数量级。

2 问题定义与概述

本文重点研究无向顶点标记图，但我们的算法可以扩展到有向图和边标记图，其中每个顶点或边都有一个标签。对于边标记图，我们可以将每条边视为连接两个端点的一个顶点，从而构建顶点标记图；新添加顶点的标签与原边的标签相同。

图表示为一个元组 $G = \{V_G, E_G, L\}$，其中 V_G 和 E_G 分别是顶点集和边集。L 是将标签（即 $L(v)$）映射到顶点 $v \in V_G$ 的函数。表2列出了本文中经常使用的符号。

2.1 问题定义

定义 2.1（子图同构）。给定一个查询图 $Q = \{V_Q, E_Q, L\}$ 和一个数据图 $G = \{V_G, E_G, L\}$，Q 与 G 子图同构，当且仅当存在一个单射函数 $f: V_Q \to V_G$，使得

(1) $\forall u \in V_Q$，有 $L(u) = L(f(u))$，其中 $f(u) \in V_G$；
并且 (2) $\forall e(u_1, u_2) \in E_Q$，有 $e(f(u_1)\ f(u_2)) \in E_G$。

子图匹配返回 G 中与 Q 同构的所有子图。每个匹配都可以表示为一组一对一匹配对 $\{(u \leftrightarrow f(u))\}$。所有匹配记为 M。

定义 2.2（流图）。如果一个图 G 按照一系列更新操作 ΔG 动态变化，则称其为流图；这些更新操作包括四种类型，即顶点添加/删除和边添加/删除。

由于我们的任务中的查询图是连通的，因此新添加的顶点显然是数据图中的孤立顶点，无法包含在任何子图匹配结果中。顶点删除仅针对孤立顶点，而非孤立顶点的删除可视为删除其所有相连边。因此，按照惯例，我们只考虑边添加和删除 [32]。

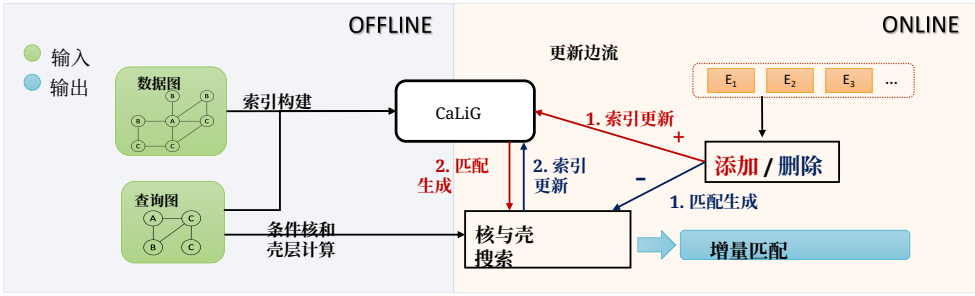


图3. 方法概述。

问题陈述 1.（连续子图匹配，简称 *CSM*）。给定一个查询图 Q、一个数据图 G和一个图更新流 ΔG，连续子图匹配任务是针对 ΔG 中的每个更新操作，查找 Q 的增量匹配（即新增或减少的子图匹配）。

示例 2.1. 考虑图*1(b)*中的数据图。假设有两个更新操作，即先删除边 $e(v_4, v_6)$，再添加边 $e(v_2, v_6)$。由于删除了边 $e(v_4, v_6)$，左侧的匹配随之消失。在添加边 $e(v_2, v_6)$后，出现了一个新的匹配（图*1(c)*右侧）。

2.2 方法概述

图3展示了所提出方法的框架，包括离线索引阶段和在线查询阶段。

在离线阶段，我们开发了一种新颖的索引，即**候**选项**点**亮**图**，简称 CaLiG。索引结构 CaLiG 根据输入查询图 Q和数据图 G构建。

定义 2.3（匹配对）。一对顶点 u 和 v 构成一个匹配对，简称为(u, v)-*MP*，当且仅当 $L(u) = L(v)$。每个(u, v)-*MP* 都具有“开启”或“关闭”两种点亮状态，分别表示 v是否可以匹配 u。令(u, v)-*MP.state* 表示点亮状态。

CaLiG 将候选匹配对组织成一个顶点对图，其中每个节点表示查询顶点 u与数据顶点 v构成的一对 (u, v)。CaLiG 引入点亮状态（“开启”或“关闭”）来表示 $v \in V_G$是否匹配 $u \in V_Q$，即 v是否为 u的候选项。由于捕获了查询图和数据图中的所有连接关系，因此可以轻松维护更严格的候选项，并支持增量匹配生成。此外，我们还为查询图中的每条边计算一个特定的条件核集合和一个壳集合。

在在线阶段，系统会根据接收到的更新操作采取不同的响应方式。对于边添加，首先更新索引 CaLiG 以获得新的候选项，然后执行子图匹配（如图 3 中红线所示）；对于边删除，则先执行子图匹配，然后更新 CaLiG（如图 3 中深蓝线所示）。

CaLiG 更新。为支持后续更新的增量子图匹配，需要实时维护索引 CaLiG。添加或删除边都可能导致候选项更新以及 CaLiG 的变化，包括结构和照明状态的变化。此外，一个匹配对状态的变化可能传播到其他匹配对，从而在 CaLiG 上启动递归传播。

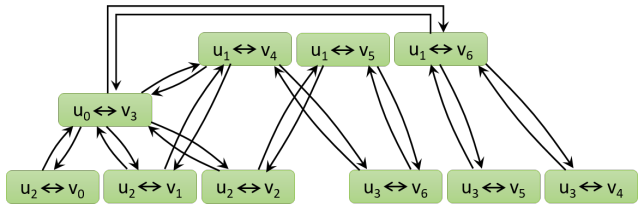


图4. 图1中查询图和数据图对应的CaLiG，其中所有匹配对的点亮状态均为“开启”（以绿色填充表示）。

增量匹配。为了高效地完成流图上的增量匹配，我们在子图匹配阶段设计了一种新的匹配范式，即核与壳搜索（KSS）。KSS 将查询图的顶点划分为条件核顶点和壳顶点。它首先将更新边的候选项作为初始部分匹配，然后扩展这些匹配以覆盖所有核顶点。最后，只需连接壳顶点的候选项即可轻松生成增量匹配，无需任何回溯。

3 CaLiG：候选项点亮图

为减少中间结果（即搜索中的失败和回溯），我们采用了精心设计的索引CaLiG。

3.1 CaLiG结构

根据子图同构的定义，顶点 $v \in V_G$ 是否与 $u \in V_Q$ 匹配取决于 v 的邻居顶点是否与 u的邻居顶点匹配。因此，CaLiG将匹配对（u，v）-匹配对组织成一张图，以便轻松维护其点亮状态。

定义 3.1（CaLiG 索引）。Q和 G 的 *CaLiG* 索引是一个有向图，其中每个节点表示一个匹配对。如果满足以下条件，则从（u_i，v_j）-匹配对到（u_k，v_l）-匹配对存在一条有向边：*1*） e（u_i，u_k）$\in E_Q$以及 e（v_j，v_l）$\in E_G$；以及 *2*）（u_i，v_j）-匹配对处于“开启”状态，或在（u_k，v_l）-匹配对之后（u_i，v_j）-匹配对被“关闭”。

在 CaLiG 中，我们使用“节点”一词来区分 Q或 G中的顶点。请注意，所提出的索引是为连续子图匹配而设计的，每个匹配对的点亮状态可能会根据图更新而更新。初始时，CaLiG 中所有节点的点亮状态均为“开启”，因为一开始所有匹配对都未被剪枝。

示例3.1。图 *4* 展示了基于图*1*中的查询图和数据图构建的 *CaLiG* 索引。查询图包含 *1* 个 *A* 顶点、*1* 个 *B* 顶点和 *2* 个 *C* 顶点，而数据图包含 *1* 个 *A* 顶点、*3* 个 *B* 顶点和 *3* 个 *C* 顶点，因此 *CaLiG* 包含 $1 \times 1 + 1 \times 3 + 2 \times 3 = 10$ 个节点。由于 e（u_1，u_2）$\in E_Q$、e（v_1，v_4）$\in E_G$以及（u_1，v_4）-匹配对均处于“开启”状态，因此存在一条从（u_1，v_4）-匹配对指向（u_2，v_1）-匹配对的边。

算法1展示了构建 CaLiG 并细化匹配对的照明状态的过程（关闭确实满足匹配要求的节点）。首先，我们生成 CaLiG 的匹配对（节点）（第2至5行），其中每个匹配对的初始照明状态为开启（第4行）。接下来，我们生成 CaLiG 中的边（第6至9行）。对于 CaLiG 中的每个节点 (u, v)-匹配对，我们遍历 u 在 Q 中的所有邻居顶点，以及 v 在 G中的所有邻居顶点。如果邻居顶点对 (u'，v') 在 CaLiG 中构成一个匹配对，则会添加一条从 (u'，v')-匹配对指向 (u，v)-匹配对的边，因为所有匹配对初始时均处于“开启”状态。最后，我们需要通过更新

照明状态（第10行）来初始化 CaLiG 的结构，具体内容将在第3.3节中详细介绍。请注意，CaLiG 初始化后，增量匹配（如果有）必须嵌入所有开启状态的节点中。

3.2 点亮状态计算

根据子图同构的定义（定义2.1），我们知道，仅当 v 的邻居也与 u 的邻居匹配时，v 才会与 u 匹配。现有的过滤规则会检查每个查询顶点的邻居是否存在候选项，但会忽略子图同构所必需的单射要求。尽管 u 的每个邻居都能在 v 的邻居中找到一个候选项，但当 v 的一个邻居被选作 u 的多个邻居的候选项时，v 的邻居仍可能无法与 u 的邻居匹配，从而违反单射要求。因此，为了计算 CaLiG 中节点的点亮状态，我们为每个匹配对引入一个二分图。

