Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction

摘要

流图在学术界和工业界引起了越来越多的关注，因为许多实际应用中的图随着时间的推移而演变。连续子图匹配（缩写为 CSM）旨在报告查询图在这种流图中的增量匹配。它涉及两个主要步骤，即候选维护和增量匹配生成，以回答 CSM。在连续子图匹配的整个过程中，在搜索空间上回溯的增量匹配生成占总成本的主导地位。然而，尽管增量匹配生成在 CSM 中非常重要，大多数先前的方法侧重于开发用于高效候选维护的技术，而对增量匹配生成的关注较少。为了最小化总体成本，我们在本文中提出了两种减少回溯的技术。我们提出了一种经济高效的索引 CaLiG，它产生更紧凑的候选维护，缩小了回溯的搜索空间。此外，我们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内核顶点和外壳顶点。利用内核顶点的匹配，可以立即通过连接外壳顶点的候选项生成增量匹配，而无需进行任何回溯。受益于减少的回溯，CSM 的经过时间显著减少。对真实图进行的大量实验证明，我们的方法比最先进的算法运行速度快几个数量级。

关键词：子图匹配；流图；回溯缩减

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、道路网络到电力网络等。在大多数实际应用中，图通过添加顶点/边或删除顶点/边的方式随时间演化，被称为流图。例如，在社交网络中，新用户注册和关闭帐户可以被视为图中的顶点添加和删除，创建和取消用户之间的交互（例如，关注）对应于边的添加和删除，这样的社交网络是典型的流图。如何高效地管理和查询流图引起了越来越多的关注 [1]。

CSM（Continuous Subgraph Matching）是指在流图（Streaming Graph）中对查询图（Query Graph）进行连续子图匹配的过程。在实际应用中，图结构通常会随时间演化，通过添加、删除顶点或边来更新。CSM 的目标是持续报告查询图在流图中的增量匹配，即由于图的更新而导致的新匹配或减少的匹配。

CSM 在广泛的应用中很有用，如推荐系统、欺诈检测和网络安全等。例如，在电子商务平台中，一个循环模式可以作为虚假交易的强烈指标，其中用户（买家或卖家）的帐户被表示为顶点，而在线交易（例如，支付活动）被表示为动态边。通过 CSM 可以检测到可疑交易，生成实时警报并触发迅速的操作。

2 相关工作

2.1 直接枚举匹配方法

一种朴素的方法是在图更新之前和之后分别枚举匹配。然而，直接应用子图匹配存在难以处理的成本，因为动态图更新的特性没有得到充分利用。特别是对于大规模图，这种方法可能会浪费大量时间。

2.2 无索引算法

一些无索引算法（如 IncIsoMat 和 Graphflow）通过扩展添加或删除的边来找到已更改的匹配，而不生成不必要的匹配。这避免了在没有匹配的情况下探索整个数据图，但仍可能导致过多基于索引的方法的时间浪费。

2.3 基于索引的方法

最近的工作主要集中在开发索引技术来维护中间结果。SJ-Tree 定义了一个左深度分解树，但由于匹配数量可能呈指数增长，因此受到了存储开销的限制。相反，TurboFlux 和 SymBi 通过利用辅助数据结构和精心设计的过滤规则详细说明了候选生成和维护。

TurboFlux 生成查询图的生成树 TQ，引入了一个称为数据中心图（DCG）的辅助数据结构，用于维护图 G 中的边是否与 TQ 中的边匹配。通过底向上的方法进行过滤，仅考虑 TQ 中的边。对于图更新，TurboFlux 更新 DCG 中边的状态，以确定每两条边是否能够匹配，以及图 G 中的每条边是否能够包含在查询图 Q 的匹配中。

SymBi 采用从查询图 Q 构建的有向无环图（DAG），检查前向或后向邻居，涵盖了 TurboFlux 忽略的非树边。SymBi 在 DAG 上采用自上而下和自下而上的动态规划。只有通过了自上而下筛选的那些顶点需要执行自下而上筛选，只有通过了两个筛选的顶点才是有效的候选。TurboFlux 和 SymBi 都强调索引的简洁性和更新效率，即使维护候选者的时间成本通常只占总成本的很小部分。在搜索阶段，TurboFlux 采用了一种具有固定匹配顺序的回溯算法，该算法取决于查询路径的估计大小。SymBi 根据查询顶点的可扩展候选的估计大小，选择一个动态匹配顺序，该顺序可以在回溯过程中自适应地改变。他们只是关注匹配顺序，而不是改进回溯框架。

