

分类号: TP391

单位代码: 10335

密 级: 公开

学 号: 12121014

浙江大学

博士学位论文



中文论文题目: 面向大规模二分图的极
大二分团枚举方法研究

英文论文题目: Research on Maximal Biclique
Enumeration in Large Bipartite Graphs

申请人姓名: 潘哲

指导教师: 何水兵

学科(专业): 计算机科学与技术

研究方向: 图计算

所在学院: 计算机科学与技术

论文递交日期 二〇二四年四月

面向大规模二分图的极

大二分图枚举方法研究



论文作者签名: 潘哲

指导教师签名: 何水兵

论文评阅人 1: 匿名

评阅人 2: 匿名

评阅人 3: 匿名

评阅人 4: 匿名

评阅人 5: 匿名

答辩委员会主席: 丁佐华 教授 浙江理工大学

委员 1: 张帅 教授 浙江财经大学

委员 2: 贺诗波 教授 浙江大学

委员 3: 李玺 教授 浙江大学

委员 4: 曾令仿 研究员 之江实验室

答辩日期 二〇二四年六月四日

Research on Maximal Biclique
Enumeration in Large Bipartite Graphs



Author's signature: Zhe Pan

Supervisor's signature: Shuibing He

External reviewers: Anonymous review
Anonymous review
Anonymous review
Anonymous review
Anonymous review

Examining Committee Chairperson:

Zuohua Ding Professor ZSTU

Examining Committee Members:

Shuai Zhang Professor ZUFE

Shibo He Professor ZJU

Xi Li Professor ZJU

Lingfang Zeng Professor Zhejiang Lab

Date of oral defence: June 4th, 2024

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得浙江大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名: 潘哲

签字日期: 2024年6月4日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

(保密的学位论文在解密后适用本授权书)

学位论文作者签名: 潘哲

导师签名: 何水兵

签字日期: 2024年6月4日

签字日期: 2024年6月4日

学位论文作者毕业后去向: 清华大学做博士后

工作单位: 清华大学

电话: 17816855724

通讯地址: 北京市海淀区清华大学

邮编: 100084

致谢

求是园韶光十载，如白驹过隙，历历在目。蓦然回首，恍如隔世，百感交集。在此，感谢一路陪伴、给予我帮助的老师、同窗以及家人朋友们。

首先，感谢我的博士生导师何水兵研究员。在科研方面，您追求卓越、严谨认真的科研态度深深地影响着我。从我刚入学开始，您就对我的科研成果进行严格的把控，将我的每一个科研成果都按照计算机领域的最高标准，撰写成了CCFA类会议论文。在与顶级会议论文审稿人的切磋交流中，我不断成长和进步，坚定了科研的信念。每当论文截稿时间临近时，您总是起早贪黑，像打磨每一件艺术品一样，对论文字斟句酌，如今回想起来，仍旧记忆犹新。在为人处事方面，您更是我的人生导师。您时常向我们现身说法，分享您几十年科研生活的经历与经验，分析其中的利弊得失。您告诉我们人生是一场马拉松，走科研路要耐得住寂寞，安心积累成果，等待机遇，不要心急。每当遇到困恼的时候，您的谆谆教诲总会在耳边回响，引导我走向正确的方向。感谢您，让我博士阶段的研究生活充实而有意义。

其次，感谢我的硕士生导师姜晓红副教授。您在我本科学习期间的计算机体系结构课程，不仅激发了我对这一领域的深厚兴趣，更引领我走上了科研之路。在硕士学习期间，您全方位关注我的成长，积极为我搭建科研实习的桥梁，并多次提供国内外学术交流的宝贵机会。您还鼓励并亲自带领我们参加FPGA设计大赛，为我提供了宝贵的实践和竞技舞台。同时，您的信任与支持，让我有机会带领本科生开展SRTP项目，这对我的科研任务完成起到了重要的辅助作用。感谢您，为我硕士阶段的研究生活增添了丰富的色彩。

再次，感谢研究生生涯中帮助过我的老师和同学们。感谢张学琛、汪睿、孙贤和、银燕龙、王洋、陈刚、曾凯、马德、顾宗华、吴健、杨莹春等老师在论文构思与写作过程中提供了宝贵的指导与建议。感谢李旭、杨斯凌、陈伟剑、陈平、陈帅犇、李振鑫、党政、洪佩怡、胡双、徐耀文、吴桐、段和霄、朱建新、宗威旭、张文捷、宋浩喆、常子汉、詹璇、王文炯、王文涛、刘榕琛、瞿浩阳、陈新宇、张瑞东、郭家豪、廖亦宸、周方、李翔、陈广、杜定益、曹聪、孙浩、聂宗涛、胡志强等同学们的陪伴，让我的研究生生活不再孤单，感谢你们给予我的支持与鼓励，愿你们每个人都能拥有一个光明且美

好的未来。

此外，特别感谢在此期间陪伴着我的家人和朋友们。感谢父母对我长期的默默支持与无私奉献，是你们给予了我读博路上不竭的动力与勇气。感谢姑父在学术探索中给予我的一次次有力的帮助与指导。感谢在这六年硕博生涯中与我保持联系的老友们，是你们的陪伴给我单调的求学生活带来了无尽的温暖与光亮。衷心感谢每一位在我的学术旅程中给予我帮助与支持的人，是你们让我的研究生涯充满了意义与价值。

最后，感谢在百忙之中为我博士论文提出宝贵意见的论文盲审专家与答辩评委，谢谢你们对我论文的严格把关，为我的博士生涯画上圆满的句号。

潘哲

二〇二四年六月于求是园

摘要

在大数据时代，二分图结构被广泛应用于表示两种不同群体之间的联系。作为二分图数据挖掘领域和图论领域中的经典问题，极大二分团枚举在理论研究和现实应用中具有重要价值，在电子商务、社交网络、基因分析以及图神经网络等领域中发挥着重要作用。然而，对于大规模二分图而言，进行极大二分团枚举是非常耗时的。具体而言，现有的极大二分团枚举方法对于包含数千万个极大二分团规模的二分图，需要数个小时来完成枚举。因此，本文重新思考了现有方法在剪枝能力、数据结构和并行实现等三个方面的局限性，并提出了三种独立的高性能极大二分团枚举算法：

(1) 针对剪枝能力受限的问题，提出主动的极大二分团枚举算法 AMBEA。该算法包括主动的集合枚举树和主动的顶点合并剪枝方法两个核心技术点。目前主流的枚举方法基于集合枚举树实现，在枚举过程中会产生大量包含非极大二分团的无效枚举节点，从而影响计算性能。主动的集合枚举树主动地将每个枚举节点的二分团扩展为极大二分团，并利用低成本的节点检查方法删除大量无效的枚举节点；主动的顶点合并剪枝方法在枚举节点生成的同时，主动合并局部计算子图中具有相同局部邻居的顶点，提升剪枝效率。实验表明，AMBEA 对无效节点的剪枝能力提升达到了 9.0 倍，并实现了 5.3 倍的性能提升。

(2) 针对静态数据结构导致的低效性问题，提出自适应的极大二分团枚举算法 AdaMBE。该算法包括基于局部计算子图的优化方法和基于位图的动态子图方法两个核心技术点。基于局部计算子图的优化方法通过动态缓存枚举过程中的计算子图，优化算法的枚举过程；基于位图的动态子图方法在枚举过程中自适应地生成位图，通过位运算加速小型计算子图中的集合运算。实验表明，AdaMBE 实现了 49.7 倍的性能提升，并且能够应用于包含超过 190 亿个极大二分团的数据集。

(3) 针对并行扩展性受限于 CPU 核心数量的问题，提出基于 GPU 的极大二分团枚举算法 GMBE。该算法包括基于枚举节点重用的迭代方法、局部邻居数量感知的剪枝方法以及负载感知的任务调度方法等三个核心技术点。基于枚举节点重用的迭代方法通过重用根节点的内存，避免为新枚举节点动态分配内存，降低了内存开销；局部邻居数量感知的剪枝方法通过批量比较顶点局部邻居的方式，在剪枝的同时最小化线程分歧

问题；负载感知的任务调度方法通过对运行时的大任务根据枚举节点信息进行进一步拆分，实现了细粒度的负载均衡。实验表明，GMBE 实现了 70.6 倍的性能提升。

综上所述，本文提出的三种算法对极大二分团枚举问题的性能提升方式进行了全面探索，将枚举介质扩展至 GPU，并且在包含超过百亿个极大二分团规模的二分图上实现极大二分团枚举。所述三种枚举方法的结合将成为未来的研究方向。

关键词: 二分图， 极大二分团枚举， 集合枚举树， 位图， GPU

Abstract

In the era of big data, bipartite graph structures are widely used to represent connections between two different groups. As a classic problem in bipartite graph data mining and graph theory, maximal biclique enumeration has significant value in theoretical research and practical applications, playing important roles in areas such as e-commerce, social networks, genetic analysis, and graph neural networks. However, for large-scale bipartite graphs, enumerating maximal bicliques is extremely time-consuming. Specifically, existing maximal biclique enumeration methods require several hours to complete the enumeration for bipartite graphs containing millions of maximal bicliques. Therefore, this paper reconsiders the limitations of existing methods in three aspects: pruning capability, data structure, and parallel implementation, and proposes three independent high-performance maximal biclique enumeration algorithms:

(1) To address the issue of limited pruning capability, the paper proposes the Aggressive Maximal Biclique Enumeration Algorithm (AMBEA). The algorithm includes two core techniques: an aggressive set enumeration tree and an aggressive merge-based pruning method. Mainstreaming enumeration methods are based on set enumeration trees, which generate a large number of invalid enumeration nodes containing non-maximal bicliques during the enumeration process, thus affecting computational performance. The aggressive set enumeration tree aggressively expands the biclique of each enumeration node into a maximal biclique and eliminates a large number of invalid enumeration nodes using a low-cost node checking method. The aggressive merge-based pruning method actively merges vertices with the same local neighbors in the local computational subgraph to improve pruning efficiency. Experiments show that AMBEA achieves a 9.0x improvement in pruning capability for invalid nodes and a performance improvement of up to 5.3x.

(2) To address the inefficiency caused by static data structures, the paper proposes the Adaptive Maximal Biclique Enumeration Algorithm (AdaMBE). The algorithm includes two core techniques: optimization based on local computational subgraphs and a bitmap-based dynamic subgraph approach. The optimization based on local computational subgraphs dy-

namically caches the computational subgraphs during the enumeration process to optimize the enumeration process. Bitmap-based dynamic subgraph approach adaptively generates bitmaps during the enumeration process and accelerates set operations in small computational subgraphs through bitwise operations. Experiments show that AdaMBE achieves a performance improvement of up to 49.7x and successfully handles the enumeration requirements of datasets containing over 19 billion maximal bicliques.

(3) To address the limitation of parallel scalability due to the number of CPU cores, the paper proposes the GPU-based Maximal Biclique Enumeration Algorithm (GMBE). The algorithm includes three core techniques: stack-based iteration with node reuse, pruning method based on local neighborhood size, and load-aware task scheduling. The stack-based iteration with node reuse avoids dynamically allocating memory for new enumeration nodes by reusing memory from root nodes, reducing memory overhead. The pruning method based on local neighborhood size minimizes thread divergence issues by comparing a batch of local neighborhood sizes of vertices while pruning. The load-aware task scheduling further splits large tasks during runtime based on enumeration node information, achieving fine-grained load balancing. Experiments show that GMBE achieves a performance improvement of up to 70.6x.

In summary, this paper comprehensively explores ways to improve the performance of maximal biclique enumeration problems, extends enumeration to GPUs, and successfully enumerates all maximal bicliques on bipartite graphs containing billions of maximal bicliques. The combination of the three enumeration methods will be a future research direction.

Keywords: Bipartite Graph, Maximal Biclique Enumeration, Set Enumeration Tree, Bitmap, GPU

缩略词表

英文缩写	英文全称	中文全称
MBE	Maximal Biclique Enumeration	极大二分团枚举
SE tree	Set Enumeration Tree	集合枚举树
DFS	Depth-First Search	深度优先搜索
ASE tree	Aggressive Set Enumeration Tree	主动的集合枚举树
AMP	Aggressive Merge-based Pruning	主动的顶点合并剪枝
PMP	Passive Merge-based Pruning	被动的顶点合并剪枝
AMBEA	Aggressive Maximal Biclique Enumeration Algorithm	主动的极大二分团枚举算法
LCG	Local Computational subGraph	局部计算子图
CG	Computational subGraph	计算子图
BDS	Bitmap-based Dynamic Subgraph	基于位图的动态子图方法
AdaMBE	Adaptive Maximal Biclique Enumeration	自适应的极大二分团枚举
INF	Infinity	无穷大，表示时间超出限制
SM	Streaming Multiprocessor	流处理器
CSR	Compressed Sparse Row	压缩稀疏行格式
GMBE	GPU-based Maximal Biclique Enumeration	基于 GPU 的极大二分团枚举

目录

致谢	I
摘要	III
Abstract	V
缩略词表	VII
目录	VIII
图目录	XI
表目录	XIII
1 绪论	1
1.1 研究背景	1
1.2 研究问题	5
1.2.1 问题定义	5
1.2.2 基本求解方法	7
1.3 国内外研究现状	12
1.3.1 穷举方法	12
1.3.2 规约方法	13
1.3.3 基于集合枚举树的枚举方法	14
1.3.4 特定约束下的枚举方法	14
1.4 研究挑战	15
1.4.1 搜索空间大, 剪枝方法欠佳	15
1.4.2 计算不规则, 静态结构低效	16
1.4.3 负载不均匀, 并行扩展性差	16
1.5 研究内容	17
1.6 本文的组织结构	18
2 主动的极大二分团枚举算法	20
2.1 现有搜索空间优化方法分析	20
2.1.1 顶点排序方法	20
2.1.2 基于枢纽顶点的剪枝方法	23

2.1.3 被动的顶点合并剪枝方法	24
2.2 研究动机	24
2.2.1 集合枚举树的结构限制	25
2.2.2 被动的剪枝方法	25
2.3 AMBEA 算法	26
2.3.1 主动的集合枚举树	26
2.3.2 主动的顶点合并剪枝方法	31
2.3.3 AMBEA 算法设计	36
2.4 实验评估	38
2.4.1 实验设置	39
2.4.2 整体评估	40
2.4.3 技术点分解评估	43
2.4.4 敏感性测试	45
2.5 本章小结	47
3 自适应的极大二分图枚举算法	48
3.1 二分图的存储结构	48
3.2 研究动机	50
3.2.1 原图中大量的无效内存访问	51
3.2.2 邻接表上高昂的集合运算开销	53
3.3 AdaMBE 算法	55
3.3.1 基于局部计算子图的优化方法	55
3.3.2 基于位图的动态子图方法	58
3.3.3 AdaMBE 算法设计	62
3.4 实验评估	64
3.4.1 实验设置	64
3.4.2 整体评估	65
3.4.3 技术点分解评估	67
3.4.4 敏感性测试	70
3.5 本章小结	73

4	基于 GPU 的极大二分图枚举算法	74
4.1	现代 GPU 架构与编程模型	74
4.2	研究动机	77
4.2.1	内存短缺	78
4.2.2	线程分歧	78
4.2.3	负载不均	79
4.3	GMBE 算法	80
4.3.1	基于枚举节点重用的迭代方法	80
4.3.2	局部邻居数量感知的剪枝方法	83
4.3.3	负载感知的任务调度方法	85
4.3.4	GMBE 算法设计	88
4.4	实验评估	89
4.4.1	实验设置	89
4.4.2	整体评估	90
4.4.3	技术点分解评估	91
4.4.4	敏感性测试	94
4.5	本章小结	97
5	总结与展望	98
5.1	工作总结	98
5.2	研究展望	100
	参考文献	102
	作者简历	107
	攻读博士学位期间取得的科研成果	108

图目录

图 1.1 电子商务场景下的二分图与极大二分团	2
图 1.2 二分图中极大二分团枚举问题与相关问题的关系图	3
图 1.3 二分图中的极大二分团枚举问题示例	7
图 1.4 对于集合 $P\{1, 2, 3, 4\}$ 的集合枚举树	8
图 1.5 算法 1.1 在二分图 G_1 上的集合枚举树	10
图 1.6 极大二分团枚举领域国内外研究情况梳理	12
图 1.7 本文的组织结构图	18
图 2.1 根据顶点的邻居数量递增排序	22
图 2.2 根据顶点的邻居数量递减排序	23
图 2.3 二分图 G_1 上的部分集合枚举树	25
图 2.4 算法 2.1 二分图 G_1 上的集合枚举树	30
图 2.5 图 2.4 中节点 m 的生成过程对比	33
图 2.6 AMP 方法执行组划分的详细过程	34
图 2.7 AMBEA 在二分图 G_1 上的集合枚举树	37
图 2.8 ASE 树、AMP 方法的技术点分解评估	43
图 2.9 顶点排序方法的分解评估（对数形式）	44
图 2.10 枚举树平衡性敏感性分析	45
图 2.11 算法扩展性敏感性分析（对数形式）	45
图 2.12 算法并行性敏感性分析（对数形式）	46
图 3.1 二分图 G_2	48
图 3.2 二分图 $G_2(U, V, E)$ 的三种存储结构	49
图 3.3 算法 1.1 在二分图 G_2 上的集合枚举树	51
图 3.4 基于 $ L $ 和 $ C $ 大小的计算子图频率分布图	52
图 3.5 真实数据集中计算图内外顶点访问百分比	53
图 3.6 邻接表与位图的集合交集运算比较实例 $N(v_0) \cap N(v_1)$	54
图 3.7 基于局部计算子图的优化方法示例	57
图 3.8 基于位图的动态子图方法示例	59

图 3.9 AdaMBE 在超大数据集上的整体评估	67
图 3.10 BDS、LCG 方法的运行时间分解评估（对数形式）	68
图 3.11 BDS、LCG 方法的内存使用分解评估（对数形式）	68
图 3.12 LCG 方法的节点剪枝效率	69
图 3.13 BDS 方法下的运行时间分解	69
图 3.14 不同阈值 τ 下 BDS 方法性能评估	70
图 3.15 顶点顺序的性能评估	71
图 3.16 可扩展性评估（对数形式）	72
图 3.17 并行性评估（对数形式）	72
图 4.1 现代 GPU 硬件架构与软件编程模型	75
图 4.2 线程分歧示意图	76
图 4.3 算法 1.1 在二分图 G_3 上的集合枚举树	77
图 4.4 在 GPU 上进行极大二分团枚举中的负载不均问题说明	79
图 4.5 在 GPU 上进行极大二分团枚举的负载不均问题示例	80
图 4.6 <i>node_buf</i> 结构示意图	81
图 4.7 局部邻居数量感知的剪枝方法示意图	84
图 4.8 枚举节点重用方法的分解评估（对数形式）	91
图 4.9 剪枝方法和任务调度方法的分解评估（对数形式）	92
图 4.10 不同调度方法中 SM 上的运行时负载比较	93
图 4.11 负载感知任务调度下阈值设置（对数形式）	94
图 4.12 参数 WarpPerSM 设置（对数形式）	95
图 4.13 GMBE 在不同型号 GPU 上的适用性（对数形式）	95
图 4.14 GMBE 在多 GPU 环境下的可扩展性	96

表目录

表 1.1 本文使用的符号及含义	5
表 2.1 AMBEA 实验数据集统计信息	39
表 2.2 AMBEA 整体运行时间评估（单位：秒）	40
表 2.3 AMBEA 整体内存使用评估（单位：MB）	41
表 2.4 AMBEA 整体剪枝效率评估 (δ/α)	42
表 3.1 AdaMBE 增加的实验数据集统计信息	65
表 3.2 AdaMBE 在常规数据集上的整体运行时间评估（单位：秒）	65
表 3.3 AdaMBE 在常规数据集上的整体内存使用评估（单位：MB）	66
表 4.1 GMBE 实验数据集统计信息	90
表 4.2 GMBE 整体运行时间评估（单位：秒）	91
表 4.3 剪枝方法使用前后生成的非极大二分团与极大二分团比值比较 δ/α	92

1 绪论

在大数据时代，每天都会产生海量的数据，涵盖各个领域，例如电子商务平台、社交网络、医疗健康等。二分图（Bipartite Graph）作为有效的数据表示方法，能准确描述两个不同群体间的关系，比如电子商务中用户与商品的购买关系，因此被广泛应用于数据建模和分析。其中，极大二分团是二分图中稠密子图，代表两个紧密连接的群体。通过枚举极大二分团，我们可以发现和理解数据中的信息，对知识挖掘、数据分析和智能决策等方面至关重要。因此，极大二分团枚举（Maximal Biclique Enumeration, MBE）问题备受研究领域关注。本文从剪枝能力、数据结构和并行实现三种角度出发，研究大规模二分图场景下高效的极大二分团枚举方法，并对应形成三种独立的解决方案。接下来，本章将依次介绍本文的研究背景、研究问题、国内外研究现状、研究挑战以及本文的研究内容和组织结构。

1.1 研究背景

随着信息技术的迅猛发展和广泛应用，人类社会正逐渐进入大数据时代，大规模数据的生成和积累已成为一种常态。这些数据涵盖了生活的方方面面，并蕴藏着丰富而有价值的信息。为了充分挖掘和利用这些数据中所蕴含的有效信息，二分图结构被广泛应用于表示两个不同群体之间的联系^[1]。在二分图中，顶点（Vertex）代表着不同的数据实体，而边（Edge）则表示实体之间的关系。二分图结构能够清晰地描述出数据实体之间的交互和连接。具体而言，在二分图中，顶点被划分为两个不同的集合，而同一集合内的顶点之间并未直接相连。例如，在如图1.1所示的电子商务的场景下，二分图描绘了用户和商品之间的关系，其中顶点可以表示用户或商品，而边则表示购买关系。二分图可以很好地描述了用户与商品之间的关联行为。此外，二分团是指在二分图中形成的一种稠密子图，它代表着数据集中那些紧密连接的群体。以电子商务为例，二分团可以表示同一群用户对同一组商品的产生的批量购买行为。这种紧密的连接揭示了数据中存在的某种规律或者共同特征，为理解群体交易行为提供了一定的线索。而极大二分团是指在二分图中那些独立于其他所有二分团的特殊二分团。它们具有独立性和独特性，不被

其他任何二分团所完全包含。识别并枚举二分图中的极大二分团，有助于发现更为细致和确切的群体信息，进一步为深入探究群体行为内部的脉络和联系提供帮助。

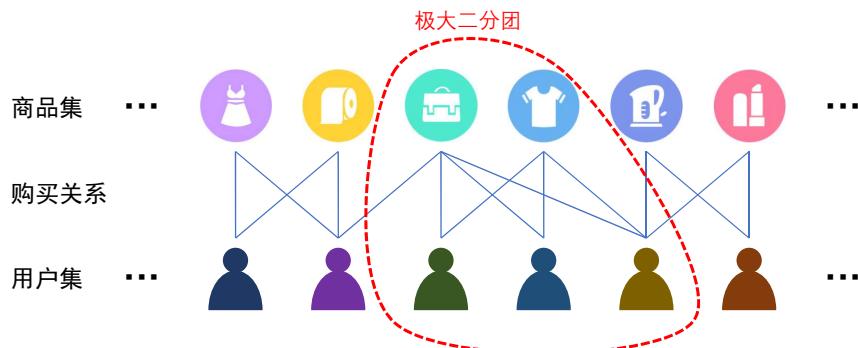


图 1.1 电子商务场景下的二分图与极大二分团

极大二分团枚举在数据挖掘领域具有广泛的应用价值。(1) 在电子商务场景下，极大二分团被广泛应用于描述用户群体对同一组商品的批量购买行为。电子商务领域的领军企业如阿里巴巴、eBay 和亚马逊通常使用二分图表示“用户—购买—商品”的交易关系^[2]。一些不法商家会利用刷单手段，雇佣一批用户购买目标商品，以提高其曝光率并扰乱市场秩序。考虑到极大二分团能有效描绘此类刷单行为，电子商务企业可以通过枚举极大二分团来发现所有可疑交易，提升对刷单行为的检测率^[2-5]。(2) 在社交网络场景下，极大二分团最大程度地描述了用户群体的相同兴趣爱好。通过极大二分团枚举，可以更好地辅助社交推荐系统。通过发现用户之间紧密的连接关系，系统可以推荐给用户其他拥有相似兴趣爱好的用户，从而增加社交互动和用户满意度。同时，极大二分团的枚举还能帮助社交网络平台理解用户行为和需求，进一步优化用户体验，提高平台的粘性和竞争力^[6-7]。此外，通过极大二分团的枚举，还有助于对社交网络进行全面分析，有助于发现社交网络中存在的异常风险信息，提升社交网络安全防护工作的能力^[8-9]。(3) 在基因分析场景下，极大二分团描述了同一组基因对同一组性状的决定作用。枚举极大二分团能够更好地帮助生物学家理解基因与性状之间的关系。通过分析不同基因之间的连接模式，可以揭示基因之间的相互作用以及它们对性状表现的综合影响。这种基于极大二分团的分析方法能够提供更全面和深入的基因功能研究视角，帮助科学家进一步进行蛋白质-蛋白质相互作用网络分析^[10]、从事务数据库中提取基因表型信息^[11]、构建最优进化树^[12]以及探索基因表达机制^[13]。此外，极大二分团的枚举还有助于准确预测基因变异对性状造成的影响，并为疾病研究、遗传工程等领域提供重要的