定义3.2（(u,v)-匹配对的二部图）。对于 (u,v)-匹配对，其二部图记为 $BI(u,v)$，是一个具有两个不相交顶点集合 $N_Q(u)$ 和 $N_G(v)$ 的二分图；当 (u_i, v_j)-匹配对是 $CaLiG$ 中 (u,v)-匹配对的入邻居时，$u_i \in N_Q(u)$与 $v_j \in N_G(v)$之间存在一条边。

对于一个匹配对（u，v），要确定 v 是否与 u 匹配，我们只需考虑 CaLiG 中（u，v）-匹配对的入邻居，而不必检查所有顶点对 $\{(u_i, v_j) | u_i \in N_Q(u) \wedge v_j \in N_G(v) \wedge L(u_i) = L(v_j)\}$。

例3.2。让我们考虑图*1*中的（u_1，v_6）-匹配对和（u_1，v_4）-匹配对。它们的二部图如图*5*所示。u_2 的邻居 u_1 没有任何候选项，这意味着当我们将（$v_6 \leftrightarrow u_1$）作为一个部分匹配时，u_2将没有可用的候选项，因此该部分匹配将失败。也就是说，v_6 不应是 u_1 的候选项。因此，（u_1，v_6）-匹配对的点亮状态应为“关闭”。显然，精确的匹配对点亮状态有助于收紧候选项，从而提高查找增量匹配的时间效率。

基于子图同构，只有当 $BI(u, v)$ 存在单射匹配时，v 的邻居才能匹配 u的邻居。

定义3.3（单射匹配）。给定一个具有两个不相交顶点集合 X和 的二部图，如果 X中的每个顶点 u都通过一条边与 Y 中的某个顶点匹配，并且匹配中的所有边彼此独立，即所有边不共享顶点，则存在单射匹配。

算法1： ConstructCaLiG(G, Q)

输入： 一个数据图 G、一个查询图 Q

输出： 一个候选照明图 $CaLiG$

```
1  CaLiG ←GenerateEmptyCaLiG();
2  对于每个 (u, v) ∈(V_Q, V_G) 执行
3      若 L(v) = L(u) 则
4          (u,v)-MP 状态 ← ON;
5          向 CaLiG 添加节点 (u,v)-匹配对;
6  对于 每个 (u,v)-匹配对 ∈ CaLiG 执行
7      对于 每个 (u′,v′) ∈(N_Q(u) N_G(v)) 执行
8          若 (u′,v′)-匹配对 ∈ CaLiG 则
9              从 (u′, v′)-匹配对 添加一条边到 (u, v)-匹配对;
10 IndexInitialization(CaLiG);
11 返回 CaLiG;
```



(a) 在二部图 $BI(u_1,\ v_6)$ 的 $(u_1,\ v_6)$-匹配对中不存在单射匹配。

(b) 在二部图 $BI(u_1,\ v_4)$ 的 $(u_1,\ v_4)$-匹配对中存在单射匹配。

图5. $(u_1,\ v_6)$-匹配对和 $(u_1,\ v_4)$-匹配对对应的二部图；蓝色边表示单射匹配。

对于一个（$u,\ v$）-匹配对，如果其二部图对于 N_Q（u）不存在单射匹配，则至少有一个顶点$u_i \in N_Q$（u）无法匹配。我们可以安全地将点亮状态设置为“关闭”。

引理3.1. 定义*3.2*中定义的二部图足以确定匹配对 (u, v)-匹配对的点亮状态。给定一个 (u, v)-匹配对，当且仅当二部图 $BI(u, v)$中不存在 $N_Q(u)$ 的单射匹配时，其点亮状态为“关闭”；否则，状态为“开启”。

证明。二部图中的边表示 CaLiG 中（$u,\ v$）-匹配对的入邻居。根据定义 3.1，无论（$u,\ v$）-匹配对处于何种状态，（$u,\ v$）-匹配对的非入邻居都必定处于关闭状态。处于关闭状态的节点表示它不属于任何匹配，因此在确定（$u,\ v$）-匹配对的状态时，无需考虑该关闭状态节点。如果 $BI(u,\ v)$ 对 $N_Q(u)$ 没有任何单射匹配，则至少有一个顶点$u_i \in N_Q(u)$ 无法匹配，从而使（$u,\ v$）-匹配对处于关闭状态。□

例 3.3。 如图*5(a)*所示，BI（u_1, v_6） 没有任何单射匹配，因此我们将（u_1, v_6）-匹配对的状态改为关闭。在图*5(b)*中，存在如蓝色线条所示的单射匹配，因此（u_1, v_4）-匹配对仍保持开启状态。

计算二部图（也称为二分图）的单射匹配，可以采用 Hopcroft–Karp 算法，时间复杂度为 [21] (O)，其中 $\sqrt{|V||E|}$ 表示二部图中的顶点数，$|V|$ 表示边数。实际应用中，只需判断是否存在这样的单射匹配，而不必关心具体的匹配结果。不存在单射匹配的情况有两种显而易见的情形。

-情况 1：二分图中的一个顶点 u_i 在 $N_Q(u)$ 中没有关联边，也就是说没有候选项。

-情况 2：二分图中至少有一条关联边的顶点数 $N_G(v)$ 小于 $|N_Q(u)|$。

第二种情况下，我们只考虑至少有一条关联边的顶点，因为每条边都表示一个候选项关系，而且二分图可能会随着边更新流而变化。

计算单射匹配与霍尔婚配定理 [15]密切相关。

定理 3.2。 （霍尔婚配定理 *[15]*）给定一个二部图，其中包含两个不相交的顶点集合 X和 Y。对于 X 的一个子集 W，令 N（W）表示 W的邻居，即 Y的一个子集，其中的顶点与 W中的某些顶点相连。当且仅当对于X的每个子集W，都有 $|W| < |N$（W）$|$时，才存在单射匹配。换言之，X的每个子集 W在 Y中都有足够多的邻居顶点。

上述两种情况确实是霍尔婚配定理的特例，其中情况 1 中的 $W = \{u_i\}$ 和情况 2 中的 $W = X$。

引理 3.3。 由 $BI(u,v)$ 中的边连接的顶点对构成 $\{(u_i, v_j)|u_i \in N_Q(u) \wedge v_j \in N_G(v) \wedge L(u_i) = L(v_j)\}$ 的一个子集。

证明。根据 CaLiG（定义3.1）和二部图（定义3.2）的定义，证明是直接的。□

通常，如果 (u,v)-匹配对的二部图包含更少的边，则计算单射匹配的时间开销可以降低。理想情况下，只保留 CaLiG 中与 (u,v)-匹配对的入邻居相对应的必要边。接下来，可以通过传播机制进一步减少 (u,v)-匹配对的入邻居。

3.3 CaLiG初始化

在 CaLiG 中，一个节点的点亮状态由其邻居的状态决定。当一个节点 (u,v)-匹配对的点亮状态变为“关闭”，即从 u 的候选集中剪除 v 时，(u,v)-匹配对的邻居将受到影响。如果其中一个邻居变为“关闭”，则其邻居也会递归地受到影响。这一过程称为关闭状态传播，它会在 CaLiG初始化中细化候选项。

关闭状态传播。利用引理 3.1，可以初步确定每个匹配对的点亮状态（即第一轮检查）。关闭状态的匹配对无法包含在任何子图匹配中。因此，可以安全地删除其他匹配对的二部图中与这些关闭状态匹配对对应的边，这可能会进一步关闭一组匹配对。该过程迭代进行，直到没有新的匹配对可以被关闭。最后，开启状态的匹配对直接生成候选项。索引 CaLiG 完成初始化，准备处理图更新。

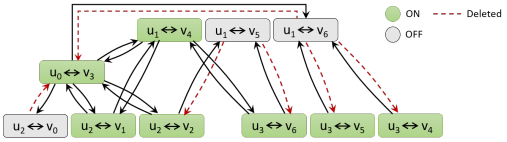
算法 2 概述了初始化过程，其中第 6 行的过程（算法3）可视为在 CaLiG 上进行传播。令 $Out_{CaLiG}(u,v)$ 表示 CaLiG 中 (u,v)-匹配对的出邻居。对于每个点亮状态为开启的节点 (u',v')-匹配对（第 2 行），我们删除从 (u,v)-匹配对指向 (u',v')-匹配对的边（第 3 行），因为 (u,v)-匹配对不会影响 (u',v')-匹配对的状态。然后检查更新后的二部图 $BI(u',v')$。如果 (u',v')-匹配对被关闭，则递归调用关闭传播过程（第 5-7 行）。

示例 3.4。考虑图 4 中的 CaLiG。由于 (u_2,v_0)-匹配对、(u_1,v_5)-匹配对和 (u_1,v_6)-匹配对的二部图都不存在单射匹配，因此它们将在第一轮被关闭。同时，我们移除出边（红色虚线边），如图 6(a) 所示。在第二轮中，如图 6(b) 所示，(u_2,v_2)-匹配对、(u_3,v_5)-匹配对和 (u_3,v_4)-匹配对被关闭。在这一轮中，只删除了从 (u_2,v_2)-匹配对指向 (u_0,v_3)-匹配对的边，因为 (u_0,v_3)-匹配对是唯一处于开启状态的出邻居。最后，没有节点可以进一步被关闭，只有节点 (u_0,v_3)-匹配对、(u_2,v_1)-匹配对、(u_3,v_6)-匹配对，以及

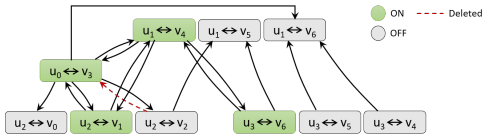
算法 2： IndexInitialization(*CaLiG*)

输入： 候选照明图 *CaLiG*

1 **对**每个(u,v)-匹配对 $\in CaLiG$ **执行**
2 为 (u,v)-匹配对构建二部图 $BI(u,v)$;
3 **对**每个(u,v)-匹配对 $\in CaLiG$ **执行**
4 **如果** (u,v)-匹配对状态 $= ON$且 $BI(u,v)$ 不存在单射匹配**则**
5 (u, v)-MP 状态 $\leftarrow OFF$;
6 关闭传播$(CaLiG, (u, v)$-匹配对$)$;