3 本文方法

3.1 本文方法概述

大多数先前的方法主要致力于候选维护，而不是直接降低匹配生成的成本。与它们不同的是，我们提出了两项技术，即一种新颖的索引结构 CaLiG（用于获取更紧凑的候选）和一种强大的增量匹配范例（以避免不必要的回溯），旨在最小化 CSM 的总体成本。图 1展示了所提方法的框架，包括离线索引阶段和在线查询阶段。

在离线阶段，我们开发了一种新型索引候选索引照明图，简称 CaLiG。CaLiG 索引结构是根据输入的查询图 Q 和数据图 G 构建的。

CaLiG 以一种顶点对图的形式组织候选匹配对，其中每个节点表示查询顶点 u 和数据顶点 v 的一对。CaLiG 利用灯光状态（“ON” 或 “OFF”）指示 $v \in V_G$ 是否与 $u \in V_Q$ 匹配，即 v 是否是 u 的候选。由于捕捉了查询图和数据图中的所有连接关系，容易维护更紧凑的候选并支持增量匹配生成。此外，我们为查询图中的每条边计算了一个特定的条件核心集和壳集。

在在线阶段，系统根据接收到的更新操作有不同的响应。对于边的添加，首先更新索引 CaLiG 以获取新的候选，然后执行子图匹配（如图 1中的红线所示）；对于边的删除，首先进行子图匹配，然后更新 CaLiG（如图 1中的深蓝线所示）。

CaLiG 更新。为了支持后续更新的增量子图匹配，索引 CaLiG 需要实时维护。添加或删除边均可能导致候选更新以及 CaLiG 结构和灯光状态的变化。此外，一对的改变状态可能传播到其他匹配对，启动对 CaLiG 的递归传播。

增量匹配。为了高效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，称为 kernel-and-shell search（KSS）。KSS 将查询图的顶点分为条件核心顶点和壳顶点。它首先以更新边的候选为初始部分匹配，然后扩展匹配以覆盖所有核心顶点。最后，通过加入壳顶点的候选，可以轻松生成增量匹配，无需任何回溯。

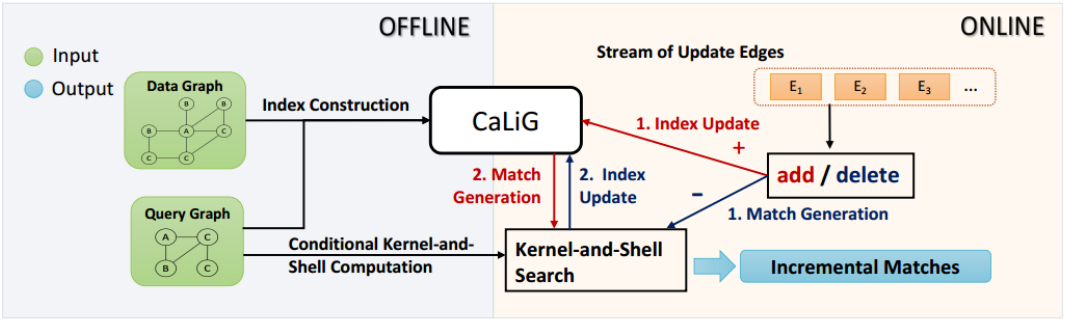

图 1. 方法概述

3.2 定义

Definition 1（Subgraph Isomorphism）. 给定查询图 $Q = \{V_Q, E_Q, L\}$ 和数据图 $G = \{V_G, E_G, L\}$，如果存在一个单射函数 $f : V_Q \to V_G$，使得

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

则 Q 是 G 的子图同构。子图匹配返回与 Q 同构的 G 的所有子图。每个匹配可以表示为一组一对一的匹配对 $\{(u \leftrightarrow f(u))\}$。所有的匹配用 M 表示。

Definition 2（Streaming Graph）. 如果一个图 G 遵循一系列的更新操作 ΔG 动态变化，其中包括四种更新，即顶点添加、顶点删除、边添加和边删除，那么图 G 被称为流图。

Definition 3（Matching Pair）. 如果顶点 u 和 v 构成一个匹配对，简称 (u, v)-MP，当且仅当 $L(u) = L(v)$。每个 (u, v)-MP 具有一个 “ON” 或 “OFF” 的状态，表示 u 是否能与 v 匹配。记 (u, v)-MP.state 表示灯光状态。

Definition 4（CaLiG Index）. 对于查询图 Q 和数据图 G，$CaLiG$ 索引是一个有向图，其中每个节点表示一个 MP。如果满足以下条件，则从 (u_i, v_j)-MP 到 (u_k, v_l)-MP 有一条有向边：

1）$e(u_i, u_k) \in E_Q$ 且 $e(v_j, v_l) \in E_G$；
2）(u_i, v_j)-MP 为 “ON” 或者在 (u_k, v_l)-MP 之后，(u_i, v_j)-MP 为” OFF”。

Definition 5 （Bigraph for (u, v)-MP）. Bigraph for (u, v)-MP，记为 $BI(u, v)$，是一个二分图，具有两个不相交的顶点集合 $N_Q(u)$ 和 $N_G(v)$，其中如果 (u_i, v_j)-MP 是 CaLiG 中 (u, v)-MP 的入邻居，则存在一条边连接 $u_i \in N_Q(u)$ 和 $v_j \in N_G(v)$。

Definition 6（(Injective Matching）.