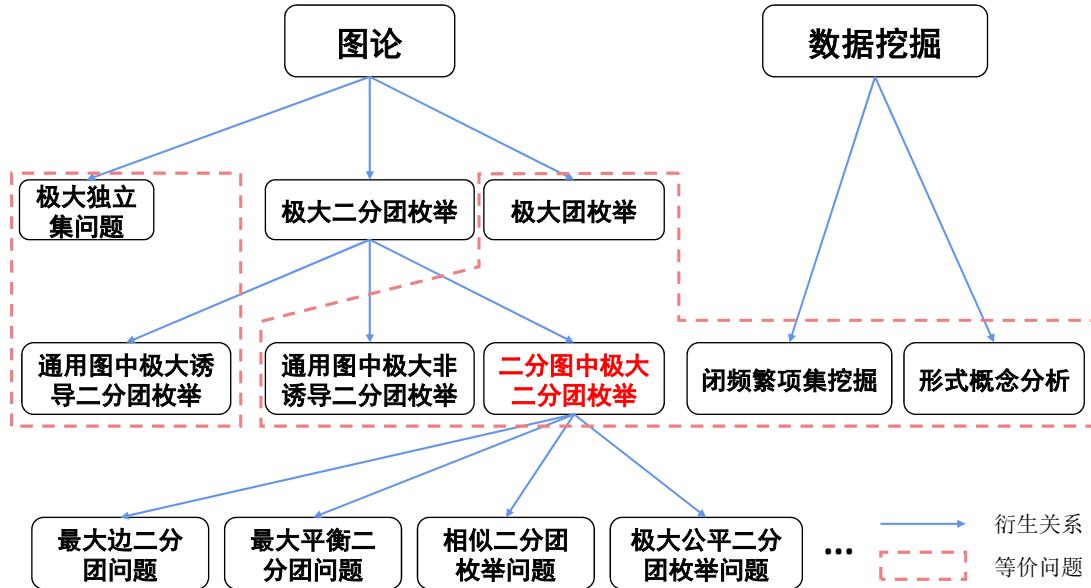


图 1.2 二分图中极大二分团枚举问题与相关问题的关系图

指导意义^[14-16]。(4) 在图神经网络 (Graph Neural Network, GNN) 领域, 极大二分团能够辅助对多个节点数据进行打包, 从而加速 GNN 信息聚合。通过识别和利用极大二分团结构, 可以将具有相似特征或者相互关联的节点分为同一个二分团, 更高效地进行信息传递和计算, 提升图神经网络的训练和推理性能^[17-19]。总而言之, 极大二分团的枚举在电子商务、社交网络、基因分析以及图神经网络等领域都发挥着重要的作用。它能够帮助揭示群体行为、发现异常的交易行为、辅助推荐系统和加速信息聚合等任务, 为相关领域的研究和实践提供有力支持。

同时, 极大二分团枚举也是图论中的一类经典组合优化问题。下面, 我们结合图 1.2, 从三个方面 (相近问题、等价问题和衍生问题) 介绍本研究问题与其他相关问题的联系。(1) 相近问题: 极大二分团是一种特殊的子图, 在通用图中同样存在。通用图中的极大诱导二分团枚举问题可转化为极大独立集问题进行求解^[20]; 通用图中的极大非诱导二分团枚举问题可转化为二分图中的极大二分团枚举问题进行求解^[21]。考虑到二分图中的极大二分团枚举问题具有广泛的应用价值, 本文仅研究二分图中的极大二分团枚举问题。(2) 等价问题: 二分图中的极大二分团枚举问题与许多图论领域和数据挖掘领域的经典问题存在一一映射关系。例如, 极大团枚举问题^[22-28] (Maximal Clique Enumeration, MCE)、闭频繁项集挖掘问题^[29-30] (Frequent Closed Itemset Mining, FCIM) 和形式概念分析问题^[31-32] (Formal Concept Analysis, FCA)。很多现有的极大二分团枚举方法都受

益于这些相关问题的优化思路，部分相关工作将极大二分团枚举问题规约到这些相关问题进行求解。我们将在 1.3 节详细介绍这些方法。相应地，对二分图中极大二分团枚举问题的优化研究间接地为解决这些问题提供了思路。(3) 衍生问题：随着二分图中极大二分团问题研究的深入，近年来，极大二分团枚举方法被应用于最大边二分团搜索^[2, 4] (Maximum Edge Biclique Search, MEB) 和最大平衡二分团搜索^[33] (Maximum Balanced Biclique Search, MBB) 等问题中，并衍生出相似二分团枚举问题^[34]、(p, q) 二分团枚举问题^[17-19, 35]、公平极大二分团枚举问题^[36] 和二分团渗透社区^[37] 等相关问题。一些研究将衍生问题推广到不确定图^[38]、带符号图^[39-40]、带权重图^[41-42] 和动态图^[43] 的场景中。这些衍生问题都基于极大二分团枚举算法，针对各自目标二分团设计特定的剪枝与优化方法，为进一步扩展和拓展极大二分团的应用领域提供了可能性。总之，二分图中极大二分团枚举问题在图论研究中占据重要地位。深入研究该问题将对其他相关问题的研究产生辐射带动作用。

然而，大规模二分图中极大二分团枚举问题面临着严峻的挑战。具体而言，这些挑战主要表现在以下三个方面：(1) 搜索空间大。极大二分团枚举问题是一个 NP-hard 问题，随着二分图规模的增大，其搜索空间呈现出指数级增长的趋势^[44]。然而，在大数据时代的背景下，二分图的规模不断膨胀。以电子商务为例，根据中国商务部的电子商务报告^[45]，2022 年全国电子商务交易额达到 43.83 万亿元，与上一年相比增长了 3.5%。此外，仅在 2020 年，阿里巴巴企业单日的交易次数已超过 1 亿次^[2]。不断增长的二分图规模使得极大二分团枚举问题的搜索空间进一步加大。(2) 计算不规则。与其他图计算问题相似，极大二分团枚举问题主要涉及集合运算。然而，真实世界中的二分图存在着不规则性^[46]，即每个顶点的邻居数量存在较大的差异，符合幂律分布的特征。这意味着只有少数顶点具有大量的邻居连接，而大多数顶点的邻居数量相对较小。因此，每次集合运算所涉及的顶点数量也各不相同。目前的方法忽视了集合运算的不规则性，导致设计出的枚举方法无法充分发挥其潜力。(3) 负载不均匀。主流的极大二分团枚举方法依赖于集合枚举树的实现^[6, 15, 47-48]。为了进一步提升枚举速度，研究者们尝试在分布式系统或多核系统中设计并行的极大二分团枚举算法^[49-50]。具体做法是将枚举树拆解成多个子枚举树，并利用充足的计算资源并行处理这些子枚举树。然而，与其他图计算问题不同的是，每个极大二分团所包含的顶点数量是不确定的，这导致子枚举树的高度无法确定，进而增加子枚举树之间的负载差异，加大了并行扩展的难度。

综上所述，二分图中的极大二分团枚举问题在电子商务中的虚假交易检测、社交网络推荐、生物医学中的基因分析等热门场景中有着广泛的应用。同时在图论领域扮演着基础问题的关键角色，并在近年来衍生出许多相关问题，成为学术研究热点。然而，在处理大规模二分图时，极大二分团枚举算法面临着挑战，包括搜索空间巨大、计算不规则以及负载分布不均等难题。因此，极大二分团问题受到工业界和学术界的广泛关注。

1.2 研究问题

本节详细介绍了本文的研究问题，包括对二分图中极大二分团枚举问题的形式化定义，以及该问题的基本求解方法。

1.2.1 问题定义

在问题定义之前，我们首先介绍图论领域的一些基础概念，并提供了随后频繁使用的符号及其含义，如表 1.1 所示。

表 1.1 本文使用的符号及含义

符号	含义
$G(U, V, E)$	一个无向二分图 G ，其中 U 和 V 是两个不相交的顶点集合， E 是二分图的边集合且 $E \subseteq U \times V$ 。
u, v	表示二分图 G 中的顶点。其中顶点 u 属于集合 U ，顶点 v 属于集合 V 。
$N(v)$	表示顶点 v 的邻居顶点集合，即 $N(v) = \{u \mid (u, v) \in E\}$ 。
$N_2(v)$	表示顶点 v 的二跳邻居顶点集合，即 $N_2(v) = \bigcup_{u \in N(v)} N(u) - \{v\}$ 。
$\Delta(v)$	表示顶点 v 的度数，即 $\Delta(v) = N(v) $ 。
$\Gamma(X)$	表示顶点集 X 内顶点的共同邻居，即 $\Gamma(X) = \bigcap_{v \in X} N(v)$ 。
$\Upsilon(X)$	表示顶点集 X 内顶点的合并邻居，即 $\Upsilon(X) = \bigcup_{v \in X} N(v)$ 。
$\Delta(X)$	表示顶点集 X 内顶点的最大度数，即 $\Delta(X) = \max_{u \in X} N(u) $ 。
$\Delta_2(X)$	表示顶点集 X 内顶点的最大二跳度数，即 $\Delta_2(X) = \max_{u \in X} N_2(u) $ 。

续下页

符号	含义
X_v^+, X_v^-	表示顶点集 X 根据顶点 v 划分成的两个子集。给定一个顶点顺序， X_v^+ 包含所有顶点比 v 更大的顶点（顺序在 v 之后的顶点），即顶点 v 的尾部顶点； X_v^- 包含包括 v 顶点在内的所有顶点比 v 更小的顶点（顺序在 v 之前的顶点），即顶点 v 的头部顶点。
L, R, C	L, R 和 C 指三个两两不相交的顶点集，其中 L 是集合 U 的子集， R 和 C 是集合 V 的子集。 L, R 和 C 共同构成一个枚举树节点，其中 (L, R) 表示枚举树节点对应的二分团， C 表示用于生成新枚举树节点的候选顶点。对于二分团 (L, R) ， L 和 R 分别表示二分团的左部顶点集和右部顶点集。
$N_L(v)$	表示顶点 v 的局部邻居。对于对应二分团 (L, R) 的枚举树节点， $N_L(v) = L \cap N(v)$ 。
\vec{v}	对于一个枚举树节点， \vec{v} 表示用于生成该枚举树节点的候选顶点，即枚举树中从父节点到子节点的边上的遍历候选顶点。
α, δ, β	对于一棵极大二分团枚举树， α 表示枚举树中产生的极大二分团的数量， δ 表示枚举树中产生的其他二分团的数量， β 表示枚举树中二分团的总数量。可知 $\beta = \alpha + \delta$ 。

二分图中的极大二分团枚举问题是在一个无向无权二分图 $G(U, V, E)$ 中的特定的图挖掘问题。随后，我们定义二分图、二分团、极大二分团以及极大二分团枚举问题。

定义 1.1. (二分图) 二分图 (Bipartite graph) $G(U, V, E)$ 是一种特殊的图结构，包含两个不相交的顶点集合 U 和 V ，以及连接这些顶点的边集合 E 。在二分图中，边集 E 中的每条边连接的两个顶点分属于不同的顶点集合，即 $E \subseteq U \times V$ 。

定义 1.2. (二分团) 二分团 (Biclique) 是二分图 $G(U, V, E)$ 中的稠密二分子图 (L, R, E') 。其中 $L \subseteq U$, $R \subseteq V$, $E' = L \times R \subseteq E$ 。为了方便，下文中我们直接用顶点集对 (L, R) 表示二分团。

定义 1.3. (极大二分团) 极大二分团 (Maximal Biclique) 是二分图 G 中的一个二分团，且该二分团不能再添加其他顶点使其成为更大的二分团。

定义 1.4. (极大二分团枚举问题) 极大二分团枚举问题 (Maximal Bipartite Clique Enumeration, MBE) 的目标是无重复、无遗漏地枚举二分图中的全部极大二分团。

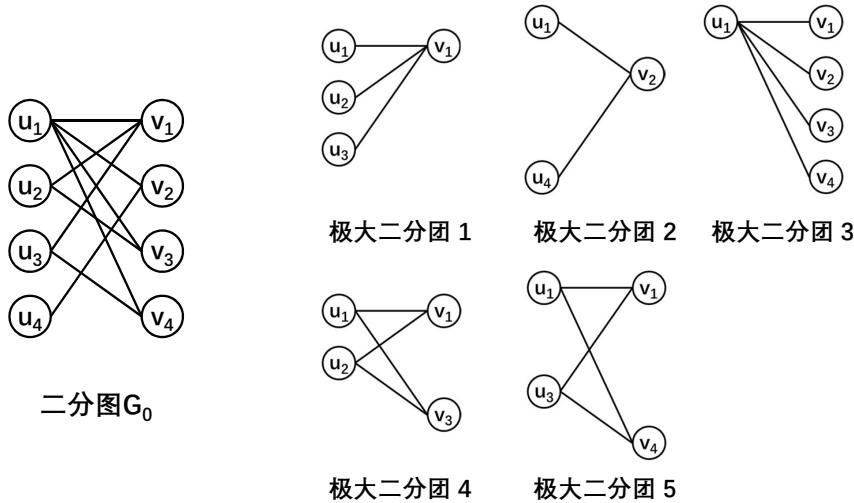


图 1.3 二分图中的极大二分团枚举问题示例

例 1.1. 图 1.3 给出了一个二分图中的极大二分团枚举问题的示例。其中左图是一个具有 8 个顶点，9 条边的二分图 G_0 ，右图展示了二分图中的全部极大二分团，共 5 个。极大二分团枚举问题即无重复、无遗漏地枚举二分图 G_0 中的全部 5 个极大二分团。

考虑到现有方法在处理大规模二分图时效率低下，本研究从剪枝能力、数据结构和并行实现等方面入手，探索面向大规模二分图场景的高效极大二分团枚举方法。

1.2.2 基本求解方法

在本节中，我们详细描述了主流的基于集合枚举树的极大二分团问题基本求解方法。我们首先介绍了极大二分团问题中的集合枚举树的定义，接着给出了基于该集合枚举树的极大二分团枚举基本算法，最后分析了该算法的复杂度。

1.2.2.1 集合枚举树介绍

集合枚举树 (Set Enumeration Tree, SE tree) 是一种用于有序地枚举特定集合全部子集的数据结构，即枚举该集合的幂集^[51]。它为解决搜索空间为特定集合幂集的子集的问题提供了一种完整且无冗余的搜索技术。具体而言，集合枚举树如图 1.4 所示，根节

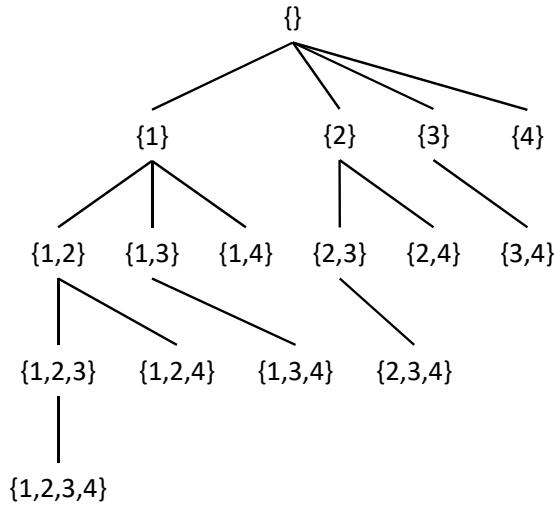


图 1.4 对于集合 $P\{1, 2, 3, 4\}$ 的集合枚举树

点表示空集，每个节点对应一个子集，其子节点代表在该子集基础上添加一个新元素所得到的子集。对于二分图中的极大二分团枚举问题而言，每个二分团的集合 R 即为二分图中集合 V 的一个独特子集。因此，通过引入集合枚举树可以无重复、无遗漏地枚举全部可能的极大二分团，进而有效地求解极大二分团枚举问题。

对于应用于极大二分团枚举问题的集合枚举树，我们从以下三个角度对其进行了规范化描述：

- 节点结构：每个树节点为一个三元组 (L, R, C) 。在一个二分图 $G(U, V, E)$ ，集合 L 是集合 U 的子集，集合 R 和集合 C 是集合 V 的两个不相交子集。集合 L 和 R 构成一个二分团，其中集合 L 包含左部顶点，集合 R 包含右部顶点。集合 C 包含用于扩展集合 R 的候选顶点。
- 节点生成：枚举树从根节点 (U, \emptyset, V) 开始遍历。对于当前节点 (L, R, C) ，枚举树按顺序访问集合 C 中的每个候选顶点 v' 以生成一个新的节点 (L', R', C') 。集合 L' 包含集合 L 和集合 $N(v')$ 中的共同顶点。集合 R' 包含集合 R 中的顶点、顶点 v' 以及集合 C 中与集合 L' 内顶点均相连且未被访问的顶点。集合 C' 包含集合 C 中与集合 L' 内顶点部分相连的且未被访问的顶点。
- 节点检查：当且仅当集合 L' 的共同邻居等于集合 R' ，即 $\Gamma(L') = R'$ 时，节点 (L', R', C') 通过节点检查，输出一个极大二分团。

此外，一些研究^[15, 48]在每个枚举节点中额外引入集合 Q 作为辅助节点检查的工具，构成四元组 (L, R, C, Q) 。其中集合 Q 于存储已访问的候选顶点，以帮助检查节点是否对应非极大二分团。具体而言，当集合 Q 中存在任意一个顶点 v_q ，并且它的邻居包含了集合 L 中的所有顶点时，根据定义 1.3 我们可以推断当前节点对应的二分团 (L, R) 可以添加顶点 v_q 构成新的二分团，即 $(L, R \cup \{v_q\})$ ，从而我们可以判定当前节点对应一个非极大二分团。然而，使用集合 Q 需要额外的存储和计算开销。幸运的是，我们观察到可以通过访问 L 中的任意顶点 u_l 的邻居 $N(u_l)$ 来高效地替代集合 Q 的作用。具体而言，当集合 $N(u_l)$ 中存在一个不在 R 集合中的顶点 v^* ，并且它的邻居包含了集合 L 中的所有顶点时，我们可以推断当前节点对应的二分团 (L, R) 可以添加顶点 v^* 构成新的二分团，从而判定当前节点对应一个非极大二分团。因此，在枚举树的介绍和相关算法中，我们不再引入集合 Q 。

1.2.2.2 基于集合枚举树的极大二分团枚举基本算法

结合上一节对集合枚举树的定义与描述，我们给出了基于集合枚举树的极大二分团枚举基本算法。

算法 1.1 基于集合枚举树的极大二分团枚举算法

输入：二分图 $G(U, V, E)$

输出：所有极大二分团

```

1: biclique_search_basic( $U, \emptyset, V$ );
2: procedure biclique_search_basic( $L, R, C$ ) :
3:   for  $v' \in C$  do
4:      $L' \leftarrow L \cap N(v')$ ;  $R' \leftarrow R$ ;  $C' \leftarrow \emptyset$ ;
5:     for  $v_c \in C$  do
6:       if  $L' \cap N(v_c) = L'$  then
7:          $R' \leftarrow R' \cup \{v_c\}$ ;
8:       else if  $L' \cap N(v_c) \neq \emptyset$  then
9:          $C' \leftarrow C' \cup \{v_c\}$ ;
10:      end if
11:    end for
12:    if  $\Gamma(L') = R'$  then
13:      输出极大二分团  $(L', R')$ ;
14:      biclique_search_basic( $L', R', C'$ );
15:    end if
16:     $C \leftarrow C \setminus \{v'\}$ ;
17:  end for
18: end procedure

```

算法 1.1 总结了基于集合枚举树的极大二分图枚举算法的基本枚举过程。具体而言，该算法从根节点 (U, \emptyset, V) 开始，递归地调用 `biclique_search_basic` 过程（第 1 行）。过程 `biclique_search_basic` 接收一个枚举树节点作为输入，即该节点对应的集合 L , R 和 C （第 2 行）。在处理当前枚举节点时，该过程会逐个遍历 C 中的顶点 v' （第 3 行），然后根据集合枚举树的节点生成规则生成新节点 (L', R', C') （第 4-11 行）。随后，过程按照节点检查规则对新生成的节点 (L', R', C') 进行检查（第 12 行）。如果该节点对应一个极大二分团，则输出该二分团（第 13 行），并递归地调用过程 `biclique_search_basic` 以节点 (L', R', C') 为根节点继续探索子枚举树（第 14 行）；否则，我们知道该节点对应非极大二分团，跳过该节点。为保证 C 中的顶点都未被访问，过程会及时从 C 中移除已访问的顶点 v' （第 16 行）。我们用下面的例子对该算法进行说明。

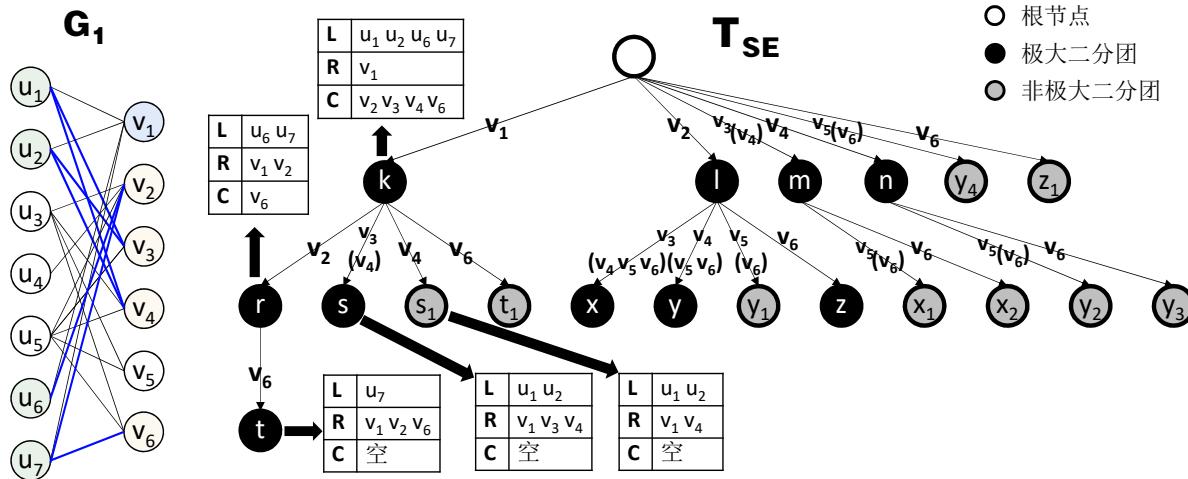


图 1.5 算法 1.1 在二分图 G_1 上的集合枚举树

例 1.2. 图 1.5 展示了算法 1.1 在二分图 G_1 上的集合枚举树 T_{SE} 。¹ 我们从根节点开始，通过深度优先搜索逐个遍历候选顶点，递归地搜索子空间。首先，我们通过遍历顶点 v_1 生成节点 k 。按照算法 1.1 中第 4-11 行的计算方法，我们可以计算得到 $L_k = N(v_1) = \{u_1, u_2, u_6, u_7\}$, $R_k = \{v_1\}$, $C_k = \{v_2, v_3, v_4, v_6\}$ 。根据节点检查规则，因为 $\Gamma(L_k) = \{v_1\} = R_k$ ，所以节点 k 输出一个极大二分团并继续探索以节点 k 为根节点的子枚举树。为便于观察，我们在图 G_1 中标记了节点 k 中的顶点，即 L_k , R_k 和 C_k 中的全部顶点，并标出了集合 L_k 与集合 C_k 之间的边。

¹ 为了方便比较，全文中具有相同字母标识的节点在枚举树中共享相同的集合 L ，只有没有下标的节点会输出极大二分团。例如，节点 s 和节点 s_1 具有相同的集合 L ，但只有节点 s 会输出一个极大二分团。我们使用集合的下标来表示该集合隶属于哪个节点。例如 L_s 表示节点 s 的集合 L 。

