硕士学位论文

ShareFlow：一种高效的多查询动态子图匹配算法

SHAREFLOW: AN EFFICIENT MULTI-QUERY DYNAMIC SUBGRAPH MATCHING ALGORITHM

研　　究　　生：李可明

指　导　教　师：唐博助理教授

南方科技大学

二〇二三年六月

国内图书分类号：TP311.13　　学校代码：14325
国际图书分类号：004-63　　密级：公开

工学硕士学位论文

ShareFlow：一种高效的多查询动态子图匹配算法

学 位 申 请 人：李可明
指 导 教 师：唐博助理教授
学 科 名 称：电子科学与技术
答 辩 日 期：2023 年 5 月
培 养 单 位：计算机科学与工程系
学位授予单位：南方科技大学

Classified Index: TP311.13
U.D.C: 004-63

Thesis for the degree of Master of Engineering

SHAREFLOW: AN EFFICIENT MULTI-QUERY DYNAMIC SUBGRAPH MATCHING ALGORITHM

Candidate:	Li Keming
Supervisor:	Assistant Prof. Tang Bo
Discipline:	Electronic Science and Technology
Date of Defence:	May, 2023
Affiliation:	Department of Computer Science and Engineering
Degree-Conferring-Institution:	Southern University of Science and Technology

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

无

答辩委员会名单

主席	Jiang Jianmin	教授	深圳大学
委员	杨双华	讲席教授	南方科技大学
	唐博	助理教授	南方科技大学
	丁宇龙	研究副教授	南方科技大学
	姜洁	副教授	中国石油大学（北京）
秘书	申乔木	研究助理教授	南方科技大学

南方科技大学学位论文原创性声明和使用授权说明

南方科技大学学位论文原创性声明

本人郑重声明：所提交的学位论文是本人在导师指导下独立进行研究工作所取得的成果。除了特别加以标注和致谢的内容外，论文中不包含他人已发表或撰写过的研究成果。对本人的研究做出重要贡献的个人和集体，均已在文中作了明确的说明。本声明的法律结果由本人承担。

作者签名：李可明　　日期：2023.05.13

南方科技大学学位论文使用授权书

本人完全了解南方科技大学有关收集、保留、使用学位论文的规定，即：

1. 按学校规定提交学位论文的电子版本。

2. 学校有权保留并向国家有关部门或机构送交学位论文的电子版，允许论文被查阅。

3. 在以教学与科研服务为目的前提下，学校可以将学位论文的全部或部分内容存储在有关数据库提供检索，并可采用数字化、云存储或其他存储手段保存本学位论文。

（1）在本论文提交当年，同意在校园网内提供查询及前十六页浏览服务。

（2）在本论文提交 ☐ 当年/ ☑ 一年以后，同意向全社会公开论文全文的在线浏览和下载。

4. 保密的学位论文在解密后适用本授权书。

作者签名：李可明　　日期：2023.05.13

指导教师签名：[illegible]　　日期：2023.05.13

摘 要

本文提出了一种在数据图流中增量子图匹配多个查询图的解决方案。强调了图匹配在计算机网络、社交媒体网络和化学物质的分子结构等多个领域的重要性。本文将现有的子图匹配相关工作分为三类：子图匹配、动态子图匹配和多查询动态子图匹配。其中，动态子图匹配包括 RapidFlow、SymBi 和 TurboFlux 等方法，逐个处理查询图，缺乏多个查询图的共享执行策略。多查询动态子图匹配，包括 TRIC 和 $\text{IncCMatch}_{\text{ann}}$，与本文的问题定义相同，但 TRIC 将查询图分为路径并合并路径，增加了大量增量计算和存储开销，以及后续匹配联接开销；而 $\text{IncCMatch}_{\text{ann}}$ 则专注于批量更新，并未提出合并查询图的方法。但是，本文的实验结果显示，TRIC 和 $\text{IncCMatch}_{\text{ann}}$ 都是低效的。本文的主要贡献是提出了一种方法，将多个查询图合并成一个标记图，以在多查询子图匹配中共享执行，并定义一个 rDAG 森林来将标记图转换为该结构，使得任何查询图都可以找到其子图同构，而不必将查询图划分为路径。基于 rDAG 森林，本文设计了一个共享增量状态图，用于存储部分匹配结果，不同查询图的增量更新和搜索共享执行，以节省计算时间。本文提出了一种匹配顺序树，以共享不同查询图的匹配执行，并提出了一种加锁-解锁方法，以剪枝搜索空间中的不必要分支。总之，本文提供了一种新的方法，用于在数据图流中高效有效地进行多查询动态子图匹配。

关键词：多查询; 共享执行; 动态子图匹配; 匹配顺序树; 加锁-解锁搜索

Abstract

This paper presents a solution to the problem of incremental subgraph matching of multiple query graphs in a data graph stream. The importance of graph matching in various fields such as computer networks, social media networks, and molecular structures of chemical substances is highlighted. The paper categorizes existing subgraph matching related work into three classes: subgraph matching, dynamic subgraph matching, and multi-query dynamic subgraph matching. Among these, dynamic subgraph matching, which includes RapidFlow, SymBi, and TurboFlux, handles query graphs one by one and lacks a shared execution strategy for multiple query graphs. Multi-query dynamic subgraph matching, including TRIC and IncCMatch$_{ann}$, has the same problem definition as our work, but TRIC divides query graphs into paths and merge paths, which brings a large amount of incremental computing and storage overhead, as well as subsequent matching join overhead, while IncCMatch$_{ann}$ focuses on batch updates and does not propose methods to merge query graphs. However, both TRIC and IncCMatch$_{ann}$ are inefficient and will be shown in our experiments. The paper's major contribution is the proposal of a method to merge multiple query graphs into an annotation graph to share execution in multi-query subgraph matching and defines a rDAG forest to turn the annotation graph into a structure where any query graph can find its subgraph isomorphism without dividing the query graph. Based on rDAG forest, the paper designs a shared incremental state graph to store partial matching results, where incremental update and searching of different query graphs share execution to save computation time. The paper proposes a matching order tree to share matching execution of different query graphs and a lock-unlock method to prune unnecessary searching branches in the search space. Overall, this paper provides a new approach for efficient and effective multi-query dynamic subgraph matching in data graph streams.

Keywords: multi-query; share execution; dynamic subgraph matching; matching order tree; lock-unlock searching

目 录

符号和缩略语说明

A_v	标记图点标记的查询图点集合
A_{vq}	标记图点标记的查询图集合
E	边集合
G_0	初始数据图
G_i	更新了 $\{\Delta op_j \mid j \in [1,i]\}$ 的 G_0
L	点集合到点标签集合的映射函数，即 $L: V \to \Sigma_V$
M	匹配集合
$N(v)$	节点 v 的所有邻点
Q	问题输入的查询图集合
V	点集合
f	一个匹配
g_a	标记图
g_r	rDAG 森林
g_s	共享增量状态图
q	一个查询图
u	查询图、标记图或 rDAG 森林的节点
v	一般指数据图 G 的节点
w	共享增量状态图节点
ΔG	数据图更新集合，是 Δop 的集合
Δop	一个数据图更新，本文一般指一条边的插入或删除
Σ_V	点标签集合

第 1 章　绪　论

1.1　背景介绍

图是由顶点或节点以及连接这些顶点的边组成的数据结构。近些年来，图已成为多个领域中用于建模信息网络的流行数据结构。而子图匹配是图中一个非常经典的算法问题，在实际中包含非常广泛的应用。比如分析蛋白质相互作用的网络[1]，社交网络分析[2-3]，化学分子结构搜索[4]，计算机通信网络[5]等。这些图非常庞大，并且由于频繁更新而在不断扩容，子图匹配逐渐成为图数据库中越来越重要的问题。

子图匹配已经被证明为是一个 NP 难问题[6]。尽管如此，由于子图匹配问题的重要性，现如今子图匹配仍然是多个不同学科领域的热门研究课题，如数据库领域使用回溯搜索的子图匹配算法 QuickSI[7]、GADDI[8]、SPath[5]、GraphQL[9]、TurboIso[10]、CFL-Match[11]和 DAF[12]等；数据库领域使用成对 join 的数据库 PostgreSQL[13]、MonetDB[14]和 Neo4j[15]等，其中 Neo4j 是专门的图数据库管理系统；AI 领域的 Ullmann[16]、VF2[17]、VF2++[18]、VF3[19]和 Glasgow[20]等；生物领域的 RI[21]和 VF2+[22]等。

此外，某些研究工作也致力于用并行办法加速查询，如 PSM[23]、Glasgow、VF3P[24]、pGlassgow[25]、PDSim[26] 、pRI[27]和 Grapes[28]等。或者是利用 GPU 做子图匹配的工作如 RPS[29]。或者是分布式工作 G-thinker[30], 具体详细关于分布式子图匹配的相关调研可以参考文章[31]。或通用多核 CPU 子图匹配并行技术文章 ISBP[32]。子图匹配问题比较热门的原因除了它日渐重要外，还因为在针对实际数据解决子图匹配时，TurboIso 指出如果能设计出高效的剪枝手段或高效的查询图的节点匹配顺序，那么算法便能够在很短的反应时间返回子图匹配的查询结果。

1.2　问题定义：多查询动态子图匹配

想要了解子图匹配，这里首先先对图论中的一些基本概念进行阐述，对应图解如图1-1：

定义 1.1 (单射, Injective)：对于两个集合 A 和 B，假设从 A 到 B 的映射是 f，$\forall v \in A$，$\forall v' \in A$，如果 $v \neq v'$，都有 $f(v) \neq f(v')$，那么我们称 f 是单射的。

定义 1.2 (满射，Surjective)：对于两个集合 A 和 B，假设从 A 到 B 的映射是 f，$\forall v' \in B$，都 $\exists v \in A$ 使得 $f(v) = v'$，那么我们称 f 是满射的。

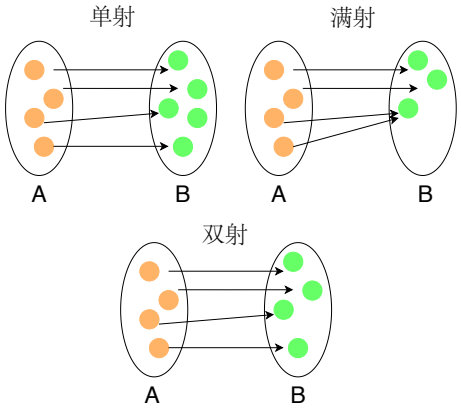


图 1-1　单射、满射、双射的例子

定义 1.3 (双射，Bijective)：如果一个映射既是单射又是满射，那么我们称这个映射是双射的。

定义 1.4 (（标签）图 G)：标签图 $G=(V,E,L)$，其中 V 是图的点集合，$E\subseteq V\times V$ 是图的边集合，$L:V\to\Sigma_V$ 是一个从点集合到点标签集合 Σ_V 的映射。**本文指的“图”一般都是指“标签图”。**

在图论中，子图匹配是指在一个图中寻找一个子图（查询图），使得这个子图中的每个节点都能在另一个图中找到一个对应节点它们的点标签一一对应相同，且这些对应节点之间的边也在另一个图中存在。子图匹配分为静态子图匹配以及动态子图匹配。

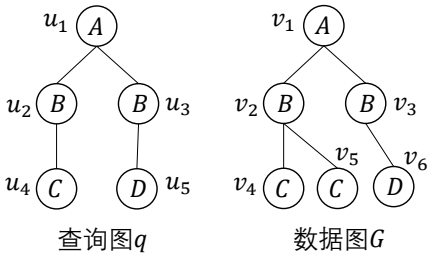


图 1-2　子图同构与子图同态的例子

一般的静态子图匹配问题中包含两个图，一个称为数据图，另一个称为查询图（也称为子图），如图1-2包含了数据图 G 与查询图 q。其中查询图 q 包含五个节点 v_1，v_2，v_3，v_4，和 v_5，其中 v_1 的点标签是 A，记为 $L(v_1)=A$，对 q 中其他点同理。对于 G 也同理。静态子图匹配问题的任务就是在一个数据图找所有匹配的查询图，匹配的方式可能为子图同构或子图同态。

同态和同构则是描述两个图之间的相似程度的概念。同态和同构都可以看作是一种图之间的“对应关系”，它们都通过一个映射把一个图中的节点对应到另一个图中的节点，从而描述两个图之间的关系。具体来说：如果两个图之间存在一

个同态，则这两个图具有一些相似的结构，它们可以被看作是同一种类型的图；如果两个图之间存在一个同构，则这两个图形状是完全相同的，它们具有相同的节点集合和边集合，只是节点之间的标号不同。

有了以上概念，接下来我们对子图同构或子图同态进行定义。

定义 1.5 (子图同构)：给定数据图 $G=(V,E,L)$，查询图 $q=(V',E',L')$，子图同构就是满足以下条件的从 V' 到 V 的**单射**函数 f:（1）$\forall u \in V'$，都有 $L(f(u))=L(u)$;（2）$\forall e'=(u,u') \in E'$，都有 $e=(f(u),f(u')) \in E$。

定义 1.6 (子图同态)：给定数据图 $G=(V,E,L)$，查询图 $q=(V',E',L')$，子图同态就是满足以下条件的从 V' 到 V 的函数 f:（1）$\forall u \in V'$，都有 $L(f(u))=L(u)$;（2）$\forall e'=(u,u') \in E'$，都有 $e=(f(u),f(u')) \in E$。

子图同构与子图同态的不同在于单射条件。子图同态允许多个点标签一样的查询图节点都映射到同一个数据图节点，如图1-2中，$f_1=\{u_1 \to v_1, u_2 \to v_2, u_3 \to v_2, u_4 \to v_4, u_5 \to v_5\}$。因为 u_2 与 u_3 都映射到了 v_2，q 中存在的一些边比如 (u_1,u_2) 对应映射点之间形成的边 (v_1,v_2) 也存在于 G，故 f_1 是一个子图同态但它并不是一个子图同构。一个子图同构的例子则是 $f_2=\{u_1 \to v_1, u_2 \to v_2, u_3 \to v_3, u_4 \to v_4, u_5 \to v_5\}$。

具体是子图同构还是子图同态依据业务场景和需求等定义。Streamworks[33]提出有一些应用场景比如化合物结构搜索、蛋白质结构搜索等需要子图同构的匹配，但 Fraud detection 白皮书[34]提出另外一些场景比如用户互动模式、诈骗模式等并不要求单射条件只需满足子图同态的匹配即可。因为子图同构与子图同态它们之间差了一个简单的单射条件的检查，本质上没有太大区别，**所以在本文中我们讨论的子图匹配只指子图同构。**

结合上述概念，接下来我们对静态子图匹配概念进行定义。

定义 1.7 (静态子图匹配)：给定一个查询图 q 与一个数据图 G，静态子图匹配就是返回所有的从 q 到 G 的子图同构或子图同态。

在实际应用中，比如社交网络并不是一直不变的，社交网络会不断地加入新的用户数据，一个用户与其他用户之间也会产生的新的互动或者关联，而用户之间的关联也可能会消失比如取消关注、删除好友等。所以对应于数据图，数据图是会一直不断变化的，如增加新节点、增加节点之间的连边、删除节点等。当数据图发生变化时，应当返回数据图变化时带来的匹配变化，于是诞生了动态子图匹配问题。

常见的动态子图匹配应用有社区流媒体（如短视频，用户之间的关注点赞会为用户数据图形成连边，当用户群体的连边符合某种模式，也就是匹配了某个子

图时，可以认为该用户群体的用户符合某种特征），计算机网络（如将 IP 地址作为节点，网络协议如 TCP 等作为边），金融交易网络（如识别金融犯罪、量化交易、国家反诈骗中心识别诈骗等），社交网络（如识别恐怖分子，研究新冠感染模式以及预防等）。

在文章 TurboFlux[35] 中提到，插入点或删除点对于子图匹配问题来说，可以看作是插入边或删除边。简单来说，插入点可以等效成同时插入与该点有关的所有边，删除点可以等效成同时删除与该点有关的所有边，故在后文的讨论中，我们只关注插入边以及删除边这两种情况，下面我们定义动态子图匹配问题中的动态数据图的定义和动态子图匹配问题的定义。

定义 1.8 (正匹配)：当数据图插入点、插入边时，会带来新的查询图的匹配，我们称此类匹配为正匹配。

定义 1.9 (负匹配)：当数据图删除点、删除边时，会使之前的部分匹配失效，这些失效的匹配实例我们称之为负匹配。

定义 1.10 (增量匹配)：当数据图时变化时，产生的正匹配或负匹配称为增量匹配。

定义 1.11 (动态数据图 G_i)：给定一个初始的数据图 G_0，一个有序的数据图更新流 $\Delta G = \{\Delta op_1, \Delta op_2, \cdots, \Delta op_n\}$，$G_i \oplus \Delta op_{i+1}$ 表示在数据图上 G_i 上施加操作 Δop_{i+1}，其中 $\Delta op_i = (+/-, e)$，$+$ 表示插入边，$-$ 表示删除边，那么 $G_i = G_{i-1} \oplus \Delta op_i$ 表示对数据图 G_0 施加 $\{\Delta op_i | i \in [1, i]\}$ 后的数据图。

动态子图匹配实际上就是返回正匹配以及负匹配，动态图由初始图和由边插入和删除组成的图更新流定义。动态子图匹配是数据图变化时更新子图匹配结果的问题，而查询图是不变的。

定义 1.12 (动态子图匹配)：给定一个查询图 q，一个初始数据图 G_0，一个有序的数据图更新流 $\Delta G = \{\Delta op_1, \Delta op_2, \cdots, \Delta op_n\}$，假设 G_i 的静态子图匹配结果是 M_i，那么动态子图匹配就是 $\forall \Delta op_i$，当 Δop_i 是插入边时返回正匹配 M_i / M_{i-1}，当 Δop_i 是删除边时，返回负匹配 M_{i-1} / M_i。因为现行的主流算法都主要在解决数据更新流的动态子图匹配问题，所以经常动态子图匹配也被称为连续子图匹配问题。

一个比较容易想到的解决动态子图匹配的方法是对更新后的数据图重新做子图匹配，并根据更新前后的数据图子图匹配结果进行差异化比较即可得到正匹配或负匹配，但显然来说这种方法是相对低效的。于是有了一些动态子图匹配的研究工作，主要研究尽可能只搜索前后的差异匹配。

现行的动态子图匹配方法包括但不限于基于索引的方法和基于过程的方法。基于索引的方法是将动态的图转换成一系列静态图，静态图记录每个数据点的部分匹配状态，并将这些静态图构建成一个索引结构，然后在这个索引结构上进

行子图匹配。目前的主流动态子图匹配方法基本都是基于索引的比如 TurboFlux、SymBi[36]和 RapidFlow[37]。这种方法的优点是能够在短时间内快速处理大规模的图，缺点是索引结构需要占用大量的内存资源，并且不能及时反映出图的动态变化。基于过程的方法是在动态的图上直接进行子图匹配。这种方法的优点是能够及时反映出图的动态变化，内存资源消耗非常少，缺点是计算量较大，难以处理大规模的图。总的来说，动态子图匹配是一个复杂的问题，目前没有一种通用的方法适用于所有场景，需要根据具体的应用场景选择适合的方法。

关于动态子图匹配研究工作有很多。在银行申请中，Fraud detection 白皮书提出诈骗组织通常形成环状的模式（即环状的查询图），在实际应用中是很有必要设计出算法能够即时地识别犯罪网络相关的模式的。Streamworks 提出网络安全应用程序也要即时识别与监控出现在计算机网络流量中符合某些模式的疑似入侵和攻击。动态子图匹配解决了子图匹配中数据图会改变的场景，但未解决实际查询过程中是多个查询图对应一个数据图的场景。在网络诈骗的应用场景中，诈骗模式是多种多样的，比如淘宝刷单诈骗、网络贷款诈骗、冒充领导诈骗等，也就意味着我们需要多个查询图来表示这些不同的诈骗模式，对于符合任意一种诈骗模式的正匹配都应该发出告警并及时提醒当事人处于诈骗过程中。

在实际的图数据库中，往往图数据库要处理大量的子图匹配查询问题，这些问题针对的是同一个数据图，但不同的用户做的子图匹配查询的查询图可能是不同的。ChemSpider 是一个拥有超过 7700 万分子数据的分子结构子图匹配查询系统[38]。面对上述如此巨大的数据图，如果不采取对多个用户的查询采取某些共享执行的策略，所有用户查询的平均反应时间随着用户数量的提升快速上涨。

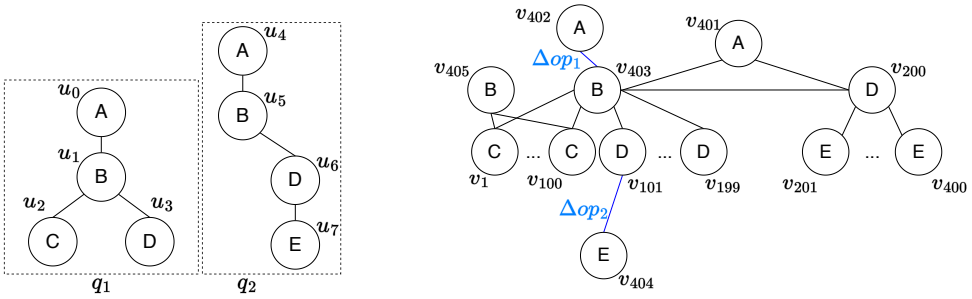
某些数据库模型采取缓存的策略，缓存部分之前用户的查询结果。但对于子图匹配问题，有研究工作[39]通过实验表明现存的大部分技术只能在指数级时间才能对子图匹配的结果进行高效的缓存以及快速重用结果。这个结论促使我们必须研究出一种能够高效同时处理大量子图匹配的算法与解决方案[40]。

基于这些实际的问题需求，提出了本课题要解决的问题——多查询动态子图匹配问题。

定义 1.13 (多查询动态子图匹配问题)：给定一个查询图数据集 $Q = \{q_1, q_2, \ldots, q_n\}$，一个数据图 G 和一个有序的数据图更新流 ΔG，多查询动态子图匹配就是在每次数据图的更新时（插入或删除边），对于查询图数据集中的每一个查询图 $q \in Q$，返回所有对应的从 q 到 G 的正匹配或负匹配。

图1-3是一个多查询动态子图匹配的例子。图1-3(a)是查询图数据集 $Q = \{q_1, q_2\}$，由 q_1 和 q_2 组成。图1-3(b)是有 402 条边的初始数据图 G_0 以及它的两

个新的插入边 $\Delta op_1 = (+, (v_{402}, v_{403}))$ 和 $\Delta op_2 = (+, (v_{101}, v_{404}))$（共 404 条边）。我们为每个操作找到所有正匹配。Δop_1 发生时，因为插入边 (v_{402}, v_{403}) 与 q_1 中的查询边 (u_0, u_1) 匹配，所以它会在数据图中产生 10,000 个正匹配，匹配结果为 $F_1 = \{\{u_0 \to v_{402}, u_1 \to v_{403}, u_2 \to v_i, u_3 \to v_j\} | i \in [1, 100], j \in [101, 200]\}$。当 Δop_2 发生时，因为它与 q_2 中的查询边 (u_6, u_7) 匹配，所以它会在数据图中产生 200 个正匹配，匹配结果为 $F_2 = \{\{u_4 \to v_i, u_5 \to v_{403}, u_2 \to v_j, u_6 \to v_{101}, u_7 \to v_{404}\} | i \in [401, 402], j \in [1, 100]\}$。



(a) 查询图集合 Q　　(b) 数据图 G

图 1-3　多查询动态子图匹配例子

1.3　贡献点

现有的单查询动态子图匹配算法，比如 TurboFlux、SymBi、RapidFlow，因为它们都是一个一个查询图返回增量匹配的，所以它们在针对多查询动态子图匹配问题时如果遇到如图1-3(a)中两个查询图非常相似的情况，并不会共享两个查询图增量匹配共同的部分，而是会重复做两次，会带来不必要的计算开销。

多查询动态匹配算法如 TRIC[41] 则会将 q_1 拆分成两个 path。即使 TRIC 在随机分 path 的情况下运气比较好，分成 $p_1 = (u_0 \to u_1 \to u_3)$ 和 $p_2 = (u_1 \to u_2)$，TRIC 可以共享 q_1 与 q_2 中公共的部分 p_1。但是因为 path 中因为每个节点少了多条边的约束部分匹配，部分匹配结果数量会显著上升；如对于 p_2，TRIC 会存储多余的 100 条边 $\{(v_{405}, v_i) | i \in [1, 100]\}$；随着 path 的长度增加，这种多余的部分匹配会指数级上涨；在 p_1 或 p_2 的部分匹配数量增加的情况下，p_1 与 p_2 的部分匹配结果的 join 操作也会难度加大。

我们这个工作的主要的贡献是：

（1）提出了一种方法，将多个查询图合并成查询图同构标记图，从而支持多查询静态子图匹配。同时定义了查询图同构的 rDAG 森林，从而支持多查询动态子图匹配，并且这个合并 rDAG 森林的方法还可以用于多个一般的数据库执行引擎 DAG 执行计划的合并，来达到共享执行的目的。

（2）新提出的标记图或者带标记的 rDAG 森林不需要拆分任何查询图，充分利用原先每个查询图的连接性来进行部分匹配筛选以及剪枝。同时，推出一个在回溯匹配方法基础上的树状多查询图匹配顺序——匹配顺序树，最大化共享匹配执行，一次性匹配所有查询图的结果。

（3）在单个查询图上的回溯方法匹配时推出一个快速回退的剪枝方法，并推广到多查询图。

总的来说，这个工作的主要贡献是提出了一个有效的多查询动态子图匹配算法，该算法能够充分利用查询图之间的连接性和公共部分，从而大大减少计算开销和重复计算，提高匹配效率。同时，该算法具有广泛的应用前景，可以用于多个一般的 DAG 执行计划的合并，从而达到共享执行的目的。

第 2 章　相关工作

2.1　通用子图匹配算法框架

理论上，子图匹配搜索属于 NP 难问题，但是如果算法利用了良好的匹配顺序和强大的剪枝技术，在实际的图数据集中可以高效地解决该问题。

有一种相对简单但是低效的方法，即将图的节点看作数据库的数据列，将寻找查询图对应的匹配实例的过程转化成传统数据库中不同表的列之间的一个 multi-way join 操作[42]。

数据库领域当前最常用来解决子图匹配问题的方法是回溯法，即用深度优先等方法对于每一个查询图节点，找到查询图节点对应的待匹配数据图节点，然后对于查询图的邻点，继续在待匹配的数据图节点的邻点做邻点之间的匹配，如果数据图不存在符合匹配的邻点则匹配失败并返回到上一个匹配的查询图节点，如果数据存在符合匹配的邻点则继续匹配查询图节点与数据图节点。

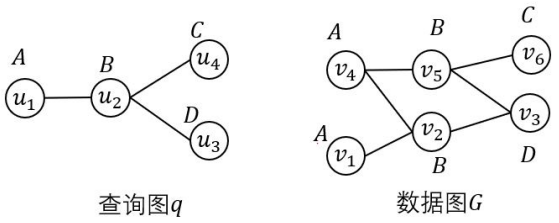


图 2-1　回溯法例子

如图2-1，u_1 匹配 v_1，u_2 匹配 v_2，u_3 匹配 v_3，对于 u_3 它还有下一个待匹配的邻点 u_4，但在 v_2 的邻点中并不存在一个点的标签与 u_4 相同，即在 v_2 的邻点中找不到 u_4 的候选点。此时进行回溯，u_3 不再匹配 v_3，且 u_2 不再匹配 v_2。第一次回溯之后，u_1 的匹配点 v_1 没有除 v_2 以外的 u_2 匹配点，此时继续回溯，u_1 不再匹配 v_1。第二次回溯之后，u_1 匹配 v_4，匹配成功；匹配 u_1 的邻点 u_2，在 v_4 的邻点中，v_2 符合条件，则将 u_2 匹配至 v_2；但同样的，因为 v_2 没有符合条件的 u_4 候选邻点，故要对 u_2 进行回溯（尽管 v_2 在之前 u_1 匹配 v_1 时已经知道 v_2 没有符合条件的 u_4 候选邻点，但这里还是做了重复的搜索并导致了比如深度搜索时更多的重复无效搜索，文章 CFL-Match 指出这种现象叫做冗余笛卡尔积）；在 v_4 的邻点中，除了 v_2，v_5 也与 u_4 的点标签一致，故可以将 u_2 匹配至 v_5；同样地，u_3 匹配至 v_3，u_4 匹配至 v_6；此时因为查询图地所有点都有对应的数据点匹配了，故我们找到了一个从查询图到数据图的匹配。