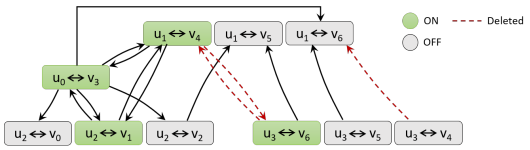


（a）步骤 1。关闭所有没有任何单射匹配的节点。

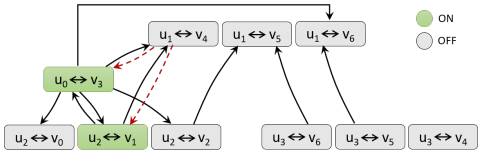


（b）步骤 2。关闭传播。

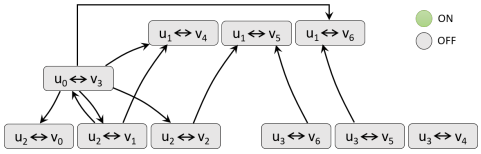
图 6。CaLiG初始化。



(a) 步骤1：删除 CaLiG 中所有更新边。



(b) 步骤2：两个节点被关闭。



(c) 步骤3：关闭传播。

图7. 删除边（v_4， v_6）时 CaLiG 的更新。

(u_1, v_4)-匹配对处于开启状态。事实上，这四个匹配对可以构成查询图的一个子图匹配。

传播过程，即 CaLiG初始化，只需传播至处于“开启”状态的节点。对于每个开启节点（u， v）-匹配对，其关闭的入邻居已在上一轮关闭入邻居时被删除，因此剩余的匹配对全部是开启节点。随后，将通过寻找单射匹配重新检查（u， v）-匹配对的状态。简而言之，关闭传播过程只会在处于开启状态的节点之间传播，先前已关闭的节点不会被重复检查。

算法3： 关闭传播($CaLiG$, (u， v)-匹配对)

输入： $CaLiG$ 以及上一轮被关闭的匹配对(u， v)-匹配对

```
1 对于 每个(u′, v′)-MP ∈ Out_CaLiG(u, v) 执行
2     如果 （u′, v′）-MP 状态 = ON则
3         删除从 (u， v)-MP 到 (u′， v′)-MP 的边;
4         更新 BI(u′， v′);
5         如果 BI（u′, v′）没有单射匹配 则
6             (u′， v′)-MP 状态 ← OFF;
7             关闭传播(CaLiG， (u′， v′)-匹配对);
```

与 SymBi 相比。SymBi 构建查询有向无环图，并基于弱嵌入细化候选项。它关注的是从 $N_Q(u)$ 到 $N_Q(v)$ 的映射是否存在，使得复用的候选项无法被排除。相比之下，CaLiG 从 $N_Q(u)$ 计算单射匹配至 $N_Q(v)$)，并沿 CaLiG 中的边进行传播。

因此，*CaLiG* 计算出的候选项与 *SymBi* 相同或更严格。

引理 3.4。 *CaLiG* 计算出的候选项与 *SymBi* 相同或更严格 .

证明。SymBi 提出一个索引 $D_i[u,v]$ 来维护动态候选空间。递归过程描述如下：

- $D_1[u,v]=1$ 当且仅当 $\exists v_p$ 与 v 相邻，使得 $D_1[u_p,v_p]=1$ 对于有向无环图 $\hat{}q$ 中 u 的每个父节点 u_p 均成立；
- $D_2[u,v]=1$ 当且仅当 $D_1[u,v]=1$ 和 $\exists v_c$ 与 v 相邻，且对于有向无环图 $\hat{}q$ 中 u 的每个父节点 u_p，均满足 $D_1[u_c,v_c]=1$。

SymBi 将 $\{v_i|D_2[u,v]=1\}$ 视为查询顶点 u 的候选项。通过将过滤迭代次数从仅执行 2 次扩展为持续迭代直到候选空间收敛，并将需要检查的顶点范围从父节点/子节点扩展到所有邻居顶点，可以得到更严格的过滤规则，即

- 当且仅当 $Cand(u)$ 中的顶点 v 与 v 相邻，且对于 q 中的 u 的每个邻居 u_p，均存在 $Cand(u)$ 中的 v_n 满足 v_n。

这就是 CaLiG 候选空间的第三个约束。因此，CaLiG 计算出的候选项至少等于 SymBi 计算出的候选项。□

复杂度分析。CaLiG 中的节点数为 $O(|V_G|\times|V_Q|)$，边数为 $O(|E_G|\times|E_Q|)$。相应地，最坏情况下，CaLiG 中节点（u， v）-MP 的二部图是完全二分图，其存储开销为 $O(|N_Q(u)|\times|N_G(v)|)$，其中 $|N_Q(u)|$ 和 $|N_G(v)|$ 分别是 $u\in Q$ 和 $v\in G$ 的度。因此，总体空间开销为 $O(|E_G|\times|E_Q|+\sum_{(u,\ v)\in CaLiG}|N_Q(u)|\times|N_G(v)|)=O(|E_Q|\times|E_G|)$。在 CaLiG 初始化过程中，CaLiG 中的每条边最多被访问一次，并以 $O(d_Q^{2.5})$ 的时间开销为关联节点计算单射匹配，其中 d_Q 是 Q 的最大顶点度。因此，总体时间复杂度为 $O(|E_Q|\times|E_G|+|E_Q|\times|E_G|\times d_Q^{2.5})=O(|E_Q|\times|E_G|\times d_Q^{2.5})$。

4 CaLiG 的动态更新

通常，我们希望尽可能快速地生成紧致的候选项。得益于 CaLiG，只需从新添加或删除边的关联节点出发探索 CaLiG，即可轻松更新候选项。核心思想是，边添加或边删除可能导致某些节点的状态改变，进而沿 CaLiG 传播。为处理这些更新，我们分别介绍如何处理边删除（第 4.1 节）和边添加（第 4.2 节）。

4.1 边删除

从数据图 G 中删除一条边，可能会导致某些候选项无法匹配查询顶点（使 CaLiG 中的某些匹配对关闭），从而减少子图匹配数。当从 G 中删除一条边时，我们首先删除 CaLiG 中所有相关的边，然后采用关闭传播，通过更新点亮状态来精炼候选项。

具体而言，令 $e(v_1,\ v_2)$ 表示待删除的数据边（数据图 G 中的边）。由于 $e(v_1,\ v_2)$ 可能是具有相同标签的任意查询边 $e(u_1,\ u_2)$（查询图 Q 中的边）的候选项，因此会导致 CaLiG 中的边删除。该过程与 CaLiG初始化类似，并采用关闭传播（算法3）。边删除的处理细节如算法4所示。首先，从 CaLiG 中删除 $(u_1,\ v_1)$-匹配对与 $(u_2,\ v_2)$-匹配对之间的边（第2至3行）。然后检查两个受影响的节点 $(u_1,\ v_1)$-匹配对和 $(u_2,\ v_2)$-匹配对。如果它们因删除边 $e(v_1,\ v_2)$ 而被关闭，则将调用关闭传播，以进一步细化其他节点。

示例4.1。 以图*1*为例，其中边 (v_4, v_6) 被删除。如图*7(a)*所示，我们删除：*(1)* (u_1, v_4)-匹配对与 (u_3, v_6)-匹配对之间的边；以及 *(2)* 从(u_3, v_4)-匹配对到(u_1, v_6)-匹配对的边。节点 (u_1, v_4)-匹配对和 (u_3, v_6)-匹配对原本处于开启状态，因此我们分别进行二部图检查。如图*7(b)*所示，由于边删除，这两个节点在二部图中都不存在单射匹配，因此会被关闭并触发关闭传播。最终，所有节点都会被关闭，如图*7(c)*所示，这表明不存在查询的匹配。

复杂度分析。令 D 表示因更新而被关闭的节点集合。对于 D中的每个节点，我们计算二部图 $BI(u, v)$的单射匹配，并将更新过程传播到其邻居。因此，删除一条边时更新 CaLiG 的时间复杂度为 $O(\sum_{(u,v)\in D}(|N_{CaLiG}(u, v)| + |N_Q(u)|^{2.5}))$，其中 $N_{CaLiG}(u, v)$是 CaLiG 中(u, v)-匹配对的邻居集合。时间开销似乎略高，但考虑到它能减少繁琐的回溯，这样的开销是合理的；换言之，更新 CaLiG 是划算的。

4.2 边添加

向数据图中添加一条边，为开启处于关闭状态的匹配对创造了机会。基于 CaLiG，不难得出新的候选项。直观来说，边添加的更新过程与边删除类似。我们首先将所有相关边添加到 GaLiG 中，然后通过开启传播检测新的候选项。在介绍处理边添加的具体细节之前，先介绍 CaLiG 的两个性质如下。

引理4.1。 对于 *CaLiG* 中的任意节点 （u、 v）-匹配对，其所有入邻居在 （u、 v）-匹配对之后要么处于开启状态，要么被关闭。

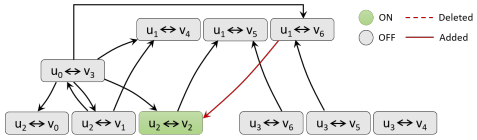
证明。假设（u'， v'）-匹配对是（u， v）-匹配对的一个入邻居，并且它在（u， v）-匹配对之前被关闭（即，当（u'， v'）-匹配对被关闭时，（u， v）-匹配对处于开启状态）。根据关闭传播，一旦（u'， v'）-匹配对被关闭，从（u'， v'）-匹配对指向（u， v）-匹配对的边就会被删除。因此，（u'， v'）-匹配对不可能是（u， v）-匹配对的入邻居。□

如第3.1节所述，所有节点的照明状态都初始化为开启，随后在离线阶段通过关闭传播进行更新。然而，这一传播过程不可逆。也就是说，如果关闭(u_1, v_1)-匹配对导致(u_2, v_2)-匹配对关闭，那么由于边添加而使(u_2, v_2)-匹配对开启，并不会使(u_1, v_1)-匹配对开启，因为当(u_2, v_2)-匹配对仍处于开启状态时，(u_1, v_1)-匹配对就已经关闭了。

