RapidMatch：子图查询处理的一种整体方法

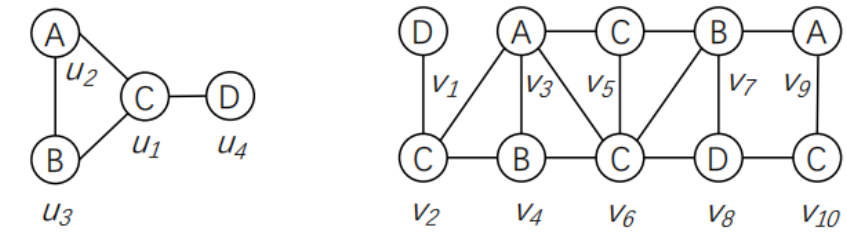
摘要

子图查询在数据图中搜索与查询图相同的所有嵌入。基于图探索和基于连接的两种算法已经被开发出来用于处理子图查询。由于算法和实现的差异，基于连接的系统可以有效地处理几个顶点的查询图，而基于探索的方法通常可以处理查询图中的几十个顶点。在本文中，我们首先比较了这两种方法，并证明了基于最先进的探索方法的结果枚举的复杂性与最坏情况最优连接的复杂性相匹配。此外，我们提出了 RapidMatch，一个集成了这两种方法的整体子图查询处理框架。具体来说，RapidMatch 不仅运行关系操作符 (如选择和连接)，而且还利用图结构信息 (如图探索) 进行过滤和连接计划生成。因此，在广泛的查询工作负载上，它的性能优于这两种方法的现有水平。

关键词：子图查询；关系过滤；子图同态

1 引言

子图查询是对图的一种基本操作，它查找数据图 G 中与查询图 Q 相同的所有嵌入。一种常见的子图查询类型是在标记图 G 和 Q 上，而 G 比 Q 大得多。考虑图 1中的示例查询图和数据图，$\{(u1,v2),(u2,v3),(u3,v4),(u4,v1)\}$ 是查询图在数据图中的嵌入，或者是匹配。子图查询处理已经在各种工作中进行了研究。在本文中，我们对现有的工作基于其建模和实现选择进行了分类，研究了基于探索和基于连接的方法，并提出了 RapidMatch，这是一种解决问题的整体方法。



(a) Query graph Q. **(b) Data graph G.**

图 1. 数据图和查询图示例

基于探索的算法采用 Ullmann [9] 提出的回溯框架，该框架通过将查询顶点映射到数据顶点来迭代扩展中间结果，从而找到所有匹配项。为了减少中间结果，最新的算法如 CFLMatch [3]、CECI [2] 和 DP-iso [4] 使用预处理生成范式执行查询。特别是，它们首先使用指定的修剪

方法为每个查询顶点生成候选顶点集。然后，他们使用贪婪方法根据候选统计数据优化匹配顺序，因为它们通常处理查询图中的数十个顶点。最后，它们沿着候选项的匹配顺序枚举结果。我们称这种包含数十个顶点的查询图为大型查询。这些大型查询在社会网络分析、计算机辅助设计和蛋白质相互作用理解中很常见。

相反，基于连接的算法将子图查询建模为关系查询，并使用关系操作符 (如选择和连接) 对查询进行评估。经典的关系系统，如 MonetDB 和 PostgreSQL，将子图查询作为一对连接操作的序列来实现，这可以看作是在每一步将中间结果扩展一条边。因此，中间结果可能大于查询的最大输出大小 $|OUT|$。最近，最坏情况最优连接 (WCOJ) 的发展改变了现状，因为它的运行时间匹配 $|OUT|$。先前的研究表明，使用 WCOJ 的最新系统显著优于经典关系系统以及原生图数据库 (如 Neo4j)。最近基于连接的方法，如 emptyheading [1] 和 Graphflow [6] 采用直接枚举范式，直接枚举数据图上的结果，易于集成到数据库系统中。它们通过穷尽搜索计划空间来生成连接计划，因为它们的目标是几个顶点的查询图。我们称这样的查询图为小查询。这种小查询通常用于检测交易网络中的周期以警告欺诈活动，以及在社交网络中搜索小集团/小集团结构以进行推荐。