接下来，节点 k 遍历顶点 v_2 生成节点 r 。同理，我们可以计算得到 $L_r = N(v_2) \cap L_k = \{u_3, u_4, u_5, u_6, u_7\} \cap \{u_1, u_2, u_6, u_7\} = \{u_6, u_7\}$ ， $R_r = R_k \cup (C_k \cap \Gamma(L_r)) = \{v_1\} \cup (\{v_2, v_3, v_4, v_5\} \cap \{v_1, v_2\}) = \{v_1, v_2\}$ 。集合 C_r 中仅包含顶点 v_6 ，因为顶点 v_3 , v_4 和 v_5 不与集合 L 中的任何顶点相连。

继续这个过程，我们可以计算得到节点 s 以及节点 s_1 。节点 s 对应二分团 $(\{u_1, u_2\}, \{v_1, v_3, v_4\})$ ，节点 s_1 对应二分团 $(\{u_1, u_2\}, \{v_1, v_4\})$ 。根据节点检查规则，因为 $\Gamma(L_{s_1}) = \{v_1, v_3, v_4\} \neq R_{s_1} = \{v_1, v_4\}$ ，所以节点 s_1 对应一个非极大二分团。具体地，与节点 s 相比，节点 s_1 不能用 v_3 来扩展该节点中的集合 R_{s_1} 。这是因为在生成节点 s_1 时，根据深度优先搜索的规则，顶点 v_3 已被访问并用于生成节点 s 。因此，在节点检查之后，我们删除了节点 s_1 。同理，其他节点可以类似地生成。

在算法 1.1 的基础上，现有的基于枚举树的极大二分团枚举算法的优化方法主要包括改变节点候选顶点的遍历顺序^[6, 15, 47-48]、设计剪枝方法以提前裁剪产生非极大二分团的节点^[15, 47-48]，以及并行优化^[49-50]。在 2.1 节中，我们将对上述优化方法进行详细说明，并介绍它们在实际应用中的效果。

1.2.2.3 算法复杂度分析

结合算法伪代码，我们从时间复杂度和空间复杂度两个方面对算法 1.1 进行分析。

时间复杂度：我们首先分析枚举树中每个节点的计算时间，随后分析枚举树中的枚举节点数量，最终得到算法 1.1 的时间复杂度。平均而言，对于每个节点 (L', R', C') 的计算包括节点生成（第 4-11 行）和节点检查（第 12 行）两个部分。由于集合 C 中最多包含 $|V|$ 个顶点，且每个顶点的集合交集运算需要 $O(\Delta(V))$ 的时间，因此节点生成过程的时间复杂度为 $O(|V|\Delta(V))$ 。而节点检查过程中，我们可以通过只访问二分图中的每条边一次来获取 $\Gamma(L')$ 的值，因此节点检查的时间复杂度为 $O(|E|)$ （或 $O(|V|\Delta_{avg}(V))$ ）。综上，每个节点的计算时间为 $O(|V|\Delta(V))$ 。为了量化算法的计算时间，我们用 β 来表示枚举树中节点的数量。最终，算法的时间复杂度为 $O(|V|\Delta(V)\beta)$ 。

空间复杂度：由于算法 1.1 按照深度优先的方式进行搜索，我们可以对枚举树中每个节点占用的空间进行分析，并结合枚举树的高度以及输入二分图所占用的空间，得到算法的空间复杂度。对于每个节点 (L', R', C') ，集合 L' 最多包含 $\Delta(V)$ 个顶点，集合 R' 和 C' 最多包含 $|V|$ 个顶点。在二分图中，集合 V 内顶点的数量通常远高于任何单个顶

点的度数，因此每个节点的空间开销为 $O(\Delta(V) + |V|) = O(|V|)$ 。在递归过程中，节点的集合 L 内的顶点数量不断减少，因此我们可以确定枚举树的高度为 $O(\Delta(V))$ 。考虑到二分图 $G(U, V, E)$ 需要占用 $O(|U| + |V| + |E|) = O(|E|)$ 的空间，最终算法的空间复杂度为 $O(|E| + |V|\Delta(V))$ 。

1.3 国内外研究现状

作为图论中的基础问题，关于极大二分团枚举问题的研究最早可追溯到 1962 年^[52]。在过去的几十年中，国内外学者对二分图中的极大二分团枚举问题进行了深入的研究，并取得了一系列重要的成果。

方法类别

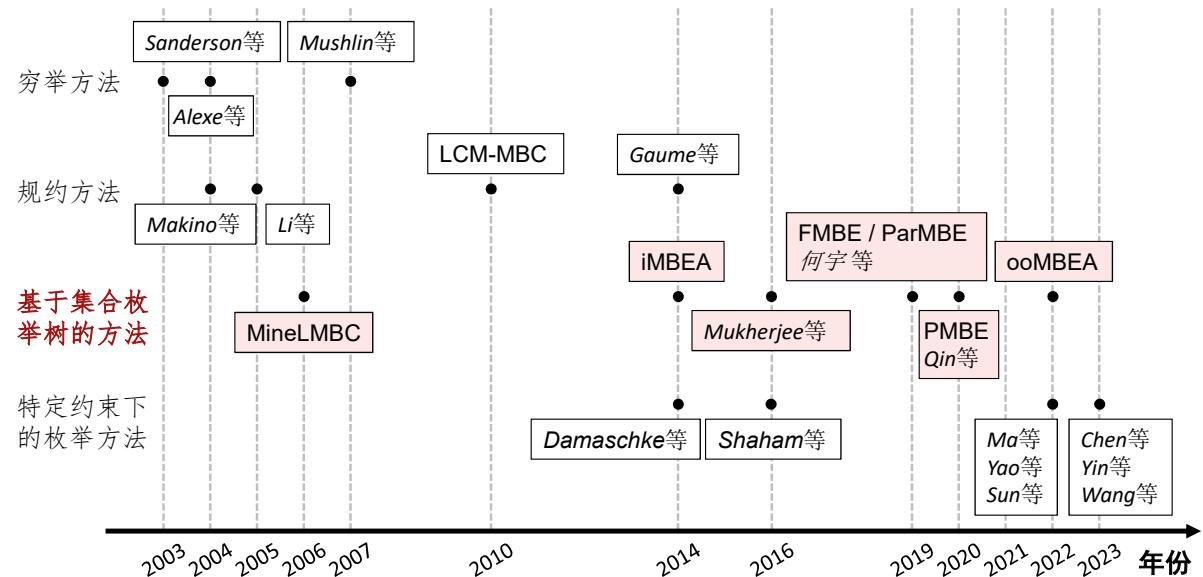


图 1.6 极大二分团枚举领域国内外研究情况梳理

我们在图 1.6 中对国内外相关研究进行了梳理和总结，将其分为四类。早期的研究主要采用穷举方法和规约方法，而当前的主流工作采用基于集合枚举树的方法。同时，近期的研究涉及到特定约束下的枚举方法。接下来，我们将详细介绍这四类方法。

1.3.1 穷举方法

解决极大二分团枚举问题的一种直观方法是使用穷举法生成并存储所有可能的二分团，然后对这些二分团进行极大性检测。早期工作以此观察为基础，提出了关于极大

二分团枚举问题的暴力穷举求解方法。Sanderson 等人提出了一种迭代算法，逐步构建出全部可能的二分团^[53]。具体而言，该算法首先选取二分图中一个顶点集中的单一顶点作为初始二分团。然后，它遍历其他顶点，逐一将其加入到当前的二分团中构成新的二分团，并对所得的新的二分团进行极大性检测。随后，Alexe 等人提出了一种共识算法，用于生成极大二分团。该算法将每个顶点及其邻居作为初始候选二分团，并通过不断进行共识分析来扩展新的候选二分团^[44]。该算法对于新生成的候选二分团进行判断，以确保其为极大二分团且不重复。整个过程会持续进行，直到候选二分团集合不再增加。此外，Mushlin 等人利用集合扩展操作构建二分团，并利用哈希表进行极大性检测。他们引入了评价指标，以支持在枚举过程中对二分团的优先级进行排序^[54]。尽管这些方法能够解决极大二分团枚举问题，但它们没有充分考虑搜索空间的剪枝优化，因此在计算过程中不可避免地会产生大量非极大二分团，导致计算和存储开销增加。同时，由于时间复杂度较高，这些方法无法应用于规模较大的二分图上的极大二分团枚举问题的求解。

1.3.2 规约方法

考虑到极大二分团枚举问题和其他问题的联系，学者们尝试将二分图中的极大团枚举问题规约到其他问题，并用其他问题中的已有方法进行求解。(1) 规约到极大团枚举问题。Makino 等人观察到二分图上的极大二分团枚举问题可以通过图膨胀的方式映射到通用图上的极大团枚举问题进行求解^[55]。具体而言，他们将二分图中的每个顶点集合内的任意两个顶点添加一条边，从而膨胀原二分图中的每个极大二分团，使其对应于膨胀后通用图中的一个极大团。通过这种方式，二分图上的极大二分团枚举问题可以被等价地转化为通用图上的极大团枚举问题。然而，需要注意的是，尽管通用图上的极大团枚举问题已经在学术界得到广泛研究，并存在一些有效的极大团枚举方法^[23, 25, 27, 56-57]，但直接应用这些方法并不实用。这是因为图的膨胀会引入大量的新边，从而导致计算难度的增加，并严重影响了计算性能。(2) 规约到闭频繁项集挖掘问题。Zaki 等人注意到事务数据库中的闭频繁项集与二分图中的极大二分团存在一一对应的关系^[29]。基于这个对应关系，Li 等人指出了数据挖掘领域中一些经典算法对于极大二分团枚举问题的解决具有帮助作用^[58]，例如 FPclose^[59] 和 LCM^[60] 等算法。通过计算闭频繁项集，可以得到极大二分团中的一个顶点集合，进而计算另一个顶点集合以构成二分团。Li 等人还在 LCM 算法的基础上提出了 LCM-MBC 算法来进行极大二分团枚举^[61]。然而，对于闭

频繁项集挖掘算法而言，在大规模事务性数据库中将支持度设置为 1 进行闭频繁项集挖掘是非常困难的^[15]。（3）规约到形式概念分析问题。Gaume 等人指出二分图中的极大二分团与形式概念分析问题中的形式概念存在等价关系^[62]。因此，现有的形式概念分析相关的工作^[31-32, 63] 可以用于极大二分团枚举问题的求解。然而，形式概念分析相关的工作主要在二进制数据上展开^[63]，即源数据通过位图方式存储并以位运算的方式开展计算。由于现实中的二分图是稀疏的，位图存储方式会带来大量的内存开销，这导致前沿的形式概念分析算法只能在小图上进行^[31-32]，在大规模二分图中会带来内存超过限制的问题。综上所述，规约方法在大规模二分图场景下是低效的。

1.3.3 基于集合枚举树的枚举方法

如 1.2.2 节所述，目前，主流且高效的极大二分团枚举算法基于集合枚举树^[51] 的数据结构实现。2006 年，Liu 等人提出 MineLMBC 算法^[6]，首次引入集合枚举树，采用分治法求解极大二分团枚举问题。此后，基于枚举树的极大二分团枚举方法得到了不断优化。为了减少搜索空间、提升计算效率，研究者们相继提出了不同的优化技术。Zhang 等人提出 iMBEA 算法^[15]，采用了顶点度数升序排序和排除顶点集等技术，以减少枚举时间。Das 等人提出 FMBE 算法^[50]，通过主动计算顶点的二跳邻居作为候选顶点集合，加速了枚举过程。Abidi 等人提出 PMBE 算法^[47]，借鉴了极大团枚举问题中的枢纽顶点思路，利用枢纽顶点进行剪枝。Chen 等人提出 ooMBEA 算法^[48]，引入了单边排序和批量枢纽顶点剪枝等技术，进一步减少了枚举时间。为了进一步提高效率，研究者们设计了并行极大二分团枚举算法。Mukherjee 等人利用 MapReduce 框架实现了分布式极大二分团枚举算法 CDFS^[49]。然而，大量的跨节点通信开销导致该算法效率较低。Das 等人提出多核算法 ParMBE^[50]，但该算法的并行性受到计算核心数量的限制。此外，国内的 He 等人提出优化的 sMBEA 算法^[64-65]，Qin 等人利用栈的特性实现了 EMBE 算法^[66]。然而，相较于主流算法（如 FMBE、PMBE、ooMBEA），这些算法存在较大的性能差距。综上所述，基于枚举树的极大二分团枚举方法是目前最高效的一类极大二分团枚举方法。

1.3.4 特定约束下的枚举方法

除了上述研究，一些学者对输入的二分图以及输出的极大二分团进行了特定的约束。Damaschke 等人提出了一种输出敏感的极大二分团枚举算法^[67]，该算法基于二分图

中顶点度数的倾斜分布。然而，该算法对输入二分图的要求较为严格，仅适用于特定情况下的求解，无法很好地适用于一般二分图。Shaham 等人提出了基于聚类的方法^[68]，在二分团搜索的子空间中利用蒙特卡洛方法获取随机种子并将其扩展为极大二分团。然而，这种方法只能枚举部分的极大二分团，无法覆盖所有可能的极大二分团。针对动态二分图，Ma 等人提出了一种有效保持最大二分团性质的框架，该框架可以在动态变化的二分图上进行高效的极大二分团枚举^[43]。此外，近年来的一些研究工作定义了特殊类型的极大二分团，并对这些特殊类型的二分团进行枚举。例如，Yao 等人定义了极大相似二分团^[34]，Sun 等人定义了极大平衡有符号二分团^[39]和最大有符号二分团^[40]，Chen 等人用多个极大二分团并集定义了二分团渗透社区^[37]，Yin 等人定义了极大二分团的公平性^[36]，而 Wang 等人定义了不确定图场景下的极大二分团^[38]。这些算法都基于极大二分团枚举算法，并在枚举过程中对输出进行适当的约束和限制，以适应特定的问题需求。然而，这些算法主要关注特定约束场景下的搜索空间剪枝与优化，难以用于加速传统的极大二分团枚举问题。综上所述，极大二分团枚举在这些研究中发挥着基础性作用。

1.4 研究挑战

尽管在极大二分团枚举领域已经有很多出色的工作，但是在处理大规模二分图时，已有方法的计算性能仍然有很大的提升空间。本节将指出主流的基于集合枚举树的枚举方法所面临的三个共性问题，并介绍解决这些问题所面临的具体挑战。

1.4.1 搜索空间大，剪枝方法欠佳

极大二分团枚举问题具有搜索空间大的特点。常见的图算法，如深度优先搜索^[69]、广度优先搜索^[70]、最小生成树^[71]、最短路径等算法^[72]，其搜索空间随着图的规模线性增长。而常见的图模式挖掘算法的目标子图往往只包含少量顶点^[73-79]，例如三角计数问题中目标子图仅包含 3 个顶点^[80]。相比之下，极大二分团枚举问题的搜索空间更大，因为它随着二分图中顶点数量的指数级增长，并且目标子图（即极大二分团）中的顶点数量相对较多。为了应对搜索空间巨大的挑战，研究人员提出了各种优化技术，旨在减少搜索空间中产生非极大二分团的无效枚举节点，进而减少枚举时间。然而，由于搜索空间的规模庞大，现有的优化方法往往难以完全覆盖所有无效节点。具体而言，通过 2.4.2 节

的实验，我们观察到现有的最新算法 ooMBEA 在 Github 数据集上需要检查并删除比极大二分团数量多 26 倍的产生非极大二分团的无效枚举节点。这些无效枚举节点带来大量的节点检查开销，严重降低了计算性能。因此，如何设计高效的剪枝方法来裁剪巨大搜索空间仍然是一个关键的挑战。

1.4.2 计算不规则，静态结构低效

极大二分团枚举问题具有计算不规则的特点。与其他图计算问题类似，在真实世界中，二分图的顶点邻居数量存在较大差异，导致每次计算涉及的顶点数量不同，即计算不规则^[46]。为了高效地解决这类问题，研究者们提出了多种存储结构来表示图的邻接关系。常用的存储结构包括位图^[31-32, 60-61, 63]、邻接表^[15, 47-48]和哈希表^[50]。不同的存储结构适用于不同场景。例如，位图结构采用位运算，具有高效的计算能力，但在稀疏图场景下会占用更多的存储资源，因此适用于稠密小图；邻接表精确地存储每个顶点的邻居信息，在处理稀疏大图时占用较少的内存，但计算过程需要执行大量的比较运算，其运行时间与顶点个数成正比，会导致计算相对低效，因此适用于稀疏大图；哈希表具有灵活性和便于快速查找的特点，可以快速判断任意两个顶点之间的连接关系，但相比邻接表，它需要更多的存储空间并且访问方式更为随机。然而，目前的研究往往采用固定的数据结构来存储顶点的邻居信息，未能充分发挥不同数据结构的优势。因此，在枚举过程中如何动态选择合适的存储结构，发挥计算潜力，是一个关键挑战。

1.4.3 负载不均匀，并行扩展性差

极大二分团枚举问题具有负载不均匀的特点，限制了问题的并行扩展能力。为了进一步提高问题求解的效率，研究人员尝试设计并行算法来处理极大二分团枚举问题^[49-50, 64]。这些算法将整个集合枚举树分解成多个子枚举树，然后利用分布式系统或多核 CPU 的大量计算资源来并行处理这些子树。然而，由于不同子枚举树的极大二分团的大小不同，导致计算负载之间存在较大差异。因此，即使有大量计算资源可用，计算任务的运行时间仍然受限于最耗时的负载。GPU 作为一种专门用于并行计算的硬件设备，由于其内部拥有大量的计算单元，非常适合处理并行任务。然而，由于 GPU 和 CPU 在体系结构、内存层次结构以及编程模型等方面存在较大差异，现有的极大二分团枚举算法无法直接迁移到 GPU 系统中。尽管 GPU 被广泛用于加速相关的图算法，如极大

团枚举^[25, 81-82]和图模式挖掘^[75, 83-85]，但在 GPU 上进行极大二分团枚举问题仍然具有挑战性。具体来说，GPU 上的极大二分团枚举问题面临着与极大二分团枚举问题类似的问题，许绍显等人指出 GPU 加速极大团枚举问题的研究极为有限^[28]。即使最新的基于 GPU 的极大团枚举算法 GBK^[25]获得的性能，也仅与 CPU 上的单线程串行算法相当。GPU 上的子图枚举问题中被枚举子图通常仅包含少量顶点，而极大二分团通常包含大量顶点，因此带来更加严重的负载不均问题，导致最新的基于 GPU 的 GPM 框架 G²Miner^[75]与 GraphSet^[79]中的优化无法直接解决 GPU 上进行极大二分团枚举面临的负载不均匀问题。因此，如何实现负载均衡并突破现有算法在并行能力方面的限制，设计基于 GPU 的并行极大二分团枚举方法是一个重要挑战。

1.5 研究内容

针对上一节提到的三个方面的研究挑战，本文分别从这三个角度对大规模二分图场景下的极大二分团枚举问题进行深入研究，并相应地提出了三个独立高效的算法，相较于现有方法均有明显的性能提升。具体研究内容如下：

第一，针对搜索空间大、现有剪枝方法效率低下的挑战，本文提出了主动的极大二分团枚举算法 AMBEA (Aggressive MBE Algorithm)。我们观察到，现有算法为了保证正确性，仅允许使用部分顶点生成新的枚举节点，导致产生大量非极大二分团；同时，现有的剪枝方法总是在节点检查后被动执行，限制了剪枝能力。因此，我们设计了以下两种方法：(1) 主动的集合枚举树，允许利用全部顶点生成新的枚举节点，突破了现有集合枚举树生成规则的限制，并通过优化父子节点之间的联系来删除重复的二分团；(2) 主动的顶点合并剪枝方法，在枚举节点生成的同时，主动合并具有相同局部邻居的顶点，提升剪枝效率。实验结果显示，AMBEA 算法相较于最优的其他现有算法，可以压缩搜索空间 9.0 倍，并获得最多 5.3 倍的性能提升。

第二，针对计算不规则、现有静态数据结构低效的挑战，本文提出了自适应的极大二分团枚举算法 AdaMBE (Adaptive MBE)。我们观察到，现有算法在计算过程中，包含活跃顶点的计算子图在动态变化，导致冗余的内存访问和低效的集合运算性能。因此，我们设计了以下两种方法：(1) 基于局部计算子图的优化方法，利用动态建立的计算子图结构完成节点生成等核心计算操作，减少了在局部计算子图外的无效顶点访问，提升

计算效率；（2）基于位图的动态子图方法，利用枚举过程中计算子图小而稠密的特点，在这些小型算子图上采用位图技术，使得集合运算变得更加规则高效，从而加速了整个计算过程。实验结果显示，AdaMBE 算法相较于最优的其他现有算法获得了最多 49.7 倍的性能提升，并且能够应用于超过百亿个极大二分团的大数据集。

第三，针对负载不均匀、现有基于 CPU 的算法并行扩展性差的挑战，本文提出了基于 GPU 的极大二分团枚举算法 GMBE (GPU-based MBE)。我们观察到，将现有算法迁移到 GPU 上主要面临内存短缺、线程分歧和负载不均三方面问题。因此，我们设计了以下三种方法：（1）基于枚举节点重用的迭代方法，通过重用父节点内存生成子节点，避免为大量子节点动态分配内存；（2）局部邻居数量感知的剪枝方法，通过对局部邻居数量这一中间结果进行批量比较，在剪枝的同时最小化线程分歧问题；（3）负载感知的任务调度方法，通过对运行时的大任务根据枚举节点信息进行进一步拆分，实现了细粒度的负载均衡。实验结果显示，GMBE 算法相比于最优并行算法获得了最多 70.6 倍的性能提升。

1.6 本文的组织结构

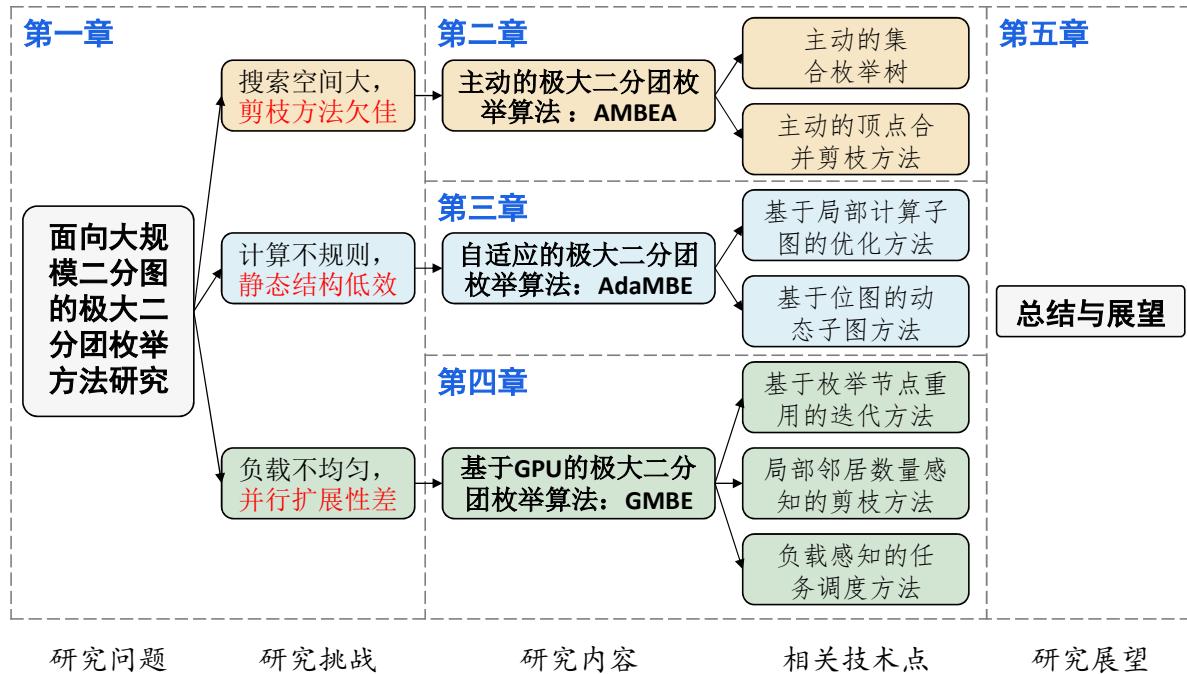


图 1.7 本文的组织结构图

图 1.7 展示了本文的研究内容和结构，共五个章节。

第一章作为绪论，首先介绍了研究背景、研究问题以及国内外的研究现状。本文主要研究大规模二分图场景下的极大二分团枚举问题。针对剪枝能力、数据结构和并行实现等三个方面的挑战，本文提出三个独立且高效的极大二分团枚举算法，分别对应于论文的第二、第三和第四章。