给定一个包含两个不相交顶点集 X 和 Y 的二部图，如果每个顶点 u 在 X 中通过一条边与 Y 中的一个顶点匹配，并且匹配中的所有边相互独立，即所有边不共享顶点，则存在一个单射匹配。

Definition 7（Kernel Set and Shell Set）.

给定一个查询图 Q ，它的核集，也称为连通顶点覆盖，是一组连接的顶点，使得 Q 中的每条边至少有一个顶点在这个集合中。核集中的每个顶点称为核心顶点。壳集是核集的补集，其中的顶点彼此独立，每个顶点称为外壳顶点。

Definition 8 （Conditional Kernel Set）:

给定一个查询图 Q 和一条边 $e(u_k, u_l) \in E_Q$，条件核集表示为 CKS，是包含顶点 u_k 和 u_l 的核集。

3.3 CALIG：候选照明图

3.3.1 CaLiG 结构

Algorithm 1 说明了构建 CaLiG 和细化匹配对的照明状态（关闭满足匹配要求的节点）的过程。

Algorithm 1: ConstructCaLiG(G, Q)

Input: A data graph G, a query graph Q
Output: A candidate lighting graph *CaLiG*

```
1  CaLiG ← GenerateEmptyCaLiG();
2  for each (u, v) ∈ (V_Q, V_G) do
3      if L(v) = L(u) then
4          (u, v)-MP.state ← ON;
5          add a node (u, v)-MP into CaLiG;
6  for each (u, v)-MP ∈ CaLiG do
7      for each (u', v') ∈ (N_Q(u), N_G(v)) do
8          if (u', v')-MP ∈ CaLiG then
9              add an edge from (u', v')-MP to (u, v)-MP;
10 IndexInitialization(CaLiG);
11 return CaLiG;
```

3.3.2 照明状态计算

在图匹配上下文中，我们首先定义匹配对，表示查询图和数据图中相同标签的节点对。然后，通过构建 CaLiG Index 来建立一个有向图，其中节点表示匹配对，有向边表示匹配对之间的关系。每个匹配对初始状态为”ON”，表示数据图中的节点是查询图中对应节点的候选。

为了确定照明状态，我们构建一个二部图（Bigraph），其中包含查询图节点邻居和数据图节点邻居。我们使用注入匹配算法（例如 Hopcroft–Karp 算法）来计算是否存在满足条件的单射匹配。如果存在单射匹配，表示每个查询图邻居都有唯一的对应候选节点，匹配对的照明状态保持为"ON"；否则，将其设置为"OFF"。

在计算单射匹配的过程中，我们应用 Hall's Marriage Theorem 检查是否存在有效的匹配。该定理确保对于每个查询图的节点集合，都有足够多的数据图节点邻居，从而满足匹配的条件。通过这个过程，我们有效地计算匹配对的照明状态，从而优化了图匹配算法的性能。

3.3.3 CaLiG 初始化

Algorithm 2 概述了初始化过程。

Algorithm 2: IndexInitialization(*CaLiG*)

Input: A candidate lighting graph *CaLiG*

for *each* (u, v)-*MP* $\in$ *CaLiG* **do**
  build a bigraph $BI(u, v)$ for (u, v)-MP;
for *each* (u, v)-*MP* $\in$ *CaLiG* **do**
  if (u, v)-*MP.state* = *ON and* $BI(u, v)$ *has no injective matching* **then**
    (u, v)-MP.state $\leftarrow$ *OFF*;
    OFF-Propagation(*CaLiG*, (u, v)-MP);

Algorithm 3: OFF-Propagation(*CaLiG*, (u, v)-MP)

Input: *CaLiG* and a matching pair (u, v)-MP that was turned off in the previous round

for *each* (u', v')-*MP* $\in$ $Out_{CaLiG}(u, v)$ **do**
  if (u', v')-*MP.state* = *ON* **then**
    delete the edge from (u, v)-MP to (u', v')-MP;
    update $BI(u', v')$;
    if $BI(u', v')$ *has no injective matching* **then**
      (u', v')-MP.state $\leftarrow$ *OFF*;
      OFF-Propagation(*CaLiG*, (u', v')-MP);