回溯法这一方向主要有三种工作：

（1）直接枚举型，直接遍历数据图 G 进行匹配，如 QuickSI 。但在文章[43]中已经提及此类工作在大的图中会面临十分严重的时间以及内存问题。

（2）索引枚举型，利用已经建立好的针对数据图 G 的索引辅助结构来帮助枚举匹配实例，主要的工作有 GADDI、SPath[5]和 SGMatch[44]。

（3）预处理枚举型，针对查询这些算法先进行预处理，用辅助的数据结构存储与查询图点一一对应的在数据图的节点集合，然后他们基于辅助数据结构的节点与边生成自己特定的匹配顺序来进行匹配，这个类型的算法是最近这几年数据库领域做子图匹配问题最为流行的方式，也是本课题重点研究的对象，典型的算法有 GraphQL、TurboISO、CFL-Match、CECI[45]、DAF 和 VEQ[46]。

当前数据库领域流行的预处理枚举型算法一般用一个自己设计的辅助结构将查询图的每一个节点对应数据图 G 上的候选节点与边都预存储起来，然后对于每一个候选节点生成一个匹配顺序然后遍历执行顺序匹配找出所有结果。这样做的好处是在执行回溯算法时，不需要遍历数据图中与查询图无关的节点、边、邻点与邻边。如图2-1所示，u_1 匹配至 v_1，当不使用辅助结构将相关的点边存储起来时，v_1 的所有邻点 v_2 到 v_7 都需要一个个确认是否点标签为 B；当使用辅助数据结构时，辅助数据结构中不存在 v_1 到 v_3，v_1 到 v_4，v_1 到 v_6 的边，这使得访问以及验证大大加速。当然，在建立辅助数据结构时我们也要考虑建立辅助数据结构所需要的时间以及辅助数据结构在对回溯过程的加速时间之间的平衡。细节上，当前辅助数据结构的通用做法是对于每个查询图节点，建立一个数据图候选点集合；当且仅当边的两个点同时存在于辅助数据结构时，数据图的边才会出现在辅助数据结构中；故辅助数据结构的构建时间空间复杂度仅仅为 $O(|V(q)|\times|V(G)|+|E(G)|\log|V(q)|)$，其中 $|V(q)|$ 代表查询图 q 的节点数目，$V(G)$ 代表数据图 G 的节点数目，$E(G)$ 代表数据图 G 的边数目，总体为多项式时间复杂度远比不上子图匹配这类 NP-hard 问题的时间复杂度，所以用辅助数据结构来加速子图匹配的查询是可行的。

回溯法除了最基本的三种工作之外，一些工作还在枚举匹配时设计了一些优化的剪枝策略来进一步地减小搜索空间。另外还有不少的工作尝试使用并行或分布式的方式（如使用多核 CPU、GPU 等）来加速查询过程[47]。

2.2　子图匹配及其变种相关工作

子图匹配的相关工作可以分为两大类，静态子图匹配（全量查询）和动态子图匹配（增量查询）。对于动态子图匹配，现有的主要工作中又按照动图的类型主要分成以下几类：

（1）静态查询图与动态数据图，如 TurboFlux、SymBi 和 RapidFlow。数据图存在边的插入与删除；正匹配是数据图的边的插入时可能带来的新的匹配实例；负匹配是数据图的边的删除时某些不再匹配的旧匹配实例。

（2）动态查询图与动态数据图，如 CEPDG[48]。查询图插入节点或边时可能带来负匹配；查询图删除节点或边时可能带来正匹配。

（3）多个静态查询图与单个动态数据图，如 TRIC 和 $\text{IncCMatch}_{\text{ann}}$[49]。数据图一条边的插入或删除可能会出发大量查询图的增量查询。

在接下来的章节中我们将动态子图匹配中的“静态查询图与动态数据图”和“动态查询图与动态数据图”归为“单查询动态子图匹配”，“多个静态查询图与单个动态数据图”归为“多查询动态子图匹配”。

2.2.1　静态子图匹配

静态子图匹配研究历史比较久远。1976 年，Ullmann 最早提出用递归算法进行匹配（算法、作者名字都为 Ullmann），随机匹配顺序对子图进行匹配，它的缺点是没有使用广度搜索或者深度搜索等图搜索方法利用图的节点之间的连接性来进行剪枝。同时在做子图匹配时，没有设计专门的机制或剪枝来尽可能避免冗余笛卡尔积。

2013 年，TurboISO，一，提出深度优先遍历的查询图匹配顺序利用查询图的连接性进行匹配，比如当前只匹配了查询图的节点 u_1，那么下一个匹配点只会选取与 u_1 相邻的子节点进行匹配；二，提出邻居等价类的概念，将查询图相同的子树合并减少冗余搜索；如图2-2的查询图 q 的 u_1 与 u_2 的子树相同，所以 q 的 u_1 与 u_2 为邻居等价类；三，在某个子节点分支都不匹配时，当前节点也可以尽早剪枝减少冗余笛卡尔带来的计算浪费；四，根据具体匹配时按节点的数目来决定下一个匹配的点标签，具体做法是按查询树叶子节点的候选点个数来决定匹配顺序，候选点个数越少匹配顺序越优先。TurboISO 也存在着一些问题，邻居等价类在大部分查询图都无效，虽然采用辅助数据结构来加速查询，但对于每个查询图节点的候选节点是使用树形从上到下的组合进行存储的，上下层查询图候选点之间的关系是一对多，如图2-3，比如 u_1 的映射候选节点有 v_1 与 v_2，u_2 的映射候选节点有 v_3 与 v_4，u_3 的映射候选节点有 v_5 与 v_6，TurboISO 会构建如 CR 那样的辅助数据结构，随着查询图深度加深以及数据图候选节点的增加，辅助数据结构内存消耗量将会指数级增加，空间复杂度达到 $O(|V(G)|^{|V(q)|})$，其中 $|V(G)|$ 为数据图 G 的节点数目，$|V(q)|$ 为查询图的节点数目。

2015 年，BoostISO[50]对 TurboISO 进一步优化，对等效的节点进行分类，并贪心地优化候选节点的匹配顺序。TurboISO 将查询图重写为 NEC 树，该树同时匹配

具有相同邻域结构的查询图点。与上述技术不同，BoostISO 的方法侧重于（1）通过将“等价”顶点分组在一起来减少搜索空间，以及（2）优化候选顶点匹配顺序以避免重复计算。BoostISO 的方法不是一种单一的算法，而是一种可以集成到所有现有回溯算法中的方法。当查询图的大小变大时，TurboISO 及其增强算法 BoostISO 会对图匹配问题非常有效。这无疑推动了开发新的、更有效的和可扩展的技术来实现子图匹配。

2016 年，CFL-Match 提出提前检查非树边减少冗余匹配。在之前的方法中，普遍都是不考虑某些边先将查询图变成查询树，再对查询树做回溯搜索，对查询树匹配的结果再检查原先未考虑的非树边。CFL-Match 指出，非树边加上对应的树边因为带有环结构，信息量更大也就意味着约束条件更强，有利于更快速地剔除最终不能匹配的中间匹配。CFL-Match 还提出新的辅助结构 CPI，来存储中间结果，解决 TurboISO 中辅助结构内存消耗指数级增长的缺点。CPI 是多层级的多对多关系，层级之间不再因上层节点使得下层节点重复枚举，使得辅助结构变为多项式空间复杂度 $O(|E(G)| \times |V(q)|)$，其中 $|E(G)|$ 是数据图 G 的边个数，$|V(q)|$ 是查询图 q 的节点个数。如图2-4中 CPI 例子，v_3、v_4 和 v_5、v_6 之间相比于 CR 结构不再有重复冗余节点与边。同时，CFL-Match 将一个查询图拆分成三个部分，核部分 V_C 包含所有非树边，森林部分 V_T 即与核相接的点以及其邻点，叶子部分 V_I 即剩余部分。CFL-Match 分别找到所有 V_C 的匹配实例，V_T 的匹配实例，V_I 的匹配实例，对三种匹配实例进行 join 即为最终的匹配。论文中列举出了优先匹配核部分 V_C 的重要性，但在不同的数据图的情况下，优先匹配核可能反而使得查询变慢。

前边提到的算法 TurboISO 和 CFL-Match 使用查询图生成的查询树来进行过滤无效点边，该过程可以找到数据图的（潜在）匹配，通过过滤过程计算的候选集被存储到辅助数据结构中。然后，因为辅助数据结构也有助于选择有效的匹配顺序，所以在辅助数据结构的指导下，通过在回溯过程中检查非树边来找到数据图的所有匹配。虽然使用查询树可以提供如上所述的良好效果，但同时也会产生局

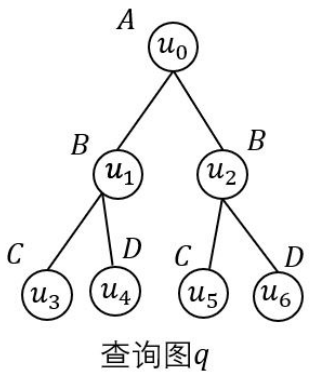


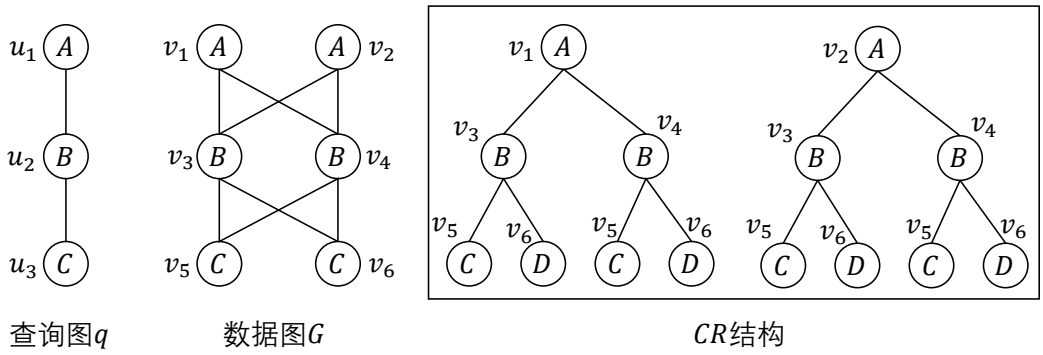
图 2-2　邻居等价类例子

限性。由于查询树不包含查询图的所有边，因此生成的辅助数据结构可能包含许多误报。

2019 年，DAF 基于上述挑战提出将查询图变成有向无环图，有向无环图再展开成查询树。相比于之前直接去掉非树边的方法，DAF 保留了所有边的检查，加强了边约束的检查，将有效降低无效的中间匹配结果。DAF 设计的辅助数据结构称为 CS 精简结构，CS 利用动态规划的方法减少要存储的候选节点，具体方法是只有子节点各自存在匹配，父节点才存在匹配，并且每层节点还要做度数检查要大于等于对应查询图节点的度数，一层层往上不断精简。匹配时也都是在精简后的辅助结构进行匹配，匹配顺序也是按照在精简结构中的预估当前中间结果数来进行匹配。DAF 还提出失败集剪枝，即在组合匹配过程中，如果匹配失败了便剪枝掉与匹配失败无关的匹配树分支。

DAF 还有一个比较大贡献是正式提出了弱匹配的概念。在之前的工作中，子图匹配算法都是去掉查询图的非树边变成查询树，实际匹配算法是用在匹配查询树的而不是查询图，但说查询树的匹配结果一定包含查询图的匹配结果，所以查询树匹配是查询图匹配的一个弱匹配。相比于匹配过程中不断检测环的存在，利用查询树这个弱匹配可以大大降低算法设计与实现的复杂度，同时快速过滤大量不符合匹配条件的中间结果，从而使得最后的端到端的查询返回时间缩短。

DAF 提出弱匹配另外一个意义在于，子图匹配算法不必再拘泥于去掉非树边变成查询树匹配，而是可以像 DAF 那样先变成 DAG，DAG 再展开成查询树，也可以像后文我们将提到的 SymBi 直接弱匹配 DAG。DAF 也存在一些不足，尽管 DAF 检查了所有边约束，但从 DAG 展开成查询树时，为了等价查询树与 DAG 的匹配结果，要对查询树的匹配结果做单射检查从而保证原先查询图的环匹配成功，因为在展开过程中 DAG 的一个节点可能变成了两个查询树的节点；记录失败组合需要额外的空间开销以及时间开销，在 DAF 论文的实验部分也展示了失败集剪枝仅仅在某些数据集上有优化，在其他数据集上甚至起到反效果使端到端的查询回



查询图q　　数据图G　　CR结构

图 2-3　CR 物化结构例子

答时间变长。

2021 年，VEQ 进一步优化了邻居等价类，提出静态邻居等价类与动态邻居等价类对匹配过程进行剪枝，该算法采用了静态等价和动态等价方法。首先，他们将查询图点的邻居等价性应用于回溯的匹配顺序，这导致验证阶段的搜索空间更小。其次，作者基于候选数据顶点的邻居等价性捕获搜索空间子树的运行时等价性，并剪掉这些搜索空间的冗余（即等价子树）。此外，作者提出了一种称为邻居安全的有效过滤方法，使 VEQ 算法能够在查询图和数据图上构建紧凑的辅助数据结构，以获得尽可能少的候选者。它改进了 TurboISO 的邻居等价类方法，提出静态邻居等价类与动态邻居等价类对匹配过程进行剪枝。

2.2.2　单查询动态子图匹配

如图2-5，静态子图匹配算法做动态子图匹配时，分别插入边 (v_0, u_{1501}) 到边 (v_0, u_{1700}) 共 200 条边时，需要对 u_2 与 u_4 的 1000×500 个组合进行 200 次验证，也就是 $1000 \times 500 \times 200$ 次匹配验证，而实际匹配的实例只有 200 个。我们需要利用动态规划的方法将某些部分匹配以某种形式物化（物化即给部分匹配定义状态并用单独的辅助数据结构存储这些部分匹配状态），或者将某些不会产生匹配的部分在物化结构中去除，使得增量匹配时减少重复的验证操作。

2015 年，SJ-Tree[51]提出对于动态子图匹配的增量辅助数据结构左深树（left-deep tree）来存储中间匹配结果。具体的，它先将查询图变成一棵查询树，不断递归地拆分查询树，使得拆分后每个部件都是一条边。SJ-Tree 先是存储每条边地候选节点，然后对相邻两条边的匹配结果进行 join 作为边数为 2 的查询图子图的匹配结果来存储；然后继续与下一条边的候选节点进行 join 作为边数为 3 的查询图子图的匹配结果来存储；以此类推，最终存储整个查询图的匹配结果。在进行边插入时，先找到插入数据边对应查询图的左深树的哪条边，更新那一条边的匹配结果，然后不断递归往边数大的子图方向更新部分匹配结果。虽然 SJ-Tree 提出了一个切实可行的增量匹配算法，但部分匹配的数量太多造成巨量的内存开销，同时

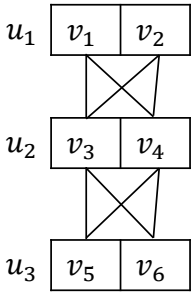


图 2-4　CPI 例子

因为 join 操作太多造成计算开销过大。

2017 年，TurboFlux 提出增量辅助数据结构 DCG，DCG 对于数据图的更新流（插入边或删除边）产生的中间结果利用动态规划以及弱匹配做快速的增量更新。具体的，因为 DCG 本质上也是匹配一棵生成树，故 DCG 定义从查询树（后文我们会提到 TurboFlux 如何将一个查询图变成一棵查询树）根节点到当前节点的路径都匹配成功的数据图节点定义为 $IMPLICIT$ 状态，定义对应查询树子树节点都有对应匹配 $IMPLICIT$ 节点的 $IMPLICIT$ 数据图点为 $EXPLCIT$ 的匹配状态；在插入树边时，在 DCG 中往下更新子树状态，回溯往上更新父节点；插入非树边，往上遍历父节点查看是否有成功匹配的实例。

在 TurboFlux 的算法中，它先为查询图 q 选择一个图搜索的开始节点 u_1（同时也是查询树的根节点），用图搜索（比如 DFS 或者 BFS）的方法从 u_1 开始搜索将查询图通过去除非树边的方式变成查询树；u_1 的选择方式是统计数据图中出现次数最少的边，从该边的两个顶点中再选择出现次数最少的顶点；去除非树边的方式是，从 u_1 开始，每次贪心地加一条边，贪心策略是预估加边后的中间结果个数，注意只加一个顶点在查询树 q' 的边，防止把环加进去。在查询树的基础上，将数据图中的边一条条插入 DCG 中，初始化的 DCG 只包含初始数据图 G_0 中的点，然后按照 DCG 的插入边的规则把 G_0 中一条条边插入 DCG，对插入边的两个顶点进行 $IMPLICIT$ 状态更新，并向下传播 $IMPLICIT$ 状态更新到它们的子节点，有必要时再从叶节点往根节点方向更新 $EXPLICIT$ 状态。将数据图的所有边插入 DCG 后，因为 TurboHOM++ 相比于 worst-case optimal SS 在真实数据集上有更优秀的表现，故采用 TurboHOM++ 算法对所有状态为根节点 $EXPLICIT$ 的数据图节点出发进行子图匹配回溯匹配，并返回 G_0 的子图匹配结果；在这一步中，因为 TurboHOM++ 只需要遍历查询树的弱匹配数据图点，大大减少了无用的遍历与搜索。在 TurboHOM++ 算法中，因为匹配顺序会很大程度影响匹配的速度，所以需要确定查询图节点的匹配顺序；TurboFlux 对 TurboHOM++ 修改的匹配顺序是，对查询树尝试一条边一条边地删除并查看 DCG 中包含多少个查询树的匹配结果，

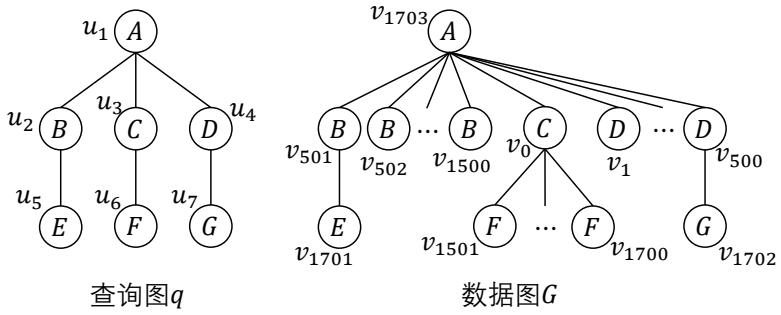


图 2-5　静态子图匹配算法做动态子图匹配问题的缺点

贪心地选择删除该边后 DCG 匹配结果最小的那条边。TurboFlux 在选择匹配顺序时也存在一些不足，比如通过删边估计的匹配树的匹配结果来决定节点在匹配图的匹配顺序，其弊有用边决定点的匹配顺序、用匹配树结果数目决定在匹配图的匹配顺序和用 DCG 中匹配结果数目来假设未来插入点边造成的匹配数目。如果问题场景需要解决子图同构问题而非子图同态问题，此时只需要对子图同态的结果做单射检查，即检查是否所有查询图节点都（匹配）映射到不同的数据图节点，通过检查的所有子图同态结果也是子图同构结果。TurboFlux 构建的辅助数据结构 DCG 的空间复杂度仅为 $O(|V(q)| \cdot |E(G)|)$，其中 $|V(q)|$ 是查询图 q 的节点个数，$|E(G)|$ 是数据图 G 的边个数。

同年 2017 年，Graphflow[52] 提出了一种旨在支持动态更新和维护大规模、复杂的图数据、提高图查询性能的图查询系统。系统采用了一种活跃的数据结构，利用活跃度量来动态监测图中节点和边的活跃程度。同时，Graphflow 还引入了一种基于启发式算法的查询优化器，用于优化复杂查询语句的执行。Graphflow 在处理大规模、复杂的图数据时具有出色的性能和可扩展性，且能够支持多种应用场景，如社交网络分析、路网导航等。

上述的活跃图数据库系统（Active Graph Database）是一种特殊类型的图数据库。相对于传统的静态图数据库，活跃图数据库能够自适应地处理图数据中的动态变化，比如节点或边的新增、删除或更新，应用场景更加灵活。

这篇文献提到的两种查询优化技术分别为：Generic Join 和 Delta Generic Join，它们被用来优化基于图模式匹配的查询语句，提高查询性能。具体来说，Generic Join 是一种用于优化传统的基于图模式匹配的查询语句的方法。在这种方法中，查询图被转换成一个关系模式，然后使用传统的关系型数据库查询技术来处理查询。但这种方法并不是设计用来支持增量查询的。

为了解决这个问题，作者随后提出了 Delta Generic Join。它是一种基于增量计算的查询优化技术，能够利用活跃度量和差分计算技术，对查询图进行动态更新，并在查询执行过程中，只计算与查询结果相关的部分，从而减少计算量和数据传输量。这种方法相较于 Generic Join 能够显著提高查询性能。

Graphflow 是用 Java 实现的单节点内存系统，共有四个重要组成部分：

（1）In-memory Property Graph Store；Graphflow 的内存图存储引擎支持以图方式存储数据，Graphflow 的数据模型是属性图，图是有向的，并且能够以 O(1) 时间访问相邻节点。此外，Graphflow 还提供了一组基本的图操作 API。

（2）Cypher++ Query Language；Graphflow 引入了一种被称为 Cypher++ 的查询语言，这是对 Cypher 语言的扩展。Cypher++ 支持更多类型的图操作，包括高级

的过滤、排序和聚合。

（3）One-time Subgraph Queries；Graphflow 对静态子图匹配的支持。

（4）Continuous Subgraph Queries；连续子图查询使得用户能够创建一些“活动查询”，并在图数据发生变化时自动更新查询结果。这种能力使得 Graphflow 能够更好地支持那些需要实时响应变化的应用程序。

2021 年，CEPDG 提出研究一个新问题，即在动态数据图与动态查询图的情况下，如何返回正向以及负向子图匹配结果。它采用初始查询图、初始数据图和图更新流（有可能是查询图的更新，也有可能是数据图的更新）作为输入，找到增量匹配结果。CEPDG 中作者主要提出了一种顶点状态转换策略 VST，来记录中间结果。

在数据图插入边或者查询图删除边都会带来正匹配，在数据图删除边或者查询图插入边都会带来负匹配；数据图的更新可能只会影响小部分的子图匹配结果，但查询图的更新将会影响几乎全部的子图匹配结果。

CEPDG:

（1）在查询图中选择一个节点作为根节点，然后用 BFS 通过去掉非树边使得查询图变为查询树；

（2）从查询树的根节点 u_r 开始，为树节点进行层数标记；

（3）构建增量辅助物化结构 TreeMat；

（4）为查询树的每个叶节点寻找候选数据节点；

（5）从叶节点层开始，从下到上构建查询树每个节点的候选节点，每个查询树节点的候选数据节点的要求是必须要与查询图下层的所有节点至少一个候选数据节点有连边；

（6）初始化各个节点的物化状态。在上一步从下往上筛选候选节点后，对根节点 u_r 的每个候选数据节点，将其加入集合 $match(u_r)$；从最上层开始，从上往下更新查询树候选节点的匹配状态，对于每个查询点 u_c 的候选点 v_c，如果存在邻点 v_p 是 u 的父节点 u_p 的 $match$ 状态，那么 v_c 的状态也为 $match$ 加入 $match(u_c)$ 中；否则，将 v_c 加入 $stree(u_c)$ 中。其中 $stree$ 代表 $sub-tree$ 子树匹配，$match$ 表示该节点在可能在某个成功匹配的实例里。

（7）按顺序处理更新流 $\Delta U = \{\Delta op_1, \cdots, \Delta op_n\}$，更新数据图 G_i 以及查询图 q_i，更新 TreeMat。在更新 TreeMat 时，按照文章中定义的 $NULL$，$stree$，和 $match$ 三者状态的转换条件进行转换。

（8）从 TreeMat 搜索增量匹配实例。CEPDG 从插入边对应的节点开始进行匹配，并且之后的匹配顺序中只匹配与已匹配的查询点有连边的查询点，保证匹

配出来的实例都是包含新插入的数据边的, 也就保证匹配结果是增量匹配。

CEPDG 从下到上构建查询图节点的候选数据节点是一种层层剪枝的策略，叶节点候选可能比较多，但通过层层的约束条件不断过滤筛选候选点，在子树匹配的节点较少的情况下实际内存花销较小。TurboFlux 与 CEPDG 的内存消耗上各有利弊。TurboFlux 的内存消耗主要有两个部分，$IMPLICIT$ 边的存储以及 $EXPLICIT$ 边的存储。TurboFlux 实际不存储 $IMPLICIT$ 的边，对于每个数据节点 $v \in V(G)$，都有一个长度为 $|V(q)|$ 的 bitmap 记录下来 $IMPLICIT$ 的状态（$IMPLICIT$ 是指从根节点到当前节点的 path 都匹配成功），这里的空间开销固定为 $O(|V(G)| \cdot |V(q)|)$。另一方面，TurboFlux 对 $EXPLICIT$ 的边的存储空间复杂度为 $O(|V(G)| \cdot |V(q)|)$。CEPDG 对于 $stree$ 以及 $match$ 的边的存储都是通过层层剪枝的，具体花销是取决于数据与剪枝效果的，但总的空间复杂度为 $O(|V(G)| \cdot |V(q)|)$。

2021 年，SymBi 指出 TurboFlux 用生成树做弱匹配进行过滤的效果效率并不高，因为生成树的方法并没有利用查询图的所有边关系进行剪枝。于是 SymBi 以将查询图变成有向无环图的方式构建辅助结构记录数据节点的中间匹配状态以及中间匹配结果。

2022 年，RapidFlow 算法被提出来。它的算法框架主要包括：

（1）针对查询图，设计一个全局索引结构，将满足查询图邻边条件的数据图点存储在全局索引结构；

（2）针对插入边、删除边，设计一个局部索引结构，按照查询图，从插入边或者删除边出发在全局索引结构搜索相关的数据图点边；

（3）基于局部索引结构里的候选数据图点后，不再需要从插入边的邻点开始匹配，而是可以从任意候选点数目较少的查询图点位置开始匹配，大大提升了匹配效率；

（4）RapidFlow 针对查询图的自同构提出了对偶匹配技术，大大减少了匹配过程中的重复计算。

作者在实验中，将 RapidFlow 与 Turboflux 和 SymBi 进行了比较，RapidFlow 在各种工作负载上的性能优于现有算法，包括 TurboFlux 和 SymBi，高达两个数量级。

2.2.3　多查询动态子图匹配

多查询动态子图匹配问题是单查询动态子图匹配问题的扩展，其在实际应用中具有更广泛的应用场景。

2020 年，TRIC 中提出了一种多查询动态子图匹配算法，先将每个查询图分解成多个 path，所有查询图的所有 path 再进行合并成一个查询 path 森林，来支持同时更新多个查询图的增量部分匹配结果，最终一个查询图的匹配结果由多个 path

的匹配结果进行 join 得到。具体 TRIC 运行的例子我们在章节1.3有提及。

2020 年，Grace Fan 等人在文章[49]中提出了多种针对边带有条件的查询图 Conditional Graph Patterns（CGP）的匹配算法，其中包括多查询动态子图匹配算法 IncCMatch$_{Ann}$。在实际生活中，图的结构日益复杂，许多边有着丰富的物理意义，只有在某种特定条件满足的情况下才可能出现，例如在线上购物网站的用户商品推荐图中，许多公司会通过判断某用户 A 的好友 B 是否购买过某商品 C 来决定是否要把商品 C 推荐给用户 A，即当 B 与 C 之间存在一条“购买”边时，A 与 C 之间才可能出现“推荐”边。因此我们需要将经典的子图匹配问题进行扩展，以适应这样的需求，这也是作者设计 IncCMatch$_{Ann}$ 的初衷。