第二章提出了 AMBEA 算法，主要包括主动的集合枚举树和主动的顶点合并剪枝方法两个核心技术点。通过深度优化剪枝方法，AMBEA 算法在搜索空间和性能方面相较于现有算法有明显提升。

第三章提出了 AdaMBE 算法，主要包括基于局部计算子图的优化方法和基于位图的动态子图方法两个核心技术点。AdaMBE 算法根据极大二分团枚举任务中计算子图动态变化的特点，采用自适应的数据结构，在性能上比现有算法有较大提升。

第四章提出了 GMBE 算法，主要包括基于枚举节点重用的迭代方法、局部邻居数量感知的剪枝方法以及负载感知的任务调度方法三个核心技术点。相比于现有算法只能在 CPU 上实现，GMBE 在 GPU 上实现了较大幅度的性能提升。

第五章对全文进行总结，并展望未来的研究方向。

2 主动的极大二分团枚举算法

针对极大二分团枚举问题搜索空间大、现有剪枝方法效率低下的问题，本章整理了相关工作中的优化方法，并指出其中的共性问题作为本章的研究动机。具体而言，我们发现通用的集合枚举树存在结构限制，即仅允许利用部分顶点生成新的枚举节点，导致产生大量包含非极大二分团的无效枚举节点；同时，已有的剪枝方法仅在当前节点通过节点检查后被动执行，剪枝效率受限。为解决这些问题，本章提出了以下优化方法：首先，针对集合枚举树的结构限制，本章提出了主动的集合枚举树。该枚举树主动地将枚举树中每个节点对应的二分团扩展为极大二分团，并利用父子节点间的联系删除枚举树中的重复二分团，从而低成本裁剪无效节点且保持树结构平衡。然后，针对现有方法被动触发的特点，本章提出了主动的顶点合并剪枝方法。该方法主动地合并具有相同局部邻居的顶点，进一步提升了剪枝效率。最后，本章结合主动的集合枚举树和主动的顶点合并剪枝方法，形成了高效的极大二分团枚举算法 AMBEA。实验证明，AMBEA 算法相比现有最优算法压缩了 2.4-9.0 倍的搜索空间，运行时间缩短了 1.2-5.3 倍。

2.1 现有搜索空间优化方法分析

为了优化极大二分团枚举问题的搜索空间，相关工作以 1.2.2 节所述的集合枚举树为基础，设计了许多优化方法。本节将这些方法总结为三类：顶点排序方法、基于枢纽顶点的剪枝方法和被动的顶点合并剪枝方法。本节将对这些方法进行详细说明。

2.1.1 顶点排序方法

顶点排序方法是一种常用的搜索空间优化技术，在枚举领域中得到了广泛应用。以极大团枚举问题为例，Eppstein 等人提出了退化排序方法^[86]。该方法根据当前顶点与未选取顶点的连接关系进行排序，将连接度低的顶点放在前面，以此限制每个枚举节点内候选顶点数量的上限来降低算法的最坏时间复杂度，从而压缩问题的搜索空间。虽然顶点排序会增加排序的开销，但是好的顶点排序方法可以显著地减小搜索空间，从而提高整体性能。因此，在近些年，研究者们广泛采用退化排序方法来解决极大团枚举问

题[23, 25, 27, 56-57] 和最大团搜索问题[87-88]。

在极大二分团枚举问题中，顶点排序方法同样适用。根据节点的生成规则，每个节点 (L, R, C) 可以按照任意顺序遍历候选顶点集 C 中的顶点生成新的节点 (L', R', C') 。由于在枚举过程中，集合 C 是二分图 $G(U, V, E)$ 中集合 V 的子集，因此极大二分团枚举问题中的顶点排序默认只对集合 V 中的顶点进行排序。基于这一理论基础，研究者们设计了多种不同的顶点排序方法。不同的顶点遍历顺序会导致产生的非极大二分团数量有所差异，相应的搜索空间大小也不同。与退化排序方法在极大团枚举问题中的地位不同，在极大二分团枚举问题中，不同的顶点排序方法各具优势。以下是对这些排序方法的梳理。

(1) 根据顶点的邻居数量递增排序。在频繁项集挖掘领域中，根据邻居数量对顶点进行递增排序的方法被广泛采用，并且实践证明这种方法对搜索空间的优化非常有效^[60]。采用这种排序方法可以使得邻居数量较少的顶点拥有较多的尾部顶点，即 $|N(v)|$ 越小， $|X_v^+|$ 越大。结合集合枚举树的节点生成规则，我们可以推导出在这种排序条件下，邻居较少的顶点对应的子搜索空间受到顶点数量的约束；而邻居较多的顶点对应的子空间受到尾部顶点数量的约束，从而提供性能保障。因此，大量的极大二分团枚举相关工作采用这种排序方式^[2, 6, 15, 49-50, 61]，代表算法包括 MineLMBC^[6]，iMBEA^[15]，FMBE 以及它的并行版本 ParMBE^[50]。此外，MineLMBC 和 iMBEA 算法也会按照局部邻居的数量递增对每个节点 (L, R, C) 内的候选顶点进行排序，即对集合 C 内的顶点 v 根据 $|N_L(v)|$ 的大小递增排列。但需要注意的是，这种排序方式会增加每个节点的计算量，因此需要在平衡排序开销和带来的性能收益方面进行考虑。

(2) 根据顶点的邻居数量递减排序。在真实的二分图场景下，很多二分图的顶点邻居数量分布呈现出倾斜的特征。Damaschke 的研究表明，如果顶点邻居数量的分布符合齐夫定律 (Zipf's law)，那么按照顶点邻居数量递减排序可以降低极大二分团枚举算法的最坏时间复杂度^[67]。然而，实际情况中很多二分图并不能完全满足这种严格的分布约束。此外，Damaschke 的研究主要关注优化算法的最坏时间复杂度。鉴于该时间复杂度相对较为宽松，因此不能准确反映极大二分团枚举任务的实际运行时间。

(3) 逆拓扑排序。在 Abidi 等人的研究中，他们发现二分图中顶点邻居的包含关系可以用来裁剪搜索空间中的非极大二分团^[47]。举个例子，如果对于两个顶点 v_1 和 v_2 ， v_1 的邻居集合包含 v_2 的邻居集合 (即 $N(v_1) \supset N(v_2)$)，那么所有包含顶点 v_2 且不包含 v_1

的二分团都是非极大二分团，可以进行剪枝。基于这个思想，Abidi 等人设计了基于枢纽的顶点剪枝方法，并设计了逆拓扑排序来辅助进行枢纽顶点的选择。具体来说，他们根据二分图中顶点邻居的包含关系构建有向图 CDAG。如果 $N(v_1) \supset N(v_2)$ ，则在 CDAG 中添加一条有向边 $v_1 \rightarrow v_2$ 。最终，根据 CDAG 对顶点进行逆拓扑排序。目前，该排序方法仅在 PMBE 算法中使用^[47]。

(4) 单边排序。Chen 等人结合极大团枚举问题中的退化排序方法^[86]和二分图的特征提出了单边排序方法。考虑到极大二分团枚举问题中候选顶点的数量取决于顶点的二跳邻居数量，单边排序根据当前顶点在未选取顶点中的二跳邻居数量进行排序，将二跳邻居数量较少的顶点排在前面。类似于根据顶点邻居数量递增排序的思路，这种排序方法可以使得二跳邻居数量较少的顶点能够拥有更多的尾部候选顶点。因此我们知道，二跳邻居数量较少的顶点对应的子搜索空间受到顶点二跳邻居数量的约束；而二跳邻居较多的顶点对应的子空间受到尾部顶点数量的约束，从而限制每个枚举节点内候选顶点数量的上限，压缩问题的搜索空间。目前，该排序方法仅在 ooMBA 算法中使用^[48]。

总体而言，在极大二分团枚举问题中，顶点排序方法的选择是灵活的，并且其性能取决于它与对应算法的配合程度。例如，逆拓扑排序方法可以辅助 PMBE 算法选择枢纽顶点，而其他排序方法不具备这种效果。因此，研究者们需要结合具体的枚举算法来选择合适的顶点排序方法。

接下来，我们将通过一个具体例子来说明顶点排序方法在极大二分团枚举问题中的实际效果：

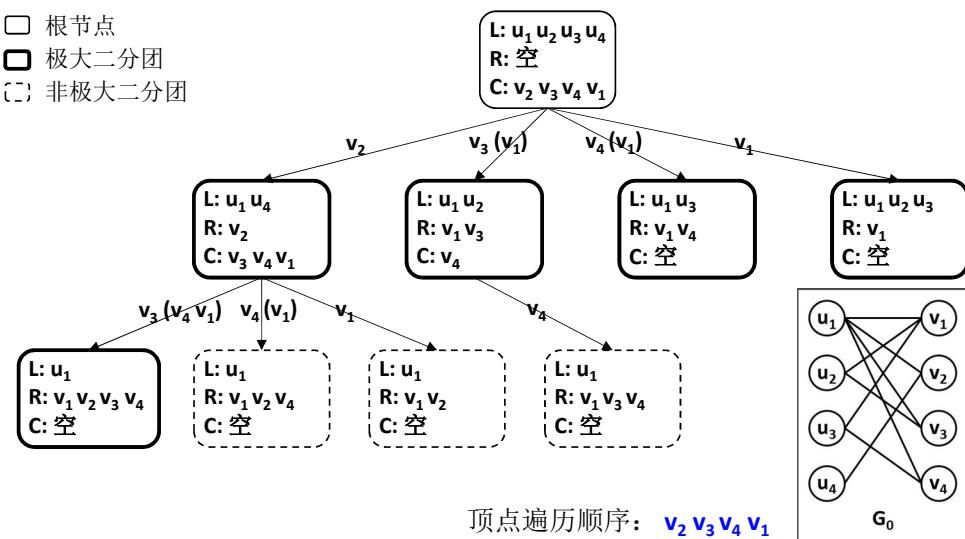


图 2.1 根据顶点的邻居数量递增排序

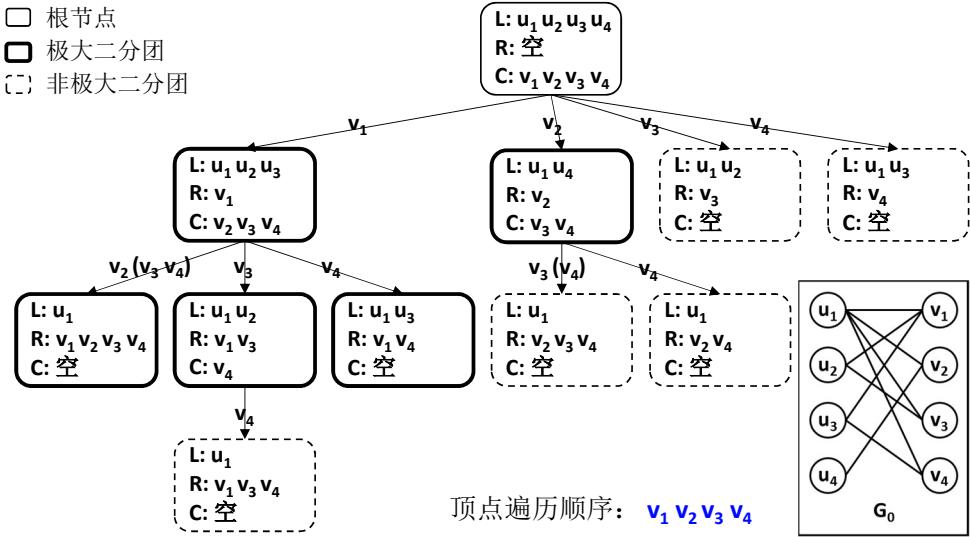


图 2.2 根据顶点的邻居数量递减排序

例 2.1. 图 2.1 和图 2.2 展示了对于同一个二分图 G_0 , 不同排序下问题搜索空间的变化。根据二分图的连接关系, 我们可知 v_1, v_2, v_3, v_4 的邻居数量分别为 3, 2, 2, 2。图 2.1 按照顶点邻居数量递增的顺序遍历候选顶点, 参考例 1.2, 可以得知这种顺序下得到 5 个极大二分团和 3 个非极大二分团; 同理, 图 2.1 按照顶点邻居数量递减的顺序遍历候选顶点, 得到 5 个极大二分团和 5 个非极大二分团。对比这两种排序方法, 对于二分图 G_0 来说, 递增排序的搜索空间优于递减排序。

2.1.2 基于枢纽顶点的剪枝方法

在极大团枚举问题中, 基于枢纽顶点的剪枝方法是一种常用的策略, 旨在减少搜索空间并提高算法效率^[86]。具体而言, 在枚举过程中选择特定的顶点作为枢纽顶点(也称为 Pivot), 并通过判断枢纽顶点与其他顶点的连接情况来裁剪掉不可能产生极大团的无效枚举节点。近年来, 研究者们结合二分图的特征, 将基于枢纽顶点的剪枝方法应用于极大二分团枚举问题中^[47-48]。

具体而言, 在节点 (L, R, C) 中, 枢纽顶点是集合 C 中的一个特殊顶点 v^* 。候选顶点集 C 中的顶点根据邻居的包含关系被分为两个部分, 其中一部分顶点 v' 满足条件 $N(v') \cap L \subset N(v^*)$ (不包括 v^*)。如果枢纽顶点 v^* 已被访问, 那么这部分候选顶点将无法用于产生极大二分团, 因为这些二分团都可以使用顶点 v^* 进一步扩展。因此我们可以对这部分分支进行剪枝。根据枢纽顶点的性质, PMBE 算法在枚举前计算不同顶点

之间的邻居包含关系，并利用这种全局包含关系进行枢纽顶点选择与剪枝^[47]。例如，在图 2.2 中，如果根节点选择 v_1 作为枢纽顶点，那么 PMBE 算法可以安全地对由 v_3 和 v_4 生成的分支进行剪枝，因为 v_3 和 v_4 的邻居是 v_1 的邻居的子集（即 $N(v_3) \subset N(v_1)$ 且 $N(v_4) \subset N(v_1)$ ）。然而，PMBE 忽略了很多运行时动态产生的局部包含关系，因此剪枝性能受到限制。ooMBA 算法在 PMBE 的基础上进行了改进，提出批量最优枢纽选择方法，即通过对候选集合 C 中的每个顶点执行深度为 2 的深度优先搜索（DFS）来选择一批最优枢纽顶点进行剪枝。这种方法可以充分利用局部的包含关系，提高剪枝效率。然而，它的计算开销相对较大。对于每个候选顶点执行深度为 2 的 DFS 操作的时间复杂度为 $O(|E|)$ ，这个时间复杂度等同于根据该顶点生成一个新节点并对该节点执行如 1.2.2.1 节所示的节点检查操作。

2.1.3 被动的顶点合并剪枝方法

被动的顶点合并剪枝方法是一种由 Zhang 等人提出的简单而有效的剪枝技术^[15]。它的基本思想是在枚举过程中，合并与当前遍历顶点具有相同局部邻居的候选顶点，以达到剪枝的效果。具体而言，当一个节点 (L, R, C) 遍历一个候选顶点 v^* 以生成一个节点 (L', R', C') 时，首先会进行节点检查。只有当节点 (L', R', C') 对应一个极大二分团时，该方法会合并集合 C 内满足 $N(v') \cap L = N(v^*) \cap L$ 的候选顶点 v' ，并从集合 C 中移除这些顶点，进而减少搜索空间。这些顶点可以被安全地移除，因为它们所对应的二分团都可以通过添加 v^* 来扩展，因此它们都不是极大二分团。被动的顶点合并剪枝方法虽然有效，但也存在一定的局限性。它只有在 v^* 生成的新节点能够产生极大二分团的情况下才能触发剪枝效果。当 v^* 生成的新节点无法通过节点检查时，类似地，这些顶点 v' 生成的新节点也无法通过节点检查，因此会分别带来节点检查开销，影响剪枝效率。

2.2 研究动机

尽管已经有许多方法来优化极大二分团枚举问题的搜索空间，但现有算法所产生的搜索空间仍然非常庞大。具体而言，在枚举过程中，现有方法仍会生成大量的非极大二分团，并对相应节点进行节点检查，导致高昂的计算开销。本节将现有方法的不足归纳为集合枚举树的结构限制以及被动的剪枝方法，并结合例子进行说明。

2.2.1 集合枚举树的结构限制

根据 1.2.2.1 节的描述, 为了保证枚举树中每个节点 (L, R, C) 对应一个独特的集合 R , 集合枚举树仅允许使用节点内未被访问的候选顶点来扩展该节点的集合 R , 我们称之为集合枚举树的结构限制。无论现有的优化方法如何安排顶点的遍历顺序或者挖掘未被访问顶点邻居之间的关系, 已访问的顶点仍然会导致大量非极大二分团产生。因此, 集合枚举树的结构限制是现有搜索空间方法的瓶颈所在。

2.2.2 被动的剪枝方法

根据 2.1 节的描述, 现有的剪枝方法的执行依赖于特定的顶点, 利用特定顶点与其他顶点之间的关系来进行剪枝操作, 我们称之为被动的剪枝方法。基于枢纽顶点的剪枝方法需要额外的计算开销来选择枢纽顶点, 并且该方法的剪枝效率严重依赖于所选择的枢纽顶点。被动的顶点合并剪枝方法仅在当前访问的顶点通过节点检查后才能触发剪枝操作。因此, 现有的剪枝方法的性能受到这些特定顶点的约束。

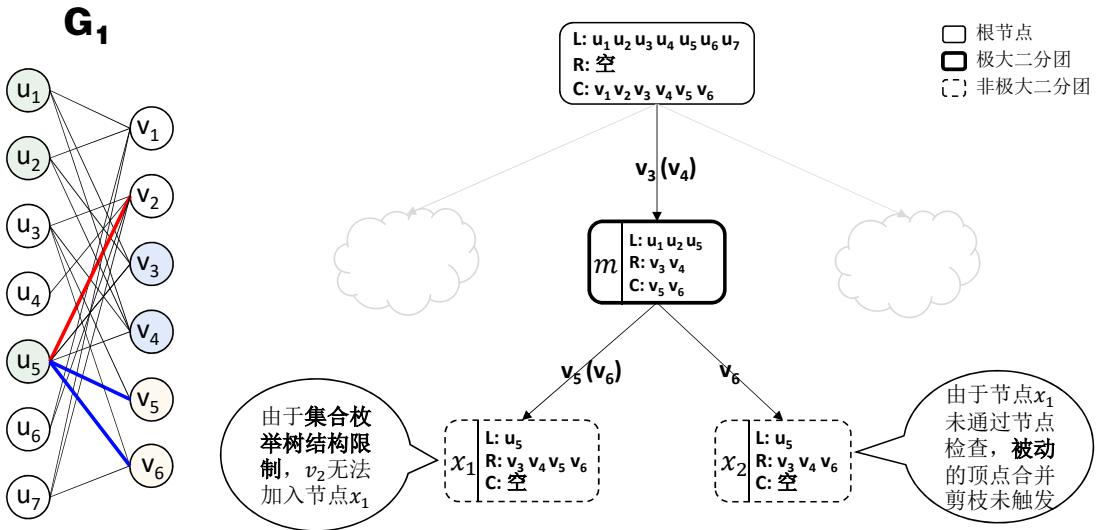


图 2.3 二分图 G_1 上的部分集合枚举树

例 2.2. 图 2.3 是图 1.5 的子图, 展示了在二分图 G_1 上, 以节点 m 为根节点的部分集合枚举树, 用于说明本章研究动机。为便于观察, 我们在图 G_1 中标记了节点 m 中的顶点, 并标记了集合 L_m 和集合 C_m 之间的边, 以及集合 L_m 和顶点 v_2 之间的边。

在所示的部分集合枚举树中, 我们观察到节点 x_1 和节点 x_2 的集合 L 都只包含顶点 u_5 , 而且顶点 v_2 与顶点 u_5 相连。根据深度优先的搜索规则, 顶点 v_2 在节点 x_1 和节点

x_2 生成之前已被访问。由于集合枚举树的结构限制，节点 x_1 和节点 x_2 不允许使用已访问的顶点 v_2 来扩展它们的集合 R ，从而导致非极大二分团的产生。同时，我们观察到在节点 m 中，候选顶点 v_5 和 v_6 在 L_m 中的邻居同是 u_5 ，这意味着可以通过顶点合并对由顶点 v_6 生成的节点 x_2 进行剪枝。然而，由于顶点 v_5 生成的节点 x_1 产生了非极大二分团，被动的顶点合并剪枝方法没有被触发，因此该方法的性能没有充分发挥。

2.3 AMBEA 算法

为了克服现有方法的不足，本节提出了主动的集合枚举树和主动的顶点合并剪枝方法，以提升极大二分团枚举算法的效率。其中，主动的集合枚举树突破了传统集合枚举树的结构限制，而主动的顶点合并剪枝方法则有效地优化了剪枝过程。最终，我们综合应用这两种技术，形成了高效的极大二分团枚举算法 AMBEA。

2.3.1 主动的集合枚举树

现有基于集合枚举树的极大二分团枚举算法存在一个限制，即它们只使用未遍历的顶点来生成的树节点（二分团）以确保区分性。因此它们通常需要进行大量节点检查以删除产生的大量非极大二分团。为了克服这一限制，我们提出了一种新颖的**主动的集合枚举树**（Aggressive Set-Enumeration Tree, ASE Tree），在节点生成过程中，它主动地引入所有顶点，将所有二分图扩展到它们的极大二分团形式，避免检查它们的极大性。考虑到这种主动的扩展可能导致生成重复的二分图，我们在节点检查过程中通过使用低成本方法来删除重复实例并输出不同的极大二分团来解决这个问题。

在展开描述之前，我们先定义一些重要的术语和符号。

定义 2.1. (\vec{v}) 对于一个枚举树节点， \vec{v} 表示用于生成该枚举树节点的候选顶点，即从父节点到子节点的边上的遍历候选顶点。例如，在图 2.3 中，节点 m 的 \vec{v} 是 v_3 。

定义 2.2. (X_v^+, X_v^-) 给定顶点顺序，顶点集 X 可以被顶点 v 划分成两个子集： X_v^+ 包含所有顶点比 v 更大的顶点（顺序在 v 之后的顶点），即顶点 v 的尾部顶点； X_v^- 包含包括 v 顶点在内的所有顶点比 v 更小的顶点（顺序在 v 之前的顶点），即顶点 v 的头部顶点。

具体而言，ASE 树保留了传统集合枚举树的节点结构，并设计了主动的节点生成方法和节点检查方法。节点生成的过程允许使用所有顶点来扩展新节点的二分团，因此每

个节点都会产生一个极大二分团，但是不同节点对应的二分团可能重复。为了删除全部重复二分团，我们利用父子节点的联系设计了新的节点检查规则。我们对 ASE 树的节点生成以及节点检查规则进行了规范化描述：

节点生成：ASE 树的遍历从根节点 (U, \emptyset, V) 开始，依次遍历当前节点 (L, R, C) 中的每个候选顶点 v' ，以生成新节点 (L', R', C') 。在新节点中， L' 表示 $R \cup \{v'\}$ 的共同邻居，即 $L' = L \cap N(v')$ 。不同之处在于， R' 包含 L' 中顶点的所有共同邻居，即 $R' = \Gamma(L')$ 。 C' 由 C 中顺序在顶点 v' 之后且与集合 L' 中任意顶点相连的顶点组成，即 $C' = C_{v'}^+ \cap (\Upsilon(L') - \Gamma(L'))$ 。

节点检查：为了识别和删除上述节点生成过程中产生的所有重复的极大二分团，我们采用**基于映射的节点检查基本方法**。具体而言，我们为每个极大二分团 (L', R') 分配唯一的目标顶点 v'_T 和唯一的目标父节点 (L_T, R_T) 。对于目标父节点选择目标顶点所产生的节点，我们输出对应的极大二分团；对于其他节点，我们删除它们产生的重复项。映射的细节如下：

$$v'_T = \min\{v \in R' \mid \Gamma(R'_v^-) = L'\} \quad (2-1)$$

$$\begin{cases} v_T = \min\{v \in R' \mid \Gamma(R'_v^-) \cap N(v'_T) = L'\} \\ L_T = \Gamma(R'_{v_T}^-) \\ R_T = \Gamma(L_T) \end{cases} \quad (2-2)$$