3.3.4 CALIG 动态删除边

从数据图 G 中删除边可能会使一些候选者无法匹配查询顶点（关闭 CaLiG 中的一些匹配对），从而减少子图匹配。从 G 中删除一条边时，我们首先从 CaLiG 中删除所有相关的边，并采用 OFF 传播来通过更新照明状态来细化候选。Algorithm 4 说明了删除边的过程。

Algorithm 4: UpdateCaLiGForDel($CaLiG$, $e(v_1, v_2)$)

Input: $CaLiG$ and an updated edge $e(v_1, v_2)$ to delete

```
for each e(u1, u2) ∈ E_Q do
    if L(u1) = L(v1) and L(u2) = L(v2) then
        delete edges between (u1, v1)-MP and (u2, v2)-MP from CaLiG;
        if (u1, v1)-MP.state = ON and BI(u1, v1) has no injective matching then
            (u1, v1)-MP.state ← OFF;
            OFF-Propagation(CaLiG, (u1, v1)-MP);
        if (u2, v2)-MP.state = ON and BI(u2, v2) has no injective matching then
            (u2, v2)-MP.state ← OFF;
            OFF-Propagation(CaLiG, (u2, v2)-MP);
```

3.3.5 CALIG 动态增加边

为了确定是否要打开匹配对 (u, v)-MP 或 (u', v')-MP，只需检查 CaLiG 中的入邻居即可。这避免了考虑所有可能的节点对，而仅需要考虑 CaLiG 的结构。

不考虑出邻居的原因是，(u, v)-MP 的状态如果由 OFF 转为 ON 也不能使得它的出邻居的状态发生改变，因此只需要考虑它的入邻居

当添加一条边时，ON-Propagation 过程用于递归地尝试打开更多的节点。如果一个节点是 ON 状态的节点，并且 ON-Propagation 尝试将这个状态传播到其入邻居。这可能导致一些节点被错误地打开，这些节点被记录在 StopSet 集合中。OFF-Propagation 过程用于确保所有新打开的节点都符合子图同构。对于 StopSet 集合中的每个节点，都会执行 OFF-Propagation 过程。如果某个节点不能通过 OFF-Propagation 将其关闭，那么它被认为是真正被打开的节点，符合子图同构。Algorithm 5 说明了增加边的过程，Algorithm 6 表示 ON-Propagation 的过程。

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

Input: $CaLiG$ and an updated edge $e(v_1, v_2)$ to add

```
for each e(u1, u2) ∈ E_Q do
    if L(v1) = L(u1) and L(v2) = L(u2) then
        add an edge from (u2, v2)-MP to (u1, v1)-MP;
        if (u1, v1)-MP.state = OFF and BI(u1, v1) has an injective matching then
            (u1, v1)-MP.state ← ON;
        if (u1, v1)-MP.state = ON then
            StopSet ← ON-Propagation(CaLiG, (u1, v1)-MP);
            while StopSet is not empty do
                (u, v)-MP ← StopSet.pop();
                OFF-Propagation(CaLiG, (u, v)-MP);
```

Algorithm 6: ON-Propagation($CaLiG$, (u,v)-MP)

Input: $CaLiG$ and a matching pair (u,v)-MP that was turned on in the previous round
Output: A set of stopping nodes S

$S \leftarrow \emptyset$;
for *each* (u',v')-$MP \in In_{CaLiG}(u,v)$ **do**
  add an edge from (u,v)-MP to (u',v')-MP;
  if (u',v')-$MP.state = OFF$ **then**
    if $BI(u',v')$ *has an injective matching* **then**
      (u',v')-MP.state $\leftarrow ON$;
      $S \leftarrow S \cup$ ON-Propagation($CaLiG$, (u',v')-MP);
    else
      $S \leftarrow S \cup (u',v')$-MP;
return S;

3.4 基于 KSS 的子图匹配

利用广泛使用的回溯搜索。这种回溯搜索逐一枚举每个查询顶点的候选者，并在每个步骤检查子图同构的约束。它对候选者进行密集的回溯，通过尝试集成候选者来构建匹配，这会产生昂贵的时间成本。

KSS 将查询顶点分解为内核顶点和外壳顶点，通过连接内核顶点和候选外壳顶点的部分匹配，可以直接提供增量匹配。

3.4.1 内核和外壳顶点

给定一个查询图 Q，它的核心集，也称为连通顶点覆盖，是一组连接的顶点，使得 Q 中的每条边至少有一个顶点属于该集合。核心集中的每个顶点称为核心顶点。外壳集是核心集的补集，其中的顶点相互独立，每个顶点称为外壳顶点。由于在 CSM 中查询图是固定的，相应的核心和外壳顶点可以提前计算并存储。