IncCMatch$_{Ann}$ 算法基于 Annotated CGPs 查询结构，Annotated CGPs 是一种边带有标记的由多个 CGP 结合生成的合成查询图，若边 $e(u_1, u_2)$ 由查询 A 和查询 B 共享，则标记 e 为 $\{A, B\}$。但是直接生成 Annotated CGPs 是十分困难的，所以作者首先生成一个 Annotated CGPs，再将该 Annotated CGPs 中的点和边进行分组，每一组代表一个查询 CGP，并依据分组进行标记，以此来模拟一个 Annotated CGPs。在获得 Annotated CGPs 后，作者首先使用 CFL-Match 中提出的自顶向下和自底向上的索引构建技术为 Annotated CGPs 建立初始索引结构 CCPI。在 IncCMatch$_{Ann}$ 算法中，其接收一个 Annotated CGPs、一个 CCPI 和一组更新边 ΔG 作为参数，提取出 ΔG 中每一条可能与 Annotated CGPs 匹配的插入（删除）边形成集合 $cand^+$（$cand^-$）。分别根据插入边集 $cand^+$ 和删除边集 $cand^-$ 对 CCPI 进行自顶向下和自底向上的更新，对于 $cand^+$ 中涉及的节点，仅对其候选列表进行添加，而对于 $cand^-$ 中涉及的节点，仅对其候选列表进行删减。更新结束后，根据更新边集 ΔG 从数据图 G 中提取被更新影响的区域 AFF，即从任意一条更新边开始，d 跳范围内的子图（d 为 Annotated CGPs 的直径）。最终在 AFF 内利用传统回溯算法搜索 Annotated CGPs 新增或减少的匹配。此外，文中还提出了新的匹配顺序，即按照边的标记数量降序进行排序，对标记数量多的边优先进行匹配。

作者将已有的子图匹配算法进行修改，并辅以高效的索引更新算法，使其适用于对多查询 CGP 的匹配任务。但 IncCMatch$_{Ann}$ 难以应用于流式更新数据图的多查询图动态子图匹配问题，因为 IncCMatch$_{Ann}$ 处理更新边的方式为批处理，即将所有的更新边一次性插入（删除）后，再对 Annotated CGPs 的索引数据结构进行更新并进行匹配。若将其用于流式处理，则每一条边的插入或删除都会带来自顶向下和自底向上的索引更新，造成大量的冗余计算，导致性能非常差。

第 3 章　ShareFlow：构建查询图同构标记图

我们的总的算法流程如算法3-1所示。我们将首先提出一个方法如何将多个查询图合并成一个标记图 g_a，也就意味着我们提供了一个简单高效的且 IncCMatch$_{ann}$ 中未解决的问题的算法；然后将标记图变成一个 rDAG（rooted Directed Acyclic Graph）森林，即多个 rDAG 组成的图，我们将其称为 rDAG 森林。给定一个 rDAG 森林，一个数据图初始图 G_0，和一个数据图更新流 ΔG, 我们的算法将在 rDAG 森林上建立一个增量辅助数据结构，共享增量状态图 g_s（Shared Incremental State Graph，SISG），来记录增量更新，最后在 SISG 上搜索增量更新。

算法 3-1　ShareFlow

```
Data: 查询图集合 Q，初始的数据图 G_0，数据图更新集合 ΔG
1  g_a ← BuildAnnGraph(Q) ;
2  g_r ← BuldRDAGForest(g_a) ;
3  g_s ← ∅ ;
4  foreach (v_i, v_j) ∈ E(G_0) do
5      if v_i.id < v_j.id then
6          SISGInsert(g_s, (v_i, v_j)) ;
7          SISGFindMatch(g_s, (v_i, v_j)) ;          /* 报告初始匹配 */
8      end
9  end
10 foreach Δop ∈ ΔG do
11     if IsInsertion(Δop.operator) then
12         G_0.InsertEdge(Δop.edge) ;
13         SISGInsert(g_s, Δop.edge) ;
14         SISGFindMatch(g_s, Δop.edge) ;            /* 报告正匹配 */
15     else
16         SISGFindMatch(g_s, Δop.edge) ;            /* 报告负匹配 */
17         SISG.DeleteEdge(g_s, Δop.edge) ;
18         G_0.DeleteEdge(Δop.edge) ;
19     end
20 end
```

我们记录增量更新或搜索增量更新的方法与主流的动态子图匹配问题的相关工作相似，基本都按如下框架返回增量匹配（因插入数据图的边带来的正匹配或因删除数据图的边带来的负匹配）：

（1）对于每一个边插入操作：

（2）　　对数据图进行更新；

（3）　对增量辅助数据结构进行更新；

（4）　在增量辅助数据结构中寻找正匹配。

（1）对于每一个边删除操作：

（2）　在增量辅助数据结构中寻找负匹配；

（3）　对增量辅助数据结构进行更新；

（4）　对数据图进行更新。

但是，在具体的增量更新或者是增量匹配过程中，我们的算法都设计成支持多个查询图的共享执行。

3.1　从多个查询图到一个标记图

在本小章中我们给出标记图、查询图同构标记图、查询图集合同构标记图和完美查询图集合同构标记图的定义；然后给出我们的算法在给定查询图集合 Q 的情况下，如何生成 Q 的完美查询图集合同构标记图 g_a；最后给出证明，证明我们的算法生成的标记图 g_a 是 Q 的完美查询图集合同构标记图。在这里我们提出查询图同构标记图的概念一个很重要的原因就是一些类似 TRIC 的相关工作它们并不是在解决一个原查询图（native query graph）的问题或者对偶问题，而是将一个查询图打散再将多个打散后的查询图合并；这里很关键的一个点是，对于原查询图来说，合并之后的共享查询图无法直接找到原查询图，需要通过将打散后的原查询图各个部分的结果进行 join 来得到原查询图结果；这样做的结果使得部分匹配结果数量非常多且需要耗费大量时间进行 join，时间开销非常高。而我们则设计一套解决方案，使得原查询图仍能在我们之后的共享查询图（也就是查询图集合同构标记图）找到一个图同构对象，且共享查询图不会有其他多余的设计（如相关查询图以外的查询图或者对一个查询图进行重复，也就是完美查询图集合同构标记图）。

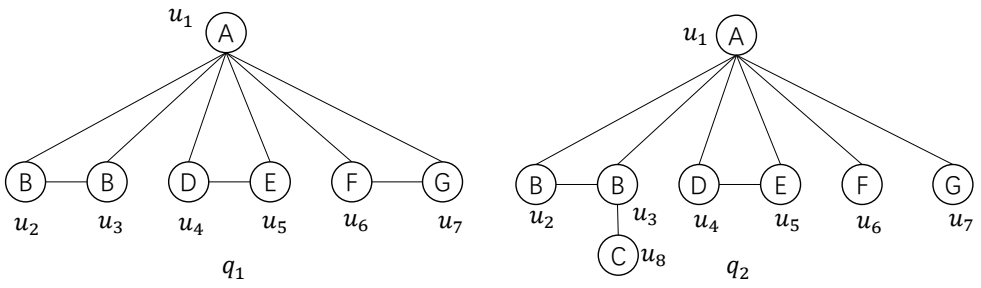
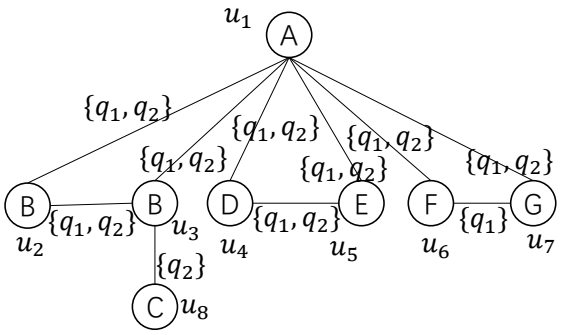


图 3-1　查询图集合 Q

定义 3.1 (标记图)：标记图是一个图，$g_a = (V, E, L, A_v, A_{vq}, A_e)$, 其中 V 是标记图的节点集合，$E \subseteq V \times V$ 是标记图的边集合，$L : V \to \Sigma_V$ 是一个从节点集合 V 的节点



vertex	A_v	A_{vq}
u_1	$\{q_1.u_1, q_2.u_1\}$	$\{q_1, q_2\}$
u_2	$\{q_1.u_2, q_2.u_2\}$	$\{q_1, q_2\}$
u_3	$\{q_1.u_3, q_2.u_3\}$	$\{q_1, q_2\}$
u_4	$\{q_1.u_4, q_2.u_4\}$	$\{q_1, q_2\}$
u_5	$\{q_1.u_5, q_2.u_5\}$	$\{q_1, q_2\}$
u_6	$\{q_1.u_6, q_2.u_6\}$	$\{q_1, q_2\}$
u_7	$\{q_1.u_7, q_2.u_7\}$	$\{q_1, q_2\}$
u_8	$\{q_2.u_8\}$	$\{q_2\}$

图 3-2　标记图 g_a

到点标签集合 Σ_V 的点标签的映射。每个标记图节点都标记了一些查询图的节点，其中 A_v 是标记图节点到被标记的查询图节点集合的映射, 且满足 $\forall u_q \in A_v(u)$ 都有 $L(u) = L(u_q)$；A_{vq} 是一个在 A_v 基础上从标记图节点到被标记查询图集合的映射；每个标记图边都标记了一些查询图，其中 A_e 是标记图边到被标记的查询图集合的映射。如图3-2是一个标记图的例子，如图中表格所示 $A_v(u_1) = \{q_1.u_1, q_2.u_1\}$ 且 $A_{vq}(u_1) = \{q_1, q_2\}$，如图所示 $A_e((u_3, u_8)) = \{q_2\}$。

定义 3.2 (查询图同构标记图)：对于查询图 q，假设标记图 g_a 中所有标记有 q 的点边构成的子图为 q'，如果 q 与 q' 图同构，那么称 g_a 是 q 的查询图同构标记图。如图3-2中的标记图 g_a 既是图3-1中 q_1 的同构标记图，也是 q_2 的同构标记图。

定义 3.3 (查询图集合同构标记图)：如果 $\forall q \in Q$，都有 g_a 是 q 的查询图同构标记图，那么 g_a 是查询图集合 Q 的同构标记图。如图3-2中的标记图 g_a 是图3-1中查询图集合 $Q = \{q_1, q_2\}$ 的同构标记图。

定义 3.4 (完美查询图集合同构标记图)：标记图 g_a 是查询图集合 Q 的完美同构标记图当且仅当：

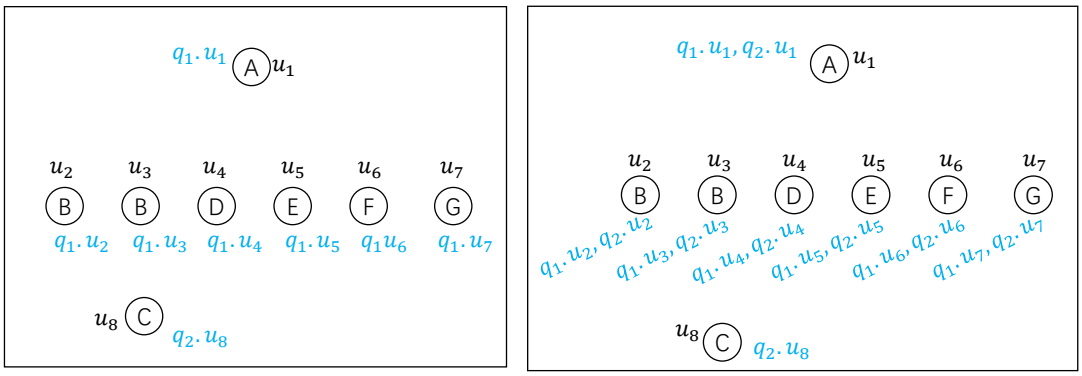
（1）g_a 是 Q 的同构标记图；

（2）g_a 中不存在任何 Q 中查询图以外的其它查询图标记。

如图3-2中的标记图 g_a 因为是图3-1中 $Q = \{q_1, q_2\}$ 的同构标记图，且 g_a 没有 q_1 或 q_2 以外的标记，所以 g_a 是 Q 的完美同构标记图。

如算法3-2所示，我们将所有查询图合并成标记图一共分为四步:

（1）初始化标记图 g_a。在算法3-2第一行中我们先初始化标记图 g_a 为空。g_a 与一般的图不同的是 g_a 的每个点或边都有查询图的标记。插入点分两种情况，第一种是插入新点并给新点赋予点标记（标记图不存在点标记为空的点），第二种是对指定的已存在的标记图点插入点标记。在插入边时如果两个点之间已经存在边



(a) 映射第一批节点后的 g_a　　(b) 映射剩余节点后的 g_a

图 3-3　标记图 g_a 点映射过程

算法 3-2　BuildAnnGraph

Data: 查询图集合 Q
Result: Q 的查询图同构标记图
1 $g_a \leftarrow \emptyset$;
2 $V_{inserted} \leftarrow$ InsertFirstBatchVertices(g_a, Q) ;
3 MapOtherVertices$(g_a, Q, V_{inserted})$;
4 MapEdges$(g_a, Q, V_{inserted})$;
5 **return** g_a ;

则给边加上查询图的标记，否则同时插入边和加上查询图的标记。

算法 3-3　InsertFirstBatchVertices

Data: 标记图 g_a，查询图集合 Q
Result: 插入到 g_a 的点集合
1 $V_{inserted} \leftarrow \emptyset$;
2 **foreach** *vertex label* l **do**
3 　$q_i \leftarrow \arg\max_{q_i \in Q} |\{v | L(v) = l, v \in V(q_i)\}|$;
4 　**foreach** $v_j \in V(q_i)$ **do**
5 　　**if** $L(v_j) = l$ **then**
6 　　　$v_{g_a} \leftarrow g_a.InsertVertex(q_i, v_j)$;
7 　　　$V_{inserted} \leftarrow V_{inserted} \cup \{\langle key \leftarrow \langle q_i, v_j \rangle, value \leftarrow v_{g_a} \rangle\}$;
8 　　**end**
9 　**end**
10 **end**
11 **return** $V_{inserted}$;

（2）对每个点标签预先在标记图插入一些查询图的点。如算法3-3所示，我们先在第一行初始化一个 key-value 结构的变量 $V_{inserted}$，这个结构的每个元素都是一对 key-value，key 是查询图 q_i 与 q_i 的节点 v_j，value 是 v_j 映射到标记图的节点 v_{g_a}。对于任意点标签 l，第三行中我们在 Q 的查询图中找到有最多数量点标签 l

的点的查询图记为 q_i。然后将 q_i 中所有点标签为 l 的点全部直接插入标签图，也就是每个查询图的点都会映射到不同的新的标签图的点，并返回每个查询图的点映射到的标签图的点 v_{g_a}。图3-1中的点标签 A，B，D，E，F，和 G 节点数量最多的查询图为 q_1，点标签 C 节点数量最多的查询图为 q_2，那么在此步骤插入 q_1 所有点标签 A，B，C，D，E，F，和 G 的节点，以及 q_2 所有点标签为 C 的节点之后，图3-1将变成如图3-3(a)所示，每个插入的点都将带上对应的查询图与节点的标记，其中黑色的是标记图点，蓝色是标记图点对应的查询图标记。

算法 3-4　MapOtherVertices

```
Data: 标记图 g_a，查询图集合 Q，已插入的查询图节点集合 V_inserted
1  foreach query graph q ∈ Q do
2      vl2vMap ← ∅ ;                                  /* 给 q_i 的节点按点标签分组 */
3      foreach v ∈ V(q) do
4          vl2vMap[L(v)] ← vl2vMap[L(v)] ∪ {v} ;
5      end
6      foreach vl2v ∈ vl2vMap do
7          if isIntersect(V_inserted, vl2v, q) then
8              continue ;                              /* 检查查询图点是否已插入 */
9          end
10         V_gal ← g_a.GetVerticesByLabel(vl2v.vl) ;
           /* v2uArray 是一个 ⟨v,u⟩ 数组，v 是 q 的节点，u 是 g_a 节点，
              要在 u 中插入标记 q.v                                          */
11         v2uArray ← FindVertexMapping(vl2v.vertices, V_gal, Q);
12         foreach v2u ∈ v2uArray do
13             g_a.insertVertex(q, v2u.v, v2u.u) ;      /* 映射 q.v 到 g_a.u */
14             V_inserted ← V_inserted ∪ {⟨key ← ⟨q, v2u.v⟩, value ← v2u.u⟩}
15         end
16     end
17 end
```

（3）插入（映射）剩余未插入的点。在这一步插入剩余查询图点的过程中，不会给标记图增加新的点，而是在现有点标签一样的标记图点中进行选择，给标记图的点增添查询图以及查询图点的标记来完成查询图点的映射。查询图点的映射遵循着尽可能让查询图点映射到相似的标签图节点中（比如邻居点标签种类以及数量作为相似依据）。

先对查询图剩下的点进行分组，使得同一个组内的点属于同一个查询图且点标签相同。组内的点分别单射到 g_a 中点标签一样的点。单射时贪心地选择每个点要映射的标记图的点。如算法3-4所示，对于每一个查询图 q，在第二行初始化一个映射函数 $vl2vMap$，这个映射函数的每一个映射都是一个 key-value 对，key 是点标签，value 是该查询图拥有该点标签的所有点。在第三到第五行我们对查询图 q

的点按照点标签进行分组。对于同一个分组的点，在第七行检查是否这些点已经插入到过标记图。因为我们在标记图插入点或者映射点时，都是按照一组同样查询图和点标签的方式插入。也就是整一个组的点都要么已经插入到标记图，要么都没插入到标记图。所以检查步骤只需要检查⟨查询图，点标签⟩这对信息是否已经在标记图已经存在即可。如果该分组的点没有映射到标签图，则 $FindVertexMapping$ 函数将为该分组的点找到不同的标签图的点进行映射。这里因为在第（2）步我们保证了拥有最多数量该点标签的查询图已经将它对应的点插入到标记图中，所以在这一步中我们总是能保证查询图同一组点标签的点能映射到足够多不同的标记图的点，也就是完成单射。最后根据找到的分组的映射，在第十一行到第十四行，我们将分组的点逐一映射到标签图的点，即在标签图的点添加 q 和 $q.v$ 的标记。

算法 3-5　FindVertexMapping

Data: 待插入查询图点组 V_{ql}，标记图待被映射点组 V_{gal}，查询图集合 Q
Result: 一个待插入查询图点到标记图点的映射 f
1 $F \leftarrow \{f | f$ is an injective mapping from V_{ql} to $V_{gal}\}$；　/* 找到所有的映射方式 */
2 $f \leftarrow \arg\max_{f \in F} CalculateScore(f, Q)$；　/* 贪心取分数最大的映射 */
3 **return** f；

如算法3-5所示，它的输入主要两组边，第一组是查询图 q 中所有点标签为 l 的点 V_{ql}，第二组是标记图 g_a 中所有点标签为 l 的点 V_{gal}。在第一行中，我们枚举所有从 V_{ql} 到 V_{gal} 的单射函数 f 并存储在 F 中，每一个单射函数都是多个 key-value 对，其中 key 都是查询图点，value 都是标记图点。在第二行中我们计算每一个单射函数的相似分数，并返回相似分数最大单射函数 f。在定理3.1中我们证明了我们这里的枚举过程一共会枚举 A_m^n 中单射。尽管单射的数目众多，因为查询图的点数非常少（在我们的默认设置中点数是 6）和 V_{gal} 的最大数目也不会超过一个查询图的点数（V_{gal} 的数目是某一个查询图某一个点标签的所有点数目），所以这个枚举计算分数过程中产生计算开销即使要计算生成 g_a 的时间也可以忽略不计。

定理 3.1：给定 m 个标记图点，n 个查询图点，其中 $m \geqslant n$，一共有 A_m^n 种从查询图点到标记图点的单射函数。

证明：从 m 个标记图点选 n 个供查询图点来单射共有 C_m^n 种可能。不妨设这 n 个查询图点是 $u_1, u_2, .., u_n$，n 个标记图点按顺序是 $v_1, v_2, ..., v_n$, 那么 $\{u_i \rightarrow v_i | i \in [1, n]\}$ 将是一个满足要求的单射函数。又任意不同顺序的标记图点按上述过程生成的映射函数都是不同的，一共有这样不同的顺序 A_n^n 个，故一共有 A_m^n 个单射函数。 ■

相似分数的计算可参考算法3-6。因为在计算一个标记图节点与查询图节点的相似性时，标记图节点已经有一些插入（或者说标记了一些）别的查询图点，假设该标记图节点是 $g_a.u$，标记了 $\{q_1.u_2, q_2.u_2, q_2.u_3\}$，那么我们可以先计算

算法 3-6　CalculateScore

Data: 一个查询图点到标记图点的单射 f，查询图集合 Q
Result: 该单射下的相似性分数 s
1 $s \leftarrow 0$;　　/* 初始化分数为 0 */
2 **foreach** $v_q \rightarrow v_a \in f$ **do**
3 　$s \leftarrow s + \max(\{overlap(v_q, v_q') | v_q' \in A_v(v_a)\})$;
4 **end**
5 **return** s ;

每个被标记点与待被插入查询图点的相似性，然后取最大值，即算法第三行的 $\max(\{overlap(v_q, v_q') | v_q' \in A_v(v_a)\})$，作为该标记图点与该被插入查询图点的相似性。这种相似性分数意味着我们并不是看标记图点所有边来计算相似性，而是被标记查询图点的邻边信息。如第三行所示，整个单射的分数则是每个映射对 $v_q \rightarrow v_a$ 的相似分数之和。

此步骤一个具体的例子如图3-3所示，插入剩余查询图的过程就是从图3-3(a)到图3-3(b)的过程。剩余的插入点包括 $q_2.u_1$，$q_2.u_2$，$q_2.u_3$，$q_2.u_4$，$q_2.u_5$，$q_2.u_6$，和 $q_2.u_7$。在此步骤中我们将映射（合并）这些点到图3-3(a)中标记图的点。因为和点 $q_2.u_1$，$q_2.u_4$，$q_2.u_5$，$q_2.u_6$，和 $q_2.u_7$ 的点标签一样的标记图点分别只有一个，我们的算法3-5将会直接返回并映射到它们各自点标签一样的标记图点。在图(b)的标记图中，$q_2.u_1$ 与 $q_1.u_1$ 合并，$q_2.u_2$ 与 $q_1.u_2$ 合并，$q_2.u_3$ 与 $q_1.u_3$ 合并，$q_2.u_4$ 与 $q_1.u_4$ 合并，$q_2.u_5$ 与 $q_1.u_5$ 合并，$q_2.u_6$ 与 $q_1.u_6$ 合并，$q_2.u_7$ 与 $q_1.u_7$ 合并。因为 $q_2.u_2$ 和 $q_2.u_3$ 拥有同样的点标签，所以它们将会被放入同一个分组并作为参数 V_{ql} 交给算法3-5处理。在标记图中点标签同样点标签为 B 的 $g_a.u_2$ 和 $g_a.u_3$ 作为另一组输入 V_{gal}。在这个例子中，算法3-5在第二行伪代码先找到所有可能的单射函数，它们分别为 $f_1 = \{q_2.u_2 \rightarrow g_a.u_2, q_2.u_3 \rightarrow g_a.u_3\}$ 和 $f_2 = \{q_2.u_2 \rightarrow g_a.u_3, q_2.u_3 \rightarrow g_a.u_2\}$。在算法3-6中我们分别计算这两个映射的相似分数。两个点标签相同的查询图点的相似度，也就是 $overlap$ 函数所计算的，就是两个点邻点中点标签相同的点数除以两个点的总邻点数。因为已经映射到 $g_a.u_2$ 的只有 $q_1.u_2$，映射到 $g_a.u_3$ 的只有 $q_1.u_3$，所以 $q_2.u_2 \rightarrow g_a.u_2$ 的相似分数也就是 $q_2.u_2$ 与 $q_1.u_2$ 的相似度；又因为 $q_2.u_2$ 和 $q_1.u_2$ 都有一个点标签为 A 的邻点，一个点标签为 B 的邻点，$q_1.u_2$ 和 $q_2.u_2$ 的总邻点数是 4，故 $q_2.u_2$ 与 $q_1.u_2$ 的相似度是 $\frac{1}{2} = \frac{2}{4}$，即 $q_2.u_2 \rightarrow g_a.u_2$ 的分数是 $\frac{1}{2}$。同理，$q_2.u_3 \rightarrow g_a.u_3$ 的分数是 $\frac{2}{5}$，$q_2.u_2 \rightarrow g_a.u_3$ 的分数是 $\frac{1}{2}$，$q_2.u_3 \rightarrow g_a.u_2$ 的分数是 $\frac{2}{5}$。所以 f_1 的总分数是 $\frac{9}{10} = \frac{1}{2} + \frac{2}{5}$，$f_2$ 的总分数是 $\frac{9}{10} = \frac{1}{2} + \frac{2}{5}$。因为两个映射分数都相同，我们将随机返回一个。如果返回的是 f_1 那么映射结果就会如图3-3(b)，$q_2.u_2$ 将在 $g_a.u_2$ 与 $q_1.u_2$ 合并，$q_2.u_3$ 将在 $g_a.u_3$ 与 $q_1.u_3$ 合并。

（4）映射查询图的边。在算法3-7中，我们将所有查询图的边都插入到标签图

算法 3-7　MapEdges

```
Data: 标记图 g_a，查询图集合 Q，已经插入的查询图节点集合 V_inserted
1  foreach q ∈ Q do
2      foreach v ∈ V(q) do
3          foreach neighbor vertex v_n ∈ N(v) do
4              if v.id > v_n.id then
5                  continue ;                          /* 防止边重复插入 */
6              end
7              g_a.InsertEdge(V_inserted[⟨q, q.v⟩], V_inserted[⟨q, q.v_n⟩], q)
8          end
9      end
10 end
```

中。对于每个查询图的每一条边，我们查找边两个点分别映射到哪两个标记图的点，在第七行中我们使用标签图 g_a 的 $InsertEdge$ 函数进行边的插入，如果边不存在则插入边并给做边标记；如果边已经存在则只给边添加边标记。最终图3-1的两个查询图将变成图3-2中的标记图。但这里要说明的是最终的标记图不一定是一个连通图，因为会出现在某个查询图集合与另外一个查询图集合没有任何相同的点标签的情况。比如假设 Q 只有两个查询图 q 和 q'，且如果 $\forall v \in q$，$\forall v' \in q'$ 都有 $L(v) \neq L(v')$，那么 $Q = \{q, q'\}$ 只会生成一个非连通的标记图。

因为 IncCMatch$_{ann}$ 不能构造标记图而只能直接用标记图作为输入再拆分成各个查询图，我们在这里生成的标记图算法可以直接作为 IncCMatch$_{ann}$ 的输入（我们在之后的实验中的实现也是这么做的）。同时标记图可以用于静态子图匹配问题（区别于动态子图匹配），指导同时匹配多个子图，这是未来可能会值得研究的方向。

假设查询图集合为 Q，标记图是 g_a，接下来我们将证明我们生成的 g_a 是 Q 的完美查询图集合同构标记图。首先，我们定义标记图的完美性质。

定义 3.5 (完美性质)：

（1）点标记单个查询图点的单射性：每个查询图的每个点都会被标记且仅被标记在一个标记图点上。从单个查询图 q 来说，不同点必须单射到不同的标记图点；但从多个查询图来说，不同查询图的点可以映射到同一个标记图点。

（2）点标记查询图点的满射性：不存在 $v \notin V(Q)$，使得 $\exists u \in V(g_a)$，$v \in A_v(u)$。

（3）标签图的每个边存在的充分必要条件：标记图 g_a 的边 (u, u') 存在当且仅当 $A_e((u, u')) \neq \emptyset$。

（4）标签图的每个边的边标记存在的充分必要条件：标记图 g_a 的边 (u_i, u_j)

有标记 q 当且仅当 u_i 有标记 q，u_j 有标记 q，且假如 u_i 标记 q 的节点为 $q.u_s$，u_j 标记 q 的节点为 $q.u_t$，那么 $(q.u_s, q.u_t) \in E(q)$。