在映射中，根据公式 2-1， v'_T 是 R' 中满足条件 $\Gamma(R'_v^-) = L'$ 的最小顶点 v ，也是仅根据极大二分团 (L', R') 能够确定的唯一目标顶点。确定了 v'_T 后，如果 $N(v'_T)$ 与 L' 相同，则 (L_T, R_T) 为根节点；否则， (L_T, R_T) 根据公式 2-2 确定。根据公式 2-2，我们首先得到中间顶点 v_T ，它是 R' 中满足条件 $\Gamma(R'_v^-) \cap N(v'_T) = L'$ 的最小顶点 v ，也是仅根据极大二分团 (L', R') 和 v'_T 能够得到的唯一顶点。然后，我们可以将 L_T 推导为 $\Gamma(R'_v^-)$ ，将 R_T 推导为 L_T 中顶点的共同邻居。

根据这种映射机制，节点 (L', R', C') 通过节点检查并输出极大二分团，当且仅当以下两个条件同时满足：

- O1：当前节点的 v 是目标顶点 v'_T 。
- O2：父节点中的极大二分团是目标父节点的极大二分团 (L_T, R_T) 。

为了提高对 ASE 树中重复节点的检查效率，我们引入了一种**低成本节点检查**方法。该方法无须进行多集合求并集运算，仅需访问候选顶点的局部邻居。如果子节点和父节点中同一候选顶点 v_c 对应的局部邻居保持不变（即 $L_{parent} \cap N(v_c) = L_{current} \cap N(v_c)$ ），我们可以安全地裁剪掉子节点选择候选顶点 v_c 所生成的节点。因为根据节点生成规则，父节点和当前节点通过遍历顶点 v_c 会生成相同的二分团。根据公式 2-2，映射机制选择满足条件的最小 v_T ，即确保目标父节点的 \vec{v} 最小。由此可知当前子节点不能对应于二分图 (L_T, R_T) ，因为 $|L_{current}| < |L_{parent}|$ 。这种低成本节点检查只需要 $O(\Delta(V))$ 的集合交集运算，而其他方法至少需要 $O(|E|)$ 的计算量。

与传统的搜索空间优化方法不同，ASE 树彻底改变了集合枚举树的节点生成方式和节点检查方式。通过采用主动的节点生成规则，枚举树中的二分团被扩展为极大二分团，并且允许重复。同时，配套的节点检查规则确保 ASE 举树能够无重复地输出二分图中的全部的极大二分团。我们在理论上为基于 ASE 树的极大二分团枚举算法提供了**正确性证明**。

定理 2.1. ASE 树能够准确的输出二分图中的全部极大二分团，并排除任何非极大二分团或重复二分团。

证明. 首先，根据节点生成规则，对于新生成的节点 (L', R', C') ，我们可以得到 $L' = L \cap N(v') = \Gamma(R \cup \{v'\})$ 且 $R' = \Gamma(L')$ 。因为 L' 和 R' 都是某个特定集合内顶点的共同邻居且 R' 内的顶点和 L' 内的顶点均相连，所以 (L', R') 是一个极大二分团。由此可知，在主动的生成规则下不会产生非极大二分团。

其次，根据节点检查规则，每个节点可以根据产生的二分团 (L', R') 计算得到唯一的目标顶点 v'_T 和唯一的目标父节点。对于对应于 (L_T, R_T) 的给定节点，我们知道 v'_T 总是包含在其候选集 C_T 中，因为 v'_T 始终大于 v_T 并与 L_T 中的一些但不是所有顶点相连。因此，如果目标父节点 (L_T, R_T, C_T) 存在，它总是可以通过遍历 v'_T 生成具有 (L', R') 的节点。

最后，根据节点检查规则，通过遵循这个过程，每个极大二分团的对应节点可以递归地找到唯一的靶父节点，直到达到根节点。因此，ASE 树总能够准确地无重复地枚举每个极大二分团。 \square

算法 2.1 基于 ASE 树的极大二分团枚举算法

输入: 二分图 $G(U, V, E)$

输出: 所有极大二分团

```

1: for  $v_s \in V$  do
2:   biclique_search_ase( $U, \emptyset, V, v_s$ );
3: end for
4: procedure biclique_search_ase( $L, R, C, v'$ ) :
5:    $L' \leftarrow L \cap N(v')$ ;  $R' \leftarrow \Gamma(L')$ ;  $C' \leftarrow \emptyset$ ;
6:    $\bar{L} \leftarrow L \setminus L'$ ;  $\bar{R} \leftarrow R'_{v'}^- \setminus (R \cup C)$ ;
7:   for  $v_c \in C$  do
8:     if  $L' \cap N(v_c) = L'$  then
9:       if  $v_c < v'$  then
10:         $\bar{L} \leftarrow \bar{L} \cap N(v_c)$ ;
11:       end if
12:     else if  $L' \cap N(v_c) \neq \emptyset \wedge v_c > v'$  then
13:        $C' \leftarrow C' \cup \{v_c\}$ ;
14:     end if
15:   end for
16:   if  $\bar{L} \neq \emptyset \wedge \bar{R} = \emptyset$  then
17:     输出极大二分团  $(L', R')$ ;
18:     for  $v'_c \in C'$  do
19:       if  $N(v'_c) \cap \bar{L} \neq \emptyset$  then
20:         biclique_search_ase( $L', R', C', v'_c$ );
21:       end if
22:     end for
23:   end if
24: end procedure

```

算法 2.1 总结了基于 ASE 树的极大二分团枚举算法的基本枚举过程。该算法依次遍历 V 中的每个顶点 v_s , 并递归地调用过程 `biclique_search_ase` (第 1-3 行)。这个过程描述了节点 (L, R, C) 访问候选顶点 v' 生成节点 (L', R', C') (第 5、12-13 行) 并执行节点检查的过程 (第 6、8-11、16 行)。接下来我们概述节点检查的执行过程。我们把节点 (L, R, C) 的 \vec{v} 记作 v 。由于节点 (L, R, C) 已通过节点检查, 这意味着 $\Gamma(R_v^-) = L$ 且 v 是 R 中满足 $\Gamma(R_v^-) \cap N(v') = L'$ 的最小顶点 (第 19 行), 因此满足条件 O1。值得注意的是, ASE 树支持使用简单的集合交集运算在 $O(\Delta(V))$ 的时间复杂度内进行**低成本的节点检查**, 而其他节点检查方法至少需要 $O(|E|)$ 的时间复杂度。根据节点的生成规则, 由于 $R \cup C$ 包含了所有与 L' 中的某些或全部顶点相连的比 v 大的顶点, 因此我们知道 \bar{R} 中的顶点都比 v 小。如果 $\bar{R} = \emptyset$ (第 16 行), 则表示在 R' 中没有比 v 小且不在 R 中的顶点。我们使用 \bar{L} 来存储与 L 中的部分顶点, 这部分顶点与 R' 内小于 v' 的全部顶点

相连且和 v' 不相连。因此，如果 $\bar{L} \neq \emptyset$ (第 16 行)，则满足条件 O2。

就时间复杂度而言，由于算法中最复杂的操作涉及最多 $|V|$ 次集合操作且每次集合操作耗时 $O(\Delta(V))$ ，因此 ASE 树中每个节点的计算时间为 $O(|V|\Delta(V))$ 。最终，算法 2.1 的时间复杂度为 $O(|V|\Delta(V)\beta)$ ，与算法 1.1 相同。

我们用下面的例子进行详细说明。

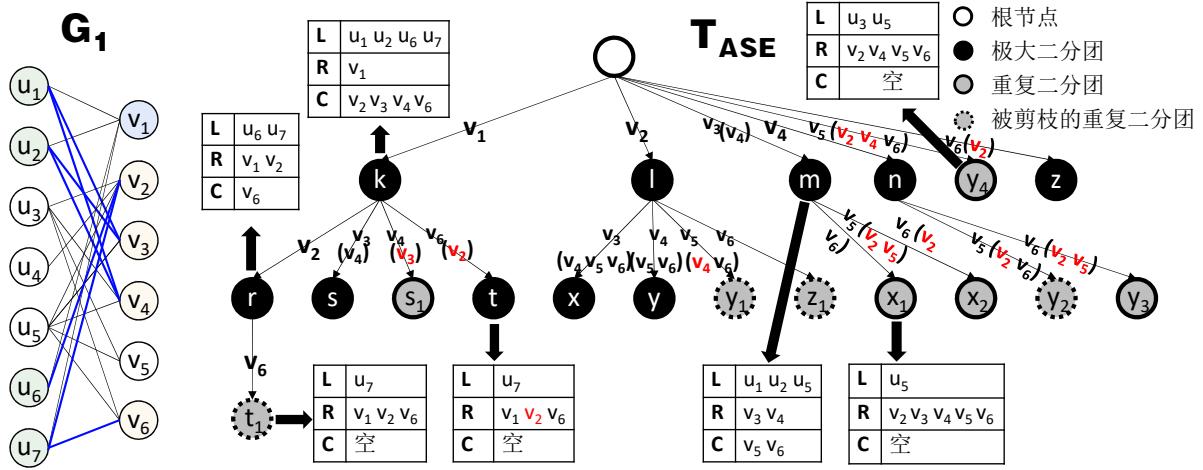


图 2.4 算法 2.1 二分图 G_1 上的集合枚举树

例 2.3. 图 2.4 展示了在二分图 G_1 上的一个枚举树 T_{ASE} 。 T_{ASE} 与图 1.5 中的 T_{SE} 之间的主要区别在于节点生成过程。具体来说，在 T_{ASE} 中，每个节点都可以通过遍历顶点进一步扩展，如图中所示。例如，在 T_{SE} 中，当节点 k 遍历 v_6 时，它生成了一个非极大二分团的节点 t_1 ，因为 T_{SE} 无法使用 v_2 扩展 R_{t_1} ，因为 v_2 已经被遍历以生成节点 r 。相比之下，在 T_{ASE} 中，节点 k 在遍历 v_6 时生成了一个极大二分团的节点 t ，因为 T_{ASE} 可以使用 v_2 扩展 R_t 。因此， T_{ASE} 中的所有二分团都是极大的，但可能存在重复的二分团，比如节点 t 和节点 t_1 。

T_{ASE} 采用了一种主动的节点检查规则来删除重复的二分团。为了说明节点检查的条件，我们将以节点 x_1 和 t_1 为例。对于节点 x_1 ，根据公式 2-1，我们按顺序将顶点从 R_{x_1} 添加到空集 X ，直到 $\Gamma(X)$ 等于 L_{x_1} 。通过这样做，我们确定节点 x_1 的 v'_T 是 v_3 ，因为 $\Gamma(\{v_2, v_3\}) = \{u_5\} = L_{x_1}$ 。因此，节点 x_1 不满足条件 O1，因为它的 \vec{v} 是 v_5 ，而不是 v_3 。对于节点 t_1 ，我们以相同的方式得到它的 v'_T 为 v_6 。类似地，使用公式 2-2，我们推断出它的 v_T 是 v_1 ，因为 $\Gamma(\{v_1\}) \cap N(v_6) = \{u_7\} = L_{t_1}$ 。然后，我们发现它的 $L_T = \Gamma(\{v_1\}) = \{u_1, u_2, u_6, u_7\}$ 。因此，节点 t_1 不满足条件 O2，因为其父节点 r 的集合

L (即 $L_r = \{u_6, u_7\}$) 与其 L_T 不相同。

与 T_{SE} 相比, T_{ASE} 输出相同的极大二分团集合。此外, T_{ASE} 通过低成本的节点检查裁剪比 T_{SE} 更多的节点, 如节点 t_1 , y_1 , z_1 和 y_2 。例如, 节点 r 主动裁剪了节点 t_1 , 因为 $L_k \cap N(v_6) = \{u_7\} = L_r \cap N(v_6)$ 。

ASE 树的主要贡献包括以下两点:

1. **支持低成本的节点检查:** ASE 树通过支持低成本的节点检查, 能够主动地裁剪无效节点, 从而减小搜索空间。具体而言, 在不考虑其他搜索空间优化方法的情况下, 我们观察到对于 T_{SE} 和 T_{ASE} 中输出相同极大二分团 (L, R) 的节点总是具有相同的 \vec{v} 和相同候选集合 C' , 即 \vec{v} 始终对应公式 2-1 中的 v'_T , 且候选集合 C' 始终是 $(\Upsilon(L') - \Gamma(L'))_{v'_T}^+$ 。例如在图 1.5 和图 2.4 中, 输出相同极大二分团的节点 k 对应的 \vec{v} 均为 v_1 , 且对应的候选集合均为 $\{v_2, v_3, v_4, v_6\}$ 。由此我们知道, 不同枚举树中输出相同极大二分团的节点总会产生 $|C'|$ 个分支, 对应地 T_{SE} 和 T_{ASE} 总是产生相同数量的枚举节点。因此, 与基本的 T_{SE} 树相比, ASE 树通过低成本的节点检查主动地裁剪无效节点, 减少了需要进行节点检查过程的节点数量。
2. **生成更加平衡的枚举树:** ASE 树通过主动的节点生成和检查规则, 倾向于降低每个枚举节点的深度, 从而生成更加平衡的枚举树, 有利于并行扩展。具体而言, 对于不同枚举树中输出相同极大二分团 (L, R) 的节点, 根据 ASE 树的节点检查规则 O1, 因为 ASE 树中该节点的父节点总是具有较小的 \vec{v} , 即公式 2-2 中的 v_T , 所以这些节点在 ASE 树中的深度更小。因此, ASE 树的高度更低且更平衡。平衡的枚举树将带来相对均衡的负载, 从而更有利子算法的并行扩展。

2.3.2 主动的顶点合并剪枝方法

为了进一步提升剪枝效率, 我们设计了一种主动的顶点合并剪枝方法, 简称 AMP (Aggressive Merge-based Pruning) 方法。与被动的剪枝方法不同, AMP 方法通过改变节点的生成过程, 在每个节点处主动地对具有相同局部邻居的顶点进行合并, 从而实现剪枝效果。此外, AMP 方法具有通用性, 可以应用到其他极大二分团枚举算法中, 以降低节点生成过程的时间复杂度。

我们首先给出顶点局部邻居的定义, 并给出顶点合并剪枝方法的理论依据。

定义 2.3. (局部邻居) 对于节点 (L, R, C) , C 中候选顶点 v 的局部邻居指 L 中与 v 相连的共同顶点, 记作 $N_L(v)$, 即 $N_L(v) = L \cap N(v)$ 。

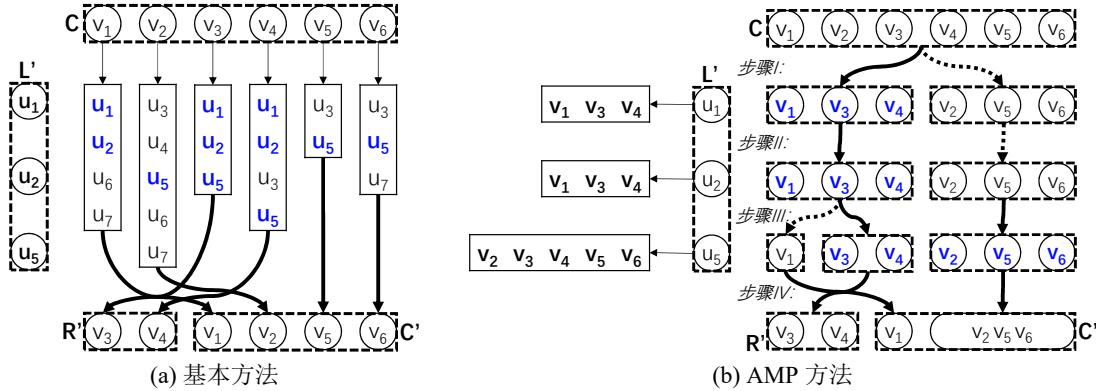
定理 2.2. 对于节点 (L, R, C) , 假设 v_1 和 v_2 是 C 中的两个候选顶点。如果 v_1 和 v_2 具有相同的局部邻居, 那么 v_1 和 v_2 在后继节点中总是具有相同的局部邻居, 因此可以被安全地合并。

证明. 由于 v_1 和 v_2 具有相同的局部邻居, 我们得到 $N(v_1) \cap L = N(v_2) \cap L$ 。对于节点 (L, R, C) 的后继节点 (L', R', C') , 我们得到 $L' \subset L$ 。进而, 我们知道 $N(v_1) \cap L' = N(v_1) \cap L \cap L' = N(v_2) \cap L \cap L' = N(v_2) \cap L'$ 。因此, v_1 和 v_2 在后继节点中总是具有相同的局部邻居。 \square

根据定理 2.2 和被动顶点合并剪枝方法的启发^[15], 我们发现通过主动合并具有相同局部邻居的顶点可以进一步释放剪枝潜力。具体而言, 通过合并这些具有相同局部邻居的候选顶点, 我们能够避免合并集合中编号较大的候选顶点产生的分支。相较于被动合并剪枝方法, 主动合并方法无需依赖特定顶点, 在剪枝效率方面更加出色。

然而, 尽管主动合并节点内具有相同局部邻居的顶点具有提升剪枝效率的潜力, 但它同时会带来一定的计算开销。一种直接的解决方法是存储所有候选顶点的局部邻居, 并对它们进行两两比较。这种方法的时间复杂度为 $O(|V|^2 \Delta(V))$, 即执行 $O(|V|^2)$ 次比较操作, 每次比较操作的时间复杂度是 $O(\Delta(V))$ 。这种方法并不实用, 因为它的复杂度超过了在每个节点上执行节点生成和节点检查的时间复杂度 $O(|V| \Delta(V))$ 。因此, 设计一种高效可行的顶点合并方法具有挑战性。

为了最小化合并开销, AMP 方法修改了节点生成的过程。具体而言, 假设节点 (L, R, C) 访问顶点 v' 生成新节点 (L', R', C') 。现有的方法通过逐个计算候选顶点的局部邻居来产生新的候选顶点集合 C' 。与现有方法不同, AMP 方法将候选顶点集合 C 视为一个整体, 并对这个整体进行划分, 使具有相同局部邻居的顶点落在同一分组中。开始时, 所有 C 中的顶点在同一个组内。然后, 依次选择 L' 中的顶点 u_l 对每个组进行划分。通过检查组内的顶点是否与传入的 u_l 相连, 每个组可以被分成两个子组。因此, 具有相同局部邻居的候选顶点总是放在同一组中。最终, 我们主动合并同组内的具有相同局部邻居的候选顶点。我们通过例子来说明 AMP 方法的执行过程。

图 2.5 图 2.4 中节点 m 的生成过程对比

例 2.4. 我们用一个例子来展示 AMP 方法。接着例 2.3, 图 2.5 展示了图 2.4 中节点 m 在不同方法下的生成过程。为了方便比较, 我们假设 C 中包含了 V 中的所有顶点。在示意图中, 我们将每个顶点的邻居在矩形框中表示, 并用蓝色标出了局部邻居。

图 2.5a 展示了基本方法按顺序逐个计算候选顶点的局部邻居。而图 2.5b 展示了 AMP 方法, 它使用 L' 中的顶点按顺序将候选顶点划分为多个组。在步骤 I 中, 由于 u_1 与 v_1 、 v_3 和 v_4 相连但与其他顶点不相连, C 被划分为两个组。在步骤 III 中, 由于 u_1 与 v_3 和 v_4 相连但与 v_1 不相连, 组 $\{v_1, v_3, v_4\}$ 被进一步划分为两个组。在步骤 IV 中, 由于 v_2 、 v_5 和 v_6 具有相同的局部邻居 $\{v_5\}$, 我们将这些顶点主动合并。因此, 使用 AMP 方法, 节点 m 可以裁剪由 v_5 和 v_6 生成的节点 x_1 和 x_2 。

相比之下, 被动合并的剪枝方法无法裁剪节点 x_2 , 因为虽然 v_5 和 v_6 具有相同的局部邻居, 但由于节点 v_5 未通过节点检查, 被动的顶点合并剪枝方法不会触发。

例 2.5. 接着例 2.4, 图 2.6 详细展示了 AMP 方法在图 2.5b 的步骤 I 中进行组划分的过程。其他步骤依此类推。首先, 我们为候选顶点集 C 中每个顶点绑定一个组索引, 每个索引指向一个记录组信息的结构。组信息结构包含三个属性: 外来顶点、下一个组和局部邻居数量。外来顶点和下一个组属性被用来将输入顶点分配到正确的组, 局部邻居数量则记录组内所有顶点的实时局部邻居数量。具体过程如下: (I) 在初始化时, C 中每个顶点的组索引都被设置为 0, 组 0 中的外来顶点和下一个组都为空, 局部邻居数量设置为 0。(II) 当输入 u_1 的邻居 v_1 时, 我们根据 v_1 的组索引找到组 0。由于组 0 的外来顶点不是 u_1 , 我们创建了一个新的组 1。组 1 中的外来顶点和下一个组被设置为空, 而且局部邻居数量比组 0 多 1 个, 因为组 1 中的顶点相比组 0 中的顶点多了 u_1 。随后, 我们设置组 0 的外来顶点为 u_1 这个邻居。然后, 我们将组 0 的外来顶点设为设置 u_1 , 下一个组

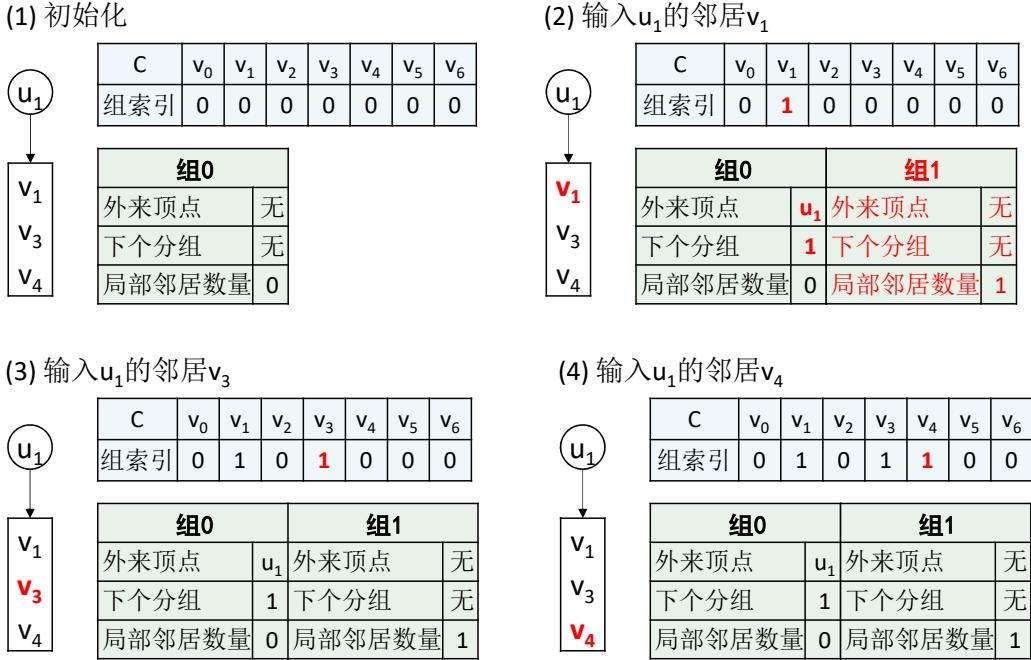


图 2.6 AMP 方法执行组划分的详细过程

设为 1，并将 v_1 的组索引更新为 1。(III) 当输入 u_1 的邻居 v_3 时，我们根据 v_3 的组索引找到组 0，并发现组 0 的外来顶点是 u_1 。因此，我们将 v_3 的组索引更新为组 0 的下个分组，也就是组 1。(IV) 同样地，在输入 u_1 的邻居 v_4 后，我们将 v_4 的组索引设置为 1。

算法 2.2 展示了 AMP 方法的分组合并过程 `partition_merge`。该过程接收原候选顶点集 C 和新集合 L' 两个输入，使用 AMP 方法合并具有相同局部邻居的顶点，并返回新候选顶点集合 C' （第 6-21 行）。具体而言，算法首先给 C 中每个顶点附加一个组索引属性 (gid)（第 7 行），用于在动态维护的组数组 `GroupArray` 中检索顶点的组信息 `GroupInfo`（第 8 行）。其中，组信息包括输入顶点 (`incomming_v`)、下个分组 (`next_gid`) 和局部邻居数量 (`ln_size`)（第 1-5 行）。初始时，`GroupArray` 只包含一个包含所有候选顶点的组（第 8 行）。然后，算法使用 L' 中的每个顶点 u_l 迭代地对候选顶点进行分组（第 9 行）。对于具有相同组索引的顶点，算法将与 u_l 相连的顶点分配到新的组中（第 10-19 行）。当 u_l 不是当前组的输入顶点时，算法创建一个新的组，并将其局部邻居数量增加 1（第 12-17 行）。为了优化内存使用，我们建议回收没有元素的组，可以使用堆栈结构实现。最后，算法合并具有相同组索引的顶点，因为它们拥有相同的局部邻居。就时间复杂度而言，由于创建组、更新组信息等操作均能在常数时间完成，且在运行过程中访问每条边最多 1 次，所以过程 `partition_merge` 的时间复杂度为 $O(|E|)$ 。