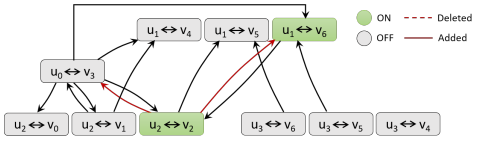
算法4： UpdateCaLiGForDel($CaLiG$、 $e(v_1, v_2)$)

输入： $CaLiG$以及要删除的更新后的边 $e(v_1, v_2)$

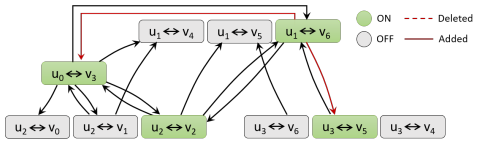
```
1 对于每个 e（u1， u2） ∈ EQ执行
2     如果 L(u1) = L(v1) 且 L(u2) = L(v2) 则
3         从 CaLiG 中删除 (u1， v1)-匹配对与 (u2， v2)-匹配对之间的边;
4         如果 （u1， v1） -匹配对状态 = ON和 BI （u1， v1） 没有单射匹配则
5             (u1， v1)-匹配对状态 ← OFF;
6             关闭传播(CaLiG， (u1， v1)-匹配对);
7         如果(u2, v2)-MP 状态 = ON且 BI(u2, v2) 没有单射匹配 则
8             (u2, v2)-MP 状态 ← OFF;
9             关闭传播(CaLiG, (u2, v2)-MP);
```



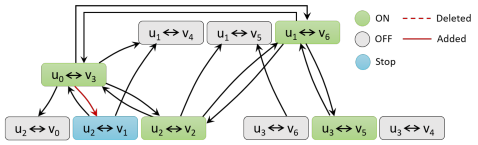
(a) 步骤 1：添加一条边，从 (u_1, v_6)-匹配对 到 (u_2, v_2)-匹配对并计算 (u_2, v_2)-匹配对的状态。



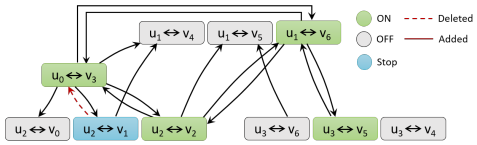
(b) 步骤 2：执行开启传播。



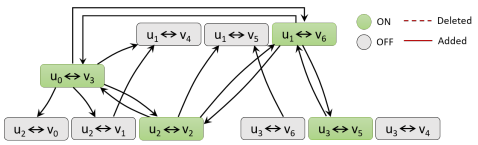
(c) 步骤 3：继续进行开启传播。



(d) 步骤 4。开启传播在 (u_2, v_1)-匹配对处停止。



(e) 步骤 5。执行关闭传播。



(f) 步骤 6。关闭传播后的 CaLiG 更新结果。

图 8。添加边 (v_2, v_6) 时 CaLiG 的更新。

引理4.2。 如果存在一条边，从处于关闭状态的(u, v)-匹配对指向处于关闭状态的(u', v')-匹配对，那么(u, v)-匹配对的开启不会导致(u', v')-匹配对开启。

证明。假设开启 (u, v)-匹配对会导致开启 (u', v')-匹配对，那么在计算 (u', v')-匹配对的点亮状态时，由于 (u, v)-匹配对处于开启状态，(u', v')-匹配对不会在 (u, v)-匹配对之前关闭，这与 (u', v')-匹配对在 (u, v)-匹配对之前关闭相矛盾。□

令 $e(v, v')$ 表示要添加的边。根据引理4.1和引理4.2，为确定是否开启匹配对 (u, v)-匹配对或 (u', v')-匹配对，其中 $L(u) = L(v)$ 且 $L(u') = L(v')$，我们只需检查其在 CaLiG 中的入邻居，而无需考虑所有顶点对 $\{(u_i, v_j) | u_i \in$

算法 5： UpdateCaLiGForAdd($CaLiG, e(v_1, v_2)$)

输入： $CaLiG$以及要添加的更新后的边 $e(v_1, v_2)$

```
1 对于每个 e（u1, u2） ∈ EQ执行
2   如果 L(v1) = L(u1) 且 L(v2) = L(u2) 则
3     将边从 (u2, v2)-匹配对添加到 (u1, v1)-匹配对;
4     如果 （u1, v1）-匹配对状态 = OFF且 BI（u1, v1）具有单射匹配则
5       (u1, v1)-匹配对.state ← ON;
6     如果 (u1, v1)-MP.state = ON 则
7       StopSet ←开启传播(CaLiG, (u1, v1)-MP);
8       当 StopSet 不为空时执行
9         (u, v)-匹配对 ← StopSet.pop();
10        关闭传播(CaLiG, (u, v)-MP);
```

$N_Q(u) \wedge v_j \in N_G(v) \wedge L(u_i) = L(v_j)\}$ 或 $\{(u_i, v_j)|u_i \in N_Q(u') \wedge v_j \in N_G(v_\prime) \wedge L(u_i) = L(v_j)\}$。在 CaLiG 中，不是 (u, v)-匹配对或 (u', v')-匹配对的入邻居的节点，已在 (u, v)-匹配对或 (u', v')-匹配对之前被关闭。同样，即使开启了 (u, v)-匹配对或 $(u', v_\prime)$-匹配对，这些节点也不会被开启。

算法 6： 开启传播（*CaLiG*，(u, v)-匹配对）

输入： *CaLiG* 以及上一轮开启的匹配对（u, v)-匹配对

输出： 一组停止节点 S

```
1  S ← ∅;
2  对于每个(u′, v′)-匹配对 ∈ In_CaLiG(u, v) 执行
3      添加一条从 (u, v)-匹配对到 (u′, v′)-匹配对的边;
4      如果(u′, v′)-MP 状态为 = OFF，则
5          如果 BI(u′, v′) 存在单射匹配，则
6              (u′, v′)-MP 状态为 ← ON;
7              S ← S ∪ 开启传播(CaLiG, (u′, v′)-匹配对);
8          else
9              S ← S ∪ (u′, v′)-匹配对;
10 返回 S;
```

算法 5 描述了基于 CaLiG 处理边添加的过程，其中涉及两个过程：开启传播（算法 6）和关闭传播。与处理边删除不同，该过程只需从 (u_2, v_2)-匹配对向 (u_1, v_1)-匹配对添加一条边，并计算 (u_1, v_1)-匹配对的点亮状态（第 2–5 行），因为它开启后会通过开启传播传播至 (u_2, v_2)-匹配对。如果 (u_1, v_1)-匹配对的状态为关闭，那么无论 (u_2, v_2)-匹配对处于开启还是关闭状态，其状态都不会改变。对于每个处于开启状态的 (u_1, v_1)-匹配对，都会调用开启传播过程，尝试开启更多节点。请注意，在开启传播之前，我们会从 (u_2, v_2)-匹配对向 (u_1, v_1)-匹配对添加一条边（算法 5 第 3 行），这表示我们逐步将 (u_2, v_2)-匹配对视为处于开启状态的节点。因此，传播可能会错误地开启某些节点。当某个节点未被开启传播开启时，该节点会被记录为停止节点。所有停止节点构成一个集合，称为 *StopSet*。对 *StopSet* 中的每个节点执行关闭传播，以确保所有新开启的节点都符合子图同构。

算法 6 展示了开启传播的详细过程。根据引理4.2，只需向其入邻居（u'、v'）-匹配对传播。如果（u'、v'）-匹配对处于关闭状态，且其对应的二部图存在单射匹配，则可以关闭（u'、v'）-匹配对，并递归调用开启传播。否则，将该节点添加到 S，即停止节点集合中。

例*4.2*。以图*8*为例，边（v_2，v_6）被添加到数据图中，如图*1(c)*中的红线所示。首先，如图*8(a)*所示，在 *CaLiG* 中从（u_1，v_6）-匹配对向（u_2，v_2）-匹配对添加一条边。随后，（u_2，v_2）-匹配对被开启。其次，我们将按照图*8(b)*所示，向（u_2，v_2）-匹配对的入邻居执行开启传播，其中节点（u_1，v_6）-匹配对和（u_0，v_3）-匹配对最初处于关闭状态。由于（u_2，v_2）-匹配对具有单射匹配，因此只有它被开启。随后，传播继续尝试开启其处于关闭状态的入邻居。相应地，（u_0，v_3）-匹配对和（u_3，v_5）-匹配对被开启，如图*8(c)*所示。当传播到（u_2，v_1）-匹配对时，它未被开启，并被标记为停止节点，如图*8(d)*所示。最后，从停止节点（u_2，v_1）-匹配对开始执行关闭传播，但在关闭传播之后，（u_0，v_3）-匹配对仍保持开启状态（图*8(e)*）。最终更新后的 *CaLiG* 索引如图*8(f)*所示。

复杂度分析。设 D 和 A 分别表示由于边添加而被关闭和开启的节点集合。边添加的时间复杂度与边删除的时间复杂度几乎相同。区别在于，我们需要考虑将 A 中的节点关闭的过程。通过添加一条边来更新 CaLiG 的总体时间复杂度为 $O\sum_{u\,v\,\in\,D\cup A}|N_{CaLiG}\,u\,v\,|+$（，）（）$|N_Q\,u\,|^{2.5}$ $N_{CaLiG}\,u\,v$ $u\,v$（，）（）），其中（，）是（，）-匹配对的邻居集合。

5 基于 KSS 的子图匹配

对于连续子图匹配，我们需要找出因边添加而产生的所有新匹配，或因边删除而减少的匹配。显然，发生变化的匹配必须包含更新边。具体而言，对于每条更新边 $e(v_i,\ v_j)$，我们需要在 CaLiG 中找到两个相连的开启状态节点 $(u_k,\ v_i)$-匹配对和 $(u_l,\ v_j)$-匹配对，它们分别包含 v_i 和 v_j。这两个节点将被视为一个部分匹配，并可扩展为完整匹配。因此，可以利用广泛使用的回溯搜索。此类回溯搜索逐个枚举每个查询顶点的候选项，并在每一步检查子图同构的约束。它通过尝试整合候选项，在候选项之间进行大量回溯以构建匹配，从而产生高昂的时间开销。