让 Q' 表示通过添加一个虚拟顶点 u_0 并将 u_0 与 Q 中的所有顶点连接而获得的图。对于 Q' 中的每条边，我们计算它的 MCKS（最小条件核集）。令 S' 表示这 $|E'_Q|$ 个 MCKS 中的最小值。通过反证法，我们可以证明 S' 是 Q' 的一个 MCVC（最小核集）。

3.4.2 内核和外壳搜索

当添加或删除一条边时，方法首先会根据更新的边初始化部分匹配（Algorithm 7），然后调用内核和外壳搜索（Algorithm 8）。对于核心顶点，KSS 通过现有的回溯计算部分匹配。增量匹配可以通过直接连接部分匹配和壳顶点的候选项来报告，无需不必要的回溯。

先找到部分匹配对，接下来，我们通过调用过程 KSS，按照先内核顶点后外壳顶点的顺序匹配剩余的查询顶点。

对于壳顶点，当其所有邻居已匹配时，提前尝试找到候选项。如果找不到候选项，可以安全地排除当前的部分匹配，否则继续找剩余核心顶点的匹配。

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

Input: $CaLiG$, an updated edge $e(v_1, v_2)$, the conditional kernel set K, and shell set S for $e(v_1, v_2)$
Output: Incremental matches due to $e(v_1, v_2)$

```
for each u1 ∈ Q do
    if L(v1) = L(u1) and (u1, v1)-MP.state = ON then
        m[u1] ← v1;
        for each u2 ∈ N_Q(u1) do
            if (u2, v2)-MP ∈ In_CaLiG(u1, v1) then
                m[u2] ← v2;
                return KSS(m, K, S).
```

Algorithm 8: $KSS(m, K, S)$

Input: The partial match m, conditional kernel set K, and shell set S
Output: Incremental matches due to $e(v_1, v_2)$

```
if m.size < |K| then
    th ← m.size;
    u ← K[th];
    Cand(u) ← generate u's candidates;
    for each v ∈ Cand(u) do
        m' ← m;
        m'[u] ← v;
        KSS(m', K, S);
else
    for each u ∈ S do
        Cand(u) ← generate u's candidates;
    return m ⋈_{u∈S} Cand(u).
```

4 复现细节

4.1 与已有开源代码对比

主要参考作者开源的代码完成复现，在此基础上主要有以下工作：

1）参考作者 C++ 源码，采用 Java 语言实现作者的项目，通过实验表明，采用 Java 语言进行流图的查询在查询图较小的时候性能相差不大，但是当查询图较大时，采用 C++ 可以取得更佳的效果。
2）改进 C++ 项目实现对匹配密度的计算。
3）通过脚本实现对多个不同查询子图的批量测试，减少重复工作。

4.2 实验环境搭建

查询图。随机采样 5% 的数据边缘作为流边缘，删除的边与添加的边的比例为 2:1。对于每个数据图，实验对 2 组子图进行采样，表示为 Q_4 ，Q_6，每组查询图 Q_i 包含 50 个查询图。

指标。所用时间以毫秒为单位报告。为了评估算法，我们会报告平均运行时间和匹配密度。

所有算法均在配有 Intel(R) Core(TM) i5-8265U CPU @ 1.80GHz 和 8GB RAM 的 Windows 服务器上实现和评估。

5 实验结果分析

整体表现。表 1分别报告了 5 个数据图上 100 个查询图的平均运行时间。正如所观察到的，它表明，最后一列 $|\Delta m|$ 表示增量匹配的数量。我们可以发现 $|\Delta m|$ 从数百万到超过 10 亿不等。结果的数量越大，所花费的时间就越多。

表 1. 平均消耗时间

Dataset	Elapsed Time (ms)	$\|\Delta m\|$
Lastfm	4.24	7,626,438
Facebook	83.44	443,817,336
Email	218.99	338,293,270
Github	8799.47	1,679,993,941
Deezer	17.85	14,327,860

更改查询图的大小。查询大小（即顶点数量）的影响通过改变 $|V_Q|$ 从 4 到 6。图 2 3 4 5 6显示了详细的结果，表示不同大小的查询子图的平均运行时间。