算法 2.2 AMP 方法中对候选顶点的分组合并过程**数据:** 二分图 $G(U, V, E)$ **输入:** 原候选顶点集 C , 新集合 L' **输出:** 顶点合并后的候选顶点集 C'

```

1: struct GroupInfo :
2:   Integer incoming_v;
3:   Integer next_gid;
4:   Integer ln_size;
5: end struct
6: procedure partition_merge( $C, L'$ ) :
7:   给  $C$  中每一个顶点附加一个属性  $gid = 0$ ;
8:   GroupArray.append(GroupInfo( $\infty, \infty, 0$ ));
9:   for  $u_l \in L'$  do
10:    for  $v_c \in C \cap N(u_l)$  do
11:       $cid \leftarrow v_c.gid$ ;
12:      if GroupArray[ $cid$ ].incoming_v  $\neq u_l$  then
13:        new_gid  $\leftarrow$  GroupArray.size();
14:        GroupArray[ $cid$ ].incoming_v  $\leftarrow u_l$ ;
15:        GroupArray[ $cid$ ].next_gid  $\leftarrow$  new_gid;
16:        GroupArray.append(GroupInfo( $\infty, \infty, GroupArray[cid].ln\_size + 1$ ));
17:      end if
18:       $v_c.gid \leftarrow GroupArray[cid].next\_gid$ ;
19:    end for
20:  end for
21:   $C' \leftarrow$  合并  $C$  内具有相同  $gid$  的顶点, 不包括  $ln\_size$  为 0 或  $|L'|$  的组;
22: end procedure

```

AMP 方法的贡献主要包括以下两点:

1. **基于主动顶点合并的剪枝方法:** 与被动的剪枝方法不同, AMP 方法在节点生成的同时执行, 不依赖于特定的顶点, 因此总是能够主动合并节点内全部具有相同局部邻居的顶点。假设集合 X 中的顶点具有相同的局部邻居, AMP 方法不仅能够裁剪掉集合 X 中除最小顶点外其他顶点所产生的无效分支, 而且能够合并集合 X 内所有的计算, 减少重复计算, 从而提升计算效率。
2. **降低节点生成过程的时间复杂度:** AMP 方法修改了节点生成的过程, 具有降低该过程时间复杂度的潜力。具体而言, 传统的节点生成过程逐个计算每个候选顶点的局部邻居, 时间复杂度为 $O(|V|\Delta(V))$ 。而 AMP 方法的时间复杂度为 $O(|E|) = O(|V|d_{avg}(V)) < O(|V|\Delta(V))$ 。因此, 作为一种通用方法, AMP 方法可以直接用于优化现有算法的节点生成过程。

2.3.3 AMBEA 算法设计

结合 ASE 树和 AMP 方法，我们设计了一种主动的极大二分团枚举算法，简称为 AMBEA（Aggressive Maximal Biclique Enumeration Algorithm）。

算法 2.3 AMBEA 算法

输入：二分图 $G(U, V, E)$

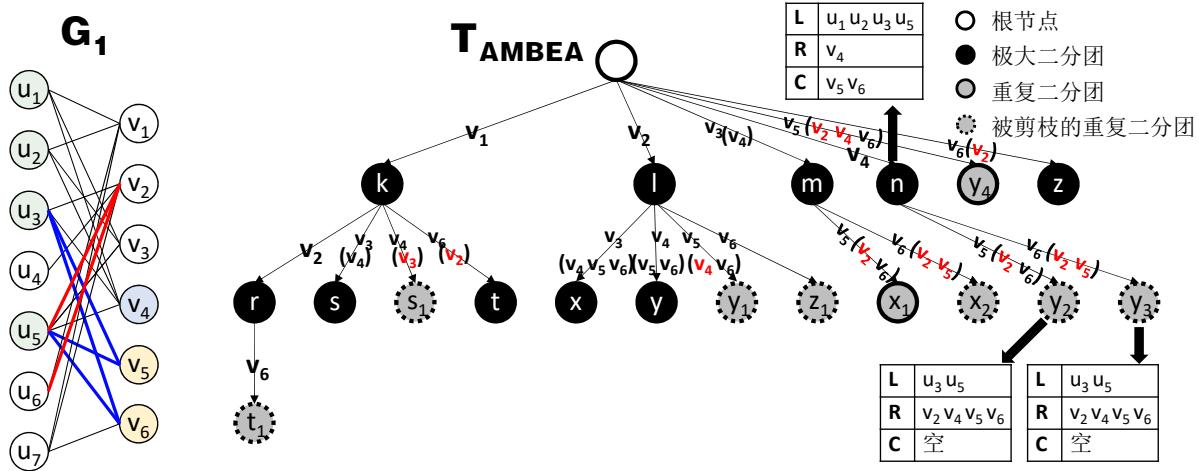
输出：所有极大二分团

```

1: 将  $V$  中顶点按照邻居数量递增排序;
2: for  $v \in V$  :
3:   bi clique\_search\_ambea( $U, \emptyset, V, v$ );
4: end for
5: procedure bi clique\_search\_ambea( $L, R, C, v'$ ) :
6:    $L' \leftarrow L \cap N(v')$ ;  $R' \leftarrow \Gamma(L')$ ;
7:    $C' \leftarrow \text{partition\_merge}(C_{v'}^+, L')$ ;
8:    $\bar{L} \leftarrow \emptyset$ ;  $\bar{R} \leftarrow R'_{v'}^- \setminus (R \cup C)$ ;
9:   for  $u_l \in L \setminus L'$  do
10:    if  $(R'_{v'}^- \setminus \{v'\}) \subseteq N(u_l)$  then
11:       $\bar{L} \leftarrow \bar{L} \cup \{u_l\}$ ;
12:    end if
13:   end for
14:   if  $\bar{L} \neq \emptyset \wedge \bar{R} = \emptyset$  then
15:     输出极大二分团  $(L', R')$ ;
16:     for  $v'_c \in C'$  do
17:       if  $N(v'_c) \cap \bar{L} \neq \emptyset$  then
18:         bi clique\_search\_ambea( $L', R', C', v'_c$ );
19:       end if
20:     end for
21:   end if
22: end procedure

```

算法 2.3 详细描述了 AMBEA。具体而言，对于二分图 $G(U, V, E)$ ，在 2.1.1 节所示的顶点排序方法中，AMBEA 选择将 V 中顶点按照邻居数量递增排序（第 1 行），因为在这种顺序下 AMBEA 算法性能最优。我们将通过下一节的实验证明这一结论。随后，该算法从根节点 (U, \emptyset, V) 开始，按顺序遍历 V 中的顶点 v ，并递归地调用 **bi clique_search_ambea** 过程（第 2-4 行）。在枚举过程中，AMBEA 同时利用了 ASE 树中的低成本节点检查技术和 AMP 方法中的主动顶点合并剪枝技术。通过两种技术的有机结合，AMBEA 高效地减小了计算空间，并取得了良好的计算性能。

图 2.7 AMBEA 在二分图 G_1 上的集合枚举树

例 2.6. 为了方便比较, 图 2.7 展示了 AMBEA 算法在二分图 G_1 上的集合枚举树 T_{AMBEA} 时, 未进行顶点排序优化的情况。为了更清晰地展示, 我们在图 G_1 中标记了节点 n 中的顶点, 并标出了集合 L_n 与集合 C_n 之间的边, 以及集合 L_n 与顶点 v_2 之间的边。在图 2.4 所示的集合枚举树 T_{ASE} 的基础上, AMBEA 利用 AMP 方法额外地裁剪节点 s_1 、 x_2 和 y_3 , 充分发挥两种方法的优势。

具体而言, 当节点 r 访问候选顶点 v_5 时, 因为 $N(v_5) \cap (L_{root} \setminus L_r) = \{u_3, u_5\} \cap \{u_4, u_6, u_7\} = \emptyset$, 所以节点 r 可以利用 ASE 树中所支持的低成本节点检查裁剪掉 v_5 对应的节点 y_2 。同时, 由于 v_5 和 v_6 具有相同的局部邻居, 算法利用 AMP 方法将它们合并, 并裁剪掉 v_6 对应的节点 y_3 。因此, 算法能够裁剪掉节点 n 的所有无效子节点。然而, 由于现有的剪枝方法只能在访问特定候选顶点后被动执行, 因此无法裁剪掉节点 m 的所有无效子节点, 从而限制了剪枝效率的提升。

结合算法 2.3, 我们从时间复杂度和空间复杂度两个方面对 AMBEA 进行分析。

时间复杂度: AMBEA 的时间复杂度可以分为两部分, 即顶点排序时间 (第 1 行) 和集合枚举树生成时间 (第 2-4 行)。顶点排序时间复杂度为 $O(|V| \log(|V|))$ 。在每个枚举节点的计算过程中, 由于 $L', R', C', \bar{L}, \bar{R}$ 的计算都可以通过之访问二分图中的每条边一次来得到, 因此每个节点的计算复杂度为 $O(|E|)$ 。为了量化算法的计算时间, 我们沿用 1.2.2.3 节中的相关定义, 用 β 来表示枚举树中节点的数量, 包括输出极大二分团的节点和产生重复二分团的节点。由此可知, AMBEA 的时间复杂度为 $O(|E|\beta + |V| \log(|V|))$ 。由于排序开销通常远小于生成集合枚举树的开销, 因此, AMBEA 的时间复杂度可以简化为 $O(|E|\beta)$ 。

就每个节点的计算复杂度而言，AMBEA 算法中每个节点的计算时间与 ooMBEA 算法相同（即 $O(|E|)$ ），且优于其他相关算法（即 $O(|V|\Delta(V))$ ）。就算法的枚举空间 β 而言，经过 2.4.2 节的实验结果验证，AMBEA 算法的枚举空间小于其他对比对象。这得益于 AMBEA 算法高效的剪枝能力。

空间复杂度：AMBEA 的空间复杂度同样分为两个部分，即输入二分图所占用的空间和极大二分团枚举过程所需要的空间。输入二分图所占用的空间为 $O(|E|)$ 。在枚举过程中，每个计算节点所需的空间为 $O(|L| + |R| + |C| + |\bar{L}| + |\bar{R}|) = O(\Delta(V) + |V|) = O(|V|)$ 。由于枚举树的高度上界为 $O(\Delta(V))$ ，我们可以得到 AMBEA 的空间复杂度为 $O(|E| + |V|\Delta(V))$ ，与其他现有算法相同。此外，由于 ASE 树倾向于产生更加平衡的枚举树，因此 AMBEA 算法具有节省空间开销的潜力。

并行扩展：为了进一步提高效率，在算法 AMBEA 的基础上，我们开发了其并行版本 ParAMBEA。受到并行算法 ParMBE^[50] 的启发，ParAMBEA 同时执行多个 AMBEA 中的 `biclique_search_ambea` 过程，并利用多线程优化过程内部的并行运算部分。按照算法 2.3 的描述，ParAMBEA 为二分图集合 V 中的每个顶点 v 分配一个线程，负责执行相应的过程（第 2-4 行），即每个线程对应枚举树中由一个顶点 v 产生的子枚举树。此外，只要有多余的线程可用，ParAMBEA 会动态创建子过程（第 18 行）。为了最大限度地发挥并行计算资源的作用，ParAMBEA 采用循环展开技术，对 `biclique_search_ambea` 过程中的所有 `for` 循环（如第 9-13 行、16-20 行）进行并行化处理。此外，根据 2.3.1 节的讨论，由于 ASE 树更倾向于生成平衡的枚举树，因此每个线程对应的负载也更加均衡，从而使 ParAMBEA 具有更佳的并行性能。

2.4 实验评估

本节同时使用真实数据集和合成数据集，通过与现有方法的对比对 AMBEA 的性能进行全面评估。首先介绍实验的环境设置，包括实验平台、数据集和比较对象。随后，通过已有算法在真实数据上的执行情况，从运行时间、剪枝效率和内存开销三个方面对 AMBEA 进行整体评估。接着，通过消融实验，对本章提出的 ASE、AMP 以及顶点排序等技术点进行分解评估。最后，对枚举树的平衡性、AMBEA 算法的并行性和扩展性进行敏感性测试。

表 2.1 AMBEA 实验数据集统计信息

数据集	目录	类型	$ U(G) $	$ V(G) $	$ E(G) $	极大二分团数量
Unicode (UL)	特征关系	国家-使用-语言	614	254	1,255	460
UCforum (UF)	交互关系	用户-发布-论坛	899	522	7,089	16,261
MovieLens (Mti)	特征关系	标签-属于-电影	16,528	7,601	71,154	140,266
Teams (TM)	归属关系	队员-属于-团队	901,130	34,461	1,366,466	517,943
ActorMovies (AM)	归属关系	电影-出演-演员	383,640	127,823	1,470,404	1,075,444
Wikipedia (WC)	特征关系	文章-属于-目录	1,853,493	182,947	3,795,796	1,677,522
YouTube (YG)	归属关系	用户-属于-群组	94,238	30,087	293,360	1,826,587
StackOverflow (SO)	评分关系	作者-收藏-帖子	545,195	96,680	1,301,942	3,320,824
DBLP (Pa)	作者关系	作者-出版-作品	5,624,219	1,953,085	12,282,059	4,899,032
IMDB (IM)	归属关系	电影-出演-演员	896,302	303,617	3,782,463	5,160,061
BookCrossing (BX)	交互关系	用户-打分-书籍	340,523	105,278	1,149,739	54,458,953
Github (GH)	作者关系	用户-参与-工程	120,867	56,519	440,237	55,346,398
LJ5	归属关系	用户-属于-群组	1,837,928	853,994	5,610,437	2,181,295
LJ10	归属关系	用户-属于-群组	2,301,031	1,421,088	11,227,130	7,430,705
LJ15	归属关系	用户-属于-群组	2,548,619	1,912,139	16,843,216	22,727,251
LJ20	归属关系	用户-属于-群组	2,704,651	2,357,485	22,456,757	61,836,924
LJ25	归属关系	用户-属于-群组	2,812,080	2,771,510	28,068,423	151,468,807

2.4.1 实验设置

实验环境设置：本节的全部实验均在一台配备有 4 个 Intel Xeon (R) Gold 5318Y 2.10GHz CPU 和 128GB 内存的服务器上完成，其中每个 CPU 拥有 24 个计算核心，服务器共计 96 个计算核心。本次实验环境的操作系统为 Linux kernel-5.4.0。在没有特殊说明的情况下，算法默认使用单个计算核心串行执行。

比较算法：AMBEA 的主要比较算法是目前 5 个最先进的基于集合枚举树的极大二分团枚举算法，包括 MBEA^[15]、iMBEA^[15]、FMBE^[50]、PMBE^[47] 以及 ooMBEA^[48]。此外，在并行性的敏感分析中，我们还将 AMBEA 的并行版本 ParAMBEA 与最先进的并行极大二分团枚举算法 ParMBE 进行比较^[50]。我们获取到了所有对比对象的源代码。为了公平比较，我们在一个统一的框架下重新实现并优化了算法 MBEA、iMBEA、FMBE 和 PMBE 的 C++ 代码，并基于此框架实现了 AMBEA 和 ParAMBEA 算法。由于 ooMBEA 和 ParMBE 算法的源代码已深度优化，我们没有重新实现这两个算法。此外，为了深入评估本章提到的技术点，我们实现了一些算法变体，并在对应实验中详细描述。

数据集：为准确评估本章提出的技术，我们同时使用了真实数据集和合成数据集。表 2.1 详细展示了实验数据集的统计信息。真实数据集来源于 KONECT 仓库^[89]，涵盖了不同的应用场景。上半部分的表格显示了真实数据集。而下半部分的表格展示

了合成数据集，这些数据集是在大数据集 LiveJournal ($|U|=7,489,073$, $|V|=3,201,203$, $|E|=112,307,385$) 上分别采样 5%、10%、15%、20% 和 25% 的边所生成，记作 LJ5、LJ10、LJ15、LJ20 和 LJ25。大数据集 LiveJournal 也来自于 KONECT 仓库。考虑到极大二分团枚举算法的运行时间与二分图中极大二分团的数量相关，因此我们按照极大二分团的数量对数据集进行了升序排列。

2.4.2 整体评估

我们使用三个指标对每个极大二分团枚举算法在 12 个真实数据集上进行整体评估：运行时间、内存使用和剪枝效率。为了保证公平性，我们为每个算法开辟一个单独的进程独立运行。测量的运行时间是进程中极大二分团枚举过程的运行时间，不包括从磁盘中加载二分图的时间。测量的内存使用是进程的最大内存使用情况。为了测量剪枝效率，我们记录了每个极大二分团算法中产生的二分团总数量 β ，即节点检查次数。由于给定的二分图中极大二分团的数量 α 是固定的，我们可以得到算法产生的无效二分团的数量 $\delta = \beta - \alpha$ 。因此，我们用比值 δ/α 来衡量不同算法的剪枝效率，比值越小表示剪枝效果越明显。

表 2.2 AMBEA 整体运行时间评估（单位：秒）

数据集	AMBEA	MBEA	iMBEA	FMBE	PMBE	ooMBEA	加速比
UL	0.00357	0.0172	0.0112	0.0306	0.0170	<u>0.00549</u>	1.5
UF	0.0809	0.576	0.314	<u>0.189</u>	0.266	<u>0.215</u>	2.3
Mti	1.7	36.0	24.6	14.1	24.1	<u>2.9</u>	1.7
TM	2.7	70.0	23.8	7.9	19.2	<u>7.0</u>	2.6
AM	8.7	205.3	73.0	123.6	199.3	<u>18.6</u>	2.1
WC	7.2	99.1	35.6	<u>13.9</u>	49.1	20.0	1.9
YG	30	1,567	347	<u>38</u>	127	131	1.3
SO	177	10,893	4,819	3,706	10,933	<u>942</u>	5.3
Pa	12	45	21	<u>14</u>	23	<u>40</u>	1.2
IM	62	1,653	754	707	1,045	<u>215</u>	3.5
BX	1,610	96,206	39,036	<u>5,934</u>	13,234	<u>9,452</u>	3.7
GH	5,085	112,488	101,207	16,616	51,647	<u>9,639</u>	1.9

表 2.2 展示了在真实数据集上，AMBEA 与主流的极大二分团枚举算法的运行时间对比。为了方便比较，我们使用下划线标记了每个数据集中性能最优的比较算法，并且展示了 AMBEA 相对于该算法的加速比。实验结果显示，现有的极大二分团枚举算法

在不同数据集上表现存在显著差异，这表明单一的极大二分团枚举算法难以满足在不同场景下的枚举性能需求。以 StackOverflow、IMDB 和 Github 等数据集为例，ooMBEA 算法明显优于 MBEA、iMBEA、FMBE 和 PMBE 算法；然而，在 Wikipedia、YouTube、DBLP 和 BookCrossing 等包含大量极大二分团的数据集上，FMBE 算法具有明显优势，比 ooMBEA 算法快 1.4-3.4 倍。相较于其他算法，本章提出的 AMBEA 算法在所有测试数据集上都展现出最短的运行时间。具体而言，AMBEA 算法在各数据集上始终比最接近的其他算法快 1.15-5.32 倍。特别是在 BookCrossing 数据集上，AMBEA 仅需 1,610 秒即可完成极大二分团枚举，而 MBEA、iMBEA、FMBE、PMBE 和 ooMBEA 算法分别需要 96,206、39,036、5,934、13,234 和 9,452 秒，这清晰地表明了 AMBEA 在处理耗时较长数据集时的高效率。通过结合 ASE 树与 AMP 方法的剪枝优势，AMBEA 算法实现了高效的极大二分团枚举，从而在性能上超越了其他算法。

表 2.3 AMBEA 整体内存使用评估（单位：MB）

数据集	AMBEA	MBEA	iMBEA	FMBE	PMBE	ooMBEA
UL	3.7	3.7	3.7	3.8	3.7	3.6
UF	3.8	4.1	4.2	4.1	4.3	4.1
Mti	5.7	5.9	5.8	5.7	8.4	12.1
TM	68	68	68	67	181	100
AM	49	45	49	45	119	175
WC	151	145	150	145	396	237
YG	14	14	14	14	33	37
SO	53	53	53	51	157	434
Pa	566	497	558	497	1,421	1,031
IM	114	105	113	104	294	525
BX	42	43	42	39	124	200
GH	20	20	19	18	55	106

表 2.3 展示了在真实数据集上，AMBEA 与主流极大二分团枚举算法的内存使用对比。实验结果显示，在所有数据集上，AMBEA、MBEA、iMBEA 和 FMBE 的内存使用情况相当，而 PMBE 和 ooMBEA 算法则明显需要更多的内存。以 StackOverflow 数据集为例，AMBEA、MBEA、iMBEA 和 FMBE 的内存使用量均为 53MB、53MB、53MB 和 51MB，而 PMBE 和 ooMBEA 的内存使用分别为 157MB 和 434MB。其中，PMBE 算法在初始化过程中需要额外的内存来存储 CDAG 索引结构，而 ooMBEA 算法在计算过程中将整个二分图划分为多个子二分图，因此需要额外的内存来存放这些子二分图。从实

验结果可以看出，我们提出的 AMBEA 算法相较于现有算法不会增加额外的内存开销。相反地，由于 ASE 树倾向于生成相对平衡的枚举树，这使得 AMBEA 在 UCforum 数据集上表现出最少的内存使用。由此可知，AMBEA 算法在保持高效性能的同时，能够更好地控制内存资源的利用，为处理大规模数据集提供了可行的解决方案。

表 2.4 AMBEA 整体剪枝效率评估 (δ/α)

数据集	AMBEA	MBEA	iMBEA	FMBE	PMBE	ooMBEA	搜索空间压缩比例
UL	1.0	36.0	15.4	27.4	22.1	<u>2.4</u>	2.4
UF	0.8	39.4	7.1	7.3	18.5	<u>7.0</u>	9.0
Mti	0.8	64.7	4.9	5.3	14.6	<u>9.2</u>	6.7
TM	0.6	6.3	<u>2.4</u>	2.5	4.1	<u>4.4</u>	3.8
AM	1.3	80.5	23.7	24.1	62.0	<u>4.6</u>	3.6
WC	0.4	5.9	<u>1.4</u>	1.5	3.1	<u>4.3</u>	3.3
YG	2.0	484.7	<u>4.7</u>	5.2	24.4	19.4	2.4
SO	1.6	2056.6	<u>45.9</u>	48.7	270.9	<u>13.7</u>	8.4
Pa	0.4	2.1	<u>0.9</u>	1.2	1.6	<u>1.3</u>	2.4
IM	1.4	149.8	<u>24.4</u>	25.5	87.5	<u>7.4</u>	5.3
BX	1.8	409.3	<u>8.5</u>	8.7	34.9	19.2	4.8
GH	3.5	2779.2	<u>14.5</u>	14.9	70.8	25.7	4.1

表 2.4 展示了在真实数据集上，AMBEA 与主流的极大二分团枚举算法的剪枝效率对比。为了方便比较，我们使用下划线标记了每个数据集中比值 δ/α 最小的比较算法，并且展示了 AMBEA 相对于该算法的搜索空间压缩比例。实验结果验证了本章的研究动机，即现有算法在枚举过程中所产生的搜索空间中仍然包含大量的无效二分团，并对相应节点进行节点检查，导致高昂的计算开销，正如 2.2 节所讨论的。以 Github 数据集为例，MBEA、iMBEA、FMBE、PMBE 和 ooMBEA 在枚举过程中分别会产生比极大二分团数量多 2779.2 倍、14.5 倍、14.9 倍、70.8 倍和 25.7 倍的无效二分团。相应地，这些算法会通过大量高开销的节点检查过程来删除这些无效二分团，从而带来巨大的性能开销。与其他算法相比，在 Github 数据集上，本文提出的 AMBEA 算法生成的无效二分团数量仅为极大二分团数量的 3.5 倍，有效地压缩了搜索空间。在不同数据集上，AMBEA 算法的比值 δ/α 比其他最接近的算法减少 2.4-9.0 倍，这意味着现有剪枝性能最优的算法产生的无效二分团数量是 AMBEA 中无效二分团数量的 2.4-9.0 倍。由此可见，AMBEA 算法中高效的剪枝方法是其比其他算法更快的关键原因。