大多数现有算法通过调整匹配顺序来加速回溯。相比之下，本节旨在探寻一种更强大的回溯框架。值得注意的是，度为一的查询顶点的匹配彼此独立，其中“独立”是指计算一个查询顶点的匹配不会影响另一个查询顶点的匹配。因此，可以同时获取这些顶点的匹配，而不必逐一探索。

因此，我们为连续子图匹配开发了一种新颖的回溯搜索框架，称为核与壳搜索（KSS）。KSS 将查询顶点分解为核顶点和壳顶点，通过连接核顶点的部分匹配与壳顶点的候选项，可以直接得到增量匹配。

5.1 核与壳顶点

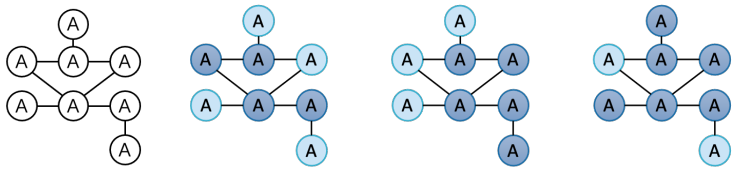
为了识别上述讨论的独立顶点，KSS 将核集合定义为连通顶点覆盖，并将壳集合定义为其补集。

定义 5.1（核集合与壳集合）。给定一个查询图 Q，其核集合（也称为连通顶点覆盖）是一个连通顶点集合，使得 Q 中的每条边至少有一个顶点属于该集合。核集合中的每个顶点称为核顶点。壳集合是核集合的补集，其中的顶点彼此独立，每个顶点称为壳顶点。

根据定义，壳顶点天然相互独立。由于 CSM 中的查询图是固定的，因此可以预先计算并存储相应的核顶点和壳顶点。

示例 5.1. 图 *9(b)*列出了查询图 *9(b)*的三组不同核集合和壳集合，其中深蓝色顶点构成核集合，其余顶点构成壳集合。

给定查询图 Q 及其核集合和壳集合，一旦确定了核集合的匹配，就可以通过检查每个壳顶点相邻的已匹配顶点来获取该壳顶点的候选项。由于壳顶点之间不存在边依赖，因此只需连接各壳顶点的候选项即可直接生成匹配，无须进行任何失败的回溯。因此，良好的分解应尽可能包含更多壳顶点，这意味着核顶点应尽可能少。如上所述，所需的增量匹配必须包含更新边 $e(v_i,\ v_j)$，且搜索从 $e(v_i,\ v_j)$ 开始。因此，核集合诱导的子图应包含边$e(u_k,\ u_l)$，以使匹配对 $(u_k,\ v_i)$-MP 和 $(u_l,\ v_j)$-MP 在 CaLiG 中处于开启状态。



（a）查询图。 （b）三个核集和壳集合。

图 9. 核顶点和壳顶点的一个示例。

定义 5.2（条件核集合）。给定查询图 Q 和边 e（u_k，u_l）$\in E_Q$，条件核集合（记为 *CKS*）是包含顶点 u_k 和 u_l 的核集合。

引理 5.1. 给定查询图 Q，通过添加一个虚拟顶点 u_0并将 u_0 与 Q中的所有顶点相连，可得到一个新的查询图 Q'。若 $S \cup \{u_0\}$ 是 Q'的最小核集，则 S是 Q的最小核集，其中 $S \subseteq V_Q$。

证明。假设 S不是 V_Q的最小核集，并且 Q存在另一个比 S更小的核集 S'。 $S' \cup \{u_0\}$将是 Q,的一个更小核集，这就导致了矛盾。□

定理 5.2. 对于查询图 Q 和边 $e(u_k, u_l) \in E_Q$，计算最小条件核集（*MCKS*）是 *NP*难的。

证明。该证明通过将 NP难的最小连通顶点覆盖（MCVC）问题归约而得 [13, 29]。

令 Q' 表示通过添加一个虚拟顶点 u_0 并将其与 Q中的所有顶点相连而得到的图。对于 Q ,中的每条边，我们都计算其 MCKS。令 S' 表示这些$|E_{Q'}|$个 MCKS 中的最小者。 S,是 Q'的一个 MCVC（即最小核集），下面通过反证法证明这一点。

假设 S'不是 Q' 的 MCVC。则必然存在另一个 Q' 的 MCVC，记为 S，使得 $|S'| > |S|$。（1）当 S不包含 u_0时，所有节点 V_Q构成 Q' 的 MCVC S，即 $S = V_Q$。如果 u_0不包含于 S'，则 $|S'| \leq |V_Q|$, 这与 $|S'| > |S|$矛盾。如果 u_0包含于 S'，则 $S' = S^* \cup \{u_0\}$，使得 $S^* \geq S$。因此，我们有 S'=$V_Q \cup \{u_0\}$。由于 $S = V_Q$， S是 Q' 的 MCKS，且存在一条边 $e(u_x, u_y)$，其中 $e(u_x, u_y) \in E_{Q'}$。这与 S' 是 Q' 的较小 MCKS 矛盾。（2）当 S 包含 u_0时，由于核集合是连通的， u_0 的至少一个邻居 u_i 将包含于 S中。 S是 Q' 的 MCKS 以及边 $e(u_0, u_i)$，这与 S' 是所有可能边中最小的一个相矛盾。

如果 $u_0 \in S$,， S,\ u_0根据引理 5.1 是一个最小核集；否则， S'是 Q的一个最小核集。显然，构造 MCVC 解的过程可在多项式时间内完成。因此，如果 MCKS 可以在多项式时间内求解，那么 MCVC 问题也可以在多项式时间内求解，这与 MCVC 问题的 NP-困难性相矛盾。□

鉴于计算 MCKS 是 NP难的，实践中提出了一种贪心算法。该算法将 u_k和 u_l加入 CKS，并从 Q中移除 u_k、 u_l及其邻居顶点。随后，它枚举剩余 Q'中的一个不相交奇环极大集合。令 t表示奇环的数量， C_i表示 i个奇环的顶点，并令 $C = \bigcup_i^t C_i$。它计算剩余二分图 $Q' \backslash C$ 的最优顶点覆盖，记为 W。如果 $C \cup W \cup \{e(u_k, u_l)\}$ 是连通的，则返回 $C \cup W \cup \{e($ u_k， $u_l)\}$；否则，我们应用 Steiner 树近似算法来获得一个连通结果。

算法 7： FindMatches($CaLiG$, $e(v_1, v_2)$, $Kernel$, $Shell$)

输入： $CaLiG$，一条更新后的边 $e(v_1, v_2)$、条件核集合 K 和壳集合 $S$for $e(v_1, v_2)$

输出： 由 $e(v_1, v_2)$ 导致的增量匹配

```
1 对每个 u_1 ∈ Q 执行
2     如果 L(v_1) = L(u_1) 且(u_1, v_1)-MP.state = ON 则
3         m[u_1] ← v_1;
4         对每个 u_2 ∈ N_Q(u_1) 执行
5             如果(u_2, v_2)-MP ∈ In_CaLiG(u_1, v_1) 则
6                 m[u_2] ← v_2;
7                 返回 KSS(m, K, S).
```

算法 8: $KSS(m, K, S)$

输入： 部分匹配 m、条件核集合 K 和壳集合 S

输出： 由 $e(v_1, v_2)$ 引起的增量匹配

```
1 如果 m.size < |K| 则
2     th ← m.size;
3     u ← K[th];
4     Cand(u) ← 生成 u的候选项;
5     对每个 v ∈ Cand(u) 执行
6         m' ← m;
7         m'[u] ← v;
8         KSS(m', K, S);
9 否则
10    对于每个 u ∈ S 执行
11        Cand(u) ← 生成 u的候选项;
12    返回 m ⋈_{u∈S} Cand（u）。
```

5.2 核与壳搜索

借助核顶点和壳顶点，我们设计了一种新颖的回溯搜索框架。当添加或删除一条边时，我们的方法首先基于更新后的边初始化部分匹配（算法 7），然后调用核搜索和壳搜索（算法 8）。对于核顶点，KSS 通过现有的回溯计算部分匹配。通过将部分匹配与壳顶点的候选项进行连接，可以立即报告增量匹配，而无需任何不必要的回溯。

算法 7 展示了基于核与壳搜索（KSS）查找增量匹配的过程。给定更新边（v_1, v_2），匹配对（u_1, v_1）-MP 必须为开启（第 2-3 行）。随后，根据（u_1, v_1）-MP 的入邻居确定匹配对（u_2, v_2）-MP（第 4-6 行）。因此，我们找到了部分匹配 $\{(u_1 \leftrightarrow v_1, u_2 \leftrightarrow v_2)\}$，记为 m。接下来，我们通过调用过程 KSS，先匹配核顶点，再匹配壳顶点，以此顺序匹配剩余的查询顶点。

算法 8 描述了如何搜索条件核与壳顶点的匹配。首先，它根据核集合确定要匹配的查询顶点（第 2-3 行），然后为该顶点生成候选项。最后，它选择一个候选项作为部分匹配，以推动匹配过程（第 5-8 行）。在所有核顶点都完成匹配后，我们为所有壳顶点生成候选项，并将这些候选项与部分匹配执行连接操作，以报告增量匹配（第 10-12 行）。该过程非常高效，因为我们通过简单的连接操作生成增量匹配，无需检查任何约束或进行回溯。

候选生成。 令 $Cand(u)$ 表示查询顶点 u 的候选项。顶点 v in$Cand(u)$ 满足以下约束：

1) v 尚未用于部分匹配。2) CaLiG 中的节点 (u, v)-匹配对处于开启状态。3) 如果 $v' \in Cand(u')$ 且 $u' \in N_Q(u)$，则 v 是 v' 的邻居。要为顶点 u 生成 $Cand(u)$，需执行两个步骤。

步骤 1。 对于每个已与 v_i 匹配的 $u_i \in N_Q(u)$，我们构建集合 $C_i = \{v' | (u', v')$-匹配对$\in In_{CaLiG}(u_i, v_i)$ 且 (u', v')-匹配对处于开启状态$\}$。然后对所有这些集合求交集 C_i，以得到 u的候选集。