定理 3.2：对于查询图集合 Q，满足所有完美性质的标记图 g_a 是 Q 的完美同构标记图。

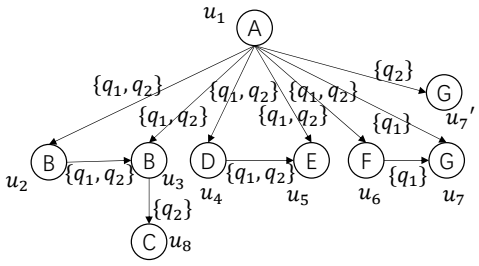
证明：由完美性质（1）可得，$\forall q \in Q$，$V = \{u | q \in A_{vq}(u), u \in V(g_a)\}$，存在一个从查询图 q 的节点到 V 的双射 f 使得 $\forall v \in V(Q)$, 都有 $v \in A_v(f(v))$；由标记图的定义3.1可得 $\forall v \in Q$，都有 $L(v) = L(f(v))$。由完美性质（3）和（4）可得，$\forall q \in Q$，$\forall (u_i, u_j) \in E(q)$，都有 $(f(u_i), f(u_j)) \in E(g_a)$，$q \in A_e((f(u_i), f(u_j)))$。又 $\forall q \in Q$，$\forall e \in E(q)$，都有 e 只会给 g_a 产生一个标记，故 g_a 是 Q 的查询图集合同构标记图。又完美性质（2）的满射性限制点标记只能是 Q 中的查询图，结合完美性质（4）边标记存在的前提条件是点标记存在导致边标记也只能是 Q 中的查询图，故 g_a 是 Q 的完美同构标记图。 ■

其中完美性质（1）和（2）由我们算法中“（2）对每个点标签预先在标记图插入一些查询图的点”和“（3）插入（映射）剩余未插入的点”这两个步骤保证了同一个查询图的点不会映射到 g_a 中的同一个点，也就保证了完美性质（1）；另一方面，g_a 的所有点标记也是这两步的输入 Q 来提供，保证了完美性质（2）。算法中“（4）映射查询图的边”保证了每个新的边标记都有唯一查询图边与之对应，故保证了完美性质（4）；另外，算法步骤“（4）映射查询图的边”并没有插入任何没有边标记的边，故保证了完美性质（3）。所以我们算法生成的标记图 g_a 满足所有的完美性质，即 g_a 是算法输入的 Q 的完美查询图集合同构标记图。

3.2　从标记图到 rDAG 森林

在本小章中我们给出 rDAG 森林、查询图同构 rDAG 森林、查询图集合同构 rDAG 森林和完美查询图集合同构 rDAG 森林的定义；然后给出我们的算法在给定 Q 的完美同构标记图 g_a 的情况下，如何生成 Q 的完美查询图集合同构 rDAG 森林 g_r；最后给出证明，证明我们的算法生成的标记图 g_r 是 Q 的完美查询图集合同构 rDAG 森林。这里我们同样定义查询图同构共享查询图（rDAG 森林）的原因与章节3.1提到的相同，只不过查询图同构标记图更适用于多查询静态子图匹配问题的共享执行（如果仅仅按照 RapidFlow 的算法流程，查询图同构标记图也适用多查询动态子图匹配问题的共享执行，这在未来可能是一个有趣且可行的研究方向），这里设计的查询图同构标记图则更适合如 TurboFlux 或者 SymBi 这样的动态子图匹配算法使得它们能在多查询动态子图匹配问题上共享执行。

定义 3.6 (DAG g_r 的根节点)：如果 $v \in V(g_r)$ 没有入边，那么 v 是 g_r 的一个根节



vertex	A_v	A_{vq}	vertex	A_v	A_{vq}
u_1	$\{q_1.u_1, q_2.u_1\}$	$\{q_1,q_2\}$	u_5	$\{q_1.u_5, q_2.u_5\}$	$\{q_1,q_2\}$
u_2	$\{q_1.u_2, q_2.u_2\}$	$\{q_1,q_2\}$	u_6	$\{q_1.u_6, q_2.u_6\}$	$\{q_1,q_2\}$
u_3	$\{q_1.u_3, q_2.u_3\}$	$\{q_1,q_2\}$	u_7	$\{q_1.u_7\}$	$\{q_1\}$
u_4	$\{q_1.u_4, q_2.u_4\}$	$\{q_1,q_2\}$	u_7'	$\{q_2.u_7\}$	$\{q_2\}$
			u_8	$\{q_2.u_8\}$	$\{q_2\}$

图 3-4　rDAG g_r

点。如图3-4中的 u_1 就是一个根节点。

定义 3.7 (rDAG)：rDAG= (V, E, L, v_r) 是一个只有一个根节点的 DAG（rooted Directed Acyclic Graph）。其中 V 是节点集合，$E \subseteq V \times V$ 是有向边集合，$L: V \to \Sigma_V$ 是从节点到点标签的映射函数，$v_r \in V$ 是 V 里面唯一一个入边数量为 0 的点。

定义 3.8 (rDAG 森林)：多个 rDAG 组成的图称为 rDAG 森林。

定义 3.9 (标记 rDAG 森林)：标记 rDAG 森林 $g_r = (V, E, L, V_r, A_v, A_{vq}, A_e)$ 既是定义3.1的标记图，也是定义3.8中的 rDAG 森林。其中 $V_r \subseteq V$ 是 g_r 中不同 rDAG 的根节点。如图3-4是一个标记 rDAG 森林，$V_r = \{u_1\}$, $A_v(u_3) = \{q_1.u_3, q_2.u_3\}$, $A_{vq}(u_3) = \{q_1, q_2\}$，$A_e((u_3, u_8)) = \{q_2\}$。**在本文一般说的 rDAG 森林都是指标记 rDAG 森林。**

定义 3.10 (查询图同构（标记）rDAG 森林)：对于查询图 q，假设标记 rDAG 森林 g_r 中所有标记有 q 的点边（将所有有向边直接变为无向边）构成的子图为 q'，如果 q 与 q' 图同构，那么称 g_r 是 q 的查询图同构 rDAG 森林。如图3-4中的标记图 g_r 既是图3-1中 q_1 的同构（标记）rDAG 森林，也是 q_2 的同构（标记）rDAG 森林。

定义 3.11 (查询图集合同构（标记）rDAG 森林)：如果 $\forall q \in Q$，都有 g_r 是 q 的查询图同构 rDAG 森林，那么 g_r 是查询图集合 Q 的同构 rDAG 森林。如图3-4中的 rDAG 森林 g_r 是图3-1中查询图集合 $Q = \{q_1, q_2\}$ 的同构 rDAG 森林。

定义 3.12 (完美查询图集合同构标记图)：rDAG 森林 g_r 是查询图集合 Q 的完美同构 rDAG 当且仅当：

（1）g_r 是 Q 的同构 rDAG 森林；

（2）g_r 中不存在任何 Q 中查询图以外的其它查询图标记。

如图3-4中的标记图 g_r 因为是图3-1中 $Q = \{q_1, q_2\}$ 的同构 rDAG 森林，且 g_r 没有 q_1 或 q_2 以外的标记，所以 g_r 是 Q 的完美同构 rDAG 森林。

定义 3.13 (DAG g_r 关于节点 u 的前向子图 g_r^u)：从 u 出发能经过的 g_r 所有点与边组成的子图（包括 u 本身）称为 g_r 关于 u 的前向子图。如在图3-4的例子中，u_3 的前向子图是点 $\{u_3, u_8\}$ 和边 $\{(u_3, u_8)\}$ 组成的图。

定义 3.14 (DAG g_r 关于节点 u 的反向子图 g_r^{-u})：从 g_r 的根节点出发可以到达 u

经过的所有点与边组成的子图称为 g_r 关于 u 的反向子图。如在图3-4的例子中，u_3 的反向子图是点 $\{u_1,u_2,u_3\}$ 和边 $\{(u_1,u_2),(u_1,u_3),(u_2,u_3)\}$ 组成的图。

本小章中我们提出的生成的标记 rDAG 森林 g_r 满足以下定义的拓扑性质（我们会在之后的算法描述中补充说明我们的算法如何满足这些拓扑性质）。

定义 3.15 (rDAG 森林的拓扑性质)：

（1）$\forall$ 有向边 $e_i=(u_s,u_d)\in E(g_r)$，$\forall e_j\in E(g_r^{-u_d})$，都有 $A_e(e_i)\subseteq A_e(e_j)$。

（2）$\forall u\in V(g_r)$，$\forall u'\in V(g_r^{-u})$，都有 $A_{vq}(u)\subseteq A_{vq}(u')$ 。

（3）假设 rDAG 森林所有标记 q 的点边组成的 DAG 是 q'，那么 q' 是一个 rDAG。此性质包含两重含义，一是 q' 只有一个根节点，二是 q' 是一个连通图。

在性质（1）中，我们描述了如果两个边 e_i 与 e_j 有拓扑先后顺序（在性质描述中 e_j 拓扑顺序先于 e_i），那么它们的标记之间存在着 $A_e(e_i)\subseteq A_e(e_j)$ 的关系；在性质（2），我们描述了如果两个点在拓扑顺序上 u' 先于 u，那么它们标记的查询图节点中的查询图满足关系 $A_{vq}(u)\subseteq A_{vq}(u')$。这两个性质在后面讲解增量更新与增量搜索会非常重要。具体满足性质的措施将在下面算法中讲解。

为了方便理解，我们会结合图3-5讲述算法3-8如何将图3-5(e)的标记图 g_a 转成图3-5(d)的完美查询图集合同构 rDAG 森林 g_r，最后再补充一些算法的细节。

如算法3-8所示，我们将一个查询图集合同构标记图变成查询图结合同构 rDAG 森林的方法主要分 3 个步骤：

（1）每次在 g_a 找到一个新 rDAG 的根节点 v_r 和对应的查询图集合 Q'；

（2）$\forall q\in Q'$，将所有标记有 q 的 g_a 的点与边插入新 rDAG 中，每次都只插入与已插入点有查询图标记交集的邻边邻点；

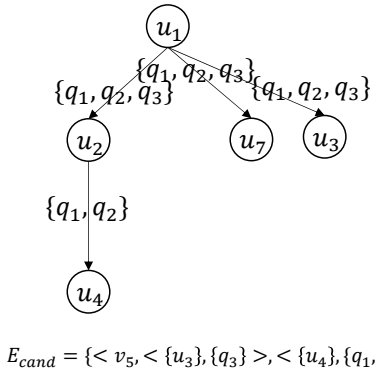
（3）回到第（1）步直到所有 g_a 的边与边标记都被插入 g_r 中。

在算法3-8第二到第四行中，我们先挑选一个标记图的一个节点 v_r 作为 rDAG 的根节点，对应例子图3-5(e)中我选择 $g_a.v_1$，并确定此次在 rDAG 中插入的查询图集合是 $\{q_1,q_2,q_3\}$，即算法第四行的 Q' 是 $\{q_1,q_2,q_3\}$。然后将 v_1 映射到 g_r 的新节点得到图3-5(a)中的 u_1，对应算法第五行。算法第七行到第十行，将 v_1 的邻边都插入 E_{cand} 中，并在 g_a 中删除 v_1 的邻边，防止重复插入。此时 $E_{cand}=\{\langle v_2,\langle\{u_1\},\{q_1,q_2,q_3\}\rangle\rangle\langle v_7,\langle\{u_1\},\{q_1,q_2,q_3\}\rangle\rangle\langle v_3,\langle\{u_1\},\{q_1,q_2,q_3\}\rangle\rangle\}$，意思是对于查询图集合 $\{q_1,q_2,q_3\}$，有一条从 u_1 出发到 v_2 的边、一条从 u_1 出发到 v_7 的边、一条从 u_1 出发到 v_3 的边共三条边待插入到 rDAG。按照算法第十二行到第二十八行逐一插入 E_{cand} 的三条边，也就是标记图的 (v_1,v_2)，(v_1,v_7)，和 (v_1,v_3) 后，因为边 (v_2,v_4) 有更多的查询图标签，它会先于边 (v_3,v_5) 插入，从而在图3-5(a)使得 v_4 对应 u_4、插入边 (u_2,u_4)。自此，插入的点与查询图标记就包括 $\langle v_1,\{q_1,q_2,q_3\}\rangle$，

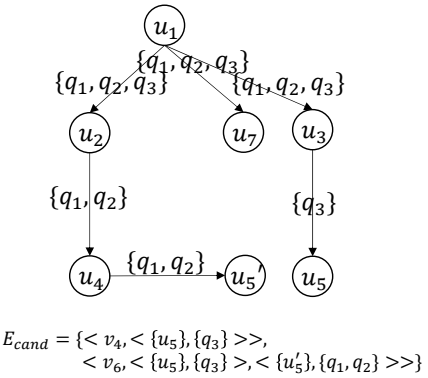
算法 3-8　BuildRDAGForest

Data: 查询图集合同构标记图 g_a
Result: 查询图集合同构 rDAG 森林 g_r

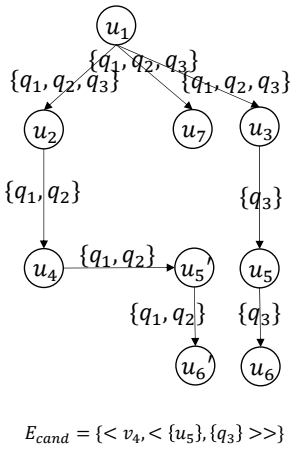
```
1  g_r ← ∅ ;
2  v_r ← FindMaxAnnVertex(g_a) ;
3  while v_r is not NULL do
4      Q' ← {q|q ∈ A_vq(v_n), v_n ∈ N(v_r)} ;          /* 本次的 rDAG 包含 Q' */
5      u_r ← g_r.InsertVertex(v_r, L(v_r), Q') ;        /* 映射 v_r 到 u_r */
6      E_cand ← ∅ ;                                    /* key = g_a.v, value = ⟨g_r.u_s, Q'⟩ */
7      foreach v_n ∈ N(v_r) do
8          E_cand[v_n].add(⟨u_r, A_e((v_r, v_n))⟩) ;
9          A_e((v_r, v_n)) ← ∅ ;
10     end
11     while E_cand ≠ ∅ do
           /* vd_usQsArray 中包含 v_d 在 E_cand 的所有边                    */
           /* vd_usQsArray.v_d，作为有向边汇点的待插入标记图点              */
           /* vd_usQsArray.usQsArray 一个 ⟨{v_s},{q}⟩ 数组                  */
12         vd_usQsArray ← FindMaxSharedAnnVertex(E_cand) ;
13         v_d ← vd_usQsArray.v_d ;
14         E_cand.remove(v_d) ;
15         foreach usQs ∈ vd_usQsArray.usQsArray do
               /* v_d 与 u_d 是一对多关系，但 A_q(u_d) 之间不重合且
                  A_q(v_d) = ∪_{u_d} A_q(u_d)                               */
16             u_d ← g_r.InsertVertex(v_d, L(v_d), usQs.qs) ;
17             foreach u_s ∈ usQs.us do
18                 g_r.InsertEdge(u_s, u_d, usQs.qs) ;
19             end
20             foreach v_n ∈ N(v_d) do
21                 Q'' = A_e((v_d, v_n)) ∩ usQs.qs ;
22                 if Q'' ≠ ∅ then
23                     E_cand[v_n].add(⟨u_d, Q''⟩) ;
24                     A_e((v_d, v_n)) ← A_e((v_d, v_n))/Q'' ;
25                 end
26             end
27         end
28     end
29     v_r ← FindMaxAnnVertex(g_a) ;
30 end
31 return g_r
```



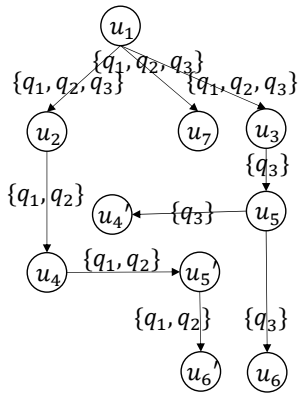
(a) 构建 rDAG 森林例子步骤 1



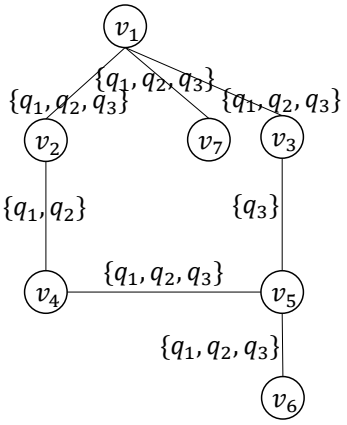
(b) 构建 rDAG 森林例子步骤 2



(c) 构建 rDAG 森林例子步骤 3



(d) 构建 rDAG 森林例子步骤 4（g_r）



(e) 标记图 g_a

图 3-5　构建 rDAG 森林例子

$\langle v_2,\{q_1,q_2,q_3\}\rangle$，$\langle v_3,\{q_1,q_2,q_3\}\rangle$，$\langle v_7,\{q_1,q_2,q_3\}\rangle$ 和 $\langle v_4,\{q_1,q_2\}\rangle$。已经插入的标记图点它们未插入 rDAG 的邻边有 $(v_4,v_5,\{q_1,q_2\})$ 和 $(v_3,v_5,\{q_3\})$，这里我们将每个点与每个边的标记图查询图标出，其中边 (v_4,v_5) 没有标出 q_3 是因为 v_4 当前只插入了 $\{q_1,q_2\}$ 标签，所以从建立 DAG 的角度，为了保证查询图同构 rDAG 森林的拓扑性质（1）和（2），我们对边 (v_4,v_5) 的边标记与 v_4 当前已插入的标记 $\{q_1,q_2\}$ 做了一个交集。

此时 E_{cand} 如图3-5(a)所示，表示对于查询图集合 $\{q_3\}$ 有一条从 u_3 出发到 v_5 的边，对于查询图集合 $\{q_1,q_2\}$ 有一条从 u_4 出发到 v_5 的边，共两条边待插入到 rDAG。因为图3-5(a)中 E_{cand} 只有一个待插入点 v_5，按边集合给 E_{cand} 中的查询图进行分类也只能分成两组 $\langle\{u_3\},\{q_3\}\rangle$ 和 $\langle\{u_4\},\{q_1,q_2\}\rangle$（如果 $E_{cand} = \langle v_5,\langle\{u_3\},\{q_3\}\rangle,\langle\{u_4\},\{q_1,q_2,q_3\}\rangle,\langle\{u_8\},\{q_3\}\rangle\rangle$，因为只有 q_1 和 q_2 的边集合完全相同，所以边会分组分成 $\langle\{u_3,u_4,u_8\},\{q_3\}\rangle$ 和 $\langle\{u_4\},\{q_1,q_2\}\rangle$ 两组），最终 v_5 按查询图拆分有两个 rDAG 节点，假设这两个 rDAG 节点是 u_5 和 u_5'，一个从 u_3 和 u_8

出发到达 u_5，边标记与 u_5 点标记都是 $\{q_3\}$；另一个是从 u_4 出发到达 u_5'，边标记与 u_5' 点标记都是 $\{q_1, q_2\}$，则在算法3-8的第十二行直接返回 $vd\_usQsArray = \langle v_5, \langle\{u_3\}, \{q_3\}\rangle, \langle\{u_4\}, \{q_1, q_2\}\rangle\rangle$。

然后算法第十五行，第一次 $Foreach$ 输入的 $usQs.qs$ 是 $\{q_3\}$，在 g_r 插入点标签是 $L(v_5)$、查询图标记是 $\{q_3\}$、点标记是 v_d 中与 $\{q_3\}$ 有关的点标记，返回的 u_d 是 u_5，也就是 $\langle v_5, \{q_3\}\rangle$ 被映射到 u_5；算法第十七行 u_s 只有 u_3，故在 g_r 建一条标记 $\{q_3\}$ 的有向边 (u_3, u_5)。同理，$\langle v_5, \{q_1, q_2\}\rangle$ 被映射到 u_5'；再次运行到第十五行的 $Foreach$ 循环后，第十七行 u_s 只有 u_4，故在 g_r 建一条标记 $\{q_1, q_2\}$ 的有向边 (u_4, u_5)。此时，图3-5(a)变成图3-5(b)。v_5 的两个边分组（一组属于 $\{q_3\}$，另一组属于 $\{q_1, q_2\}$）在算法第二十行到二十六行，找到它们邻边的标记与两个分组的标记的交集（第二十一行），第一组找到第一条邻边 $\langle v_4, \langle\{u_5\}, q_3\rangle\rangle$，表示从点 u_5（$\langle v_5, \{q_3\}\rangle \rightarrow u_5$）出发的边到达 v_4；第一组找到第二条邻边 $\langle v_6, \langle\{u_5\}, q_3\rangle\rangle$，表示从点 u_5（$\langle v_5, \{q_3\}\rangle \rightarrow u_5$）出发的边到达 v_6。第二组找到邻边 $\langle v_6, \langle\{u_5'\}, q_1, q_2\rangle\rangle$，表示从点 u_5'（$\langle v_5, \{q_1, q_2\}\rangle \rightarrow u_5'$）出发的边到达 v_6。

因为此时 E_{cand} 的边中 v_6 共享最多标记，那么在算法第十二行会返回 $vd\_usQsArray \leftarrow \langle v_6, \langle\{u_5\}, \{q_3\}\rangle, \langle\{u_5'\}, \{q_1, q_2\}\rangle\rangle$，并在算法第十七到第十九行插入 v_6 相关的边，$\langle v_6, \{q_1, q_2\}\rangle$ 将映射到 u_6'，插入边 (u_5', u_6')，$\langle v_6, \{q_3\}\rangle$ 将映射到 u_6，插入边 (u_5, u_6)。又因为 v_6 没有标记有 $q \in Q'$ 的邻边可继续插入 g_r（因为 v_5 在 E_{cand} 插入与 v_6 有关的边时，已经在 g_a 删除 (v_5, v_6) 上删除相关查询图标记，也就是 (v_5, v_6) 的所有标记，从而 (v_5, v_6) 也在 g_a 删除），所以图3-5(b)变成图3-5(c)。

因为 E_{cand} 只剩下边 $\langle v_4, \langle\{u_5\}, \{q_3\}\rangle\rangle$，算法第十二行也将返回这条边。在算法第十六行到第十九行时，得到 $v_4, \langle q_3\rangle$ 的映射点 u_4'。然后用 u_5 作为起点、$\{q_3\}$ 作为标签，在 g_r 中插入有向边 (u_5, u_4')，最终 E_{cand} 为空，完成一个 rDAG 的建立。事实上每个 rDAG 都有属于自己要处理的查询图，不同的 rDAG 之间要处理的查询图 Q' 相互之间没有交集，最终这些处理不同查询图集合的 rDAG 组成 rDAG 森林。

至此，图3-5的例子讲述完毕，我们将进一步详细讲述算法3-8中的实现细节以及设计思路。

总的来说，我们的算法3-8在第一行我们初始化 rDAG 森林 g_r 为空，然后在标记图中找到 v_r 作为新的 rDAG 的根节点，每次经历第三行的 While 循环时都是我们在 rDAG 森林中建立一个新的 rDAG。在 g_a 剩余的节点中，找到一个共享标签分数最大的节点作为 g_r 的根节点。g_a 剩余节点 u 的共享标签分数表示为 $\Sigma_{e_i \in N_E(u)}(|A_e(e_i)| - 1)$，其中 $N_E(u)$ 表示 u 剩余的邻边集合，$|A_e(e_i)|$ 表示边 e_i 的

标记数。比如图3-2中的点 u_3 它有三个邻边，边 (u_1,u_3) 带来的共享标签分数是 2 个标签-1，也就是 1；边 (u_2,u_3) 带来的共享标签分数是 2 个标签-1，也就是 1；边 (u_3,u_8) 带来的共享标签分数是 1 个标签-1，也就是 0；那么 u_3 当前的共享标签分数是三条边分数之和 2。

第四行中，根据 v_r 现有的邻边的标记 Q' 作为当前 rDAG 要处理的查询图集合。第六行到第十行，初始化当前 rDAG 的待插入边 E_{cand} 成 v_r 的邻边，并在标记图 g_a 中删除已经插入 E_{cand} 的边来避免重复处理。如果 g_a 的某条边删除一些标记后已经一个标记都没有那么会将这条边也删除；如果 g_a 的某个点某个标记 q 的邻边都删除后，该点的标记 q 也会删除；如果 g_a 的某个点 v 所有邻边都删除后，v 也会从 g_a 中删除。E_{cand} 中包含了所有已插入点含 $q \in Q'$ 标记但未插入 rDAG 的邻边，第十一行的 While 循环会一直到所有含 $q \in Q'$ 标记的点与边插入完毕才跳出。

因为 E_{cand} 的边都如图3-5按标记图节点进行分组，所以在第十三行中，我们在 E_{cand} 选择一个标记图 g_a 的点 v_d，v_d 是 E_{cand} 中由 FindMaxSharedAnnVertex 函数贪心选择的可能共享最多标记的标记图点，FindMaxSharedAnnVertex 同时还会将 v_d 在 E_{cand} 的边按标记的查询图进行分组，每个查询图对应一组。如假设 E_{cand} 有 $e' = \langle v_8, \langle \{u_8\}, \{q_8, q_9, q_{10}\} \rangle, \langle \{u_9\}, \{q_8, q_9, q_{10}\} \rangle, \langle \{u_8\}, \{q_8\} \rangle \rangle$，那么 FindMaxSharedAnnVertex 会先将 e' 分组成 $\langle v_8, \langle \{q_8\}, \{u_8, u_9, u_{10}\} \rangle, \langle \{q_9\}, \{u_8, u_9\} \rangle, \langle \{q_{10}\}, \{u_8, u_9\} \rangle \rangle$，然后根据每个组拥有的有向边起点集合合并不同的组。在这个例子中，FindMaxSharedAnnVertex 将会合并 $\langle \{q_9\}, \{u_8, u_9\} \rangle$ 和 $\langle \{q_{10}\}, \{u_8, u_9\} \rangle$ 成 $\langle \{q_9, q_{10}\}, \{u_8, u_9\} \rangle$，最后返回 $vd\_usQsArray \leftarrow \langle v_8, \langle \{u_8, u_9, u_{10}\}, \{q_8\} \rangle, \langle \{u_8, u_9\}, \{q_9, q_{10}\} \rangle \rangle$。最终返回的两个分组的共同特点是，同一个分组里面每个查询图都有所有它标记边的起点，且同一个分组里面的所有查询图都有相同边的起点集合，如 q_9 和 q_{10} 都有且只有有向边起点 u_8 和 u_9。每个分组都会将 v_d（这个例子中的 v_8）映射到不同的 rDAG 节点，但是对每个查询图来说这种映射还是单射关系，因为一个查询图只会在一个分组里面。最终这样分组的好处是每个新映射的 rDAG 节点，它们的每个入边都有相同的标记，标记的查询图之间完全共享已经建立的 rDAG 从根节点到当前节点的拓扑结构。把 rDAG 森林当作一个查询图，利用这种共享的拓扑结构，在增量更新或者匹配时，也就能实现查询图之间的计算或则内存共享。FindMaxSharedAnnVertex 函数在 E_{cand} 选择下一个插入点的策略就是返回分组后的共享分数最高的点，一个点的分数具体计算方式就算每个组分数之和，每个组分数的计算方式就是 $|E| \times (|Q''| - 1)$，其中 $|E|$ 是边数，Q'' 是标记的查询图数。

如 v_8 的第一组边有三条边起点为 $\{u_8, u_9, u_{10}\}$，查询图数只有 q_8 一个，故该组分数为 0；第二组有两条边起点为 $\{u_8, u_9\}$，查询图数有 q_9 和 q_{10}，故该组分数为 2；所以 v_8 在 E_{cand} 的共享分数是 2。

FindMaxSharedAnnVertex 函数找到的点以及相应的不同分组，会在算法3-8的第十六行中对每个分组插入一个新的 rDAG 节点并得到 rDAG 插入的新点 u_d。每个新节点的查询图标记就是分组中的标记，然后在第十七到第十九行中，依据当前分组里的每个 rDAG 入点 u_s 和新插入点 u_d 之间建边并给边做查询图标记。在第二十行到第二十六行，找 v_d 的边标记与 u_d 标记有交集的邻点 v_n（因为已经插入 E_{cand} 的边每次都会在标记图 g_a 中删除，所以我们保证了不会重复处理任何边或边标记，并在第二十三行中在 E_{cand} 插入该边，其中 $E_{cand}[v_n]$ 表示 E_{cand} 中 v_n 的边，$\langle u_d, Q'' \rangle$ 表示该 rDAG 边的起点将会是 u_d 且边的查询图标记集合是 Q''。

算法3-8第十一行的 While 循环结束也就意味着标记有 $q \in Q'$ 的点与边都插入至 rDAG 中，我们会在第三十行为新的 rDAG 重新找一个新的根节点 v_r，在第四行重新找到新 rDAG 要处理的查询图集合 Q'。当标记图 g_a 中的所有点都被删除时，FindMaxSharedAnnVertex 会返回 $NULL$，算法第三行的 While 循环将会结束，至此，整个算法3-8描述完毕。