在本文中，我们建议研究这两种算法，并设计一个综合利用这两种算法的整体框架。具体而言，我们比较了它们在时间复杂度方面的差异，并证明了基于探索的方法中结果枚举的最坏情况复杂度与 WCOJ 方法相匹配。此外，我们的详细分析表明，尽管基于连接的方法使用关系运算符实现，而基于探索的方法没有，但是两种方法的结果枚举没有根本区别。因此，一种有希望的方法是采用基于连接的方法，并改进小型和大型查询的连接计划生成和执行。

我们提出了一个基于连接的子图查询处理框架，称为 RapidMatch，它利用图结构信息进行过滤和连接计划生成。首先，给定查询图 Q 和数据图 G，关系过滤器组件基于顶点标签构建关系，然后用半连接过滤这些关系，以删除不会出现在结果中的数据边。其次，连接计划生成器根据前面组件获得的统计数据生成一个连接计划。第三，开发了关系编码器，根据联接计划优化关系数据布局，加快后续枚举的速度。最后，在结果枚举器中进一步扩展了查询计划执行的一些优化，例如高级集合交集方法 [1,5]、交集缓存 [6] 和失败集修剪 [4]。我们的实验表明，在广泛的工作负载下，RapidMatch 在基于探索和连接的方法上都优于目前的技术水平。总之，我们做出了以下贡献。

- 我们研究了基于探索和基于连接的方法，并弥合了它们之间的差距。
- 我们提出了 RapidMatch，一个基于连接的子图查询处理引擎，可以有效地评估小查询和大查询。
- 我们设计了一个基于半连接的关系过滤器来减少关系的基数。该过滤器将最先进的修剪技术从基于探索的方法映射到基于连接的方法。
- 为了减少结果枚举的搜索空间，提出了一种基于查询图核分解的连接计划生成器。
- 我们对各种工作负载进行了广泛的实验，并证明 RapidMatch 优于最先进的基于探索和基于连接的方法。

2 相关工作

子图查询可以基于以下两种方式来定义: 用 ISO 表示的子图同构或用 HOM 表示的子图同态。子图同态是一个函数，它保留图中的边关系，而子图同构是一个双射函数，它同时保留图中的边关系和顶点之间的一一对应关系。两者之间的唯一区别是 ISO 使用内射函数，而 HOM 使用映射。

基于探索的算法查找从 Q 到 G 的所有子图同构，而基于连接的方法枚举所有子图同态。HOM 允许结果包含重复的数据顶点，而 ISO 不允许。因此，ISO 必须是 HOM，反之亦然。一种特殊情况是，当每个查询顶点都有一个不同的标签时，由于定义中对标签的约束，HOM 是一个 ISO。然而，判断是否存在 ISO 或 HOM 是 NPC 问题，两种算法经过简单的修改可以得到同样的结果。

2.1 基于探索的算法

基于探索的算法是一种常见的子图查询处理算法，其核心思想是通过遍历数据图中的顶点和边，逐步扩展匹配的子图，最终找到所有与查询图匹配的子图。该算法具有简单、易于实现等优点。但是，由于其时间复杂度较高，对于大规模和复杂的查询图处理效率较低。根据执行范式，基于探索的方法可分为三类。

第一类算法，如 QuickSI [7]，遵循直接枚举范式，其核心思想是从查询图的一个顶点开始，逐步遍历数据图中的顶点和边，直到找到所有与查询图匹配的子图。

第二类算法，如 GADDI [10]，利用索引枚举框架，在 G 上构建索引结构，并在索引的帮助下评估所有查询。常见的索引结构包括邻接矩阵、邻接表、前缀树等。索引枚举算法的核心思想是利用索引结构来快速定位匹配的顶点和边，从而减少遍历数据图的时间。

第三类算法，如 CFLMatch [3]、CECI [2]、DP-iso [4] 采用预处理生成框架。通过预处理数据图和查询图的特征信息来加速子图查询处理。常见的预处理方法包括候选集生成、匹配顺序优化、剪枝等。预处理枚举算法的核心思想是通过预处理过程来减少遍历数据图的时间和中间结果的数量，从而提高查询处理的效率。

以往的性能研究表明：索引枚举方法由于索引的构建存在严重的可扩展性问题；预处理生成方法一般是其中性能最好的 [8]。因此，本文采用了预处理枚举方法。

2.2 基于连接的算法

基于连接的算法是另一种常见的子图查询处理算法，其核心思想是将子图查询问题转化为关系查询问题，利用关系数据库系统中的连接操作来处理子图查询。基于连接的算法通常包括两个主要步骤：关系构建和连接操作。