步骤 2。 我们从候选集中移除已用于部分匹配的数据顶点。完成此步骤后，我们得到 u的最终候选集。

提前剪枝。 KSS 首先计算核顶点的部分匹配，这有助于匹配生成，但也会引发新的问题。例如，当我们确定了所有核顶点的部分匹配后，某个壳顶点的候选集可能为空，但只有在为该壳顶点生成候选项时，我们才能发现这个失败的部分匹配。因此，我们采用一种剪枝策略，以尽可能早地检测此类失败。对于每个壳顶点 u，一旦其所有邻居顶点都已完成匹配，我们就提前尝试为 u 寻找候选项。如果找不到候选顶点 u，则可以安全地排除当前部分匹配。否则，继续寻找其余核顶点的匹配。

复杂度分析。对于 KSS，只有核部分通过回溯搜索进行匹配，其最坏情况复杂度为 $O(|V_G|^{|K|})$，其中 $|K|$ 是核顶点的数量，远小于传统回溯搜索的时间复杂度 $O(|V_G|^{|V_Q|})$。

6 实验评估

在本节中，我们评估所提出的方法 CaLiG，并将其与两种最先进算法 TurboFlux [24] 和 SymBi [32] 进行比较。

6.1 实验设置

数据图。 表3列出了实验中使用的图，这些图均从 SNAP [28], 下载，但 Netflow（从 CAIDA [1] 下载）除外。我们随机抽取 5% 的数据边作为流式边，其中删除边与添加边的比例为 2:1。

查询图。 对于每个数据图，我们通过将顶点数量从3调整到16，采样7组子图，分别记为 $Q3$、 $Q4$、 $Q6$、 $Q8$、$Q10$、 $Q12$、 $Q14$ 和 $Q16$，作为查询图。每组 Qi 包含50个查询图。

度量指标。 运行时间以毫秒为单位报告。由于子图匹配具有NP-困难性，某些查询可能需要极长时间。按照惯例，我们为每个查询设置20分钟的超时限制。如果某个查询无法在时间限制内完成，则称其为未完成。为了全面评估算法，我们报告平均耗时、完成率、平均峰值内存使用量以及匹配密度。

所有算法均使用 C++ 实现，并在配备 Intel(R) Xeon(R) CPU E5-2640 @ 2.60GHz 和 128G 内存的 Linux服务器上进行评估。

表3。数据图。

数据集	Abb.	|V|	|E|	平均度
Last.fm	lfm	7,624	27,806	7.3
Facebook	fbm	22,470	170,823	15.2
电子邮件	em	36,692	183,831	10.0
GitHub	gh	37,700	289,003	15.3
Deezer	dz	41,773	125,826	6.0
Twitch	tw	168,114	6,797,557	80.9
Skitter	sk	1,696,415	11,095,298	13.1
Netflow	nf	3,114,895	16,668,683	10.7

表4. 平均耗时。

	耗时（毫秒）			
数据集	TurboFlux	SymBi	CaLiG	$\|\Delta m\|$
Last.fm	49,341.9	13,848.3	**641.6**	7,626,438
Facebook	2,510,220.9	1,745,489.0	**94,174.2**	443,817,336
电子邮件	2,836,668.9	1,873,555.0	**82,417.3**	338,293,270
GitHub	4,177,493.4	3,411,865.8	**189,060.7**	1,679,993,941
Deezer	368,280.5	99,535.6	**4,669.0**	14,327,860
Twitch	1,957,440.8	1,107,821.3	**51,247.5**	464,933,035
Skitter	1,460,160.6	321,649.7	**9,447.0**	130,260,509
Netflow	727,494.2	119,694.9	**122.3**	48,801

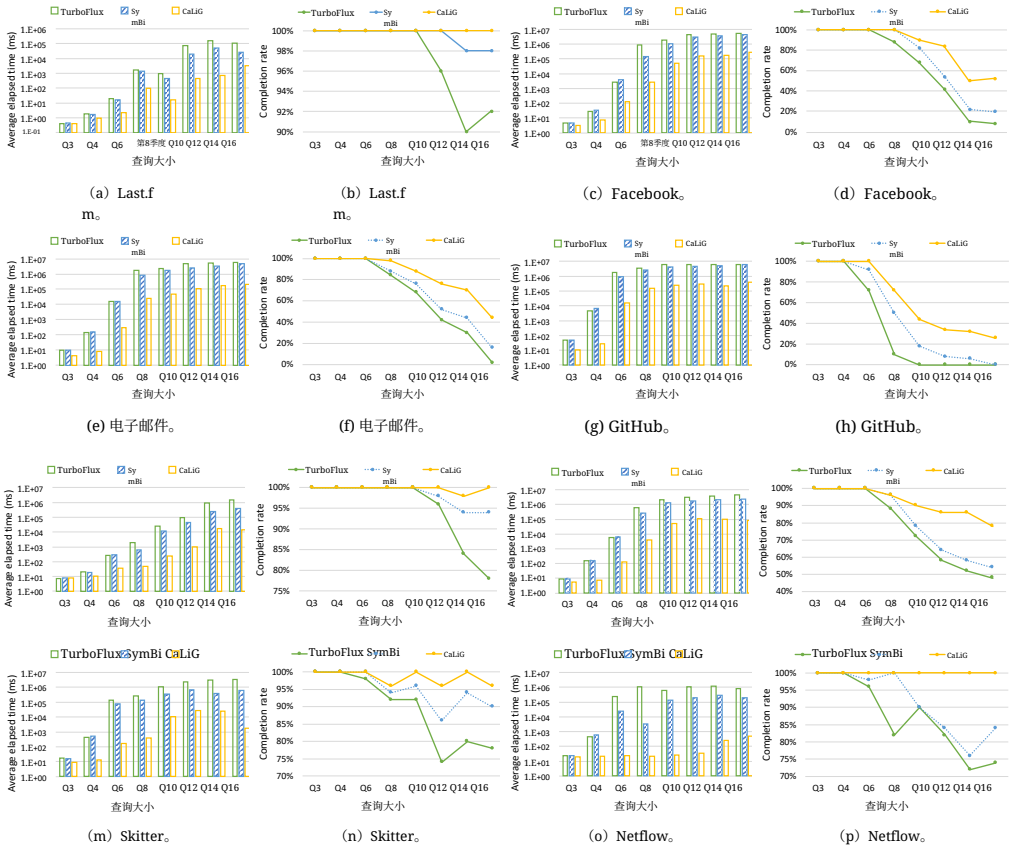
6.2 实验结果

总体性能。 表4分别报告了8个数据图上400个查询图的平均耗时。表4。如图所示，CaLiG在所有图上的运行速度都远快于TurboFlux和SymBi，分别比TurboFlux（22.10倍至5963.07倍）和SymBi（18.05倍至978.69倍）实现了显著加速。最后一列 $|\Delta m|$ 表示增量匹配的数量。我们可以发现， $|\Delta m|$ 从数百万增加到超过10亿。结果数量越多，所需时间越长。

改变查询图的大小。通过将 $|V_Q|$ 从 3 调整到 16，评估查询大小（即顶点数量）的影响。图10展示了详细结果，其中每个数据集包含两个子图，分别表示同一大小的每组查询的平均耗时和完成率。可以看出，CaLiG 相较于其他算法具有显著优势。

对于所有数据图，在时间效率方面，CaLiG 的表现始终优于 SymBi，速度约快 1 到 3 个数量级，同时完成率更高。Q_{12} 组查询的平均加速比高达 5472 倍（在 Netflow 上）。此外，CaLiG 相较于 TurboFlux 也取得了更高的加速比。随着查询大小的增加，时间开销也随之增加。对于较大的查询，CaLiG 的完成率显著更高。

改变数据图的大小。 为评估可扩展性，我们通过 LSBench [27] 随机生成了 7 个不同大小的数据图：包含 0.1、0.2、0.5、1、2、5 和 1000 万个顶点，同时数据图的其他特征保持不变。本实验使用了两组查询图，分别包含 8 个和 12 个顶点。结果如图11(a)所示。虚线表示



（a）Last.fm。 （b）Last.fm。 （c）Facebook。 （d）Facebook。

（e）电子邮件。 （f）电子邮件。 （g）GitHub。 （h）GitHub。

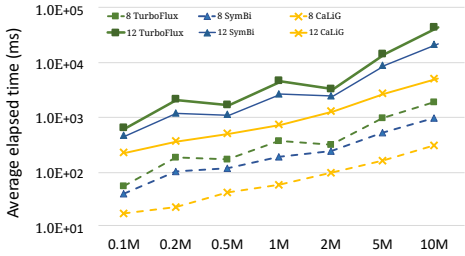
（m）Skitter。 （n）Skitter。 （o）Netflow。 （p）Netflow。

图10. 不同数据图中变化的查询图。

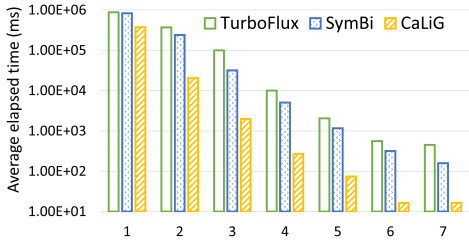
包含 8 个顶点的查询图的结果，实线表示包含 12 个顶点的查询图的结果。与 TurboFlux 和 SymBi 相比，无论数据图大小如何，CaLiG 始终具有更快的速度。此外，随着数据集规模的增长，所有算法的运行时间都呈上升趋势。

改变数据图的顶点标签数量。我们通过随机为顶点分配标签来评估标签数量的影响。对于每个原始数据图，我们生成了7个标签数量不同的数据图，标签数量分别为1、2、3、4、5、7和9。图11(b)展示了数据图*Deezer*上的结果。可以看到，CaLiG算法比SymBi快2.2至19.7倍，比TurboFlux快2.4至50.5倍。随着标签数量增加，加速效果也随之提升。此外，我们发现标签数量会影响运行时间。总体而言，时间开销与标签数量成反比，即标签越多，连续子图匹配的运行时间越短。