易证算法3-8BuildRDAGForest 处理了所有在 g_a 上的所有查询图，首先我们会证我们算法3-8生成的图是 rDAG 森林，然后证明 rDAG 森林是完美查询图集合同构 rDAG 森林，最后证明它符合所有的 rDAG 森林拓扑性质。

定理 3.3：$\forall q \in Q$，假设算法3-8构建的 g_r 中带有标记 q 的所有点边组成的子图是 q'，那么 q' 与 q 互为图同构。

证明：假设算法3-8的输入完美查询图集合同构标记图为 g_a，那么

（1）$\forall q \in Q$，假设所有标记为 q 的 g_a 的点边组成的子图 q'。因为 g_a 的完美性质以及 q 是连通图，所以 q' 也是连通图；

（2）g_a 的所有点边都只会被放入 E_{cand} 一次；

（3）尽管点边从 E_{cand} 拿出时，算法第十二行的 FindMaxSharedAnnVertex 会将一个标记图点按照查询图标记拆分成多个 rDAG 的点，一条边也会按照图标记拆分成多条 rDAG 的边，但标记图点的查询图标记既没有增加也没有减少，边的查询图标记也没有增加也没有减少，所以从查询图的任意一个点或者一条边的角度来看，它们都只会被从 E_{cand} 拿出一次；

（4）对于单独一个查询图 q 来说，它的连边信息都在 E_{cand} 中保留，且在 g_r 中插入边 $(u_i, u_j) \in E(q)$ 时，不妨设算法将它以有向边 (u_i, u_j) 的方式插入 g_r，算法保留了标记有 $q.u_i$ 的 rDAG 点 u_s，标记有 $q.u_j$ 的 rDAG 点 u_d 的信息，在 g_r 插

入边 (u_s, u_d) 并给边添加标记 q；

所以 q' 是 q 的同构。 ■

定理 3.4：$\forall q \in Q$，假设算法3-8构建的 g_r 中带有标记 q 的所有点边组成的子图是 q'，那么 q' 是一个 rDAG。

证明：对单个查询图 q 来说，算法3-8每次都会在算法的第八行、第二十三行将边在 rDAG 森林的方向定为总是从已插入点到未插入点，且关于 q 的点边都是在算法第二行或者第二十九行选完出发的根节点 v_r 后一次性添加完毕，所以所证成立。 ■

定理 3.5：算法3-8构建的 g_r 是一个 rDAG 森林。

证明：由定理3.4以及算法3-8的第十六行和第十八行可得，边的方向总是从已插入点到新建节点，故 g_r 是一个 rDAG 森林。 ■

定理 3.6：算法3-8生成的（标记）rDAG 森林是查询图集合 Q 的完美同构 rDAG 森林。

证明：由定理3.3可得我们生成的 rDAG 森林满足完美性质的“点标记单个查询图点的单射性”，“标签图的每个边存在的充分必要条件”，和“标签图的每个边的边标记存在的充分必要条件”。

又所有的点边标记都由完美查询图同构标记图 g_a 而来，且无点边有其他查询图标记，故我们算法生成的 rDAG 森林满足完美性质的“点标记查询图点的满射性”。

至此，我们的 rDAG 森林满足所有的完美性质，证毕。 ■

定理 3.7：算法3-8构建的 rDAG 森林 g_r 满足 rDAG 森林的拓扑性质。

证明：由算法3-8第十六行和第十八行可得 g_r 中的点边以及对应的查询图标记都是一次性插入不再更新。又由第二十一行、第二十三行可得，$\forall (u_s, u_t) \in E(g_r)$，$A_{vq}(u_t) = A_e((u_s, u_t)) \subseteq A_{vq}(u_s)$。由数学归纳可得 g_r 满足 rDAG 森林的拓扑性质（1）和（2）。又由定理3.4可得 g_r 满足所有的 rDAG 森林的拓扑性质，证毕。 ■

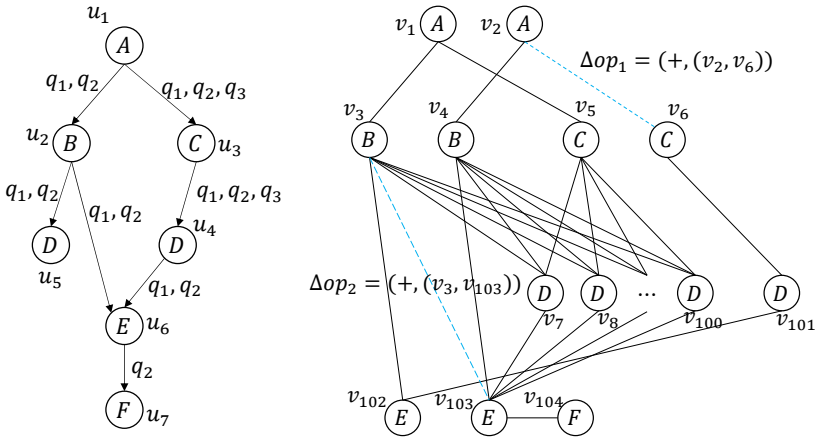
第 4 章　ShareFlow：共享增量状态图与增量匹配

4.1　共享增量状态图

在动态图子图匹配最近的工作中，都会用各自设计的增量辅助数据结构来存储部分匹配结果。每个工作的增量辅助数据结构都不一样，在不同数据集上存储的多余的数据图点边也不一样，也就是剪枝能力不一样，但它们的共同特点都是保证至少存储了有匹配结果的数据图子图。这样每个算法都可以在自己的增量辅助数据结构上搜索增量匹配结果。

如 TurboFlux 的 DCG，先在查询图找一个点作为树根节点，然后去掉非树边，生成一个查询树，DCG 则用于存储查询树的部分匹配结果，数据图有更新时，更新也会反馈到 DCG，然后 TurboFlux 在 DCG 上进行搜索。如 SymBi 的 DCS，将查询图变成一个 DAG，然后按照边的方向来存储部分匹配结果，数据图更新时也会在 DCS 上进行搜索而不是在数据图上。如 RapidFlow，维护一个 GlobalIndex，主要针对查询图节点的邻边信息来存储那些邻边度数符合查询图节点约束的数据图点，在数据图进行更新时，在 GlobalIndex 上从插入边出发按查询图搜索所有相关的点边作为 LocalIndex，最后在 LocalIndex 上搜索匹配结果。如 TRIC，它的问题跟我们一样输入是多个查询图、一个初始的数据图 G_0，更新 G_0 时同时返回所有查询图的增量更新；它将每个查询图贪心拆分成多个 path，然后从 path 第一个节点相同开始合并 path，节点或边不同时开始分叉，最终将所有查询图变成一个森林，但是同一个查询图会被拆分成不同的 path 分布在森林的不同树上；数据图进行更新时，按照 path 森林存储部分匹配同时进行增量更新，在搜索匹配时则对每个查询图，先找到查询图分布在森林的哪些树上，然后对不同树上的匹配结果进行 join 得到最终一个查询图的匹配结果。在本小章中我们介绍如何在我们的 rDAG 森林上建立一个共享增量状态图，Shared Incremental State Graph（SISG），以及在数据图进行更新时，SISG 如何做增量更新。在前面我们已经提到，$\forall q \in Q$，我们的 rDAG 森林上标有 q 的点与边组成的子图与 q 同构，也就是说只要匹配了对应的子图就匹配了 q。

定义 4.1 (共享增量状态图)：共享增量状态图，Shared Incremental State Graph（SISG），是一个图 g_s，点 $V(g_s) \subseteq V(g_r) \times V(G_i)$，边 $E(g_s) \subseteq V(g_s) \times V(g_s)$。假设共享增量状态图的节点表示为 $w_s = \langle u_i, v_j \rangle$，那么 u_i 称为 w_s 的查询图部分，可表示为 $w_s.u = u_i$；v_j 称为 w_s 的数据图部分，可表示为 $w_s.v = v_j$。每个节点都有两个可



(a) rDAG　　(b) $G_0 \oplus \Delta op_1 \oplus \Delta op_2$

图 4-1　rDAG 与 $G_0 \oplus \Delta G$

满足的状态，分别是 $state_1$ 和 $state_2$。$state_1$ 状态是一个布尔值只有 $True$ 或者 $False$ 状态，$state_2$ 状态是一些查询图标记。

我们要求共享增量状态图满足的一些性质:

（1）$\forall w_s = (u_i, v_j) \in V(g_s)$，其中 $u_i \in V(g_r), v_j \in V(G_i)$，都有 $L(u_i) = L(v_j)$。

（2）假设 $e = (w_s, w_d)$，其中 $w_s = \langle u_{si}, v_{si} \rangle$，$w_d = \langle u_{di}, v_{di} \rangle$，那么 $e \in E(g_s)$ 当且仅当 $(u_{si}, u_{di}) \in E(g_r)$ 且 $(v_{si}, v_{di}) \in E(g)$。

定义 4.2 (共享增量状态图节点 $state_1$)：每个共享增量状态节点的 $state_1$ 只能是 $True$ 或者 $False$。

（1）$\forall w_s = (u_i, v_j) \in V(g_s)$，如果 u_i 是 rDAG 森林的一个根节点，那么 w_s 是 $state_1$ 状态；

（2）对于 $w_d = (u_i, v_j) \in V(g_s)$，如果 $\forall u_n = N_{in}(u_i)$，$\exists e = (w_s, w_d) \in E(g_s)$ 满足 $w_s.u = u_n$ 且 $w_s.state_1 = True$，那么 $w_d.state_1 = True$。

定义 4.3 (rDAG 的叶节点)：$\forall u \in rDAG$，$Q_{in} = \cup_{e_i \in N_{inE}(u)} A_e(e_i)$，$Q_{out} = \cup_{e_i \in N_{outE}(u)} A_e(e_i)$，如果 $Q' = Q_{in}/Q_{out} \neq \emptyset$，那么 u 是 rDAG 中相对于 Q' 的叶节点。

对于图4-1(a)中的 rDAG 例子，因为 u_4 的所有入边标记集合是 $\{q_1, q_2, q_3\}$，所有出边标记集合是 $\{q_1, q_2\}$，所以 u_4 是 $\{q_3\}$ 的叶节点；u_5 的所有入边标记集合是 $\{q_1, q_2\}$，所有出边标记集合是 $\emptyset$，所以 u_5 是 $\{q_1, q_2\}$ 的叶节点；u_6 的所有入边标记集合是 $\{q_1, q_2\}$, 所有出边标记集合是 $\{q_2\}$，所以 u_6 是 $\{q_1\}$ 的叶节点；u_7 的所有入边标记集合是 $\{q_1\}$，所有出边标记集合是 $\emptyset$，所以 u_7 是 q_2 的叶节点；其他节点如 u_1，u_2 和 u_3 都因为入边标记集合与出边标记集合一致，故它们都不是任何查询图的叶节点。

定义 4.4 (共享增量状态图节点 $state_2$)：每个共享增量状态图节点 $w_s = (u_i, v_j)$ 它的 $state_2$ 状态都是一些查询图标记，且只能是 u_i 在 rDAG 的标记的子集。比如图4-1(a)的 u_6 标记为 $\{q_1, q_2\}$，那么 $\forall w_s = (u_6, v_j)$，$w_s.state_2 \subseteq \{q_1, q_2\}$。满足以下性质的共享增量状态图节点将有 $state_2$ 标记 q：

（1）$\forall w_s = (u_i, v_j) \in V(g_s)$，如果 w_s 是 rDAG 森林对于 q 的一个叶节点且 $w_s.state_1 = True$，那么 $q \in w_s.state_2$；

（2）对于 $w_s = (u_i, v_j) \in V(g_s)$，如果 $\forall u_n \in \{u_n | q \in A_{vq}(u_n), u_n \in N_{out}(u_i)\}$，都有 $\exists e = (w_s, w_d) \in E(g_s)$ 满足 $w_d.u = u_n$ 且 $q \in w_d.state_2$，那么 $q \in w_s.state_2$。

由定义4.4可得一个共享增量状态图节点的 $state_2$ 状态有多种不同的状态。

定义 4.5 (共享增量状态图节点状态变为 $state_2$)：如果一个共享增量状态图节点 w_s 的 $state_1 = True$，且 $state_2$ 从 $q \notin state_2$ 变成 $q \in state_2$，那么我们称 w_s 的状态变为 $state_2$。特别地，我们称 w_s 的状态增加关于 q 的 $state_2$。

定义 4.6 (共享增量状态图节点状态从 $state_2$ 变为 $state_1$)：如果一个共享增量状态图节点满足 $q \in w_s.state_2$，变成 $q \notin w_s.state_2$，那么我们称 w_s 的状态变为 $state_1$（尽管 w_s 可能还相对一些查询图是 $state_2$ 状态）。特别地，我们称 w_s 的状态减少关于 q 的 $state_2$。

定义 4.7 (共享增量状态图节点 $state_0$)：如果共享增量状态图节点 $w_s.state_1 = False$，我们也称 w_s 的状态是 $state_0$。

4.2　在共享增量状态图进行增量更新

我们将共享增量状态图中的点分成三类，$state_0$ 的点，仅仅 $state_1$ 的点（也就是仅仅 $state_1 = True$ 且 $state_2 = \emptyset$），$state_2$ 的点（也就是 $state_2 \neq \emptyset$ 的点）。为了高效更新我们的共享增量状态图，我们不在共享增量状态图存储任何状态为 $state_0$ 的点的出边。共享增量状态图节点的状态从 $state_0$ 变化到 $state_1$ 或从 $state_1$ 到 $state_2$ 发生在数据图有边插入时；节点的状态从 $state_2$ 变化到 $state_1$ 或从 $state_1$ 到 $state_0$ 发生在数据图有边删除时；数据图插入边时不一定会带来节点状态的变化，也有可能改变多个节点的状态（但肯定都是将节点从 $state_0$ 转成 $state_1$ 或 $state_1$ 转成 $state_2$ 或 $state_0$ 转成 $state_2$）；数据图删除边时不一定会带来节点的变化，也有可能改变多个节点的状态（但一定都是将节点从 $state_1$ 转成 $state_0$ 或 $state_2$ 转成 $state_1$ 或 $state_2$ 转成 $state_0$）。下面我们先给出三种状态的点之间如何转化的规则，然后讲述我们的算法以及对应例子在点发生状态变化时会如何在共享增量状态图传递变化又会在哪里停止状态变化传递。

从 $state_0$ 到 $state_1$。对于状态为 $state_0$ 的节点 $w_d = \langle u_i, v_j \rangle$，如果 u_i 是 rDAG

森林 g_s 的一个根节点，那么 w_d 的状态将转为 $state_1$，也就是 $w_d.state_1 = True$。另外，如果 $\forall u_k \in N_{in}(u_i)$，$\exists w_s$ 满足 $w_s.u = u_k$，$(w_s, w_d) \in E(g_s)$，和 $w_s.state_1 = True$，那么 w_d 的状态也将转为 $state_1$。在实际的实现当中，我们会为每个共享增量状态图节点 $w_d = \langle u_i, v_j \rangle$ 配备大小为 $|N_{in}(u_i)|$ 的数组，数组每个元素初始化为 0，分别记录 w_d 有各种类型的入边有多少个（入边的类型按照 $N_{in}(u_i)$ 的类型进行划分）；如果某个元素从 0 变为 1，我们会检查是否数组所有元素都是非 0，如果是则将该点的状态改为 $state_1$。

从 $state_1$ 到 $state_2$。当共享增量状态图节点 $w_s = \langle u_i, v_j \rangle$ 更新成状态 $state_1$ 时，如果 u_i 是 rDAG 森林对于查询图集合 Q_l 的一个叶节点，那么更新 $w_s.state_2$ 使得 $Q_l \subseteq w_s.state_2$。如果（1）$w_s.state_1 = True$，$q \notin w_s.state_2$；（2）$\forall u_n \in \{u_n | q \in A_{vq}(u_n), u_n \in N_{out}(u_i)\}$，都有 $\exists e = (w_s, w_d) \in E(g_s)$ 满足 $w_d.u = u_n$ 且 $q \in w_d.state_2$；同时满足以上两个条件，那么我们会更新 $w_s.state_2$ 使得 $q \in w_s.state_2$。在实际的实现当中，我们为每个 rDAG 森林的节点 u_i 设置一个查询图叶节点的变量，如果一个共享增量状态节点 $w_s = \langle u_i, v_j \rangle$ 更新成 $state_1$，我们会查询 u_i 的查询图叶节点 Q_l 是否为空，如果不为空则更新 $w_s.state_2 \leftarrow Q_l$；另外我们会为每个 $state_1$ 的共享增量状态图节点配置一个长度为 $\Sigma_{u_n \in N_{out}(u_i)} |A_{vq}(u_n)|$ 的出边数组，数组每个元素初始化为 0，分别记录 w_s 有各种类型的 $state_2$ 出边对于每个查询图有多少个（出边的类型按照 $N_{out}(u_i)$ 的类型进行划分）；如果某个类型的边关于查询图 q 使得某个元素从 0 变成 1，我们检查 u_i 关于 q 的每个出边是否都有 $state_2$ 状态共享状态图边（也就是查询图的出边类型数量是否都不为 0），如果满足则更新 $w_s.state_2 \leftarrow w_s.state_2 \cup \{q\}$。

从 $state_0$ 到 $state_2$。对于所有从 $state_0$ 转换到 $state_1$ 的点，我们都会检查它是否满足 $state_2$ 的条件，如果满足则给该节点的 $state_2$ 状态增加相关的查询图标记。事实上从 $state_0$ 到 $state_2$ 可以由从 $state_0$ 到 $state_1$ 和从 $state_1$ 到 $state_2$ 组合完成。

从 $state_2$ 到 $state_1$。对于共享增量状态图节点 $w_s = \langle u_i, v_j \rangle$，如果 $\exists u_n \in N_{out}(u_i)$ 使得 $\forall w_d = N_{out}(w_s)$ 都有 $w_d.u \neq u_n$，那么节点将按如下更新状态 $w_s.state_2 \leftarrow w_s.state_2 / A_{vq}(u_n)$。如果 $\exists u_n \in N_{out}(u_i)$，$q \in w_s.state_2$ 使得 $\forall w_d = N_{out}(w_s)$，都有“$w_d.u \neq u_n$”或者“$w_d.u = u_n$ 但 $q \notin w_d.state_2$”，则 w_s 更新状态成 $w_s.state_2 \leftarrow w_s.state_2 / \{q\}$。在实际实现当中，在删除 w_s 的 $state_2$ 出边时（假设出点是 $w_d = \langle u_m, v_n \rangle$，$Q'' \in w_d.state_2$）。$\forall q \in Q''$，我们会更新“从 $state_1$ 到 $state_2$”中那个长度为 $\Sigma_{u_n \in N_{out}(u_i)} |A_{vq}(u_n)|$ 的出边数组，更新对应出边位置元素的值使其减 1，如果元素的值变为 0 也就是没有任何一条该类型该查询图

q 的出边，我们会更新 $w_s.state_2 \leftarrow w_s.state_2/\{q\}$。

从 $state_1$ 到 $state_0$。对于共享增量状态图节点 $w_d = \langle u_i, v_j \rangle$，如果 $\exists u_n \in N_{in}(u_i)$ 使得 $\forall w_s = N_{in}(w_d)$ 都有 $w_s.u \neq u_n$，那么 w_d 的状态将会按如下更新：$w_d.state_2 = \emptyset$，$w_d.state_1 = False$。实际实现当中，我们使用“从 $state_0$ 到 $state_1$”中大小为 $|N_{in}(u_i)|$ 的数组，更新删除的入边对应数组元素，如果元素变为 0，则更新状态。

从 $state_2$ 到 $state_0$。同“从 $state_1$ 到 $state_0$”。

算法 4-1　SISGInsert

```
Data: 共享增量状态图 g_s，插入边 (v_1, v_2)
SISGInsertD(g_s, v_1, v_2, L(v_1), L(v_2));
if L(v_2) ≠ L(v_1) then
    SISGInsertD(g_s, v_2, v_1, L(v_2), L(v_1));
end
```

我们的更新算法与 SymBi 的算法相似，但我们的更新要更多考虑如何将单个查询图的增量数据更新变成同时做多个查询图的增量数据更新（也就是我们的共享增量状态图）。同时，区别于 SymBi 每次在搜索时要对一个点的所有邻边进行访问，我们的更新算法会将所有的 $state_2$ 状态的点以及对应的边从共享增量状态图中复制到一个叫状态 2 的图，在后面部分的匹配搜索我们都会实际在状态 2 图中完成，从而避免访问不必要的邻边。如算法4-1所示，对于每一个数据图的更新 Δop_i（在本文中我们仅讨论边插入，因为对于共享增量状态图或状态 2 图的点，大度点的邻边我们都用哈希链表进行存储，一般度数的点我们都用二分平衡树存储它们的邻边，所以边删除就是边插入的简单逆向操作），我们都需要正着或者反着各一次交由共享匹配查询图处理。

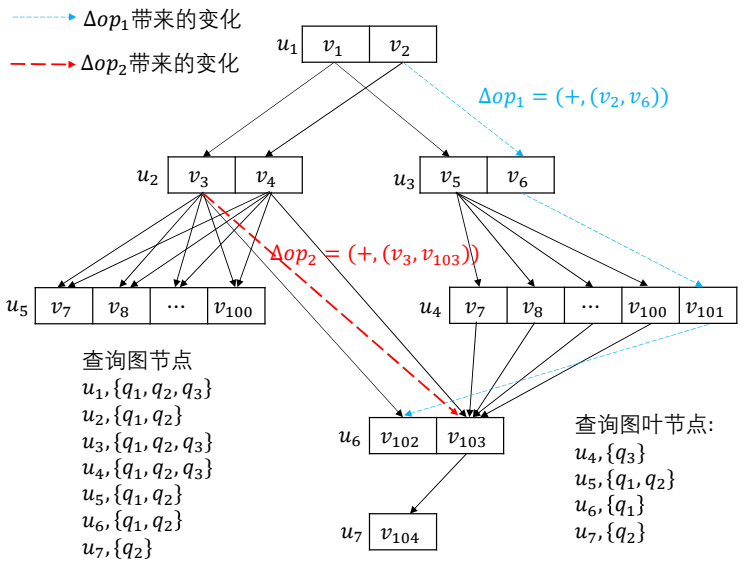
假设我们合并的三个查询图 $\{q_1, q_2, q_3\}$ 成 rDAG 时如图4-1(a)所示，u_1 是该 rDAG 的根节点，它的查询图标记是所有出边标记的并集 $\{q_1, q_2, q_3\}$，它的点标签是 A, 其他点的查询图标记则是所有入边标记的并集（事实上我们在前面有强调过我们生成的 rDAG 有一个特点就是一个点的所有入边查询图标记集合都一样，如有向边 (u_2, u_6) 和 (u_4, u_6) 的查询图标记集合都是 $\{q_1, q_2\}$，如 u_6 的查询图标记集合就是 $\{q_1, q_2\}$）。初始的数据图 G_0 是图4-1(b)中的所有点以及所有黑色实线边；蓝色的虚线是更新流 $\Delta G = \{\Delta op_1 = (+, (v_2, v_6)), \Delta op_2 = (+, (v_3, v_{103}))\}$；$v_1$ 和 v_2 的点标签都是 A；v_3 和 v_4 点标签是 B；v_5, 和 v_6 点标签是 C；$\forall i \in [7, 101], L(v_i) = D$；$v_{102}$ 和 v_{103} 的点标签都是 E；v_{104} 的点标签是 F。

而图4-2则展示了对应图4-1的共享增量状态图 g_s。图的左下角记录了 g_s 对应的 rDAG 的查询图标记，主要用于当节点变成 $state_1$ 状态时，初始化我们前面讲

算法 4-2　SISGInsertD

Data: 共享增量状态图 g_s，插入边源节点 v_s，插入边汇节点 v_d，源节点标签 l_s，汇节点标签 l_d

```
1  E_u ← g_s.g_r.FindMatchQEdges(l_s, l_d) ;
2  foreach e_u ∈ E_u do
3      u_s ← e_u.sourceV ;
4      u_d ← e_u.destV ;
5      w_s ← g_s.GetVertex(u_s, v_s) ;
6      w_d ← g_s.GetVertex(u_d, v_d) ;
7      if w_s.state_1 = True then
8          UpdateState1(w_s, w_d) ;
9      end
10     if w_d.state_2 ≠ ∅ then
11         UpdateState2(w_s, w_d, w_d.state_2) ;
12     end
13     while queue_1 ≠ ∅ do
14         w_s ← queue_1.pop() ;
15         U_d ← {u_d | u_d ∈ N_out(w_s.u)} ;
16         foreach u_d ∈ U_d do
17             V_d ← N_out(w_s, L(u_d)) ;   /* L(u_d) type neighbor edges */
18             forall w_d ∈ V_d do
19                 UpdateState1(w_s, w_d) ;
20             end
21         end
22     end
23     while queue_2 ≠ ∅ do
24         ⟨w_d, Q'⟩ ← queue_2.pop() ;
25         U_s ← {u_s | u_s ∈ N_in(w_d.u)} ;
26         foreach u_s ∈ U_s do
27             V_s ← N_in,state_1(w_d, u_s) ; /* u_s state_1 neighbor vertices */
28             forall w_s ∈ V_s do
29                 UpdateState2(w_s, w_d, Q') ;
30             end
31         end
32     end
33 end
```



vertex	$state_1$	$state_2$
$\langle u_1, v_1 \rangle$	True	q_1, q_2, q_3
$\langle u_1, v_2 \rangle$	True	q_1, q_3
$\langle u_2, v_3 \rangle$	True	q_1, q_2
$\langle u_2, v_4 \rangle$	True	q_1, q_2
$\langle u_3, v_5 \rangle$	True	q_1, q_2, q_3
$\langle u_3, v_6 \rangle$	*True*	q_1, q_3
$\langle u_4, v_7 \rangle$	True	q_1, q_2, q_3
⋯	True	q_1, q_2, q_3
$\langle u_4, v_{100} \rangle$	True	q_1, q_2, q_3
$\langle u_4, v_{101} \rangle$	*True*	q_1, q_3
$\langle u_5, v_7 \rangle$	True	q_1, q_2
⋯	True	q_1, q_2
$\langle u_5, v_{100} \rangle$	True	q_1, q_2
$\langle u_6, v_{102} \rangle$	*True*	q_1
$\langle u_6, v_{103} \rangle$	True	q_1, q_2
$\langle u_7, v_{104} \rangle$	True	q_2

图 4-2　共享增量状态图 g_s

到节点状态转化时用到的一些关于 $state_2$ 的出边数组。图中间的每个框实际上代表共享增量状态查询图的一个点，如最上面的 v_1 与它所在表格的前面的 u_1 组成节点 $w_{1,1} = \langle u_1, v_1 \rangle$。图上的所有点与所有黑色实线组成了 G_0 的共享增量状态图，细蓝虚线是 Δop_1 给共享增量状态图带来的边的变化；红色的则是 Δop_2 给共享增量状态图带来的边的变化。最右边表格则是记录了共享增量状态图每个节点的状态，其中黑色部分都是 G_0 对应的状态，蓝色加粗斜体部分都是 Δop_1 带来变化的部分，红色部分都是 Δop_2 带来变化的部分。如 $\langle u_5, v_{100} \rangle$ 一行的表格数据表示在数据图还是 G_0 时（在 g_1 或者 g_2 时也是如此，因为 Δop_1 或者 Δop_2 没有给它带来任何的变化），它的 $state_1 = True$，它的 $state_2 = \{q_1, q_2\}$。如 $\langle u_6, v_{102} \rangle$ 一行的表格数据是蓝色的代表是 Δop_1 给共享增量状态图带来的变化，从 $state_1 = False$ 变成 $state_1 = True$，从 $state_2 = \emptyset$ 变成 $state_2 = \{q_1\}$。中间下面还记录了 rDAG 每个节点分别是哪些查询图的叶节点（如果某个节点不是任何查询图的叶节点那么就省略），如 u_4 是 q_3 的叶节点，u_5 是 q_1 和 q_2 的叶节点。

在数据图 G_0 更新操作 Δop_1 时，如图4-1(b)，G_0 会插入边 (v_2, v_6)，然后根据算法4-1在 rDAG 森林 g_r 中找有向边 (v_2, v_6) 和 (v_6, v_2) 会对应 g_r 中哪些边来更新共享增量状态图 g_s（对应算法4-2第一行）。进入算法4-2后，由第一行得到插入边对应 g_r 中可能要更新的边位置集合 E_u，如果是插入有向边 (v_2, v_6)，这里 E_u 会返回 $\{(u_1, u_3)\}$（只有一条边），$e_u = (u_1, u_3)$，w_s 就是图4-2中的点 $\langle u_1, v_2 \rangle$，w_d 就是 $\langle u_3, v_6 \rangle$。