关系构建：在关系构建阶段，将查询图和数据图转化为关系结构，其中查询图的顶点和边分别对应关系数据库中的表和关系。通过构建关系结构，可以将子图查询问题转化为关系查询问题，从而利用关系数据库系统的优化技术和索引结构来加速查询处理。

连接操作：在连接操作阶段，利用关系数据库系统中的连接操作（如内连接、外连接等）来执行子图查询。通过连接操作，可以有效地在数据图中查找与查询图匹配的子图，从而实现高效的子图查询处理。

基于连接的算法的优点在于可以利用关系数据库系统的优化技术和索引结构来加速查询处理，尤其适用于处理大规模和复杂的子图查询。此外，基于连接的算法还可以与现有的关系数据库系统集成，方便实际应用和部署。基于最坏情况下的最优连接（WCOJ）的算法是一种用于处理关系查询的算法，特点是在最坏情况下具有与查询的最大输出大小相匹配的运行时间。WCOJ 算法的提出是为了解决传统关系数据库系统中连接操作可能产生的指数级增长的中间结果的问题。最近的研究表明，利用 WCOJ 的系统在性能上明显优于传统的关系数据库系统和原生图数据库系统，因此基于连接的算法在处理子图查询方面具有很大的潜力 [1]。

3 本文方法

3.1 本文方法概述

本文主要研究无向顶点标记图 $g = (V, E)$，其中 V 是一组顶点，E 是一组边集。$N(u)$ 表示顶点 u 的邻居顶点。$d(u)$ 表示 u 的度。L 是一个标签函数，它将一个顶点与标签集 Σ 中的一个标签相关联。Q 和 G 分别表示查询图和数据图。Q_C 表示 Q 的 2-core 结构子图（每个顶点的度大于等于 2），Q_F 等于 $Q - Q_C$，即 $E(Q_F) = E(Q) - E(Q_C)$，$V(Q_F)$ 是 $E(Q_F)$ 中边的顶点集，注意 Q_C 是连接的，而 Q_F 是一组树。我们将 Q_C 和 Q_C 分别命名为 Q 的核心结构和森林。

在介绍本文方法之前，需要明确 2 个基本概念：

核分解是一种用于识别图中密集子图的方法。在图数据库和图算法中，核分解被用于寻找图中的“核心”结构，即密集连接的子图。这种方法可以帮助识别图中的重要模式和结构，对于图数据的分析和查询处理非常有用。核分解推广了 k-core 和 k-truss 分解，并在不同层次上发现密集的子图。直观地说，$k-(r,s)$ 核是 G 的一个连通子图，它满足密度和连通性约束，而核森林 $T_{r,s}$ 描述了基于 (r,s) 核包含关系的层次结构。$k-(1,2)$ 核恰好是 k 核，$k-(2,3)$ 核是 k-truss 聚集，除了 k-truss 的定义外，还需要三角形连通性。

LFTJ 算法是一种高效的子图查询处理算法，主要用于图数据库和图算法中的子图查询处理。LFTJ 算法的核心思想是使用跳跃指针技术来加速连接操作，从而提高查询处理的效率。具体来说，LFTJ 算法使用一个基于 trie 树的数据结构来存储查询图和数据图中的顶点和边，然后使用跳跃指针技术来快速定位匹配的顶点和边。在连接操作中，LFTJ 算法使用一种基于 set intersection 的方法来计算匹配的顶点和边，从而避免了传统的基于排序的方法中的昂贵的排序操作。此外，LFTJ 算法还使用了一种基于哈希的方法来加速连接操作，从而进一步提高了查询处理的效率。

RapidMatch 的算法流程之前需要进行一定的数据分析处理，首先进行核分解，得到查询图 Q 的核心结构 Q_C、森林 Q_F、核森林 $T_{1,2}, T_{2,3}, T_{3,4}$ 以识别查询图 Q 的密集子图及其层次关系。我们采用核森林 $T_{1,2}, T_{2,3}, T_{3,4}$ 是因为：(1) $T_{1,2}$ 包含所有查询顶点；(2) 前人对 $T_{r,s}$ 的研究表明，设置 $r = 3$ 和 $s = 4$ 是一个最佳点，可以得到密度相当的密集子图，并且可以高效地计算。完成核分解后，RapidMatch 进行子图查询处理。图 2展示了 RapidMatch 的四个组件的处理框架。