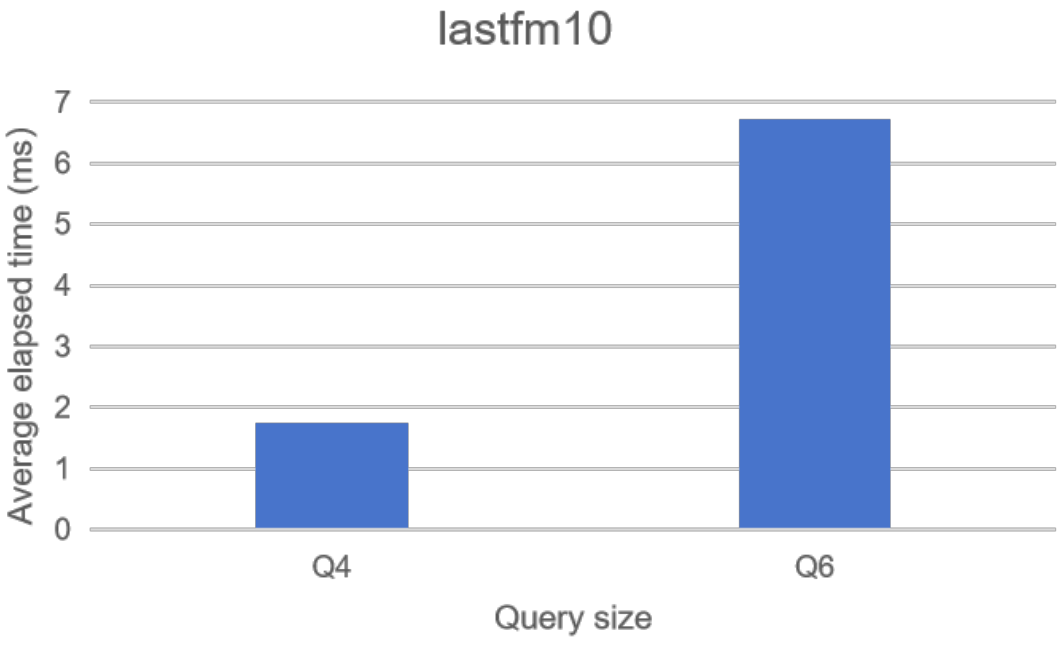

图 2. 不同子图下 lastfm10 的消耗时间

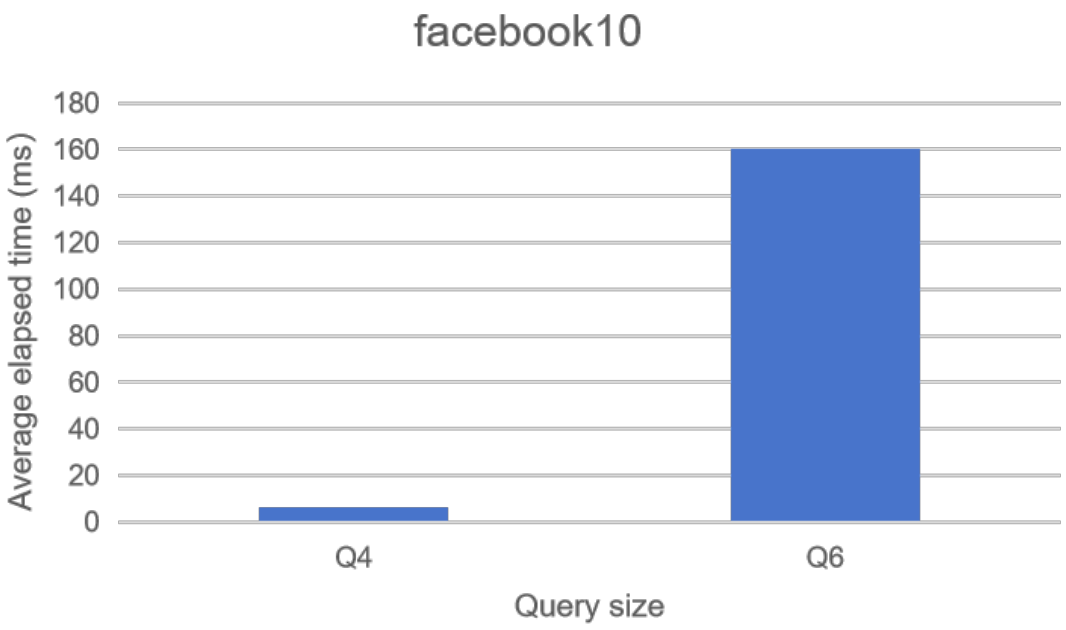

图 3. 不同子图下 facebook10 的消耗时间

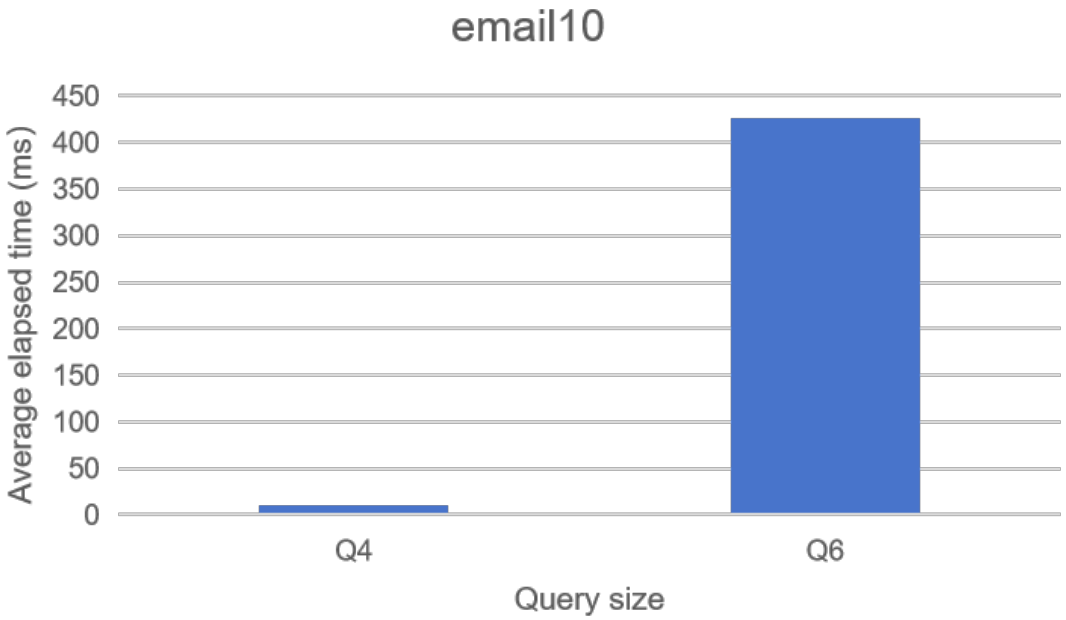

图 4. 不同子图下 email10 的消耗时间

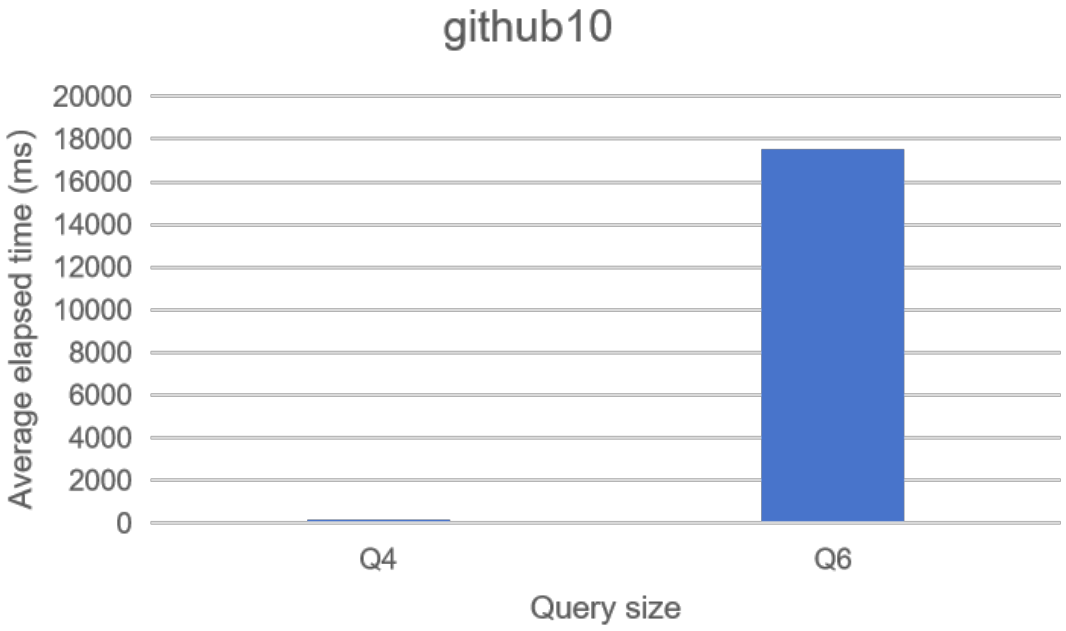

图 5. 不同子图下 github10 的消耗时间

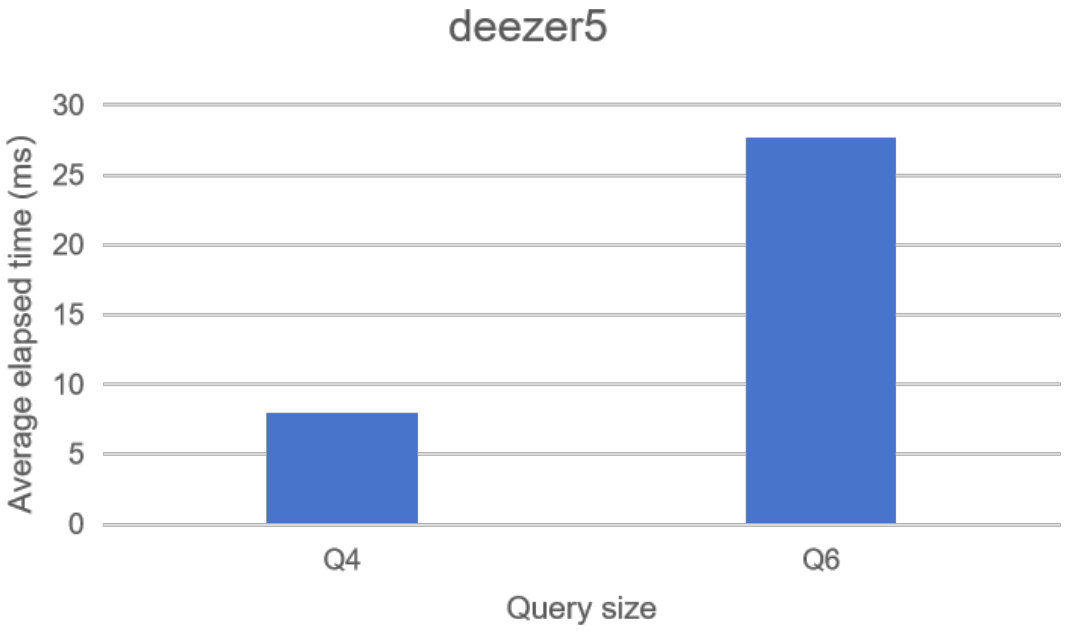

图 6. 不同子图下 deezer5 的消耗时间

匹配密度。为了讨论回溯搜索的有效性，匹配密度（MD）被定义为增量匹配回溯与回溯次数的商。MD 越高，在更少的回溯和更短的时间内可以找到相同数量的匹配。图 7列出了关于 dz、lfm 和 sk 的结果。