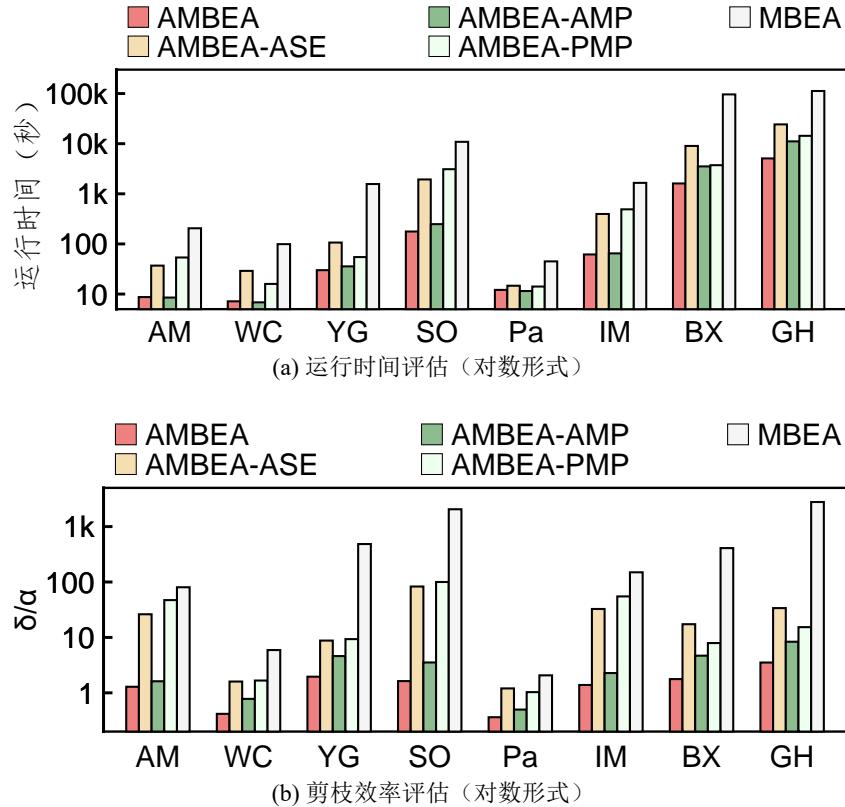


图 2.8 ASE 树、AMP 方法的技术点分解评估

2.4.3 技术点分解评估

我们设计消融实验，沿用整体评估中的运行时间和剪枝效率指标。具体而言，我们在 8 个极大二分团数量较多的真实数据集上，对 ASE 树、AMP 方法以及顶点排序方法进行技术点分解评估。

ASE 树的效果：为了分解评估 2.3.1 节提出的主动的集合枚举树技术，我们在算法 MBEA^[15] 的基础上应用了该技术，形成了 AMBEA-ASE 算法。我们选择使用 MBEA 算法作为基础，因为它严格按照算法 1.1 执行，并且没有进行其他优化。如图 2.8 所示，AMBEA-ASE 的运行时间和 δ/α 的值在所有数据集上均小于 MBEA。具体而言，以 YouTube 数据集为例，AMBEA-ASE 的运行时间是 106.5 秒， δ/α 的值为 1.96；而 MBEA 的运行时间是 1,566.8 秒， δ/α 的值为 484.7。由此可见，AMBEA-ASE 裁剪掉了 MBEA 算法中 $(484.7 - 1.96)/484.7 = 98.2\%$ 的无效二分团，并因此带来 $1,566.8/106.5 = 14.7$ 倍的性能提升。这是因为在 ASE 树中引入了低成本节点检查技术，可以裁剪掉大量无效节点，有效的降低 δ 的值，并且带来性能提升。

AMP 方法的效果：为了对 2.3.2 节提出的主动的顶点合并剪枝方法进行更详细的评估，我们设计了两个算法变体：AMBEA-AMP 和 AMBEA-PMP。其中，AMBEA-AMP 在 MBEA 算法的基础上引入了主动的顶点合并剪枝方法，而 AMBEA-PMP 则采用了 2.1.3 节中介绍的被动顶点合并剪枝方法（Passive Merge-based Pruning, PMP）。如图 2.8 所示，与 MBEA 相比，AMBEA-AMP 和 AMBEA-PMP 通过顶点合并技术裁剪了大量无效节点，在所有数据集上都取得了性能提升。此外，AMBEA-AMP 的运行时间和剪枝效率一直优于 AMBEA-PMP。具体来说，在 StackOverflow 数据集上，MBEA、AMBEA-AMP 和 AMBEA-PMP 的 δ/α 比值分别为 2,056.6, 3.5 和 100.1。这说明 AMBEA-AMP 相较于 AMBEA-PMP 额外裁剪了 96.5% 的无效二分团。因此，相较于 AMBEA-PMP，AMBEA-AMP 的执行时间从 3,097 秒缩短至 248 秒，带来了 12.5 倍的性能提升。这是因为 PMP 方法只有在特定顶点生成的极大二分团通过节点检测后才会合并具有相同局部邻居顶点，而 AMP 方法则始终合并具有相同局部邻居顶点，因此带来额外的剪枝效果。

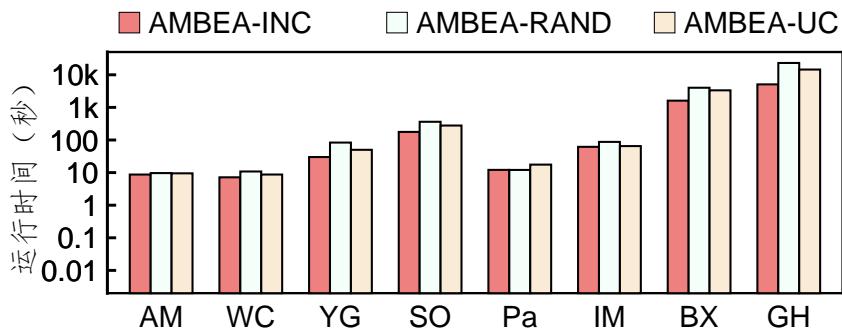


图 2.9 顶点排序方法的分解评估 (对数形式)

顶点排序方法的效果：为了分解评估 2.3.3 节提到的按照顶点邻居数量递增排序的顶点排序方法，我们设计了三个算法变体：AMBEA-INC, AMBEA-RAND, AMBEA-UC。这三个变体均构建在 AMBEA 算法的基础上，唯一的区别在于集合 V 中顶点的排序方式不同。其中，AMBEA-INC 按照顶点邻居数量递增排序，AMBEA-RAND 不进行任何排序，而 AMBEA-UC 则采用单边排序方式^[48]，这种方式在 2.1.1 节中已详细介绍。如图 2.9 所示，AMBEA-INC 的运行时间始终小于 AMBEA-RAND 和 AMBEA-UC，特别是在具有许多极大二分团的数据集上表现更为明显。具体来说，在 Github 数据集上，AMBEA-INC 的运行时间为 5,085 秒，而 AMBEA-RAND 和 AMBEA-UC 分别为 23,067 秒和 14,500 秒。由此可见，按照顶点邻居数量递增排序的方法对于处理执行时间较长的数据集至关重要。因此，我们选择按照顶点邻居数量递增排序集合 V 中的顶点。

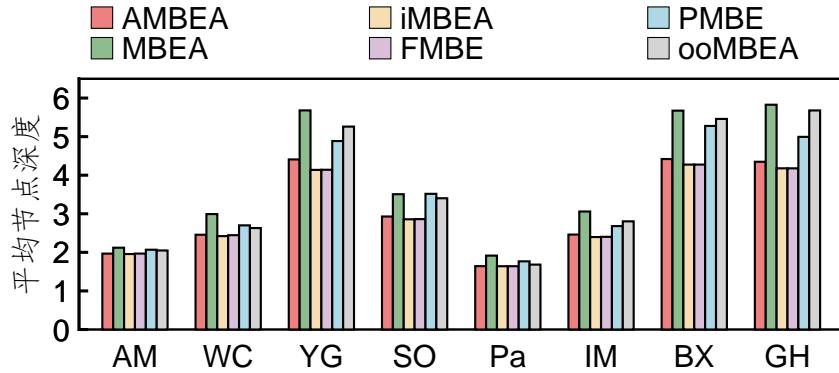


图 2.10 枚举树平衡性敏感性分析

2.4.4 敏感性测试

我们对 ASE 树的平衡性、AMBEA 算法的扩展性和并行性进行了敏感性测试。

ASE 树对枚举树平衡性的影响: 为了探索 AMBEA 对枚举树平衡性的影响，我们测量了不同算法生成的枚举树中每个节点的深度，并计算得到输出最大二分团节点的平均深度，具体结果见图 2.10。由于 AMP 方法不会影响输出极大二分团的节点的生成过程，因此 AMBEA 的平均节点深度与基本 ASE 树中的平均节点深度相同。我们观察到，与 MBEA 相比，AMBEA 的平均节点深度始终更小，这说明 ASE 树相对于基本集合枚举树更加平衡。PMBE 和 ooMBEA 通常具有较大的平均节点深度，这是因为它们使用了 2.1.2 节中提到的基于枢纽顶点的剪枝方法。具体而言，枢纽顶点 v^* 会导致满足 $N_L(v') \subset N_L(v^*)$ 条件的顶点 v' 在当前节点不产生新节点，进而加深了顶点 v' 产生新节点的深度，导致生成的枚举树更加不平衡。iMBEA 和 FMBE 通过在每个节点进行顶点重排序来生成平衡的枚举树。然而，在每个节点进行顶点重排序会增加额外的计算开销。总而言之，算法的选择显著影响枚举树的平衡性，与其他极大二分团枚举算法相比，AMBEA 生成了相对平衡的集合枚举树，这验证了 2.3.1 节中有关 ASE 树平衡性的观点。

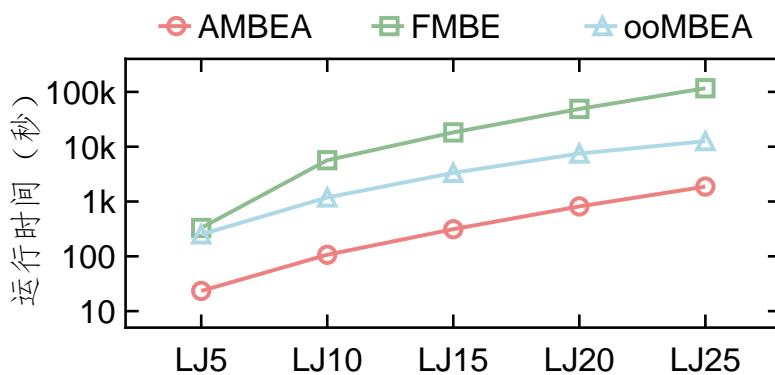


图 2.11 算法扩展性敏感性分析（对数形式）

可扩展性评估：为了探索 AMBEA 的可扩展性，我们选择了 FMBE 和 ooMBEA 作为对比算法，因为它们在整体性能实验中在大规模数据集上表现出色。如图 2.11 所示，实验结果表明，尽管随着数据集规模的增加，不同算法的运行时间呈指数级增加，但是 AMBEA 算法始终在大规模合成数据集上表现出比现有方法更好的性能。具体而言，AMBEA 算法在大规模合成数据集上的运行速度比 FMBE 快 47.2 倍至 141.7 倍，比 ooMBEA 快 6.7 倍至 11.1 倍。特别是在 LJ25 数据集上，AMBEA 算法的完成时间为 1,870 秒，而 FMBE 和 ooMBEA 分别需要 116,582 秒和 12,614 秒。从实验结果可以看出，AMBEA 算法具有良好的可扩展性，在处理大规模数据集时显示出明显的优势。

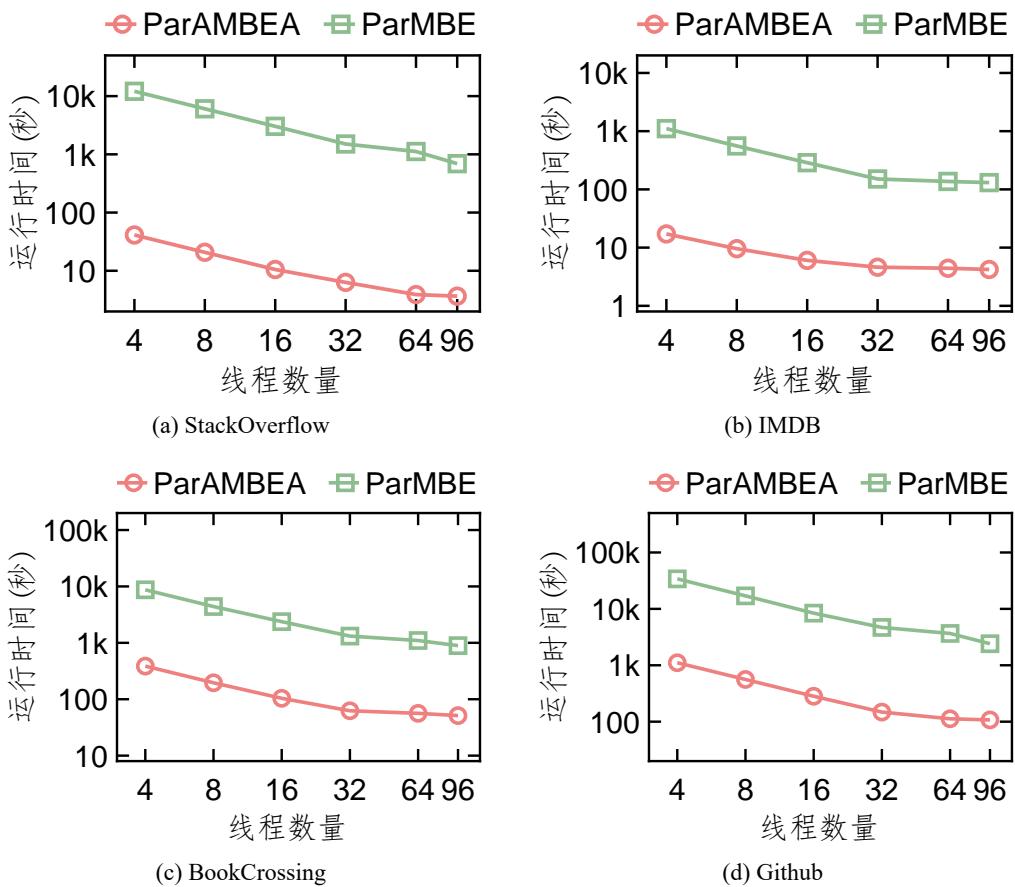


图 2.12 算法并行性敏感性分析（对数形式）

并行性评估：为了探索 AMBEA 的并行性，我们将其并行版本 ParAMBEA 与最先进的并行算法 ParMBE 进行比较。我们在四个运行时间最长的数据集上进行了实验：StackOverflow、IMDB、BookCrossing 和 Github。我们分别使用 4、8、16、32、64、96 个线程在不同数据集上分别运行 ParAMBEA 和 ParMBE 算法。由于实验服务器中共包

含 96 个计算核，我们将线程数量上限设置为 96。如图 2.12 所示，ParAMBEA 相较于 ParMBE 具有明显的性能优势，运行时间缩短了 17.4-293.8 倍。同时，从图表中我们可以观察到，随着线程数量的增加，ParAMBEA 的运行时间几乎呈线性下降趋势，这意味着它可以有效地利用更多的计算资源来加速算法的执行。值得注意的是，通过并行化，我们将 BookCrossing 数据集上 AMBEA 算法的运行时间从 1,610 秒大幅缩短至仅需 51 秒。这个结果充分证明了 ParAMBEA 在并行性能方面的优越性，并显示出其在大规模数据集上的应用潜力。

2.5 本章小结

本章提出了主动的集合枚举树 (ASE 树) 和主动的顶点合并剪枝 (AMP) 两种剪枝优化方法，并结合两种方法形成高效的极大二分团枚举算法 AMBEA，旨在解决极大二分团问题中搜索空间大的挑战。首先，针对现有集合枚举树的局限性，即只允许使用部分顶点扩充枚举节点内二分团的结构限制，ASE 树允许使用全部顶点扩展二分团，从而释放剪枝潜力，并生成更加平衡的集合枚举树。其次，针对现有剪枝方法的依赖特定顶点被动执行的问题，AMP 方法通过修改节点生成过程，以较小的成本主动合并具有相同局部邻居的顶点，同时提供了降低节点生成过程时间复杂度的新思路。最后，我们将上述两种技术整合，实现了 AMBEA 算法及其并行版本 ParAMBEA。实验结果充分证明了 AMBEA 算法的高性能以及本章中所有剪枝方法的具体作用。

3 自适应的极大二分团枚举算法

针对极大二分团枚举问题计算不规则、静态数据结构效率低下的问题，本章整理了相关工作中不同的图存储数据结构，详细描述不同数据结构的优劣以及适用场景，并指出静态数据结构所导致的局限性，作为本章的研究动机。具体而言，现有算法通常使用静态图结构完成枚举，但忽略了枚举过程中计算子图动态变化的特性，导致在原图中进行大量无效内存访问；同时，现有算法采用邻接表存储二分图，导致高昂的集合运算开销。在这些因素的限制下，现有算法仅能处理包含不超过 1 亿极大二分团的小数据集。为了解决这些问题，本章提出以下优化方法：首先，本章提出了基于局部计算子图的优化方法，通过动态缓存枚举过程中的计算子图，优化算法的枚举过程，从而减少无效的顶点访问、集合运算和枚举节点。然后，本章提出基于位图的动态子图方法，利用位图，通过规则的位运算加速小型计算子图中的集合运算。最后，本章结合上述两个技术点，形成自适应的极大二分团枚举算法 AdaMBE。实验证明，AdaMBE 算法比现有工作的运行时间缩短 1.6-49.7 倍，且能够应用于超过 190 亿极大二分团的 TVTropes 数据集。

3.1 二分图的存储结构

二分图的存储结构对极大二分团枚举算法中集合运算的效率具有直接影响，在极大二分团枚举过程中扮演着关键角色。图 3.2 展示了图 3.1 中二分图 G_2 的三种二分图存储结构，包括邻接表、位图和哈希表。每种数据结构在时间和空间开销上都有其独特的优势和局限性，因此在实际应用中需要权衡各种因素，选择最适合的存储结构解决问题。

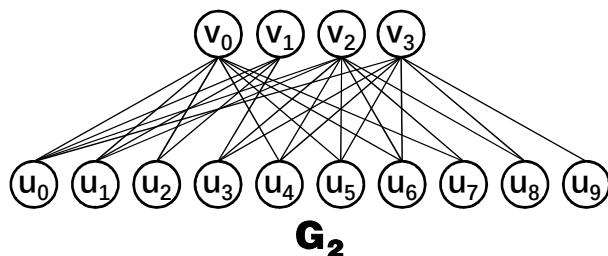


图 3.1 二分图 G_2

图 3.2 二分图 $G_2(U, V, E)$ 的三种存储结构

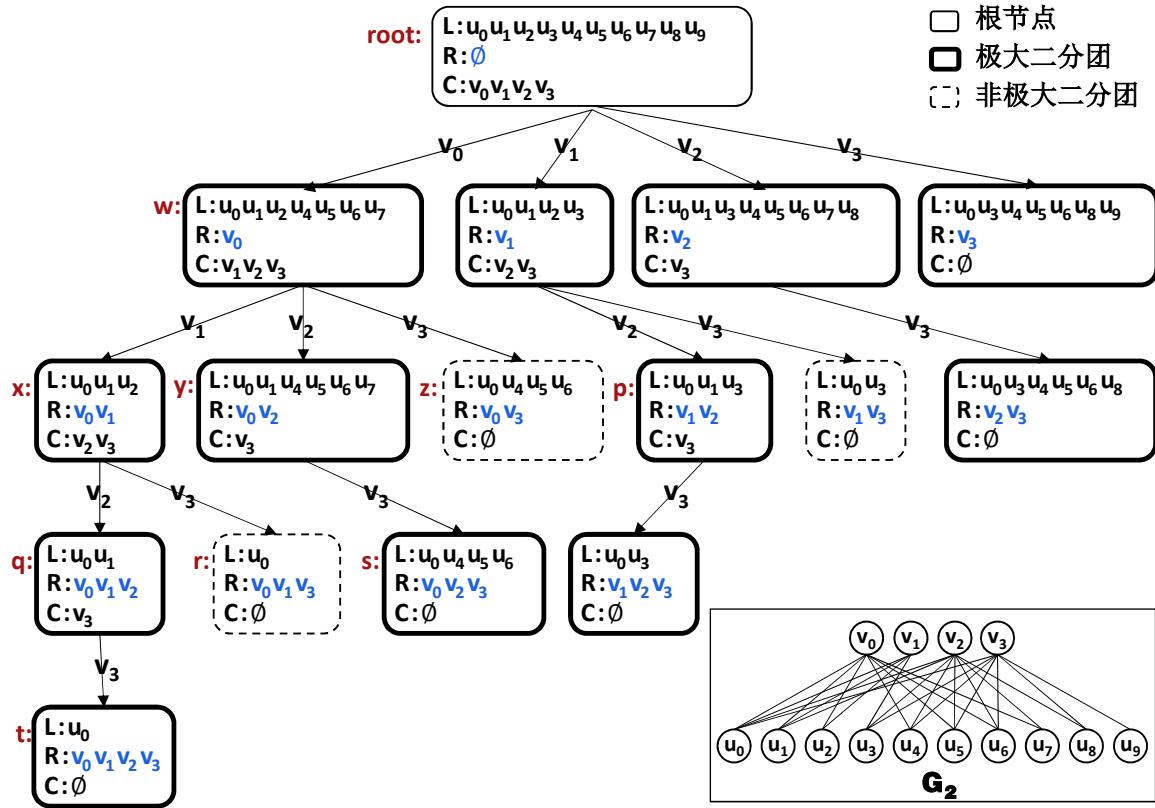
邻接表：邻接表是一种常见的图存储结构，具有低存储使用的优点，特别适用于稀疏图。在邻接表中，每个顶点都与一个包含其相邻顶点的链表相关联。这种存储结构节约内存，因为它只存储实际存在的边，而不需要额外的空间来存储缺失的边，因此它的空间开销仅为 $O(|E|)$ 。此外，邻接表使得遍历图变得更加高效，因为我们可以遍历每个顶点的邻居链表来访问相邻顶点。邻接表在许多图算法中表现良好，特别是针对稀疏图的算法。然而，对于集合交集运算，邻接表可能会受到顶点度数较大的影响，因为每次交集操作都需要按顺序访问一个顶点的邻居，时间复杂度为 $O(\Delta(V))$ 。在处理稠密图时，邻接表可能不如其他存储结构效率高，因为缺失的边在这类图中的占比较小，导致邻接表的低存储使用的优点无法凸显。邻接表存储结构广泛应用于最新的极大二分图枚举算法中，例如 PMBE^[47] 和 ooMBEA^[48]。

位图：位图是另一种常见的图存储结构，在小图中具有计算高效的特点，可以通过位运算实现集合交集运算，通常用于小图或密集图的存储。如图 3.2 所示，在位图中，每个元素即一个位 (Bit)，代表两个顶点的连接关系。与邻接表不同，位图在执行集合交集操作时不受顶点度数限制，因为它直接对应于所有可能的边。这使得位图在处理高度连通图时表现出色。在二分图中，通过两个 $|U|$ 位的位图之间进行按位与操作 ($\&$)，可以计算出两个顶点的共同邻居，这个集合交集运算的时间复杂度为 $O(|U|)$ 。对于小图，通过几次按位与操作，它可以在 $O(1)$ 内高效地执行集合交集。然而，位图可能需要大量内存，特别是对于大规模稀疏图，因为需要为所有缺失的边分配位。这可能会导致内存消耗过大，不适用于资源有限的情况。因此，位图存储结构仅应用于早期的基于小图的极大二分团枚举算法中，包括 LCM-MBC^[61] 和 iMMEA^[15]。

哈希表：哈希表是一种基于散列函数的数据结构，适用于快速查找和插入元素的场景。在哈希表存储法下，二分图中每个顶点的邻居存放在一张哈希表中，每张哈希表通过哈希函数，能够在 $O(1)$ 的时间内快速地查找某顶点是否在这张哈希表中。因此，在实现集合交集操作 $A \cap B$ 时，假设集合 A 顶点数多于集合 B 的顶点数，我们可以通过将集合 B 中的每个元素分别在集合 A 对应的哈希表中进行查找的方式实现，因此集合交集运算的复杂度为 $O(|B|)$ 。由此可知，哈希表在两个输入集合差异显著时，计算效果尤为明显。同时，部分完美哈希函数^[90-91] 能够实现空间复杂度与输入集合长度相同，因此得到与邻接表类似的空间复杂度。基于哈希表的存储结构，Jovan 等人优化了极大团枚举算法的时间复杂度^[23]。然而，相比于邻接表，哈希表存储结构可能涉及大量的随机内存访问和更高的内存使用，效率不够理想。哈希表存储结构仅在 ParMBE^[50] 中使用。