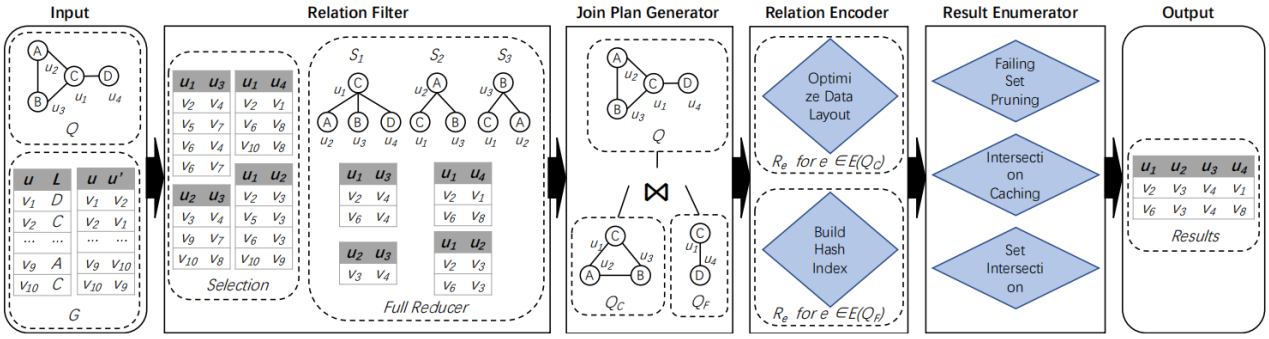


图 2. RapidMatch 框架图

关系过滤器 (RelationFilter) 接收给定输入，基于顶点标签为每个查询边构建一个关系。然后，它将 Q 分解为一组树结构的子查询（因为后续的 full-reducer 过程不能处理循环查询），并将 full-reducer（full-reducer 是一个有限的半连接操作序列，它可以删除关系中所有悬挂的元组）应用于每个子查询，以消除不会出现在任何连接结果中的数据边。

连接计划生成器 (JoinPlanGenerator) 基于 RelationFilter 中得到的关系统计值和 Q 的核森林生成一个连接计划变量 φ。连接计划是首先用 LFTJ 算法对 Q_C 执行计算，然后在随后的查询上执行一系列哈希连接以找到最终结果。因为在 G 中密集子图 (如 4-clique) 通常比稀疏子图 (如 4-cycle) 出现的频率要低，变量 φ 定义优先考虑密集区域的查询顶点。此外，所有连接都将在管道中执行，以避免实现中间结果。

关系编码器 (RelationEncoder) 在运行时基于连接计划优化关系的数据布局以加速随后的枚举。在 Q_C 中对关系中的数据顶点的 id 进行编码，并将它们存储在一个 trie 结构中，以协助 LFTJ 的计算。它还为 Q_F 中的关系构建哈希索引，以服务于哈希连接。

最后，结果枚举器 (ResultEnumerator) 对编码的关系执行连接计划。此外，RapidMatch 还采用了高级集合交集方法 [1,5]、交集缓存 [6] 和失败集修剪 [4] 等多种优化方法。下面，我们将详细介绍这四个组件。

3.2 关系过滤器

传统生成关系的方法忽略了底层的图结构，这可能导致关系中出现大量悬空元组，即不会出现在任何最终结果中的数据边。这个问题直接降低了连接操作的性能。此外，悬空元组会对连接计划生成的有效性产生负面影响，因为关系的统计信息是计划优化的重要因素 [1,3,4,6]。然而，由于子图同态 (或同构) 问题的难度，删除 Q 的所有悬空元组是一个 NP-hard 问题。

由于难度大，我们开发了一种利用图结构特征来去除悬空元组的启发式方法。具体来说，将 Q 分解为一组树状结构的子查询 Q'，并对每个 Q' 应用 full-reducer 程序来删除悬空元组。在执行 full-reducer 程序后，关系中每个数据边最终会出现在从 Q' 到 G 的子图同态中。

算法 1 演示了关系过滤器的流程，它为每个查询边构建关系并修剪悬空元组。第 2-3 行基于标签为每个查询边生成一个关系。然后，第 4-5 行使用 full-reducer 删除每个 $Q_T \in Q_F$ 的悬空元组。接下来，将 Q_C 分解为一组树 S_u，每棵树以 $u \in V(Q_C)$ 为根，而 u 的邻居顶点点为叶，并对每个树应用 full-reducer 程序。具体来说，第 8 行生成查询顶点的顺序，以确定树的处理顺序。我们把这个顺序称为过滤顺序 (δ)。在 S_u 上应用 full-reducer 过程中，希望涉及尽可能多的已被处理的关系，以利用来自前面步骤的修剪结果。因此，过滤顺序的生成按