流数据规模的变化。接下来，我们探究流规模对算法的影响。我们分别设置了两组实验：一组用于测试混合更新流，即更新流中同时包含删除边和添加边，二者的比例为 2:1；另一组中，流内仅允许包含删除边。这两组更新流均从原始数据边中随机采样，其规模分别设置为数据边数量的 2%、4%、6%、8% 和 10%。



(a) 改变数据图的规模。



(b) 改变顶点标签的数量。

图11. 改变顶点标签的数量和数据图的规模。

表5. 不同查询规模下的匹配密度

方法	数据集	4	6	8	10	12	14
TurboFlux	*dz*	0.530	0.240	0.040	0.048	0.021	0.0001
	lfm	1.858	1.190	0.152	0.005	0.002	0.002
	sk	4.562	1.514	0.214	0.123	0.119	0.009
SymBi	*dz*	0.557	0.392	0.252	0.110	0.024	0.0009
	lfm	2.012	1.265	0.509	0.371	0.104	0.270
	sk	**35.83**	12.59	1.243	0.567	0.183	0.070
CaLiG	*dz*	**2.139**	**22.88**	**102.8**	**52.83**	**10.79**	**77.05**
	lfm	**9.834**	**199.7**	**65.36**	**422.4**	**156.8**	**1028**
	sk	30.05	**2617**	**156588**	**504142**	**113778**	**357529**

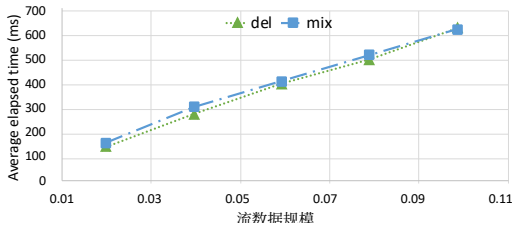
图1 2(a)报告了数据图电子邮件上的结果。结果显示，无论是混合流还是边删除流，算法性能都与更新流规模呈线性关系，表明 CaLiG 具有良好的可扩展性。此外，我们还可以发现，CaLiG 在处理边添加和边删除时表现出相近的性能。

匹配密度。为讨论回溯搜索的有效性，匹配密度（MD）定义为

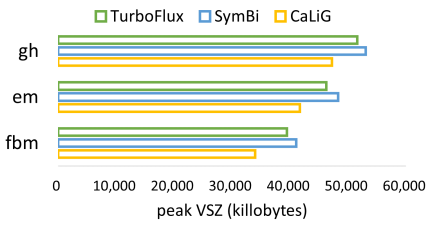
$$\text{MD} = \frac{\#\text{ of Incremental Matches}}{\#\text{ of Backtrackings}}$$

MD 越高，相同数量的匹配就能通过更少的回溯并在更短的时间内找到。表5列出了在*dz*、*lfm*和*sk*上的结果。CaLiG 的匹配密度比 TurboFlux 和 SymBi 高出最多 7 个数量级：对于大多数查询，TurboFlux 或 SymBi 中的一次回溯都无法生成一个匹配，而 CaLiG 一次回溯就能生成数千个匹配。此外，随着查询大小的增加，TurboFlux 和 SymBi 的 MD 降低，而 CaLiG 的 MD 同时增加，这表明 CaLiG 在解决大型查询方面具有更显著的优势。

内存使用量。我们通过报告峰值内存来评估内存使用量，峰值内存定义为“ps”工具输出中的虚拟内存集大小（VSZ）的最大值。图12(b)展示了在图*Facebook*、电子邮件和*GitHub* 上针对包含8个顶点的查询图得到的结果。在所有图上，CaLiG的内存使用量都低于TurboFlux和SymBi。



（a）流数据规模变化。



（b）内存使用量。

图 12. 流数据规模变化和内存使用量。

消融实验。 我们评估了本文提出的每项独立技术的效果。用于比较的方法包括

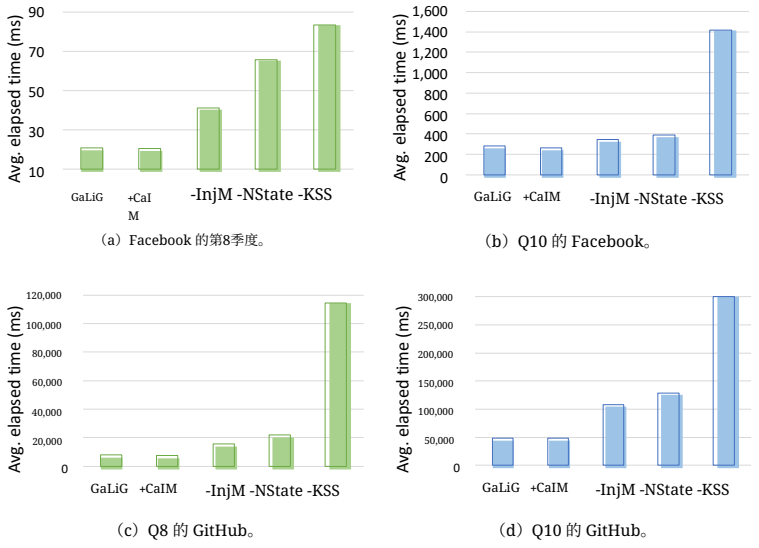
(1) *CaLiG*，所提出算法的完整版本；(2) *+CaIM*，缓存已计算出的单射匹配，以避免不必要的重新计算。如果更新边不涉及缓存的单射匹配，则无需重新计算单射匹配；(3) *-InjM*，禁用单射匹配检查，仅考虑二部图中的 $N_Q(u)$ 每个顶点是否都有相连的边；(4) *-NState*，禁用点亮状态的更新和检查；(5) *-KSS*，用每次匹配一个扩展顶点的方式替代 KSS 技术。

图13展示了在 Facebook 和 GitHub 上的结果，其中 $Q8$ 和 $Q10$ 分别表示包含 8 个和 10 个顶点的查询图。我们可以观察到，缓存单射匹配带来的改进微乎其微，因为计算单射匹配的开销远小于总体开销。相比之下，禁用其他技术时，运行时间会显著增加。其中，KSS 的效果最为突出，平均实现了 4.0X~14.7X 的加速，因为增量匹配生成占据了总开销的大部分。

7 相关工作

流图引擎。 近期关于流图引擎的研究提出了以顶点或边为中心的编程模型，用于迭代式增量图计算。Kineograph [8] 和 GraphInc [7] 是两个支持单调算法（如最短路径）增量计算的系统。Chronos [18] 假设流式图数据以持久化方式存储，并在多个快照上同时计算通用图算法。Auxo [17] 可与 Chronos 互补，并可用作其数据源。GraphBolt [30] 提出了一种通用增量模型，用于处理置信传播等非单调算法；但对于单调算法，其开销高于 KickStarter。GraphS [34] 设计了一种名为 GraphS 的实时流系统，用于环检测。RisGraph [12] 面向逐更新分析，与其他系统相比，可提供更低的延迟和更详细的信息。然而，上述系统都未关注连续子图匹配问题。

子图匹配。 子图匹配是图数据库的一项基本要求，近几十年来一直受到广泛研究。QuickSI [35] 遵循直接枚举框架，直接探索数据图以枚举所有结果。大多数基于状态空间的表示模型（即每个状态表示一个中间结果）都采用该框架，例如 Ri [5], VF2++ [22]、Graphql [20], CFL[4], CECI[3], 和 DP-ISO[16]。得益于有效的过滤和排序方法，这些模型在总体性能方面取得了显著提升。然而，这些



（a）Facebook 的第8季度。 （b）Q10 的 Facebook。

（c）Q8 的 GitHub。 （d）Q10 的 GitHub。

图 13。CaLiG 的不同组件的影响。

算法关注的是静态数据图中的问题。对于流图而言，如果分别在更新前后的图上执行子图匹配，将无法满足实时要求。

连续子图匹配。 IncIsoMat [11] 和 Graphflow [23] 从更新后的边扩展匹配，无需额外的索引结构，因此实时性能有限。SJ-Tree [10] 具体化子图的所有部分匹配，导致巨大的存储开销和索引更新开销。TurboFlux [24] 采用简洁的 DCG（一种完整的多重图）来表示中间结果，从而降低存储开销。类似地，CEPDG [40]采用 TreeMat 存储树的部分匹配。它同时支持模式图和数据图的变化。然而，TurboFlux 和 CEPDG 利用查询图的生成树来筛选候选项，将非树边交由 DCG 维护。SymBi [32] 引入辅助数据结构 DCS，用于存储有向无环图的弱嵌入作为中间结果。与 TurboFlux 使用的 DCG 相比，DCS 同时考虑查询图的有向无环图中的树边和非树边。尽管如此，SymBi 构建的索引仍不够经济高效；也就是说，可以进一步收紧候选项，以降低总体成本。

8 结论

本文提出了一种用于连续子图匹配的高性价比索引 CaLiG。CaLiG 以较低的成本生成比以往方法更精确的候选项，从而显著加快匹配生成。为进一步加快增量生成，我们提出了一种名为 KSS 的新型子图匹配范式。借助核顶点的部分匹配，KSS 只需连接壳顶点的候选项即可生成增量匹配，无需进行任何回溯。实验结果表明，所提方法效率很高。