运行至算法4-2第七行时，判断条件因为 u_1 是 rDAG 的根节点所以为真，在

算法 4-3　UpdateState1

Data: 共享增量状态图某个边的源节点 w_s，汇节点 w_d

```
if w_d.s1InEdgeNum[w_s.u] = 0 then
    w_d.s1InEdgeType ← w_d.s1InEdgeType + 1;
    if w_d.s1InEdgeType = |N_in(w_d.u)| then
        w_d.state_1 ← True ;
        queue_1.push(w_d) ;
        if g_r.isLeaf(w_d.u) then
            w_d.state_2 ← g_r.getLeafQ(w_d.u) ;
            queue_2.push(⟨w_d, g_r.getLeafQ(w_d.u)⟩) ;
        end
    end
end
w_d.s1InEdgeNum[w_s.u] ← w_d.s1InEdgeNum[w_s.u] + 1 ;
```

算法 4-4　UpdateState2

Data: 共享增量状态图某个边的源节点 w_s，汇节点 w_d，要向 w_s 更新的 $state_2$ 状态相关的查询图集合 Q'

```
Q'' ← ∅ ;
foreach q ∈ Q' do
    if w_s.s2OutEdgeNum[q, w_d.u] = 0 then
        w_s.s2OutEdgeType[q] ← w_s.s2OutEdgeType[q] + 1;
        if w_s.s2outEdgeType[q] = |N_out,q(w_s.u)| then
            w_s.state_2 ← w_s.state_2 ∪ {q} ;
            Q'' ← Q'' ∪ {q} ;
        end
    end
    w_s.s2OutEdgeNum[q, w_d.u] ← w_s.s2OutEdgeNum[q, w_d.u] + 1 ;
end
if Q'' ≠ ∅ then
    queue_2.push(⟨w_s, Q''⟩);
end
```

UpdateState1 函数更新 w_d 的 $state_1$ 状态。进入算法4-3，首先算法第一行判读 w_d 是否已经有 $w_s.u$ 类型的 $state_1$ 邻点，如果没有则在第二行给 w_d 的 $state_1$ 的入边类型加一；如果 w_d 的 $state_1$ 的入边类型数量满足 $w_d.u$ 在 g_r（或者 rDAG）的入边数量，表示 w_d 刚刚有所有 $w_d.u$ 的所有入边类型；此时可以直接设置 $w_d.state_1 = True$，即图4-2中表格 $\langle u_3, v_6\rangle$ 的 $state_1$ 列变为 True，并在队列 $queue_1$ 中加入要往出边方向更新邻点状态的点 w_d；同时，判断 $w_d.u$ 是否是 rDAG 森林 g_r 中某些查询图的叶节点，但从图4-2看，$w_d.u = u_3$ 并不是任何查询图的叶节点，于是运行至第十二行给 w_d 的 $state_1$ 的 $w_s.u$ 类型的边数加一。

运行至算法4-2第十行时，此时 $w_d = \langle u_3, v_6\rangle$ 还未是 $state_2$ 状态故跳过。

但在第十三行时 $queue_1$ 中放有 $\langle u_3, v_6\rangle$ 表示应该去更新 $\langle u_3, v_6\rangle$ 的出点的状态，让 $\langle u_3, v_6\rangle$ 的每个出点更新多一个 u_3 类型的入点。此时第十四行 w_s 更新成 $\langle u_3, v_6\rangle$，U_d 是 $\{u_6\}$，V_d 是 $\{\langle u_4, v_{101}\rangle\}$，在第十九行进入函数 UpdateState1 更新 $\langle u_4, v_{101}\rangle$ 的入点信息；$\langle u_4, v_{101}\rangle$ 在算法4-3会更新状态 $\langle u_4, v_{101}\rangle.state_1 \leftarrow True$，将自己放入 $queue_1$；并且因为 u_4 是 q_3 的叶节点，会更新状态 $\langle u_4, v_{101}\rangle.state_2 \leftarrow \{q_3\}$，也就是表格中 $\langle u_4, v_{101}\rangle$ 那一行里面蓝色的 q_3，并在 $queue_2$ 中放入 $\langle\langle u_4, v_{101}\rangle, \{q_3\}\rangle$。

之后继续第十三行的循环，w_s 更新成 $\langle u_4, v_{101}\rangle$，$U_d$ 是 $\{u_6\}$，V_d 是 $\{\langle u_6, v_{102}\rangle\}$。因为 $\langle u_6, v_{102}\rangle$ 有 $state_1$ 状态 u_2 类型的入点 $\langle u_2, v_3\rangle$ 和 u_4 类型的入点 $\langle u_4, v_{101}\rangle$，故在表格中 $\langle u_6, v_{102}\rangle$ 的 $state_1$ 也将更新为 $True$，$queue_1$ 将加入 $\langle u_6, v_{102}\rangle$。又因为 u_6 是 q_1 的叶节点，所以表格中 $\langle u_6, v_{102}\rangle$ 的 $state_2$ 加入 q_1，$queue_2$ 中也会加入 $\langle u_6, v_{102}\rangle$。

继续回到第十三行循环，w_s 更新成 $\langle u_6, v_{102}\rangle$，$U_d$ 是 $\{u_7\}$，V_d 为空集，又因为 $queue_1$ 已经为空，故跳出此循环。自此，按照边的方向更新 $state_1$ 状态结束，我们依次更新了 $\langle u_3, v_6\rangle$，$\langle u_4, v_{101}\rangle$，和 $\langle u_6, v_{102}\rangle$ 的 $state_1$ 状态。

运行至第二十三行，$queue_2$ 中有 $\langle\langle u_4, v_{101}\rangle, \{q_3\}\rangle$ 和 $\langle\langle u_6, v_{102}\rangle, \{q_1\}\rangle$。先对 $\langle\langle u_4, v_{101}\rangle, \{q_3\}\rangle$ 进行处理，U_s 是 $\{u_3\}$，V_s 是 $\{\langle u_3, v_6\rangle\}$。进入算法4-4后，输入的 w_d 即 $\langle u_4, v_{101}\rangle$，$w_s$ 是 $\langle u_3, v_6\rangle$，$Q' = \{q_3\}$。第三行发现 w_s 之前没有关于 q_3 是 $state_2$ 的 u_4 类型的邻点，于是给 w_s 关于 q_3 的 $state_2$ 的邻点类型数目加一，然后检查 w_s 关于 q_3 的 $state_2$ 出边类型都有（因为 $w_s.u$ 的出边只有 u_4，而 w_s 有一个满足 q_3 的 $state_2$ 状态的出点 $\langle u_4, v_{101}\rangle$），于是给 w_s 的 $state_2$ 状态增加 q_3，并且在 $queue_2$ 中放入 $\{w_s, \{q_3\}\}$ 在之后继续更新 w_s 的入边状态。

继续回到算法4-2第二十三行的循环，$queue_2$ 中有 $\langle\langle u_6, v_{102}\rangle, \{q_1\}\rangle$ 和 $\langle\langle u_3, v_6\rangle, \{q_3\}\rangle$。先处理 $\langle\langle u_6, v_{102}\rangle, \{q_1\}\rangle$，这次会更新 $\langle u_4, v_{101}\rangle$，使得

$\langle u_4, v_{101}\rangle.state_2$ 从 $\{q_3\}$ 变成图4-2中表里的 $\{q_1, q_3\}$（不更新 $\langle u_2, v_3\rangle$ 是因为 $q_1 \in \langle u_2, v_3\rangle.state_2$），同时将 $\langle\langle u_4, v_{101}\rangle, \{q_1\}\rangle$ 放入 $queue_2$ 中。

继续回到第二十三的循环，之后依次更新 $\langle u_1, v_2\rangle$ 增加 q_3 的 $state_2$ 状态，更新 $\langle u_3, v_6\rangle$ 增加 q_1 的 $state_2$ 状态，更新 $\langle u_1, v_2\rangle$ 增加 q_1 的 $state_2$ 状态。自此关于 Δop_1 的共享增量状态图更新完毕。

更新 Δop_2 虽然不会给共享增量状态图带来任何新增的边（共享增量状态图的边与节点的 $state_1$ 状态有关，状态二图的边与节点更新 $state_2$ 才有关），但如图4-2所示会给 $\langle u_2, v_3\rangle$ 的 $state_2$ 增加 q_2，给 $\langle u_1, v_1\rangle$ 的 $state_2$ 增加 q_2。

4.3　在共享增量状态图进行增量匹配搜索

定义 4.8 (从查询图到共享增量状态图的匹配)：一个从查询图节点 $V(q)$ 到共享增量状态图节点 $V(g_s)$ 的单射映射函数 f 是一个匹配当且仅当

（1）$\forall u \in V(q), L(f(u)) = L(u)$；

（2）$\forall (u, u') \in E(q), (f(u), f(u')) \in E(g_s)$；

（3）$\forall u_i \in V(q), f(u_i).u = u_i$；

（4）$\forall u_i \in V(q), u_j \in V(q), u_i \neq u_j$，都有 $f(u_i).u \neq f(u_j).u$。

定理 4.1：给定查询图 q，数据图 g，共享增量状态图 g_s，从 $V(q)$ 到 $V(g)$ 的匹配集合 M，从 $V(q)$ 到 $V(g_s)$ 的匹配集合 M'，满足 $\forall u \in V(q)$，$\forall f \in M$，都有 $\exists f' \in M'$，$f(u) = f'(u).v$。

证明：我们将 f' 定义成 $\{u \to w | w = \langle u, f(u)\rangle, u \in V(q)\}$。又 $\forall u \in V(q), \langle u, f(u)\rangle \in V(g_s)$；且 $\forall (u_i, u_j) \in E(q)$，$(\langle u_i, f(u_i)\rangle, \langle u_j, f(u_j)\rangle) \in E(g_s)$；则 f' 满足 $f(u) = f'(u).v$ 的一个从 $V(q)$ 到 $V(g_s)$ 的子图同构映射函数，证毕。 ■

定理4.1证明了我们不需要在数据图上 G_i 进行匹配，而是可以直接在剪枝好（比如去掉 G_i 中 $state_0$ 或者仅仅只是 $state_1$ 的邻点）的共享增量状态图 g_s 上进行匹配搜索。接下来我们将讲述如何在给定完美查询图集合同构 rDAG 森林 g_r，共享增量增量状态图 g_s，以及插入边的情况下，决定增量匹配顺序以及如何利用我们后面设计的匹配顺序树来共享执行匹配过程。

4.3.1　决定匹配顺序

在图4-1(b)的数据图 G 中插入边 (v_2, v_6) 后，我们在上一小章33提到对 (v_2, v_6) 在共享增量查询图 g_s 中更新各个点的状态，更新后如图4-2所示。因为点 $\langle u_1, v_2\rangle$ 的 $state_2$ 状态为 $\{q_1, q_3\}$，而点 $\langle u_3, v_6\rangle$ 的 $state_2$ 状态也为 $\{q_1, q_3\}$，故要从部分匹配 $f' = \{u_1 \to v_2, u_3 \to v_6\}$ 开始，尝试 q_1 与 q_3 的部分匹配。在接下来我们将配合

图4-3中的例子讲解我们如何找到一个匹配顺序，一次性同时匹配多个查询图。

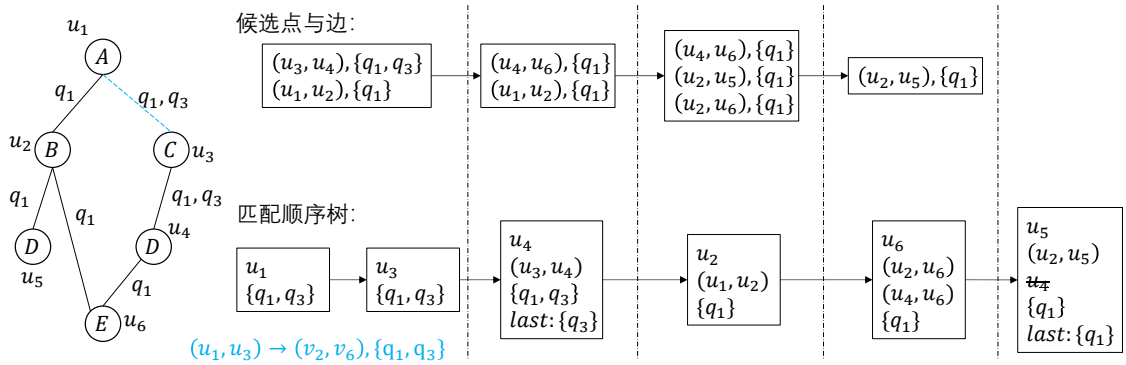


图 4-3　对 $\Delta op_1=(v_2,v_{101})$ 的匹配顺序森林

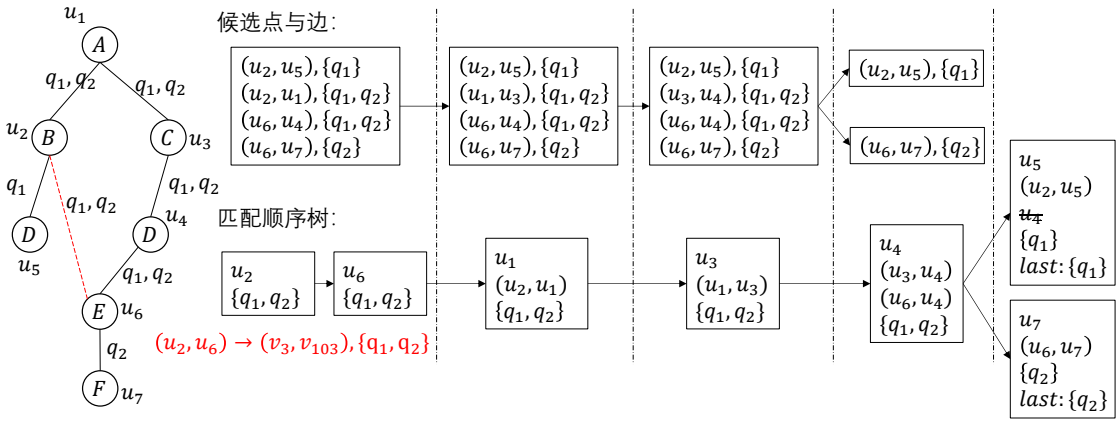


图 4-4　对 $\Delta op_2=(v_3,v_{103})$ 的匹配顺序森林

图4-3中的标记图是从图4-1(a)中的找到所有含有 q_1 或 q_3 点边组成的子图，蓝色的边 (u_1,u_3) 正是 Δop_1 要被映射的标记图边，剩下的点边都是将要匹配的，我们将为这个标记图上的查询图 q_1 和 q_3 生成一个匹配顺序树。

其中在“匹配顺序树”中每一个框的节点表示当前要匹配的标记图点。查询图集合表示成功或失败匹配这个点会成功或失败部分匹配哪些查询图。如果某一个框跟着不为空 $last$ 查询图集合，这个 $last$ 查询图集合表示匹配完当前点后就完成了哪些查询图的匹配。如匹配顺序树中的第一个框，表示第一个匹配点是 u_1，匹配它代表完成了 q_1 和 q_3 的部分匹配。第一个框跟第二个框是在第一条竖直虚线之前的表示在实际匹配前，第一个和第二个匹配点已经匹配好了，在这里对应着插入边的两个映射 $u_1\rightarrow v_2$ 和 $u_3\rightarrow v_6$。

要找到真正第一个待匹配点，首先在标记图中找到 u_1 与 u_3 的邻边，也就是“候选点与边”中第一个框有 u_3 的标记有 $\{q_1,q_3\}$ 邻边 (u_3,u_4)，有 u_1 的标记有 $\{q_3\}$ 的邻边 (u_1,u_2)，这两个邻边去掉已经匹配的点后预示这下一个匹配点可以选择 u_4 或者 u_2。我们选择下一个匹配点遵循下面几点原则：

（1）优先选图标记更多的点；

（2）上面条件一样选择边数更多的点，利用 join 来减少候选点数量；

（3）上面条件一样选择预估候选点数量更少的点。比如对于 u_2，预估候选点数量的方法是用 u_2 类型 $state_2$ 的点数量来预估。

（4）上面条件一样则随机选择一个。

匹配顺序树第三个框表示依据上述规则我们选择的第三个匹配点是 u_4，(u_3,u_4) 表示 u_4 的匹配点必须与 u_3 匹配的点有边（事实上筛选 u_4 的候选点的方式就是在 u_3 匹配点的邻点中找 $state_2$ 包含 q_1 或者 q_3 的），匹配它就是在同时匹配 q_1 和 q_3。同时因为 $last$ 有查询图 q_3，表示如果按照匹配顺序完成前面的点的匹配（包括 u_4），可以将当前的所有映射作为查询图 q_3 的一个匹配结果；如假设 u_4 按照图4-2映射到 v_{101}（即在共享增量状态图 g_s 中的访问顺序依次是 $\langle u_1,v_2\rangle$，$\langle u_3,v_5\rangle$，$\langle u_4,v_{101}\rangle$），那么将在 q_3 关于 $\Delta op_1=(+,(v_2,v_6))$ 的增量匹配结果中增加匹配 $f=\{u_1\to v_2,u_3\to v_6,u_4\to v_{101}\}$。

“候选点与边”中第二个框表示选定 u_4 作为第三个匹配点后，删除“$(u_3,u_4),\{q_1,q_3\}$”，增添 u_4 未进入过“候选点与边”的邻边“$(u_4,u_6),\{q_1\}$”。在 u_2 和 u_6 之间，我们假设随机选择 u_2 作为下一个匹配点。

“匹配顺序树”第四个框表示第四个匹配点 u_2，u_2 的候选点需要是 u_1 匹配点的关于 q_1 的 $state_2$ 邻点。“候选点与边”在选择 u_2 作为第四个匹配点后，第三个框展示如果先匹配 u_6，那么它有两条边可以 join（分别是 (u_4,u_6) 和 (u_2,u_6)），而 u_5 只有一条，故我们选 u_6 作为下一个匹配点。“匹配顺序树”第五个框表示第五个匹配点是 u_6，u_6 的候选点需要是 q_1 的 $state_2$ 状态，与 u_2 已匹配的数据图节点有连边且与 u_4 已匹配节点有连边。最后一个框表示匹配顺序树最后一个匹配点 u_5，需要与 u_2 的已匹配点有连边和本身是 q_1 的 $state_2$ 状态，u_4 被划掉意思是需要对 u_5 的候选点与 u_4 的匹配点做单射检查来满足子图同构条件（如果是子图同态，则不需要检查，最后的结果就是子图同态的），$last$ 集合表明 u_5 这个点匹配完后当前已经有的映射就是 q_1 的一个匹配，可以直接放入 q_1 关于 Δop_1 的增量匹配结果中。

在这里我们已经讲完针对 Δop_1 我们如何生成它的匹配顺序树。而针对 Δop_2，不同的地方在于“候选点与边”第四步被分成了两个框（分支），这是因为这一步的所有候选边中不存在任何一个边的标记查询图集合是其他所有边的标记查询图集合的超集（superset）。我们将这一步的所有边拆分成多个分支（在这个例子中是两个分支），使得树分支满足以下性质:

（1）每个分支里的边，至少存在一个边的查询图标记集合是分支里所有边的

查询图标记集合的超集；

（2）不同的分支不会出现同一个查询图标记。

在“候选点与边”进行树分叉后，后续的“匹配顺序树”也会跟着进行相应的树分叉，最终形成图4-4中最后一条竖直虚线后的两个树分叉。第一个分叉表示在匹配完 u_4 后，如果通过边 (u_2,u_5) 找到关于 $\{q_1\}$ 的 $state_2$ 状态的 u_5 的候选点（并且与 u_4 的匹配点进行单射检查后），那么当前的映射集合可以作为 $\{q_1\}$ 的匹配结果（因为 $last$ 是 $\{q_1\}$）。第二个分叉表示在匹配完 u_4 后，如果通过边 (u_6,u_7) 找到关于 $\{q_2\}$ 的 $state_2$ 状态的 u_7 候选点，那么当前的映射集合可以作为 $\{q_2\}$ 的匹配结果。

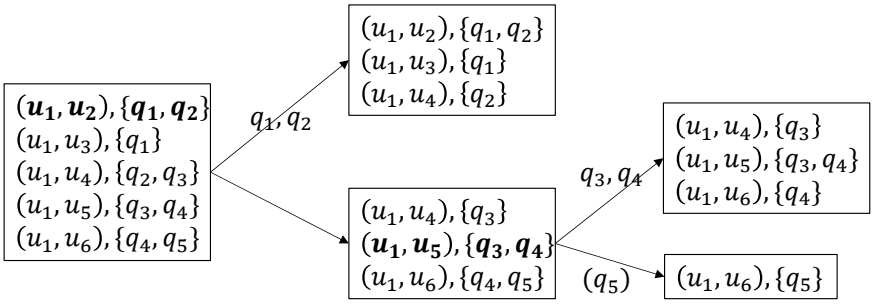


图 4-5　对候选点与边进行树分叉的过程例子

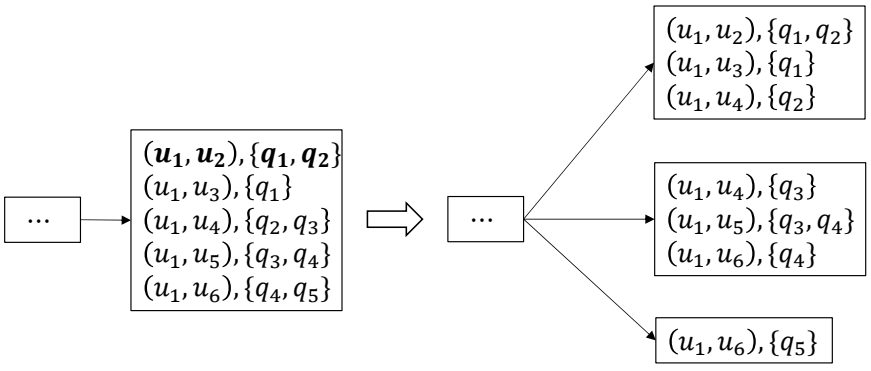


图 4-6　对候选点与边进行树分叉的前后

我们通过图4-5中的例子来说明我们的算法如何拆分时使树分支满足以上性质。假设待匹配边就是最左边框里的五条边，因为里面没有任何一条边的查询图标记集合是其他边的查询图标记集合的超集，所以要对它进行树分叉。由于找不到任何一条边有最大的查询图标记数，我们选取查询图标记数为 2 的一条边 (u_1,u_2) 的查询图标记 $\{q_1,q_2\}$ 作为初次拆分的依据，第一个树分支将分走所有包含 q_1 或 q_2 的边。特别地，因为边 (u_1,u_4) 有两个标记，q_2 属于 $\{q_1,q_2\}$，而 q_3 不属于 $\{q_1,q_2\}$，

这里我们会先将“$(u_1,u_4),\{q_2,q_3\}$”拆分成“$(u_1,u_4),\{q_2\}$”和“$(u_1,u_4),\{q_3\}$”，第一条边归属树分支 $\{q_1,q_2\}$，第二条边进入剩余部分等待继续树分叉。第一次树分叉过后，剩余三条边 (u_1,u_4)，(u_1,u_5)，和 (u_1,u_6)，因为剩余部分也不满足树分支的性质，我们将对剩余部分继续进行树分叉。假设我们选取 $\{q_3,q_4\}$ 作为第二个树分叉的依据，拆分的两个部分如图所示，第一部分边的查询图集合超集都是边 (u_1,u_5) 的 $\{q_3,q_4\}$，第二部分查询图集合超集都是边 (u_1,u_6) 的 $\{q_5\}$，在这里所有部分都满足性质，所以两个部分都成为树的一个分支，最后如图4-6所示会变成三个候选点与边的树分支，之后也会使匹配顺序树产生三个树分支。

4.3.2　匹配算法

在针对某一个图更新 Δop_i 更新完共享增量状态图 g_s 后，我们需要在共享增量状态图中搜索增量匹配。如算法4-5所示，因为我们在算法4-6的 FindMatchQEdges 函数更新状态或者搜索返回的 rDAG 森林对应匹配边都是有向的，所以我们会将 v_1 与 v_2 互换一次共两次调用算法4-6。在算法4-6中，我们先调用 FindMatchQEdges 函数找到所有匹配 (l_1,l_2) 的 rDAG 森林的可能匹配有向边集合 E_u。

算法 4-5　SISGFindMatch

```
Data: 共享增量状态图 g_s，插入边 (v_1, v_2)
1 SISGFindMatchD(g_s, v_1, v_2, L(v_1), L(v_2)) ;
2 if L(v_1) ≠ L(v_2) then
3 |   SISGFindMatchD(g_s, v_2, v_1, L(v_2), L(v_1)) ;
4 end
```

图4-3中 Δop_1 对应算法4-5的 E_u 是 $\{(u_1,u_3)\}$。对于每个 rDAG 森林匹配的边 e_u，找到插入边对应在共享增量状态图的两个点 w_s 和 w_d，对应图4-2中的 $\langle u_1,v_2\rangle$ 和 $\langle u_3,v_6\rangle$，它们的 $state_2$ 交集是 $\{q_1,q_3\}$。匹配顺序树将按图4-3生成，mo 是图中匹配顺序树的第三个框，表示第一个正式的匹配点从 u_4 开始。算法4-7将针对 (u_s,u_d) 和 (v_s,v_d) 对查询图 q_1 和 q_3 同时进行搜索。当 $mo.u = u_4$ 时，第一行的 candidate 函数先查看 $mo.E$，得知需要从 u_3 的匹配点 v_6 的邻点中寻找，通过图4-2的点 $\langle u_3,v_6\rangle$ 找到 $V_{cand} = \langle u_4,v_{101}\rangle$。因为 u_4 是查询图 q_3 的最终匹配点，故算法第六行到第十四行先将 u_4 映射到 v_{101}，然后在第十行将当前的映射 f 放入查询图 q_3 的增量匹配结果集合。在第十六行中，我们将 u_4 映射到 v_{101} 并在第十八行选择匹配顺序树节点 mo 的一个子节点（比如图4-3匹配顺序树 u_4 节点的一个子节点就是 u_2）作为回溯算法4-7的输入。之后将 $u_2 \to v_4$ 放入 f 中。但当匹配到 u_6 时，由图4-3匹配顺序树的第五个框我们知道 u_6 的候选点必须是 u_2 的匹配点 v_4 的邻点且必须是 u_4 的匹配点 v_{101} 的邻点，因为事实上不存在这样的 u_6 候选点，所

算法 4-6　SISGFindMatchD

```
Data: 共享增量状态图 g_s，插入边源节点 v_s，插入边汇节点 v_d，源节点标签 l_s，
      汇节点标签 l_d
1  E_u ← g_s.g_r.FindMatchQEdges(l_1, l_2) ;
2  foreach e_u ∈ E_u do
3      u_s ← e_u.sourceV ;
4      u_d ← e_u.destV ;
5      w_s ← g_s.GetVertex(u_s, v_s) ;
6      w_d ← g_s.GetVertex(u_d, v_d) ;
7      Q' ← w_s.state_2 ∩ w_d.state_2 ;          /* Q' 是待匹配的查询图集合 */
8      if Q' = ∅ then
9          continue;
10     end
11     mo ← GetMatchingOrder(g_s.g_r, u_s, u_d, Q') ;
12     f = {u_s → v_s, u_d → v_d} ;
13     MatchMultiQueries(g_s, f, mo) ;          /* mo 是树深的第三个节点 */
14 end
```

以到这里匹配失败。在回退时，u_2 没有 v_4 以外的候选点也将回退；u_4 没有 v_{101} 以外的候选点继续回退；此时回退到算法4-6进入第二行，因为 E_u 只有一条边所以继续回退到算法4-5第二行，进入第三行时因为对应算法4-6的 E_u 为空集则继续回退；最终我们发现 Δop_1 只给 q_3 带来了一个增量匹配 $\{u_1 \to v_2, u_3 \to v_6, u_4 \to v_{101}\}$。