照以下理论：首先将只属于 $V(Q_F)$ 的顶点添加到过滤顺序中，然后选择与过滤顺序中结点拥有最多相邻的顶点作为下一个顶点。第 9-11 行按照过滤顺序对树应用 full-reducer 修剪悬空元组。此过程利用 S_u 上的修剪结果来过滤 $S_{u'}$ 中的关系，其中顶点 u 在过滤顺序中位于顶点 u' 之前，并且 u' 是 u 的邻居。为了用 $S_{u'}$ 上的修剪结果过滤 S_u 中的关系，第 13-15 行按照相反顺序执行相同的操作。

Algorithm 1 RelationFilter

1: **procedure** RELATIONFILTER(Q,G,Q_C,Q_F)
2: **for** $e(u,u') \in E(Q)$ **do**
3: $R(u,u') \leftarrow \{(v,v')|e(v,v') \in E(G) \wedge L(v) = L(u) \wedge L(v') = L(u')\}$;
4: **end for**
5: **for** $Q_T \in Q_F$ **do**
6: fullreducer(Q_T)
7: **end for**
8: $\delta \leftarrow generateFilteringOrder(Q, Q_C, Q_F)$;
9: **for** $u \in V(Q_C)$ 根据 δ 顺序选择 **do**
10: 计算 S_u 为以 u 为根节点，$N(u)$ 为叶子节点的树
11: fullreducer(S_u)
12: **end for**
13: **for** $u \in V(Q_C)$ 根据 δ 反顺序选择 **do**
14: 计算 S_u 为以 u 为根节点，$N(u)$ 为叶子节点的树
15: fullreducer(S_u)
16: **end for**
17: **end procedure**
18: **function** GENERATEFILTERINGORDER(Q,Q_C,Q_F)
19: $\delta \leftarrow ()$;
20: 把 $u \in V(Q_F) - V(Q_C)$ 顶点加入到 δ 中
21: **while** $|\delta| \neq |V(Q)|$ **do**
22: $\delta \leftarrow argmax_{u \in V(Q_C)-\delta} |N(u) \cap \delta|$
23: **end while**
24: **return** δ
25: **end function**

3.3 连接计划生成器

连接计划是指在执行子图查询时，按照一定的顺序连接不同的关系，并选择合适的连接算法和优化连接操作的过程。连接计划的目的是最大程度地减少计算成本并提高查询性能。RapidMatch 中的连接计划 φ 主要特点是将核心顶点 $V(Q_C)$ 放在非核心顶点 $(V(Q) - V(Q_C))$ 前面。根据 k-core 定义，一个非核心顶点在 φ 中恰好有一个后向邻居顶点。在生成连接计划后，RapidMatch 首先以 $\varphi[1 : |V(Q_C)|]$ 为顺序，用 LFTJ 算法执行计算 Q_C 中边关系的连接，

然后用哈希连接计算 Q_C 和 Q_F 连接。在实践中，RapidMatch 以管道方式执行所有连接，即一个连接生成的结果立即发送到后续连接，以避免实现中间结果。

现有方法是基于成本模型 $\cos t(\varphi) = \sum_{i=1}^{|V(Q)|} |R(Q[\varphi[1:i]])|$，该成本模型等于在枚举期间生成的中间结果的总数。然而对查询图的子查询输出大小的基数估计是非常具有挑战性的，特别是当 Q 很大的时候，而估计的偏差可能导致非常无效的连接计划，所以我们从图结构角度来优化。在 LFTJ 算法中，当 $X_M(\{u\})$（在 M 查询顶点匹配下 u 目前的候选匹配点）为空时，中间结果 M 不能被进一步扩展，即不能出现在任何最终结果中。可以在早期终止这些无效的搜索路径来优化变量 φ 的定义，当 φ 将具有更多向后邻居的查询顶点放在一开始时，算法往往表现得更好，因此定义了效用函数 $utility(\varphi) = \sum_{i=1}^{|V(Q)|} \sum_{j=1}^{i} |N_+^{\varphi}(\varphi[j])|$。但是当 Q 很大时，通过列出 $V(Q)$ 的所有排列来最大化效用函数的代价是非常昂贵的。因为 $\sum_{j=1}^{i} |N_+^{\varphi}(\varphi[j])|$ 等于 $|E(Q[\varphi[1:i]])|$，即 Q 的顶点诱导子图中的边数，所以可以通过优先考虑图中密集区域的顶点来优化效用函数，在论文中主要使用核心分解方法生成连接计划顺序，因为该方法可以高效地找到具有详细层次结构的高质量密集区域。此外，在执行一系列哈希连接操作时倾向于连接基数较小的关系。