致谢

本研究得到中国香港研究资助局资助（No.14203618、No. 14202919 和 No. 14205520）。Weiguo Zheng 为通讯作者。

参考文献

[1] [无日期。]。2013年匿名化互联网轨迹。https://catalog.caida.org/details/dataset/passive_2013_pcap。[2] Ehab Abdelhamid、Mustafa Canim、Mohammad Sadoghi、Bishwaranjan Bhattacharjee、Yuan-Chi Chang 和 Panos Kalnis。2018年。大型演化图上的增量频繁子图挖掘。载于2018年IEEE第34届数据工程国际会议（ICDE）。1767–1768。[3] Bibek Bhattarai、Hang Liu 和 H. Howie Huang。2019年。CECI：用于可扩展子图匹配的紧凑嵌入聚类索引。载于2019年数据管理国际会议论文集（荷兰阿姆斯特丹）（SIGMOD '19）。美国计算机协会，纽约州纽约，美国，1447–1462。[4] Fei Bi、Lijun Chang、Xuemin Lin、Lu Qin 和 Wenjie Zhang。2016年。通过延迟笛卡尔积实现高效子图匹配。载于2016年数据管理国际会议论文集（美国加利福尼亚州旧金山）（SIGMOD '16）。美国计算机协会，纽约州纽约，美国，1199–1214。https://doi.org/10.1145/2882903。2915236[5] Vincenzo Bonnici、Rosalba Giugno、Alfredo Pulvirenti、Dennis Shasha 和 Alfredo Ferro。2013年。子图同构算法及其在生化数据中的应用。BMC生物信息学14，增刊7（2013年4月22日）。https://doi.org/10.1186/1471-2105-14-S7-S13 资助信息：本文作为增刊的一部分发表。本文的出版费用由PO拨款——FESR 2007-2013第4.1.1.2号干预线、CUP G23F11000840004资助。。[6] Yazan Boshmaf、Ildar Muslukhov、Konstantin Beznosov 和 Matei Ripeanu。2011年。社交机器人网络：机器人何时为名利而社交。载于第27届年度计算机安全应用会议论文集。93–102。[7] Zhuhua Cai、Dionysios Logothetis 和 Georgos Siganos。2012年。促进实时图挖掘。载于第四届云数据管理国际研讨会论文集。美国计算机协会，1–8。[8] Raymond Cheng、Ji Hong、Aapo Kyrola、Youshan Miao、Xuetian Weng、Ming Wu、Fan Yang、Lidong Zhou、Feng Zhao 和 Enhong Chen。2012年。Kineograph：感知快速变化且相互连接的世界。载于第7届ACM欧洲计算机系统会议论文集。美国计算机协会，85–98。[9] Sutanay Choudhury、Lawrence Holder、George Chin、Abhik Ray、Sherman Beus 和 John Feo。2013年。StreamWorks：动态图搜索系统。载于2013年ACM SIGMOD数据管理国际会议论文集。1101–1104。[10] Sutanay Choudhury、Lawrence B. Holder、George Chin Jr.、Khushbu Agarwal 和 John Feo。2015年。流图中连续模式检测的选择率方法。载于第18届扩展数据库技术国际会议，EDBT 2015，比利时布鲁塞尔，2015年3月23日至27日。157–168。[11] Wenfei Fan、Xin Wang 和 Yinghui Wu。2013年。增量图模式匹配。ACM数据库系统汇刊38，3，文章18（2013年9月），47页。[12] Guanyu Feng、Zixuan Ma、Daixuan Li、Shengqi Chen、Xiaowei Zhu、Wentao Han 和 Wenguang Chen。2021年。RisGraph：用于演化图的实时流系统，支持每秒数百万次操作下的亚毫秒级逐更新分析。513–527。[13] Michael R. Garey 和 David S. Johnson。1990年。计算机与不可解性：NP完全性理论指南。W. H. Freeman；Co.，美国。[14] Pankaj Gupta、Venu Satuluri、Ajeet Grewal、Siva Gurumurthy、Volodymyr Zhabiuk、Quannan Li 和 Jimmy Lin。2014年。实时Twitter推荐：大型动态图中的在线模体检测。VLDB基金会会议录7，13（2014年8月），1379–1380。[15] P. Hall。1935年。关于子集代表元。伦敦数学学会学报s1-10，1（1935），26–30。https://doi.org/10.1112/jlms/s1-10.37.26 arXiv：https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/jlms/s1- 10.37.26[16] Myoungji Han、Hyunjoon Kim、Geonmo Gu、Kunsoo Park 和 Wook Shin Han。2019年。高效子图匹配：统一动态规划、自适应匹配顺序和失败集。载于SIGMOD 2019——2019年数据管理国际会议论文集。1429–1446。[17] Wentao Han、Kaiwei Li、Shimin Chen 和 Wenguang Chen。2019年。Auxo：时态图管理系统。大数据挖掘与分析2，1（2019），58–71。https://doi.org/10.26599/BDMA.2018.9020030[18] Wentao Han、Youshan Miao、Kaiwei Li、Ming Wu、Vijayan Prabhakaran、Wenguang Chen、Enhong Chen、Fan Yang 和 Lidong Zhou。2014年。Chronos：用于时态图分析的图引擎。载于EuroSys。[19] Huahai He 和 A.K. Singh。2006年。Closure-Tree：图查询的索引结构。载于第22届数据工程国际会议（ICDE'06）。38–38。https://doi.org/10.1109/ICDE.2006.37

[20] Huahai He 和 Ambuj K. Singh。2008年。Graphs-at-a-Time：图数据库的查询语言和访问方法。载于2008年ACM SIGMOD数据管理国际会议论文集（加拿大温哥华）（SIGMOD '*08* 405 418 10 1145 1376616 1376660）。美[21] 1971 5 2国计算机协会，纽约州纽约，美国，–。https://doi.org/。/。John E. Hopcroft 和 Richard M. Karp。无日期。二分图中的N / 最大匹配算法。载于第12届年度开关与自动机理论研讨会（SWAT 1971）（SWAT '71）论文集。IEEE计算机学会，美国， 122 125 10 1109 1971 1 –。https://doi.org/。/SWAT。/。Alpár Jüttner 和[22] 2018Péter Madarasi。无日期。VF2++——改进的子图同构算法。离散应用数学242（2018），69–81。https://doi.org/10.1016/j.dam.2018.02.018 组合优化中的计算进展。[23] Chathura Kankanamge、Siddhartha Sahu、Amine Mhedbhi、Jeremy Chen 和 Semih Salihoglu。2017年。Graphflow：主动图数据库。载于2017年ACM数据管理国际会议论文集。1695–1698。[24] Kyoungmin Kim、In Seo、Wook-Shin Han、Jeong-Hoon Lee、Sungpack Hong、Hassan Chafi、Hyungyu Shin 和 Geonhwa Jeong。2018年。TurboFlux：用于流式图数据的快速连续子图匹配系统。载于2018年数据管理国际会议论文集。411–426。[25] Seongyun Ko、Taesung Lee、Kijae Hong、Wonseok Lee、In Seo、Jiwon Seo 和 Wook-Shin Han。2021年。ITurboGraph：扩展并自动化增量图分析。美国计算机协会，977–990。[26] Pradeep Kumar 和 H. Howie Huang。2019年。GraphOne：用于演化图实时分析的数据存储。载于第17届USENIX文件与存储技术会议（FAST 19）。249–263。[27] Danh Le-Phuoc、Minh Dao-Tran、Minh-Duc Pham、Peter Boncz、Thomas Eiter 和 Michael Fink。2012年。关联流数据处理引擎：事实与数据。载于语义网——ISWC 2012，Philippe Cudré-Mauroux、Jeff Heflin、Evren Sirin、Tania Tudorache、Jérôme Euzenat、Manfred Hauswirth、Josiane Xavier Parreira、Jim Hendler、Guus Schreiber、Abraham Bernstein 和 Eva Blomqvist（编）。施普林格柏林海德堡，柏林，海德堡，300–312。[28] Jure Leskovec 和 Andrej Krevl。2014年。SNAP数据集：斯坦福大型网络数据集集合。http://snap.stanford. edu/data。[29] Yuchao Li、Wei Wang 和 Zishen Yang。2019年。K正则图中的连通顶点覆盖问题。组合优化杂志。38, 2 2019 635 645 10 1007 10878 019 00403 3（2019年8月），–。https://doi.org/。/s -[30] 2019- - Mugilan Mariappan 和 Keval Vora。无日期。GraphBolt：流图的依赖驱动同步处理。载于第十四届 EuroSys 会议2019论文集。美国计算机协会，文章25。[31] Andrew McGregor。2014年。图流算法：综述。SIGMOD记录（2014年5月），9–20。[32] Seunghwan Min、Sung Gwan Park、Kunsoo Park、Dora Giammarresi、Giuseppe F. Italiano 和 Wook-Shin Han。2021年。通过双向动态规划实现对称连续子图匹配。VLDB基金会会议录14，8（2021），1298–1310。[33] Mohammad Hossein Namaki、Keyvan Sasani、Yinghui Wu 和 Tingjian Ge。2017年。BEAMS：图流中的有界事件检测。载于2017年IEEE第33届数据工程国际会议（ICDE）。1387–1388。[34] Xiafei Qiu、Wubin Cen、Zhengping Qian、You Peng、Ying Zhang、Xuemin Lin 和 Jingren Zhou。2018年。大型动态图中的实时约束环检测。VLDB基金会会议录（2018年8月），1876–1888。[35] Siddhartha Sahu、Amine Mhedhbi、Semih Salihoglu、Jimmy Lin 和 M. Tamer Özsu。2017年。大型图的普遍性与图处理的意外挑战。VLDB基金会会议录11，4（2017年12月），420–431。[36] Dennis Shasha、Jason T. L. Wang 和 Rosalba Giugno。2002年。树与图搜索的算法及应用。载于第二十一届ACM SIGMOD-SIGACT-SIGART数据库系统原理研讨会论文集。39–52。[37] Chunyao Song、Tingjian Ge、Cindy Chen 和 Jie Wang。2014年。图流上的事件模式匹配。VLDB基金会会议录8，4（2014年12月），413–424。[38] Nan Tang、Qing Chen 和 Prasenjit Mitra。2016年。图流摘要：从大爆炸到大收缩。载于SIGMOD 2016——2016年数据管理国际会议论文集。1481–1496。[39] Alex Hai Wang。2010年。不要跟踪我：Twitter中的垃圾信息检测。载于2010年国际安全与密码学会议（SECRYPT）。1–10。[40] Qianzhen Zhang、Deke Guo、Xiang Zhao 和 Xi Wang。2021年。动态图数据上演化模式的连续匹配。万维网24，3（2021），721–745。

收稿于2022年4月；修订于2022年7月；录用于2022年8月