图4-4中的按匹配顺序树的顺序进行匹配时，u_1 将映射到 v_1，u_3 将映射到 v_5，u_4 将按顺序映射到 $v_i \in \{v_i | i \in [7, 100]\}$。然后运行到算法4-7的第十七行，$mo.u = u_4$，$mo.children$ 是图中“匹配顺序树”的 u_4 到 u_5 的 u_5 树分支和 u_4 到 u_7 的 u_7 树分支。在进入 u_5 分支时，如果 u_4 的匹配点是 v_i，那么 u_5 可以按顺序映射到 $v \in \{v_j | j \neq i, j \in [7, 100]\}$；在 u_5 一共会得到 $8742 = 94 \times 93$ 个 q_1 的匹配。退出 u_5 分支后，进入 u_7 分支，因为 u_7 只有候选点 v_{104}，故在 u_7 一共会得到 94 个 q_2 的匹配。

4.4　匹配剪枝：匹配状态回退

图4-7将三个查询图合并成一个 rDAG 森林 g_r；g_s 是 g_r 对于某个数据图的共享增量状态图，其中更新边（插入边）是 (v_{201}, v_{202})，里面的所有点都是对应 rDAG 森林点的所有查询图标记的 $state_2$ 状态，比如 $\langle u_4, v_1 \rangle.state_2 = \{q_1, q_2\}$。

在插入边 (v_{201}, v_{202}) 后，生成图4-8的匹配顺序树，插入边两个点，v_{202} 被 u_0 匹配，v_{201} 被 u_1 匹配，实际匹配从 u_2 开始。匹配顺序 DAG 是根据匹配顺序树每个点的匹配顺序将 rDAG 森林的边的方向改变，展示每个点在匹配时，与之前已经匹配的点的关联关系；比如因为有一条有向边 (u_2, u_5) 表示 u_2 在匹配顺序树的拓

算法 4-7　MatchMultiQueries

Data: 共享增量状态图 g_s，当前的匹配映射 f，当前匹配顺序树节点 mo

```
1  V_cand ← candidate(g_s, f, mo) ;
2  V_cand ← V_cand/{v|(u → v) ∈ f} ;                    /* 单射检查 */
3  if V_cand = ∅ then
4  |   return;
5  end
6  if mo.leaf ≠ ∅ then
7  |   foreach v ∈ V_cand do
8  |   |   f.push(mo.u ← v);
9  |   |   foreach q ∈ mo.leaf do
10 |   |   |   M[q].push(f);                  /* 或者直接输出 q 的匹配 f */
11 |   |   end
12 |   |   f.pop();
13 |   end
14 end
15 foreach v ∈ V_cand do
16 |   f.push(mo.u → v) ;
17 |   foreach mo_c ∈ mo.children do
18 |   |   MatchMultiQueries(g_s, f, mo_c) ;
19 |   end
20 |   f.pop();
21 end
```

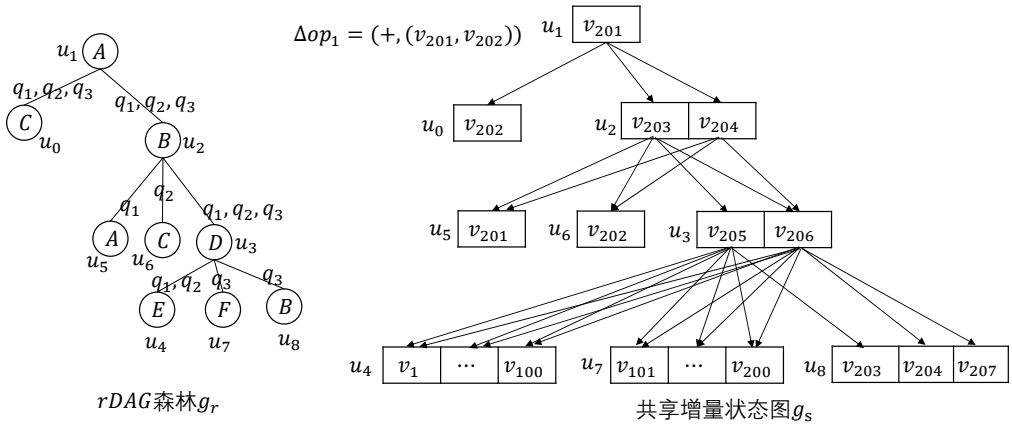
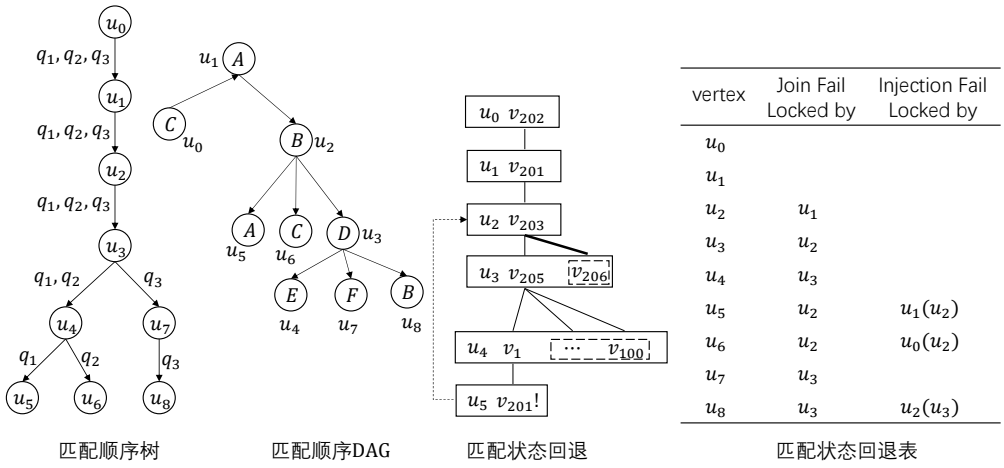


图 4-7　rDAG 森林 g_r 和共享增量状态图 g_s



vertex	Join Fail Locked by	Injection Fail Locked by
u_0		
u_1		
u_2	u_1	
u_3	u_2	
u_4	u_3	
u_5	u_2	$u_1(u_2)$
u_6	u_2	$u_0(u_2)$
u_7	u_3	
u_8	u_3	$u_2(u_3)$

匹配状态回退表

图 4-8　匹配状态回退动机例子

扑顺序先于 u_5，且 u_5 的匹配候选点应该与 u_2 已匹配点有连边，也就是如果 u_5 匹配不到合适的候选点时，与 u_2 有关。“匹配状态回退”展示了在图4-7的共享增量状态图进行匹配搜索时的搜索树，当前的部分匹配是 $f = \{u_0 \to v_{202}, u_1 \to v_{201}, u_2 \to v_{203}, u_3 \to v_{205}, u_4 \to v_1\}$。当 f 尝试匹配 u_5 时，因为 u_2 的匹配点 v_{203} 有的 u_5 类型的邻点只有 v_{201}，故 u_5 的候选匹配点是 v_{201}；因为 u_1 已经匹配到 v_{201}，也就是子图同构要求的单射条件不满足（同一个查询图的不同的查询图点需要映射到不同的数据图点），这时匹配失败；这种情况属于 join 条件（也就是与已匹配点有连边条件）或单射条件不满足导致没有合适的候选点。在原先的匹配逻辑中，应该回溯到 u_4 的其他匹配候选点，如从 v_1 换到 v_2，v_3,…, 和 v_{100} 等。但是很明显 u_5 匹配失败的原因只与 u_5 在匹配顺序 DAG 的祖先节点（join 条件）或者在匹配顺序树中拓扑顺序比 u_5 先且与 u_5 点标签相同的点（单射条件）有关，在这里无论 u_4 匹配什么点都不会影响 u_5 会匹配失败的结局。对于 u_5 这个匹配顺序树分支来说，这里可以对匹配状态进行一个回退，直接回退到让 u_2 切换一个候选数据图点，剪枝掉尝试匹配 u_4 和 u_3 其他候选点的搜索树分支。匹配状态回退表展示了匹配顺序树中的节点在没有合适的候选节点或者因为单射约束导致没有合适的候选节点时与哪些之前已经匹配的匹配顺序树节点有关；比如因为匹配顺序 DAG 在尝试匹配 u_5 时如果没有合适的候选点，应该在算法4-7中回退到匹配 u_2 的函数栈，让 u_2 切换一个候选数据图点，故表中 u_5 的“Join Fail Locked by”是 u_2；如果 u_5 有合适的候选点，但是因为单射条件冲突导致最终没有任何匹配点，单射条件冲突与 u_1 的匹配点有关，又因为 join 失败有关点 u_2 在匹配顺序树或者匹配顺序 DAG 上的拓扑顺序比 u_1 先，所以在单射失败时也会回退到 u_2（表中括号点为实际回退匹配点）。

图4-8中的“匹配状态回退”与“匹配状态回退表”展示了匹配顺序树如何在单

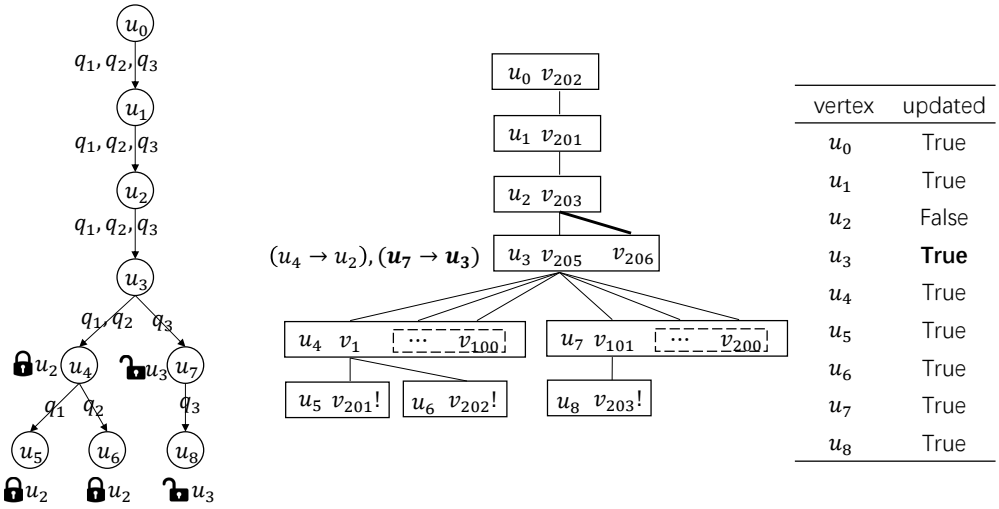


图 4-9　匹配状态回退运行例子步骤一

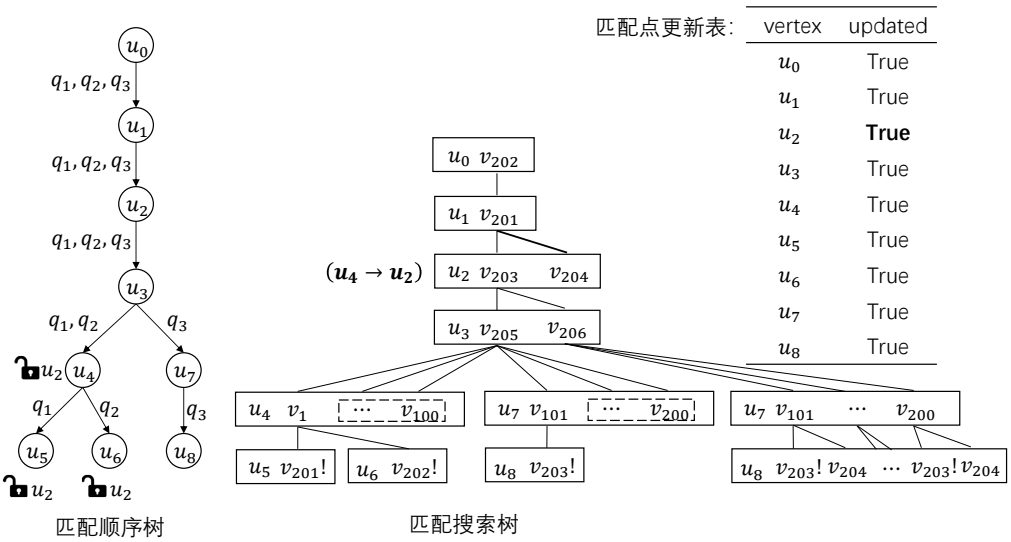


图 4-10　匹配状态回退运行例子步骤二

个查询图进行匹配回退，接下来在图4-9与图4-10展示如何在整个匹配顺序树（多个查询图）做匹配回退。

在匹配顺序树中，不同节点在进行回退时需要回退到不同的已匹配节点来进行剪枝。为此，我们一对设计匹配加锁与匹配解锁操作，给应该剪枝的匹配顺序树节点进行加锁，按匹配顺序树顺序匹配时，不匹配当前无法解锁的匹配顺序树节点（树分支）。当某个匹配顺序树节点没有合适的匹配候选点时，依据图4-8中的“匹配状态回退表”对当前树节点进行“加锁”标记上应该回退的节点。加锁一共分三个步骤，在图4-9中的因为 u_5 没有合适的匹配点需要 u_2 的匹配点更新才有可能有匹配，于是第一步先给匹配顺序树中设置 $u_5.locked = u_2$，第二步在匹配点更新表 u_2 的 $updated$ 设置成 $False$（所有点的 $updated$ 默认为 $True$，且每当一个匹配顺序树节点匹配新的数据图点（共享增量状态图点）时，都会将该匹配顺序树节点的 $updated$ 置为 $True$），第三步回溯对父节点以及祖父节点等进行加锁。在回溯时，u_5 的父节点是 u_4，但是 u_4 还有另外一个下一个匹配点是 u_6 的匹配顺序树分支；如果一个匹配顺序树节点有多个子节点，那么该节点的锁节点是所有子节点的锁节点中匹配顺序最晚的锁节点；在匹配顺序树的 u_4 节点需要继续往子节点 u_6 的方向进行匹配。但从图4-9中的 $u_4 \to v_1$ 走到匹配 u_6 时，u_6 也没相应的匹配候选点，按图4-8中的匹配回退表可以直接回退到匹配 u_2，使 u_2 切换候选点。因为图4-9匹配顺序树中 u_4 的子节点 u_5 与 u_6 的锁节点都是 u_2，故 u_4 的锁节点也是 u_2。回溯到 u_3 时，u_3 会往 u_7 的分支进行匹配。当 u_7 匹配 v_{101} 后，u_8 并没合适的匹配点，u_8 会设置锁节点为 u_3; 因为 u_7 只有子节点 u_8，同样对 u_7 设置锁节点为 u_3；这期间“匹配点更新表”的 u_3 为 $False$。但是回溯到匹配 u_3 时，匹配点从 v_{205} 切换到 v_{206}，此时“匹配点更新表”的 u_3 会更新为 $True$。u_3 会再次尝试往匹配顺序树的叶节点 u_4 方向进行匹配；u_4 查询当前被 u_2 锁住，而查询到匹配点更新表中 u_2 的匹配点仍未更新，所以将会放弃往 u_4 的匹配顺序树分支进行匹配。图4-9展示了进行完上述步骤后的匹配顺序树，匹配搜索树，和匹配点更新表。

继续进行到图4-10，因为匹配顺序树中 u_3 的树分支 u_7 被 u_3 锁住，所以 u_7 会再次查询匹配点更新表确认 u_3 是否已经更新了匹配点，确认结果是将对 $u_7.locked$ 进行重置，并从 $u_3 \to v_{206}$ 运行到 $u_7 \to v_i, i \in [101, 200]$，运行到 $u_8 \to v_{204}$。这期间会返回 q_3 的匹配结果，$\{f | f = \{ u_0 \to v_{202}, u_1 \to v_{201}, u_2 \to v_{203}, u_3 \to v_{206}, u_7 \to v_i, u_8 \to v_{204}\}, i \in [101, 200]\}$。在回溯算法正常回溯到 u_3 后，u_3 没有其他候选点，会正常回溯到 u_2，u_2 会将匹配点从 v_{203} 换到 v_{204}，图4-10的“匹配点更新表”中 u_2 的状态从 $False$ 变成 $True$。在之后向树叶节点方向的匹配过程中，u_4、u_5、和 u_6 都会查询到 u_2 的匹配点已经更新从而被解锁可以正常匹配。

第 5 章　实验设置与实验结果

我们设计了多组实验来评估 ShareFlow 算法性能，在这一章节中，我们将展示我们的实验结果，并说明 ShareFlow 算法的有效性。

5.1　实验设置

我们在实验中与近年来提出的多种高效的增量动态子图匹配算法进行了对比，这些算法分为单查询算法与多查询算法，单查询算法每次运行仅能处理单个查询图，而多查询算法在查询时将多个查询图融合为一个共享查询图进行匹配，一次运行即可获取多个查询图的匹配结果。我们的对比目标包括三种单查询算法 RapidFlow，SymBi 与 TurboFlux，和两种多查询算法 IncCMatch$_{Ann}$ 和 TRIC。RapidFlow 发表于 2022 年，该算法是截止目前最为高效的单查询匹配算法；SymBi 和 TurboFlux 分别发表于 2021 年和 2018 年，它们的性能虽然弱于 RapidFlow，但其算法架构与我们的算法较为相似，因此我们也需要与其比较；而 IncCMatch$_{Ann}$ 和 TRIC 均表于 2020 年，它们是目前为止最为优秀的多查询匹配算法。

我们将 ShareFlow 与这些算法进行了多方面的比较，包括在查询图类型不同、数据集大小不同、边插入率不同和共享查询图包含查询图的数量不同等场景。我们对 IncCMatch$_{Ann}$ 和 TRIC 进行了实现，并且尽可能对它们进行了优化。此外，实验中使用的 RapidFlow 的代码由其作者提供，SymBi 和 TurboFlux 的代码由[53]的作者提供，所有算法均使用 C++17 进行实现，并使用 gcc 7.5.0 进行编译。我们所有实验均在装配有 Gold 5122 3.6GHz CPU 和 128GB 内存的机器上进行，操作系统均为 Ubuntu 18.04.5 LTS。

5.1.1　数据集

为尽量保持实验场景与相关工作一致，我们使用与 RapidFlow 相同的两个现实中真实存在的动态图数据集 Amazon[54]和 LiveJournal[55-56]进行试验。Amazon 数据集是亚马逊公司购物平台上的商品网络，它是由亚马逊的“购买 X 的人也购买了 Y”功能提及的商品组成的。网络中的节点表示产品，从 A 到 B 的边表示产品 A 经常与产品 B 共同购买。LiveJournal 数据集是 LiveJournal 的在线社交平台提供的社交网络，其节点表示用户，A 到 B 的边表示 A 与 B 为朋友关系。

在原始数据集中，Amazon 和 LiveJournal 的点和边均未标记标签，RapidFlow

表 5-1　数据集的详细统计数据，$|V(G)|$ 代表数据图点的数量，$|E(G)|$ 代表边的数量，$|\Sigma_V|$ 代表点标签集合的大小，d_{avg} 代表所有节点的平均度数，d_{max} 代表单个节点的最大度数

数据集	$\|V(G)\|$	$\|E(G)\|$	$\|\Sigma_V\|$	d_{avg}	d_{max}
Amazon	0.4M	2.4M	6	12.2	0.2M
LiveJournal	4.9M	42.9M	30	18.1	4.3M

的作者将分布较为集中的标签随机分配给了各个顶点，表5-1显示了数据集的详细统计信息。为了从多方面来对比各个算法的性能，我们还对以上两组数据集进行了部分修改，如边的数量和插入率等，具体的操作方法将会在后续实验描述中详细说明。

5.1.2　查询集

我们实验的查询集基本使用由 RapidFlow 作者提供的查询集，是在原始数据图中随机提取连通子图而得到的。在 RapidFlow 中其作者对查询图进行了分类，先将查询集根据是否含有环分为有环查询图和树查询图（Tree Queries）两类，再进一步根据查询图节点的平均度数将含环查询图分为稠密图（Dense Queries）和稀疏图（Sparse Queries），稠密图的节点平均度数不小于 3，反之则为稀疏图。Amazon 和 LiveJournal 两个数据集分别对应拥有稠密查询集，稀疏查询集和树查询集各一组，每组含有 100 个互不相同的查询图，查询图的节点数量均为 6，边的数量从 5 至 14 不等。

5.1.3　评估指标

实验的时间开销主要包括读图存图、预处理、增量查询三部分时间。因为每种算法均采取相同的数据结构和读图方法来处理查询图与数据图，所以我们并未统计读取或存储图的时间。此外，算法的预处理主要包括多查询算法合并查询图、算法对查询图与数据图进行修饰、初始化为加速增量查询所设计的辅助数据结构和搜索初始匹配。在现实场景下，增量动态子图匹配通常是在数据图 G_0 和初始匹配 M_0 已经存在的情况下，在线地对新增（删除）的边进行正（负）匹配搜索，所以我们并不关心以上两类时间（其中多查询算法合并查询图基本都在 10^{-1} 时间级别以内完成可以完全忽略），在接下来的实验中我们会将重点聚焦在增量查询时间上。增量查询时间（Incremental Query Time）主要包括两部分时间，一是更新增量辅助数据结构，二是从增量辅助数据结构中搜索匹配。需要注意的是，ShareFlow、TRIC 和 $\text{IncCMatch}_{\text{Ann}}$ 是多查询算法，而其他算法均为单查询算法。所以在实验中我们分别统计单查询算法处理 100 条查询所用的增量查询时间的总和，与多查询

算法同时处理 100 个查询图的时间进行对比。此外，为确保程序运行时间在合理范围内，我们为单个查询设置匹配时间上限为 2 小时，而对于多查询算法，为公平起见，时间上限则应设置为 200 小时。在正式实验开始前，我们会对使用单查询算法对每个查询图进行评估，如果某个查询图在任意单查询算法上达到了时间限制，我们将该查询替换为新的可在 2 小时内运行完毕的查询图，这样处理的原因是在保持查询集的大小为 100 的同时防止多查询算法的效率被某个困难的查询图“拖累”。若仍有多查询算法无法在 200 小时内匹配完毕，我们记时间为 200 小时。因为大部分的算法是较为高效的，所以这种做法并不会过分影响实验的公平性与科学性。

5.2　实验结果

5.2.1　实验一：各算法在查询图类型不同时的性能对比

在实验一中我们将对上面提到的所有算法进行增量查询时间，数据集为 Amazon 和 LiveJournal，我们在原始完整的数据图 G 中随机抽取 10% 的边作为图的更新流 ΔG 用于插入，保留剩余 90% 的边和所有顶点作为初始数据图 G_0。这里我们定义数据图的插入率 α（Insertion Rate）为更新流 ΔG 中边的数量占原始完整数据集 G 中边的数量的百分比，即 $|\Delta G|/|E(G)| \times 100\%$，则此实验插入率为 10%；查询集为对应的 Dense、Sparse 和 Tree 查询集。

图5-1展示了各算法在两个数据集的三种查询集上运行的增量查询时间。从三种查询集的平均增量查询时间来看，整体上 ShareFlow 算法性能比最好的 RapidFlow 还要分别快 4.13 倍和 6.12 倍，其中对 Tree 类型增量查询的加速效果尤为显著，分别是 RapidFlow 的 15.30 倍和 26.95 倍，比其余算法快 300 倍以上。但是在 Dense 和 Sparse 类型的查询集上，ShareFlow 的性能大部分略差于 RapidFlow，在 Amazon Dense 中性能为 RapidFlow 的 0.72 倍，在 Amazon Sparse 中为 0.61 倍，在 LiveJournal Dense 中为 0.68 倍。ShareFlow 仅在 LiveJournal 的 Sparse 查询集上优于 RapidFlow，是 RapidFlow 的 1.24 倍。原因之一为环的减少降低了我们在3.2中提到的 rDAG 森林的复杂程度，如图 3-5(d) 所示，若标记图中存在环，rDAG 中则存在有 u_4'，u_5' 与 u_6' 这些重复节点，若环的数量和复杂程度增加，则会使得 rDAG 森林中重复的节点数量急剧增加从而使匹配的开销增大。而在处理 Tree 类型的查询集时，我们的标记图没有环结构出现，因此使 rDAG 森林的冗余度降低，提高了匹配效率。另一原因有关我们在4.4中讲述的剪枝技术，在 Tree 类型的查询图中，分支之间互不关联，剪枝技术会更加有效地减少不完全匹配的产生。

我们观察到 TRIC 和 $\text{IncCMatch}_{\text{Ann}}$ 与其他算法的性能差距较大，TRIC 在 Ama-

zon Dense 和 Sparse 中的用时分别比前一名的 TurboFlux 要慢 63.71 倍和 326.26 倍，在其余组中均运行时间超限；而 $\text{IncCMatch}_{\text{Ann}}$ 则是在所有组别中均达到了设置的时间上限。这是因为 TRIC 基于查询图的边对子图进行匹配，TRIC 会先将查询图随机分解为多条 path，在数据图中为每一条 path 寻找候选，最终将每条 path 的候选集合 join 起来，判断是否匹配成功，而这样的 join 操作则是非常耗时的；而对于 $\text{IncCMatch}_{\text{Ann}}$ 来说，$\text{IncCMatch}_{\text{Ann}}$ 是为批量处理更新流而设计的，即一次性向初始数据图 G_0 中插入多条边，再按照插入边在数据图中的最大影响范围对其增量辅助数据结构 CCPI 进行自顶向下和自底向上的更新，最后根据更新后的 CCPI 对 Annotated CGPs 重新搜索所有的匹配后找到新增的匹配，所以若用 $\text{IncCMatch}_{\text{Ann}}$ 处理流式的数据图更新，则每一条边的插入都造成一次对 CCPI 进行自顶向下和自底向上的更新，这会带来大量多余的开销，导致性能较差。因此，在后续实验的介绍中，我们会将重点放在 ShareFlow 和 RapidFlow、SymBi 与 TurboFlux 的性能对比上，不再展示 TRIC 和 $\text{IncCMatch}_{\text{Ann}}$ 的实验结果。

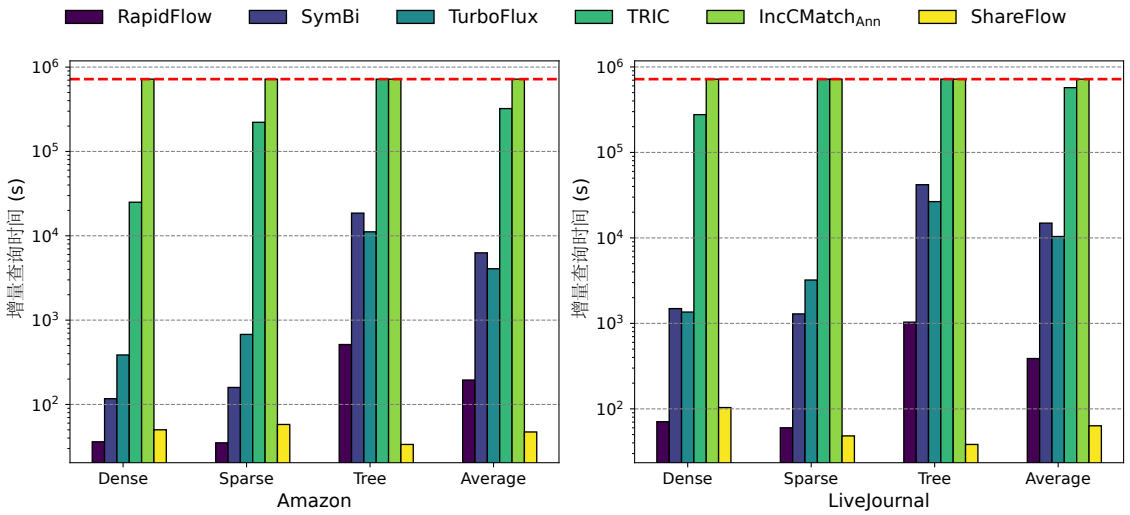


图 5-1　各算法在查询图类型不同时的增量查询时间对比

5.2.2　实验二：各算法在数据图大小不同时的性能对比

在实验二中使用与实验一完全相同的更新流 ΔG，固定为原图边数的 10%；数据图方面分别从 Amazon 和 LiveJournal 剩余的 90% 的边中随机抽取 10%、40%、70% 和 100% 作为不同大小的 G_0，即当抽取量为 10% 时，G_0 中边的数量为原始数据图 G 中边的数量的 0.09 倍（10% × 90%），当抽取 100% 的边时，获得的 G_0 与实验一中完全相同。查询集仍为两个数据集对应的 Dense、Sparse 和 Tree。

图5-2(a)-(c) 分别表示在 Amazon 数据集上的对 Dense、Sparse 和 Tree 进行增量查询的时间，(d)-(f) 分别表示在 LiveJournal 数据集上的对 Dense、Sparse 和 Tree 进行增量查询的时间。我们发现实验二与实验一的结果基本一致，即在查询集为