Algorithm 2 JoinPlanGenerator

```
1: function JoinPlanGenerator(Q, T_{1,2}, T_{2,3}, T_{3,4})
2:     T ← ConstructDensityTree(T_{1,2}, T_{2,3}, T_{3,4}); Ω ← {};
3:     for χ ∈ T.leaves do
4:         e*(u, u') ← arg min_{e(u,u')∈E(χ)}(d(u) + d(u'));
5:         φ ← (u, u')whend(u) ≥ d(u');
6:         Traverse(Q, T, φ, χ); 将 φ 加入 Ω;
7:     end for
8:     return arg max_{φ∈Ω} utility(φ)
9: end function
```

算法 2 给出了基于核分解的连接计划生成器。第 2 行通过 ConstructDensityTree 函数首先构建了 Q 的密度树 T，树的节点是分解的核。第 3-6 行从叶节点遍历 T，通过遍历函数将 Q 的密集区域中的顶点放在 φ 开始。我们添加在 $e(u,u') \in E(\chi)$ 中具有最大度和的顶点，并在第 4-6 行开始遍历。接下来将介绍关于这两个函数的更多细节：ConstructDensitTree 函数和 Traverse 函数。

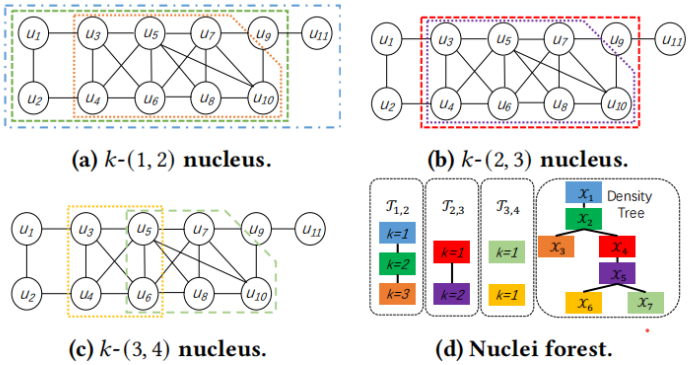


图 3. 密度树示例图

ConstructDensityTree: 密度树的构建示例可以观察图 3，对于每个核树 $T \in T_{3,4}$ 我们将 T 的根节点的父节点设置为 $T_{2,3}$ 中的最小的包含该根节点的核 χ（第 2-3 行）。第 5-6 行对每个 $T \in T_{2,3}$ 进行类似的操作。因此，构造密度树函数生成一个树 T，因为 (1) 1-（1,2）核包含所有查询顶点；(2) 给定一个 $k-(r,s)$ 核，必须有一个 $k'-(r-1,s-1)$ 核在 $k' \geq k$。

Algorithm 3 JoinPlanGenerator

1: **function** ConstructDensityTree($T_{1,2}, T_{2,3}, T_{3,4}$)
2: **for** $T \in T_{3,4}$ **do**
3: 设置 T 的根的父结点为 $T_{2,3}$ 中包含该根节点的最小节点
4: **end for**
5: **for** $T \in T_{2,3}$ **do**
6: 设置 T 的根的父结点为 $T_{1,2}$ 中包含该根节点的最小节点
7: **end for**
8: **return** $T_{1,2}$ 的树
9: **end function**

Traverse: 如果 χ 已经被访问过，则返回（第 2 行）。否则，请将其标记为已访问过。如果 χ 中的所有顶点都存在于 φ 中，那么就跳转到它的父节点（第 3 行）。由于 χ 的密度通常低于它的子节点，5-9 行首先考虑它的子节点。我们优先考虑与 φ 有更强联系的子节点 χ'，这在式 1 中定义。$|SP(\varphi, \chi')|$ 是 χ 中从 $\chi[\varphi]$ 到 χ' 通过顶点的最短路径的长度。与 φ 有更多共同顶点或者与 φ 接近的子节点得分较高。