CaLiG 的匹配密度比 TurboFlux 和 SymBi 的匹配密度高达 7 个数量级：TurboFlush 或 SymBi 中的一次回溯无法为大多数查询生成一个匹配，而 CaLiG 可以在一次回溯中生成数千个匹配。此外，随着查询大小的增长，TurboFlux 和 SymBi 的 MD 降低，但 CaLiG 的 MD 同时增加，这表明 CaLiG 在解决大型查询方面会有更显著的进步。

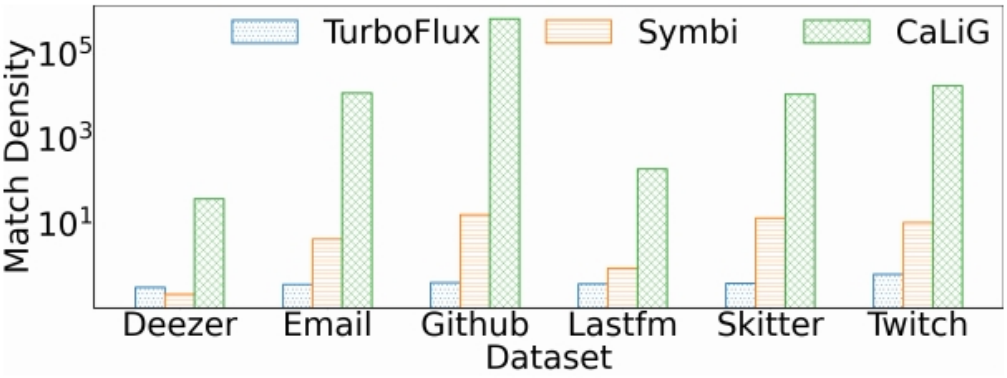

图 7. 对每个图的 50 个随机查询的不同算法的平均匹配密度

通过在两个查询图 Q_4，Q_6 下的匹配密度实验结果也如表 2所示，实验表明，CaLiG 的匹配密度效果非常不错。

表 2. 匹配密度

Dataset	$\|\Delta m\|$	Match Density
Lastfm	7,626,438	53.25242
Facebook	443,817,336	2100.0704
Email	338,293,270	6498.008
Github	1,679,993,941	490299.036
Deezer	14,327,860	93.68212

6 总结与展望

在本文中，我们提出了一个用于连续子图匹配的具有成本效益的指标 CaLiG。与以前的方法相比，CaLiG 以低成本产生了更紧密的候选者，有助于显著加快匹配生成的速度。为了进一步加速增量生成，我们开发了一种新的子图匹配范式，称为 KSS。利用核顶点的部分匹配，KSS 可以通过在没有任何回溯的情况下连接壳顶点的候选者来产生增量匹配。实验证明了我们提出的方法的有效性。

考虑进一步优化算法以提高性能，可能通过进一步改进数据结构、算法实现或并行计算等手段。将算法应用于实际场景，如社交网络分析、推荐系统或欺诈检测等领域，以验证其在实际应用中的有效性。

参考文献

[1] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.