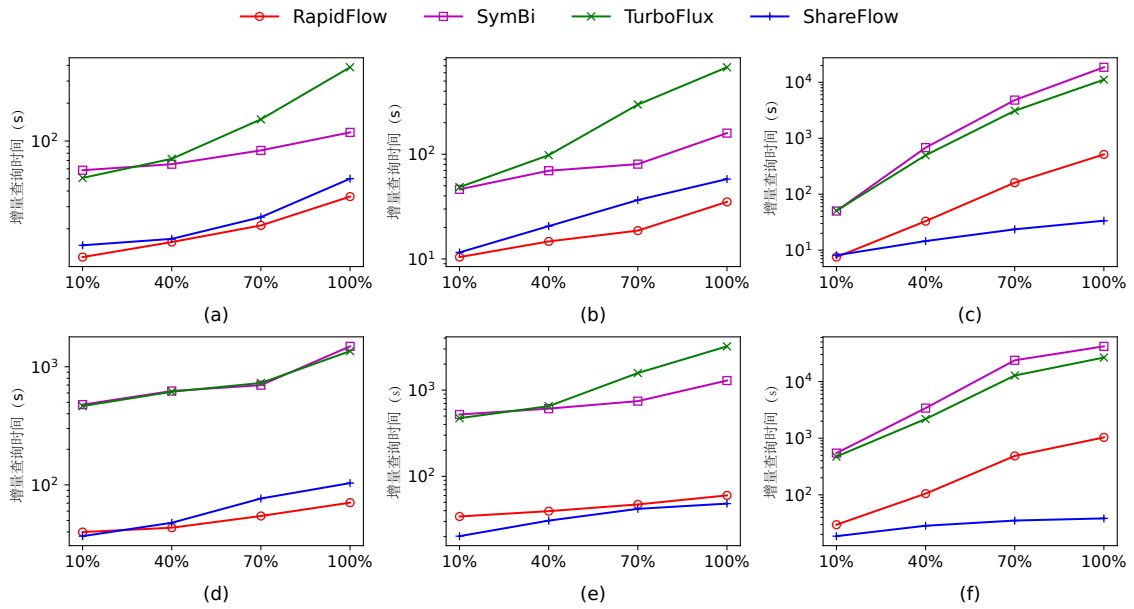


图 5-2　各算法在数据图大小不同时的增量查询时间对比

Tree 时，无论初始数据图 G_0 的大小如何变化，ShareFlow 的增量查询性能几乎全部优于 RapidFlow （1.59-26.95 倍），远远优于 SymBi 和 TurboFlux（6.25-1092.52 倍），并且当查询图大小由 10% 逐步增大至 100% 时，ShareFlow 在性能上与其它算法的差距也逐步扩大。仅有 RapidFlow 在数据集为 Amazon（10%），查询集为 Tree 这一组实验中优于 ShareFlow，为 ShareFlow 的 1.08 倍。我们分析了出现这种结果的原因：初始数据图 G_0 仅包含原图 G 中极少量的边，而查询图是随机从原始数据图 G 中提取得到的，因此这组实验中很多查询仅能搜索到较少的匹配甚至不存在匹配，而 RapidFlow 则会在更新辅助数据结构和枚举匹配前进行剪枝，提前判断是否有必要进行更新和枚举，这样便可以避免匹配较少或不存在情况下算法的开销。除 Tree 类型查询集之外，ShareFlow 在数据集为 LiveJournal（10%-100%），查询集为 Sparse 这几组中，性能也均优于其它算法。在其它几组实验中，ShareFlow 的增量性能略微弱于 RapidFlow，但优于 SymBi 和 TurboFlux 一至三个数量级。

5.2.3　实验三：各算法在插入率大小不同时的性能对比

我们在实验四中使用 Amazon 和 LiveJournal 数据集，从原始数据图中固定抽取 10% 的边作为 G_0，将剩余 90% 的边作为更新流 ΔG，我们将插入率 α 以 10% 为步长从 10% 增加至 90%，对两个数据集的 Dense、Sparse 和 Tree 三个查询集分别进行实验。实验结果如图5-3所示，(a)-(c) 分别表示在 Amazon 数据集上的对 Dense、Sparse 和 Tree 进行增量查询的时间，(d)-(f) 分别表示在 LiveJournal 数据集上的对 Dense、Sparse 和 Tree 进行增量查询的时间。可以观察到，ShareFlow 在绝大多数情

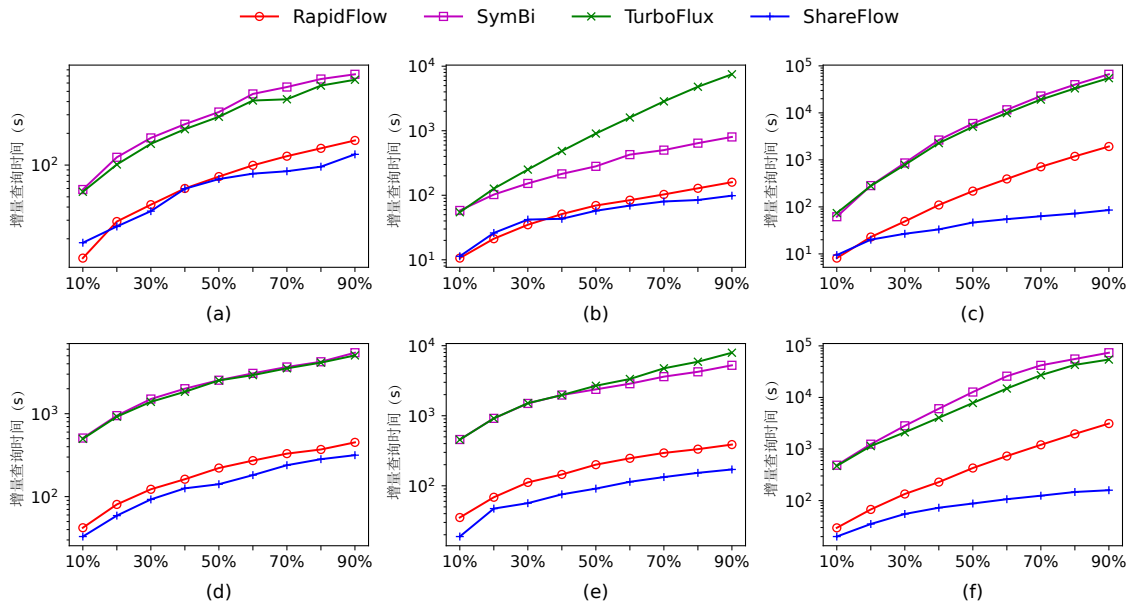


图 5-3　各算法在插入率大小不同时的增量查询时间对比

况下的增量查询用时低于 RapidFlow，仅有在 Amazon 中少量低插入率的情况下高于 RapidFlow，原因我们已在实验二中提到，因为 RapidFlow 在匹配较少甚至不存在的情况下会在更新前进行剪枝，从而避免了额外的开销。与 SymBi 和 TurboFlux 相比，ShareFlow 的性能均比他们高出一至三个数量级。值得注意的是，ShareFlow 折线相较于其他三条折线上升更为缓慢，表明我们的算法对插入率变化的敏感程度更低。而在 Tree 类型查询集的两组实验中，折线几乎接近线性增长，这种性能优势得益于我们在4.4中提到的匹配剪枝技术，这也说明该剪枝技术是十分有效的。

5.2.4　实验四：ShareFlow 在批量大小不同时的性能对比

ShareFlow 为多查询增量匹配算法，我们在此实验中探索了批量大小 Batch Size，即一个合成标记图中包含的单个查询图数量，对 ShareFlow 的性能影响。在此实验中我们使用 Amazon 数据集，固定插入率为 10%，则剩余 90% 的边形成 G_0。我们取 Batch Size 为 1、5、10、20、50 和 100，这些数字可以整除 100，使得算法完成 100 个查询所需的批处理次数为整数。为避免分组导致的偶然性，我们分别对 Dense、Sparse 和 Tree 查询集中的 100 个查询图根据不同的 Batch Size 进行随机分组；比如 Batch Size 是 5，意味着 100 个查询图分成 20 组每组有 5 个查询图，然后统计 ShareFlow 这 20 组查询图增量查询的总时间作为一次实验的结果；三个查询图数据集各进行 30 次这样随机分组实验后取平均值作为最后的实验结果，绘制出图5-4的 Dense，Sparse 和 Tree 三条折线，图5-4中的 Average 折线是将上述三条折线取平均后绘制得来。

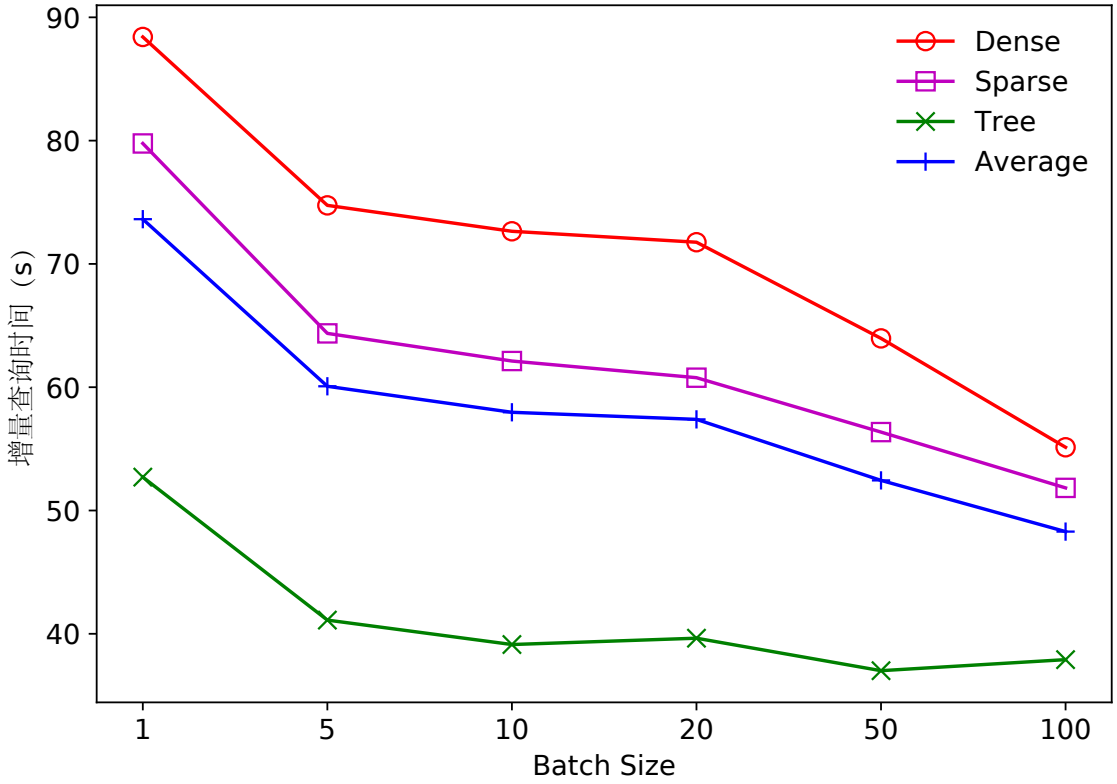


图 5-4　ShareFlow 在批量大小不同时的增量查询时间对比

从图5-4可以看出，我们的实验结果是基本符合随着 Batch Size 增大，增量查询时间呈现下降趋势。这是因为当融合的查询图增多，在合成标记图中共享的节点和边随之增加，因此增量辅助数据结构共享增量查询图中将会有更多共用的候选节点，能够有效减少后续搜索过程的匹配次数；此外，更多边和点的共享可以有效地减少共享增量查询图更新过程中的冗余的开销，使得性能提升。我们注意到，在 Batch Size 从 1 调整至 5 时，三种类型的查询集增量查询时间降幅均最大，分别下降了 15.45%（Dense）、19.31%（Sparse）和 22.00%（Tree），这也说明我们多查询策略相比单查询来说是有效的。

不过我们注意到在 Tree 查询集中 Batch Size 为 100 时，查询时间相比 Batch Size 为 50 时有所上升。现象出现的原因有以下几点：第一，在 Batch Size 为 50 和 100 时，合成查询图的边和点的共享率差距不大，即在 Batch Size 为 50 左右时，边与点的共享率已接近峰值，导致区分度不明显；第二，当融合的查询图比较多时，rDAG 森林或共享增量状态图某些节点的度数会变得非常大，在更新相邻节点和边时会造成大量访问开销。但从平均折线来看，增量查询的时间仍随着 Batch Size 的增加呈明显的下降趋势。

结 论

本文对多查询动态子图匹配问题进行了研究，提出了一种高效的多查询算法ShareFlow。我们首先基于查询图之间的相似性，设计了一种映射方法将多个查询图合并成一个标记图，这个标记图可以用于多查询静态子图匹配问题；然后将标记图转化为一个 rDAG 森林，专门用于多查询动态子图匹配问题；并根据 rDAG 森林构建增量辅助数据结构——共享增量状态图，该数据结构用于保存双向动态规划算法在查询图和动态数据图之间进行计算时得出的中间结果，可以有效地降低搜索空间的大小。基于上述方案，我们进一步设计了一种快速有效的共享增量状态图的流处理更新和枚举算法，同时作用于多个查询图进行高效的共享增量查询，同时减少增量更新或增量搜索时的冗余计算。此外我们还设计了针对多查询匹配顺序树的匹配状态的回退过程对匹配树进行剪枝，实验表明我们这种剪枝策略在 Tree 查询图数据集上十分高效。同时，在多个数据集上的大量实验证实，我们所设计的多查询的优化方法是十分有效的，ShareFlow 时间性能上优于所有的 SOTA 多查询动态子图匹配算法，优于大部分的单查询动态子图匹配算法，甚至在一些数据上，比如 Tree 查询图数据集，比最先进的算法 RapidFlow 要高出一定数量级。

在未来的工作中：

（1）我们设计的将标记图变成 rDAG 森林的方法不仅仅只是局限于用来合并查询图来加速多查询动态子图匹配。事实上，数据库所有的数据查询执行都可以解析为一个个 DAG. 如果我们用我们的合并方法将不同的数据查询执行进行合并来共享查询执行，那么将可以大大减少数据查询执行的计算资源开销。

（2）从实验中我们可以看出不一定合并越多的查询图最终效果越好，我们会尝试找到一种方法将查询图集合划分成不同的组，使得组内的查询图集合相似性高从而在共享执行时不会因为执行一部分查询图的增量更新或者搜索时被另一部分无关的查询图集合影响太多性能，并且探索查询图之间的相似度等其它因素对多查询算法性能的影响。

（3）此外，我们会进一步提升 ShareFlow 在不同查询场景下的效率，比如将 RapidFlow 对单查询的优化改进成适合同时多查询的增量更新与增量匹配，增强 ShareFlow 在 Dense 或者 Sparse 数据集上的性能。

（4）同时也希望可以使用异构算力（比如 GPU）和提升并行度（比如多核 CPU 和多线程等）来提升算法的效率。

参考文献

[1] PRŽULJ N, CORNEIL D G, JURISICA I. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks[J]. Bioinformatics, 2006, 22(8): 974-980.

[2] SNIJDERS T A, PATTISON P E, ROBINS G L, et al. New specifications for exponential random graph models[J]. SM, 2006, 36(1): 99-153.

[3] FAN W. Graph pattern matching revised for social network analysis[C]//Proceedings of ICDT. 2012: 8-21.

[4] YAN X, YU P S, HAN J. Graph indexing: A frequent structure-based approach[C]//Proceedings of SIGMOD. 2004: 335-346.

[5] ZHAO P, HAN J. On graph query optimization in large networks[J]. Proceedings of VLDB, 2010, 3(1-2): 340-351.

[6] GAREY M R. A Guide to the Theory of NP-Completeness[J]. Computers and intractability, 1979.

[7] SHANG H, ZHANG Y, LIN X, et al. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism[J]. Proceedings of VLDB, 2008, 1(1): 364-375.

[8] ZHANG S, LI S, YANG J. GADDI: distance index based subgraph matching in biological networks[C]//Proceedings of EDBT. 2009: 192-203.

[9] HE H, SINGH A K. Graphs-at-a-time: query language and access methods for graph databases [C]//Proceedings of SIGMOD. 2008: 405-418.

[10] HAN W S, LEE J, LEE J H. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases[C]//Proceedings of SIGMOD. 2013: 337-348.

[11] BI F, CHANG L, LIN X, et al. Efficient subgraph matching by postponing cartesian products [C]//Proceedings of SIGMOD. 2016: 1199-1214.

[12] HAN M, KIM H, GU G, et al. Efficient subgraph matching: harmonizing dynamic programming, adaptive matching order, and failing set together[C]//Proceedings of SIGMOD. 2019: 1429-1446.

[13] MOMJIAN B. PostgreSQL: introduction and concepts: volume 192[M]. 2001.

[14] BONCZ P A, et al. Monet: A next-generation DBMS kernel for query-intensive applications [Z]. 2002.

[15] MILLER J J. Graph database applications and concepts with Neo4j[C]//Proceedings of SAIS: volume 2324. 2013.

[16] ULLMANN J R. An algorithm for subgraph isomorphism[J]. JACM, 1976, 23(1): 31-42.

[17] CORDELLA L P, FOGGIA P, SANSONE C, et al. A (sub)graph isomorphism algorithm for matching large graphs[J]. TPAMI, 2004, 26(10): 1367-1372.

[18] JÜTTNER A, MADARASI P. VF2++—An improved subgraph isomorphism algorithm[J]. Discrete Applied Mathematics, 2018, 242: 69-81.

[19] CARLETTI V, FOGGIA P, SAGGESE A, et al. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3[J]. TPAMI, 2017, 40(4): 804-818.

[20] ARCHIBALD B, DUNLOP F, HOFFMANN R, et al. Sequential and parallel solution-biased search for subgraph algorithms[C]//CPAIOR. 2019: 20-38.

[21] BONNICI V, GIUGNO R, PULVIRENTI A, et al. A subgraph isomorphism algorithm and its application to biochemical data[J]. BMC bioinformatics, 2013, 14(7): 1-13.

[22] CARLETTI V, FOGGIA P, VENTO M. VF2 Plus: an improved version of VF2 for biological graphs[M]//GbRPR: volume 9069. 2015: 168-177.

[23] SUN S, LUO Q. Parallelizing recursive backtracking based subgraph matching on a single machine[C]//ICPADS. 2018: 1-9.

[24] CARLETTI V, FOGGIA P, RITROVATO P, et al. A parallel algorithm for subgraph isomorphism[C]//GbRPR. 2019: 141-151.

[25] MCCREESH C, PROSSER P. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs[C]//CP. 2015: 295-312.

[26] BOUHENNI S, YAHIAOUI S, NOUALI-TABOUDJEMAT N, et al. Efficient parallel edge-centric approach for relaxed graph pattern matching[J]. The Journal of Supercomputing, 2022, 78(2): 1642-1671.

[27] KIMMIG R, MEYERHENKE H, STRASH D. Shared memory parallel subgraph enumeration [C]//IPDPSW. 2017: 519-529.

[28] GIUGNO R, BONNICI V, BOMBIERI N, et al. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures[J]. PloS one, 2013, 8(10): e76911.

[29] GUO W, LI Y, TAN K L. Exploiting reuse for GPU subgraph enumeration[J]. TKDE, 2020, 34 (9): 4231-4244.

[30] YAN D, GUO G, KHALIL J, et al. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing[J]. The VLDB Journal, 2021: 1-34.

[31] BOUHENNI S, YAHIAOUI S, NOUALI-TABOUDJEMAT N, et al. A survey on distributed graph pattern matching in massive graphs[J]. ACM Computing Surveys (CSUR), 2021, 54(2): 1-35.

[32] HANG Q, YONG D. Isolate-Set-Based In-Memory Parallel Subgraph Matching Framework [C]//Proceedings of CSSE. 2022: 129-134.

[33] CHOUDHURY S, HOLDER L, CHIN G, et al. Streamworks: a system for dynamic graph search[C]//Proceedings of SIGMOD. 2013: 1101-1104.

[34] SADOWSKI G, RATHLE P. Fraud detection: discovering connections with graph databases [J]. White Paper-Neo Technology-Graphs are Everywhere, 2014, 13.

[35] KIM K, SEO I, HAN W S, et al. Turboflux: A fast continuous subgraph matching system for streaming graph data[C]//Proceedings of SIGMOD. 2018: 411-426.

[36] MIN S, PARK S G, PARK K, et al. Symmetric continuous subgraph matching with bidirectional dynamic programming[J]. Proceedings of VLDB, 2021, 14(8): 1298–1310.

[37] SUN S, SUN X, HE B, et al. RapidFlow: an efficient approach to continuous subgraph matching [J]. Proceedings of VLDB, 2022, 15(11): 2415-2427.

[38] PENCE H E, WILLIAMS A. ChemSpider: an online chemical information resource[Z]. 2010.

[39] KATSAROU F, NTARMOS N, TRIANTAFILLOU P. Performance and scalability of indexed subgraph query processing methods[J]. Proceedings of VLDB, 2015, 8(12): 1566-1577.

[40] DUONG C T, HOANG T D, YIN H, et al. Efficient streaming subgraph isomorphism with graph neural networks[J]. Proceedings of VLDB, 2021, 14(5): 730-742.

[41] ZERVAKIS L, SETTY V, TRYFONOPOULOS C, et al. Efficient continuous multi-Query processing over graph streams[C]//Proceedings of EDBT. 2020: 13-24.

[42] GOTTLOB G, GROHE M, MUSLIU N, et al. Hypertree decompositions: Structure, algorithms, and applications[C]//WG. 2005: 1-15.

[43] LEE J, HAN W S, KASPEROVICS R, et al. An in-depth comparison of subgraph isomorphism algorithms in graph databases[J]. Proceedings of VLDB, 2012, 6(2): 133-144.

[44] RIVERO C R, JAMIL H M. Efficient and scalable labeled subgraph matching using SGMatch [J]. KIS, 2017, 51(1): 61-87.

[45] BHATTARAI B, LIU H, HUANG H H. Ceci: Compact embedding cluster index for scalable subgraph matching[C]//Proceedings of SIGMOD. 2019: 1447-1462.

[46] KIM H, CHOI Y, PARK K, et al. Versatile equivalences: speeding up subgraph query processing and subgraph matching[C]//Proceedings of SIGMOD. 2021: 925-937.

[47] TRAN H N, KIM J J, HE B. Fast subgraph matching on large graphs using graphics processors [C]//DASFAA. 2015: 299-315.

[48] ZHANG Q, GUO D, ZHAO X, et al. Continuous matching of evolving patterns over dynamic graph data[J]. WWW, 2021, 24(3): 721-745.

[49] FAN G, FAN W, LI Y, et al. Extending graph patterns with conditions[C]//Proceedings of SIGMOD. 2020: 715–729.

[50] REN X, WANG J. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs[J]. Proceedings of VLDB, 2015, 8(5): 617-628.

[51] CHOUDHURY S, HOLDER L, CHIN G, et al. A selectivity based approach to continuous pattern detection in streaming graphs[J]. Proceedings of EDBT, 2015.

[52] KANKANAMGE C, SAHU S, MHEDBHI A, et al. Graphflow: An active graph database[C]// Proceedings of SIGMOD. 2017: 1695-1698.

[53] SUN X, SUN S, LUO Q, et al. An in-depth study of continuous subgraph matching[J]. Proceedings of VLDB, 2022, 15(7): 1403-1416.

[54] LESKOVEC J, ADAMIC L A, HUBERMAN B A. The dynamics of viral marketing[J]. ACM Trans. Web, 2007, 1(1): 5–es.

[55] BACKSTROM L, HUTTENLOCHER D, KLEINBERG J, et al. Group formation in large social networks: membership, growth, and evolution[C]//Proceedings of SIGKDD. 2006: 44-54.

[56] LESKOVEC J, LANG K, DASGUPTA A, et al. Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters[J]. Internet Mathematics, 2008, 6: 29-123.

致　谢

4+3 年的南科大生活让我感触非常深刻。

在本科阶段，王琦教授的 Java 编程课与离散数学课带我进入计算机的大门，让我在南方科技大学选择了计算机专业，他就像一位朋友，为你答疑解惑，亲切、智慧、耐心。Stephane Faroult 教授给我教过非常多的基础计算机课如 C++ 编程、数据库原理等，他本人教学能力出众，知识深入浅出，使我受益匪浅，为后来打下了良好基础。Hisao Ishibuchi 教授与姚新教授教会了我很多科研的方法，教会了我如何进行 critical thinking，教会了我如何将研究一步步推向领域顶尖。

在硕士阶段，疫情肆虐。唐博教授与 Kyriakos Mouratidis 教授带我完整地完成一个科研工作，投稿到 SIGMOD 会议，一个数据库领域公认数一数二的会议。在他们身上，每次的开会与讨论，我总能从他们身上学到非常多的知识与经验。尔后，香港理工博士生现如今华为员工章佳豪博士带我再次参与了一次 SIGMOD 层次的工作，从这个工作的每个老师身上，我都学习到了不同的研究风格，学会了与每个人在同一个工作中相互学习和帮助。

去年十月，突然决定申请，万般焦灼，裸考的雅思，裸考的 GRE，飘过录取最低分，突然感染新冠。毫无准备，从零开始写个人简历、个人陈述、选校，在这期间晏潇教授给我的个人简历以及个人陈述很有建设性的建议。在这期间，我最为感谢是三位为我写推荐信的老师，分别是唐博教授，Kyriakos Mouratidis 教授，和王琦教授。也感谢最后在今年博士申请如此激烈的情况下，给我发全奖博士 offer 的 UCI 的 Sharad Mehratra 教授，UMass Amherst 的 Alexandra Meliou 教授和 Marco Serafini 教授。特别是 Sharad Mehratra 教授为我成功申请院长奖学金。

我想，我最应该感谢的是我的父母，尽管他们并不能直接帮助我，但是他们尽己所能给了我他们最好的。他们一直支持着我读书学习知识，支持着我走我想走的道路。最重要但我觉得不可否认的是，他们给了我优秀的基因，聪明、善良、耐心、运动天赋、乐于助人。

最后，感谢在写这篇论文时给我送咖啡的罗炎同学，感谢三年来在数学问题上给我答疑解惑的里昂高师的数学博士生杨绮云，非常感谢东北大学的博士生卫雪和南科大的本科生原佩琦对本课题的参与，感谢华为高斯部门对本课题的资助，感谢我的导师唐博教授三年来的指导。所有的苦难一次次将我杀死，你们一次又一次将我拯救，没有你们，没有今天的我！

个人简历、在学期间完成的相关学术成果

个人简历

1997 年出生于广东高州。

2016 年 9 月考入南方科技大学计算机科学与工程系计算机科学与技术专业，2020 年 7 月本科毕业并获得工学学士学位。

2020 年 9 月至今，在南方科技大学大学计算机科学与工程系电子科学与技术专业攻读工学硕士学位。

在学期间完成的相关学术成果

学术论文

[1] Mouratidis K, **Li K**, Tang B. Marrying Top-k with Skyline Queries: Relaxing the Preference Input while Producing Output of Controllable Size[C]//Proceedings of SIGMOD. 2021: 1317-1330. (EI 收录, DOI:10.1145/3448016.3457299, CCF A 类会议, 数据库顶级国际会议.)

[2] Zhang J, Tang B, Yiu M L, Yan X, **Li K**. τ-LevelIndex: Towards Efficient Query Processing in Continuous Preference Space[C]//Proceedings of SIGMOD. 2022: 2149-2162. (EI 收录, DOI:10.1145/3514221.3526182, CCF A 类会议, 数据库顶级国际会议.)

申请及已获得的专利

[3] 唐博, **李可明**. 一种用户查询推荐方法、装置、电子设备及存储介质, CN114528478A[P]. 2022-05-24.

参与的科研项目

[4] Mouratidis K, **Li K**, Tang B. Quantifying the Competitiveness of a Dataset in Relation to General Preferences. （VLDB 期刊投稿中, Minor Revision 状态, 影响因子 4.243, CCF A 类期刊，数据库顶级国际期刊.）

[5] Zhi X, Yan X, **Li K**, Tang B. Dynamic Graph Storage: An Experimental Survey. （VLDB 会议投稿中，CCF A 类会议，数据库顶级国际会议.）