Algorithm 4 JoinPlanGenerator

1: **function** Traverse(Q, T, φ, χ)
2: 如果 χ 被访问过了，则返回
3: 如果 $V(\chi) - \varphi = \emptyset$, 转移到 15 行
4: **while** $\chi.children \neq \emptyset$ **do**
5: $\chi^* \leftarrow \arg\max_{\chi' \in \chi.children} connection(\varphi, \chi')$;
6: **if** $V(\chi*) \cap \varphi = \emptyset$ **then**
7: 把 SP 中的 u 点加入到 Ω，如果 $u \notin \varphi$
8: **end if**
9: Traverse($Q, T, \varphi, \chi*$)
10: 把 $\chi.children$ 从 $\chi*$ 移出去
11: **end while**
12: **while** $V(\chi*) \cap \varphi = \emptyset$ **do**
13: $u* \leftarrow \arg\max_{u \in V(X)-\varphi} |N(u) \cap \varphi|$, 把 $u*$ 加入到 φ 中
14: **end while**
15: **if** $\chi \neq T.root$ **then**
16: $Traverse(Q, T, \varphi, \chi.parent)$;
17: **end if**
18: **end function**

$$connection(\varphi, \chi') = |\varphi \cap V(\chi')| + \frac{1}{1 + |SP(\varphi, \chi')|} \tag{1}$$

在访问了 χ 的所有子节点之后，第 34-35 行将 χ 中的剩余顶点添加到 φ 中。在每一步中，我们选择具有最大 $|N(u) \cap \varphi|$ 的顶点 u 来优化效用函数。我们打破了与 (1) 高顶点度的联系；以及 (2) 小关系基数。第 37 和 38 行遍历 χ 的父节点。最后，第 8 行返回具有最大效用值的 φ 作为连接计划。

3.4 关系编码器

它在运行时基于连接计划优化关系的数据布局，以加速后续的枚举过程。以下是关系编码器的详细实现细节：

1. 数据布局优化：根据连接计划在运行时优化关系的数据布局，以加速后续的枚举过程。它对 Q_C 中的数据顶点的 ID 进行编码，并将其存储在一种 trie 结构中，以协助计算 LFTJ。它还为 Q_F 中的关系构建哈希索引，以用于哈希连接。

2. Trie 结构编码：将关系存储为两级 trie，其中第一级存储 u 的值，第二级记录已排序的邻居。这种编码的 trie 结构通过对关系中的顶点进行编码来优化关系的数据布局。

3. 加速枚举: 通过优化关系的数据布局并使用编码的 trie 结构，关系编码器旨在加速后续的枚举过程.

3.5 结果枚举器

结果枚举器是 RapidMatch 算法的一个组件，用于执行连接计划并枚举查询结果。以下是结果枚举器的详细实现步骤：

1. 执行连接计划：基于连接计划生成器的操作，按照计划顺序执行查询的连接操作。

2. 加速枚举：利用了几种优化方法，包括高级集合交集方法、交集缓存和失败集合修剪等。它还采用了编码的 trie 结构和哈希索引，以加速执行连接计划。

3. 优化数据布局：根据连接计划在运行时优化关系的数据布局，以加速枚举过程。

4 复现细节

4.1 与已有开源代码对比

该论文代码已经开源，四个组件的代码实现基本参照开源代码。但是在复现该论文实验的过程中发现，代码仅限于标签为数字的图查询。基于此，我设想扩展一下代码的适用性，修改了关系过滤器的代码，使该算法适应标签为字符串的图匹配查询，当然也兼容标签为数字的图匹配查询，算法的速度和准确度保持不变。

4.2 实验环境搭建

数据集：该论文复现工作主要用了 LiveJournal 数据集和 DBLP 数据集。LiveJournal 数据集是一个公开可用的社交网络数据集，包含了 LiveJournal 社交网络平台的用户关系数据。该数据集记录了用户之间的好友关系，以图的形式表示。DBLP 图数据集是基于 DBLP 数据库构建的一个图数据集，用于表示计算机科学领域的学术合作网络和引用关系网络。该数据集将 DBLP 中的作者和论文作为节点，以及作者之间的合作关系和论文之间的引用关系作为边，形成一个图结构。

实验设置：在子图查询处理中，查询图的大小和复杂度对算法的性能和效率有很大的影响。一般来说，可以将查询分为大查询和小查询两类，复现实验工作主要也分为大查询和小查询。小查询主要以 LiveJournal 数据集来进行实验。由于该图数据集最初没有标签，我们从一组不同标签的中均匀随机选择一个标签，并将其分配给顶点；我们将不同标签的数量设置为 4 个，因为如果有大量不同的标签，工作负载将难以评估。查询图的生成依据一组固定的结构，并随机分配标签，结构如图 4所示。大查询主要以 DBLP 数据集来进行实验。该数据集没有标签，我从 Σ 中均匀随机地选择一个标签，并将其分配给该顶点。将 |Σ| 的大小从 10 改变到 30，选择这个大小这样我们就可以在不破坏大多数算法的情况下演示竞争算法的能力。然后我们根据该数据集使用随机游走算法，生成密集查询图集合和稀疏查询图集合，每一个集合包括了 200 个有相同数量顶点的连通图。复现实验主要基于小查询。

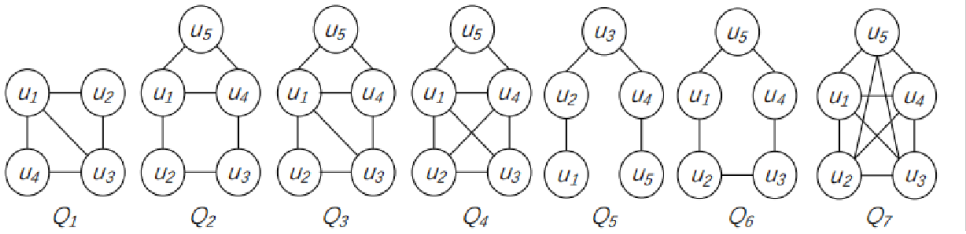


图 4. 查询图结构图

评估指标：查询处理时间：评估算法执行子图查询的时间开销，包括查询图的匹配和结果枚举等阶段的时间消耗。最终结果数量：评估算法生成的最终查询结果的数量，这可以反映算法的准确性和完整性。

5 实验结果分析

在复现论文的实验过程中，按照上述步骤对基础图查询和拓展字符串图查询都进行了小查询实验，设置标签数量为 4，随机分配给数据图和查询图。大查询实验按照随机游走算法生成查询图之后，还未来得及进行大查询实验。然后比较查询时间，查询结果准确性也可以得到验证，通过在其他算法中计算可知。结果如表 1 和表 2，时间单位为 ms，DB 为 DBLP 数据集，LJ 为 LiveJournal 数据集。可以看出来，RapidMatch 查询速度较快，在经过小修改之后，字符串图查询速度也依然很快。

表 1. 数字标签查询时间表

	查询时间(DB)	查询时间(LJ)
Q1	0.716189	11.667483
Q2	1.193314	62.636602
Q3	1.793742	58.234913
Q4	2.046732	84.128394
Q5	6.374813	302.37489
Q6	4.128374	134.17391
Q7	1.468292	76.247823

表 2. 字符标签查询时间表

	查询时间(DB)	查询时间(LJ)
Q1	0.346189	22.933579
Q2	1.038473	73.348626
Q3	2.084759	83.137485
Q4	1.836459	79.173812
Q5	7.374613	365.11382
Q6	3.136791	127.37492
Q7	1.132494	80.128392

6 总结与展望

根据这篇论文，我复现 RapidMatch 算法的图查询并对代码进行扩展实现字符串图查询。通过该实现，我对 RapidMatch 算法的关系编码器和结果枚举器有了更深入的了解。关系编码器通过优化数据布局和使用编码的 trie 结构加速了查询过程，而结果枚举器则执行了连接计划并利用了多种优化方法以提高查询结果的枚举效率。将来我们将研究基于密度的连接计划生成方法，结合先进的基数估计方法，考虑其他指标（例如集交集的成本和 ISO 冲突的影响），并扩展规划空间。

参考文献

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4), oct 2017.

[2] Bibek Bhattarai, Hang Liu, and H. Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1447–1462, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1429–1446, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1587–1602, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *CoRR*, abs/1903.02076, 2019.

[7] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. Efficient parallel subgraph enumeration on a single machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 232–243, 2019.

[8] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1083–1098, New York, NY, USA, 2020. Association for Computing Machinery.

[9] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, jan 1976.

[10] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, page 192–203, New York, NY, USA, 2009. Association for Computing Machinery.