3.2 研究动机

尽管现有研究已经提出了许多优化方法来提高枚举效率，但现有的极大二分团枚举算法通常在单一的静态图结构下完成枚举，忽略了枚举过程中计算子图动态变化的特性，导致算法的性能受到静态数据结构的限制。本节将现有算法的数据结构限制归纳为两个方面：在原图中的大量无效内存访问和在邻接表上高昂的集合运算开销，并结合例子进行说明。为了方便表述，本章中默认以算法 1.1 在二分图 G_2 上的集合枚举树为例，如图 3.3 所示。

图 3.3 算法 1.1 在二分图 G_2 上的集合枚举树

3.2.1 原图中大量的无效内存访问

目前，现有的极大二分团算法通常以静态方式存储原始二分图，并在计算过程中直接访问原图中的顶点。然而，我们观察到由活跃顶点产生的计算子图在枚举过程中会动态变化。访问原图中不在当前计算子图的顶点对计算结果并无影响，因此是无效的。这导致现有方法中存在大量的无效内存访问，从而造成性能瓶颈。本节首先定义了计算子图，然后给出了三个关于计算子图特征的逐步深入的观察，最终结合实验数据指出了现有方法的不足之处。

定义 3.1. (计算子图) 对于当前节点 (L, R, C) ，极大二分团枚举算法中的计算子图指的是由 L 和 C 中顶点组成的子图，该子图包含原始二分图中 L 和 C 中的所有边。在后续内容中，我们使用 CG 表示计算子图。

结合 1.2.2 节提到的算法 1.1 以及该算法在真实数据集上的执行结果，关于计算子图的特征，我们得到以下三个逐步深入的观察：

- O1: 计算子图是大小动态变化的。**根据计算子图的定义，每个节点的 $|L|$ 和 $|C|$ 大小不同，因此计算子图的大小也会随着节点 $|L|$ 和 $|C|$ 的大小动态变化。为了进一步说明这一点，我们进行了一系列实验，并在图 3.4 中展示了在 BookCrossing 和 Github 数据集上计算图大小的频率分布情况。图中的每个单元格表示特定 $|L|$ 和 $|C|$ 组合在计算子图中出现的频率，并已用总出现次数对频率进行了归一化处理。值得注意的是，相对于原图，大多数计算子图相对较小，其中 90% 的计算子图中的 $|L|$ 和 $|C|$ 均小于 32。

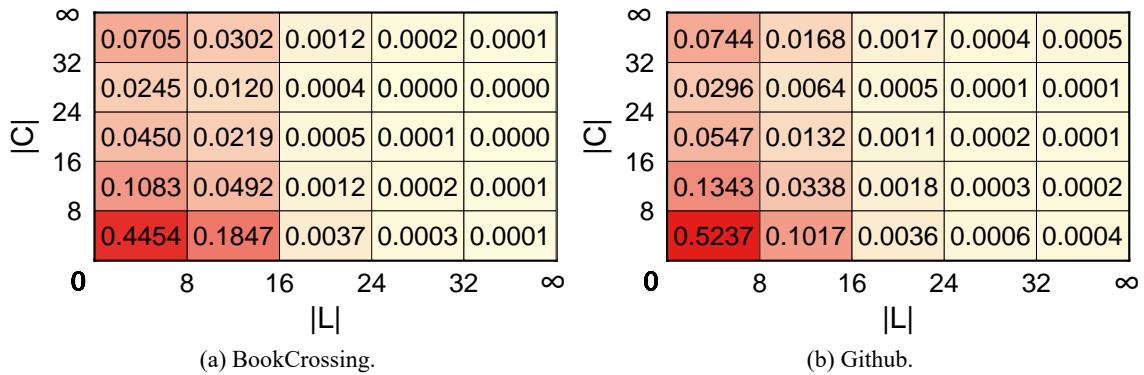


图 3.4 基于 $|L|$ 和 $|C|$ 大小的计算子图频率分布图

- O2: 当前枚举节点的计算子图可直接用于节点生成。**具体而言，在算法 1.1 中，当前节点 (L, R, C) 通过遍历 v' 生成节点 (L', R', C') 。我们知道 L' 是 L 的子集（第 4 行）。根据我们的观察，在节点生成过程中，我们可以将对原图上顶点邻居的访问全部替换为对当前计算子图内顶点局部邻居的访问。具体而言，我们将 $L \cap N(v')$ （第 4 行）推导为 $L \cap N(v') = L \cap (L \cap N(v'))$ ，并将 $L' \cap N(v_c)$ （第 6、8 行）推导为 $L' \cap N(v_c) = (L' \cap L) \cap N(v_c) = L' \cap (L \cap N(v_c))$ ，其中 $L \cap N(v')$ 和 $L \cap N(v_c)$ 表示当前计算子图中顶点 v' 和 v_c 的局部邻居。这样的替换方式可以减少对原图的访问次数，进而提高算法的效率。
- O3: 现有算法需要访问其当前计算子图外的顶点。**尽管计算子图可以直接用于节点生成，但现有算法默认访问原图。然而，这种做法导致大量对当前计算子图外的无效的顶点访问，严重影响了算法的计算性能。如图 3.5 所示，在绝大多数数据集上，这些无效的顶点访问占据了超过 90% 的内存访问量，尤其在耗时较长的 BookCrossing 数据集上，这一比例达到 99.9%。

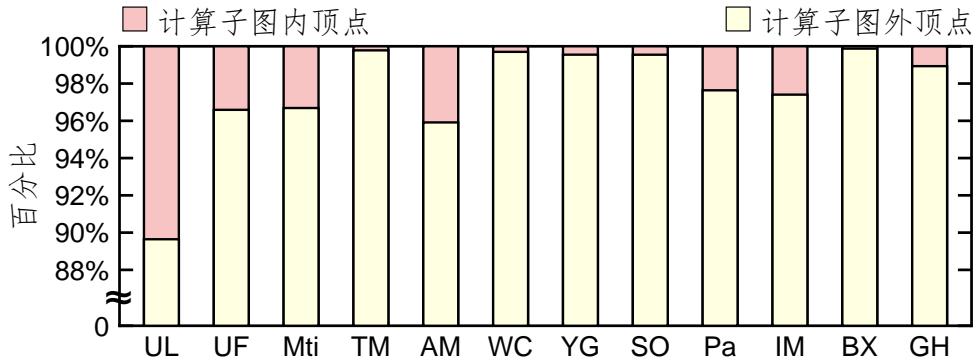


图 3.5 真实数据集中计算图内外顶点访问百分比

总而言之，现有的方法在计算过程中默认对原图进行操作，导致了大量的计算子图外的无效顶点访问。如果我们能够充分利用计算子图内的内存访问，就能在减少无效顶点访问的同时，进一步减少重复集合运算和节点剪枝的操作，从而带来更大的优势。我们将在 3.3.1 节进行详细说明。

3.2.2 邻接表上高昂的集合运算开销

近年来，极大二分图枚举算法通常以邻接表作为存储二分图的首选方式。这主要是因为在处理大规模稀疏图的情况下，相比于其他图存储结构，邻接表结构具有最小的存储开销^[47-48]。然而，基于邻接表进行集合操作需要对两个集合对应的邻接表进行顺序串行比较，带来较大的计算开销。具体而言，在图 3.6 中我们比较了邻接表与位图的集合运算实例。由于邻接表采用顺序连续存储的方式，进行集合操作通常需要串行访问和比较，导致操作时间与集合大小成正比，效率较低。相比之下，位图始终以等长方式存储顶点的所有邻居，因此每次集合操作可以通过固定长度的位运算（如与操作）实现。对于小规模的稠密图，位图能够在常数时间内高效执行集合操作；然而，对于大规模稀疏图，由于其较高的内存需求，使用位图变得不切实际。因此，我们需要一种能平衡内存消耗和集合操作效率的自适应数据结构。

例 3.1. 图 3.6 展示了二分图 G_2 上 v_0 和 v_1 的共同邻居在邻接表和位图中的计算过程。图 3.6a 则展示了一种基于邻接表的集合交集运算计算过程，采用双指针法。具体而言，首先，我们初始化两个指针，分别指向两个已排序集合的起始位置，并在集合中用绿色和蓝色标记指针位置。随后，我们不断地比较两个指针指向的元素。如果相等，则将该元素添加到交集中，并同时将两个指针向后移动一位（步骤 I、II、III）；如果不相等，则

$\begin{array}{l} N(v_0) \\ \cap \\ (I) \quad N(v_1) \end{array} = \boxed{\begin{array}{ccccccccc} u_0 & u_1 & u_2 & u_4 & u_5 & u_6 & u_7 \end{array}} = \boxed{u_0}$ <p>$\boxed{u_0} = \boxed{u_0}$, 输出 $\boxed{u_0}$, 移动 $\boxed{\text{green}}$ 和 $\boxed{\text{light blue}}$。</p>	$\begin{array}{l} \cap \\ (II) \end{array} = \boxed{\begin{array}{ccccccccc} u_0 & u_1 & u_2 & u_4 & u_5 & u_6 & u_7 \end{array}} = \boxed{\begin{array}{cc} u_0 & u_1 \end{array}}$ <p>$\boxed{u_1} = \boxed{u_1}$, 输出 $\boxed{u_1}$, 移动 $\boxed{\text{green}}$ 和 $\boxed{\text{light blue}}$。</p>
$\begin{array}{l} \cap \\ (III) \end{array} = \boxed{\begin{array}{ccccccccc} u_0 & u_1 & u_2 & u_4 & u_5 & u_6 & u_7 \end{array}} = \boxed{\begin{array}{ccc} u_0 & u_1 & u_2 \end{array}}$ <p>$\boxed{u_2} = \boxed{u_2}$, 输出 $\boxed{u_2}$, 移动 $\boxed{\text{green}}$ 和 $\boxed{\text{light blue}}$。</p>	$\begin{array}{l} \cap \\ (IV) \end{array} = \boxed{\begin{array}{ccccccccc} u_0 & u_1 & u_2 & u_4 & u_5 & u_6 & u_7 \end{array}} = \boxed{\begin{array}{ccc} u_0 & u_1 & u_2 \end{array}}$ <p>$\boxed{u_4} > \boxed{u_3}$, 移动 $\boxed{\text{light blue}}$。</p>
$\begin{array}{l} \cap \\ (V) \end{array} = \boxed{\begin{array}{ccccccccc} u_0 & u_1 & u_2 & u_4 & u_5 & u_6 & u_7 \end{array}} = \boxed{\begin{array}{ccc} u_0 & u_1 & u_2 \end{array}}$ <p>$\boxed{\text{light blue}}$ 超出边界, 计算结束。</p>	

(a) 邻接表

$\begin{array}{l} N(v_0) \\ \cap \\ (V) \end{array} = \boxed{\begin{array}{ccccccccc} u_0 & u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 & u_8 & u_9 \end{array}} = \boxed{\begin{array}{ccccccccc} u_0 & u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 & u_8 & u_9 \end{array}}$	$\begin{array}{l} \cap \\ (V) \end{array} = \boxed{\begin{array}{ccccccccc} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{array}} = \boxed{\begin{array}{ccccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}}$
--	--

(b) 位图

图 3.6 邻接表与位图的集合交集运算比较实例 $N(v_0) \cap N(v_1)$

将指向较小元素的指针向后移动一位 (步骤 IV)。最后, 我们重复以上过程, 直到其中一个集合的指针到达末尾为止 (步骤 V)。除双指针法外, 基于邻接表的集合运算方法还包括融合路径法^[92-93]、二分法^[80, 94]等。然而, 由于邻接表顺序连续存储的特性, 集合运算至少需要对两个输入集合中的一个进行串行顺序访问。这意味着集合运算所需的时间与集合的大小直接相关, 导致了较低的运算效率。

图 3.6b 展示了一种基于位图的集合交集运算计算过程, 采用位运算。具体来说, 在二分图 $G_2(U, V, E)$ 中, 集合 V 中每个顶点 v 的邻居存储在一个长度为 $|U|$ 的位图中, 其中每一位代表 v 的邻居中是否存在对应元素。在位图存储下, 我们通过对两个位图进行按位与 (&) 操作, 即可得到运算结果。因此, 在 $|U|$ 较小的情况下, 由于位操作的计算复杂度较低, 基于位图的集合交集运算更加高效。

幸运的是，考虑到计算子图动态变化的特性，以及大多数计算子图规模较小的情况（如图 3.4 所示），因此利用位图操作加速小型计算子图中的集合运算是一个可行的方案。我们将在 3.3.2 节对此进行详细讨论。

3.3 AdaMBE 算法

为了克服现有方法的不足，本节提出了基于局部计算子图的优化方法和基于位图的动态子图方法，以提升极大二分团枚举算法的效率。其中，基于局部计算子图的优化方法利用计算子图中的局部邻居信息，加速了极大二分团枚举算法的核心操作，同时减少了算法中的无效的顶点访问、集合运算以及枚举节点；而基于位图的动态子图方法利用位图，通过规则高效的位运算加速了小型计算子图中的集合运算，在集合运算性能和访存效率之间进行了良好的折中。最终，我们结合这两种技术，形成了高效的自适应极大二分团枚举算法 AdaMBE。

3.3.1 基于局部计算子图的优化方法

为了减少原图中的无效顶点访问，我们提出了一种基于局部计算子图的优化方法，即 LCG (Local Computational subGraph) 方法。其核心设计思想是 **动态缓存局部计算子图，并基于该子图重新设计算法的核心操作**，以提升枚举性能。具体而言，我们观察到每个节点需要计算候选顶点的局部邻居作为中间结果。因此，我们可以通过缓存候选顶点的局部邻居这一中间结果，来得到局部计算子图。这里，顶点 v 在节点 x 中的 局部邻居被定义为 $N(v)$ 和 L_x 的交集，在本章中被记作 $N_x(v)$ 。为了提升计算性能，我们利用计算子图中的局部邻居重新设计了 1.2.2 节算法 1.1 中的三个核心操作：

(1) **减少计算 R' 和 C' 时的无效顶点访问。** 在当前的极大二分团枚举算法中，计算当前节点的集合 R' 和 C' 需要计算每个候选顶点 v_c 在当前节点的局部邻居 $L' \cap N(v_c)$ 。这一过程对应于算法 1.1 中的第 6 和第 8 行。该过程默认访问顶点 v_c 在原图中的邻居 $N(v_c)$ 。然而，根据第 3.2.1 节中的观察 O2，我们采用了一种优化策略：使用顶点 v_c 在当前节点父节点 x 对应计算子图中的局部邻居 $N_x(v_c)$ 替代原图中邻居 $N(v_c)$ 的访问。由于计算子图中的局部邻居 $N_x(v_c)$ 通常规模较小，远远少于原图中的全部邻居 $N(v_c)$ ，因此计算 $L' \cap N_x(v_c)$ 的开销明显低于计算 $L' \cap N(v_c)$ 。为了便于获取顶点的局部邻居，我

们将每次计算候选顶点 v_c 在当前节点局部邻居 $L' \cap N(v_c)$ 这一中间结果缓存起来，以避免不必要的顶点访问。

局部邻居 $L' \cap N(v_c)$ 的计算是现有极大二分团枚举算法中的必要步骤。通过缓存局部邻居这一中间结果，我们可以得到每个节点对应的计算子图，进而消除计算子图之外的无效顶点访问，提高枚举效率。具体而言，通过缓存局部邻居，我们平均减少了 97.1% 的计算子图外的无效顶点访问，如图 3.5 所示。鉴于极大二分团枚举问题是计算密集型任务，内存使用较少，利用一些额外的内存来存储局部邻居以减少运行时间是值得的。

(2) 消除计算 L' 时的重复集合运算。我们注意到现有极大二分团枚举算法中存在重复的集合运算。具体而言，根据算法 1.1，计算当前节点集合 L' 的过程 $L \cap N(v')$ （第 4 行）与其父节点 (L, R, C) 计算候选顶点 v' 局部邻居的过程（第 8 行）完全相同。因此，通过缓存 v' 的局部邻居，我们可以避免重复计算 L' 的集合运算。尽管这个问题在现有的极大二分团枚举算法中很常见，但还没有得到解决。

(3) 节点生成前裁剪无效枚举节点。根据我们的观察，我们注意到每个顶点的局部邻居始终对应于枚举节点的集合 L ，如算法 1.1 的第 4 行所示。因此，我们可以知道相同的局部邻居将导致具有相同集合 L 的节点，从而产生无效枚举节点。具体而言，当节点 q 是节点 p 的子节点且顶点 v 的局部邻居大小在两个节点中相等，即 $|N_p(v)| = |N_q(v)|$ 时，节点 p 可以安全地裁剪掉通过遍历顶点 v 节点生成的无效枚举节点。因为 $N_p(v)$ 总是包含 $N_q(v)$ 中的所有顶点，邻居大小的相等即可表明 $N_p(v)$ 和 $N_q(v)$ 是相同的。因此，我们可以通过确保每个局部邻居仅生成一个节点来实现枚举节点的剪枝。

接下来，我们用一个实例进行说明：

例 3.2. 图 3.7 展示了在图 3.1 中节点 w 为根节点的子树中使用 LCG 方法的过程。LCG 方法在枚举过程中，将候选顶点的局部邻居默认缓存到当前节点对应的计算子图中。为了区分不同节点对应的计算子图和局部邻居，我们使用不同的颜色进行标示。

首先，LCG 方法通过访问顶点在计算子图中的局部邻居，避免了对顶点全部邻居的遍历，从而消除了在当前计算子图之外的无效顶点访问。例如，节点 x 利用其父节点 w 的计算子图中缓存的局部邻居 $N_w(v_2)$ 计算 $N_x(v_2)$ ，有效地避免了对当前计算子图之外的无效顶点的访问，如 u_3 和 u_8 。其次，LCG 方法通过直接访问父节点的计算子图获取当前节点的集合 L ，避免了重复的集合交集运算。例如，节点 x 通过直接访问其父节点 w 的计算子图中 v_1 的局部邻居，即 $N_w(v_1)$ ，得到 L_x 。最后，LCG 方法通过比较父节点

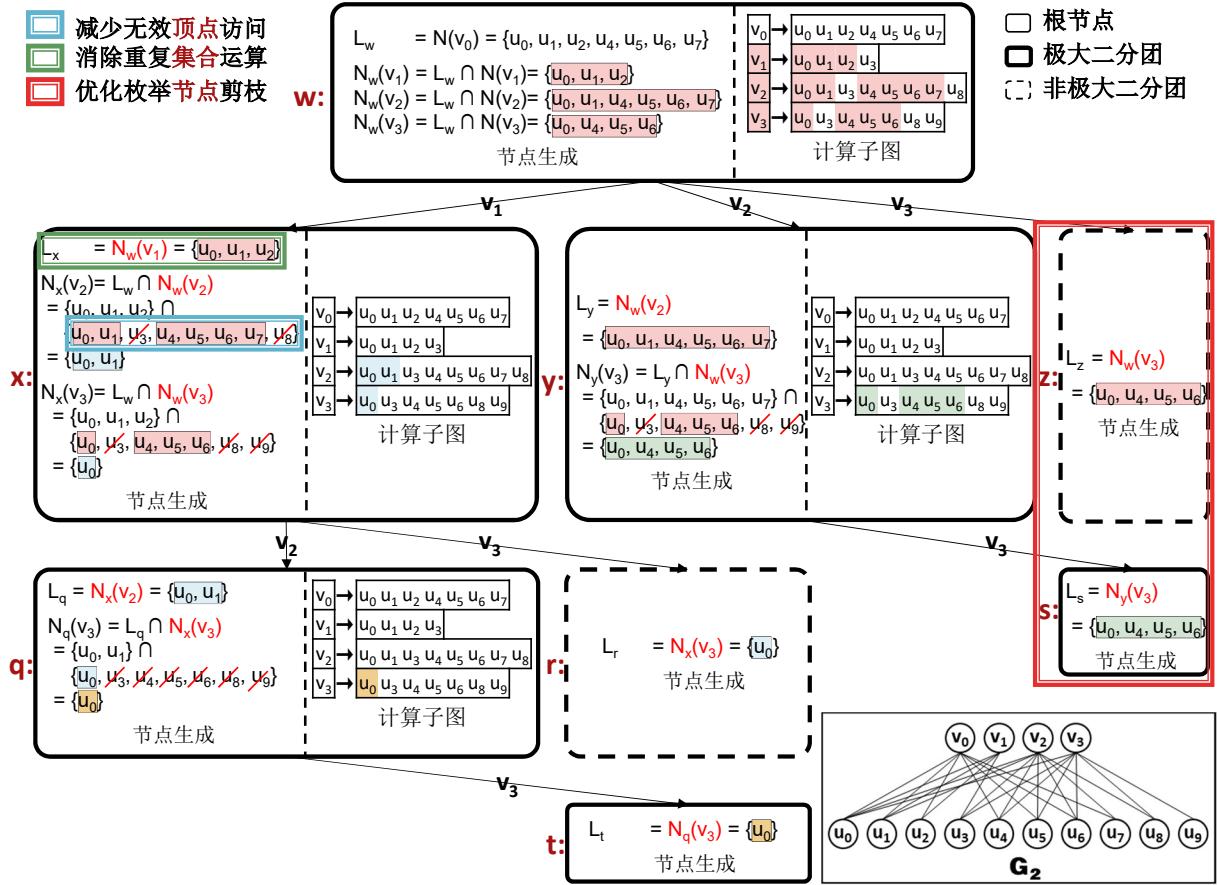


图 3.7 基于局部计算子图的优化方法示例

和子节点的中候选顶点的局部邻居大小来实现节点剪枝。例如，节点 w 通过比较节点 w 和 y 中顶点 v_3 的局部邻居大小相等，即 $|N_w(v_3)| = |N_y(v_3)| = 4$ ，实现对由顶点 v_3 产生的节点 z 的剪枝。

LCG 方法的主要贡献包括以下两点：

- 提升中间结果的利用率。**在现有的极大二分团枚举算法中，局部邻居作为必要的中间结果，扮演着至关重要的角色。LCG 方法通过缓存这些中间结果来动态缓存节点的计算子图，并利用该子图实现了对枚举过程的优化。虽然 LCG 方法需要额外的内存来存储局部邻居，但是如图 3.5 所示，这一方法平均减少了无效顶点的内存访问量达到了 97.1%，从而提升了效率。鉴于内存并非计算密集型极大二分团枚举问题的瓶颈，使用额外的内存空间来缩短计算时间是值得的。
- 同时解决枚举过程中的多种问题。**在缓存了计算子图之后，我们充分利用了这些子图，解决了枚举过程中的多个问题，包括无效顶点访问、重复集合运算和无效

枚举节点。与传统的优化方法只注重剪枝无效枚举节点不同，LCG 方法通过巧妙地利用缓存的计算子图，高效解决了极大二分团枚举中的无效顶点访问问题和重复集合运算问题。当前，图挖掘领域的最新算法开始关注重复集合运算导致的计算低效性问题^[95-96]。然而，与这些现有方法相比，我们提出的 LCG 方法额外关注了更细粒度的顶点访问层面的问题，并解决了这些问题，为解决图挖掘领域中无效顶点访问的问题提供了全新的思路。

3.3.2 基于位图的动态子图方法

为了优化邻接表上高昂的集合运算开销，我们提出了一种基于位图的动态子图方法，即 BDS (Bitmap-based Dynamic Subgraph) 方法。其核心思想是利用位图加速小型计算子图中的集合运算操作。如 3.1 节所述，计算子图的存储结构是计算效率和内存效率之间的权衡。因此，BDS 方法采用混合图存储结构：对于原图或者大的计算子图，我们用邻接表来表示，以提高内存效率；对于小的计算子图，我们用动态建立的位图来表示，以提高集合运算效率。

具体而言，我们观察到枚举过程中计算子图是不断缩小的。根据算法 1.1 所述，新节点 (L', R', C') 中的集合 L' 和 C' 中的全部顶点均源自父节点 (L, R, C) 对应的集合 L 和 C (第 4、8-9 行)，即 $L' \subset L$ 且 $C' \subset C$ 。由此可知，随着枚举过程的深入，根据节点内集合 L 和 C 产生的计算子图也在不断缩小。因此，我们设计的 BDS 方法可以在当前节点对应的计算图大小小于给定的阈值时动态建立子位图，并充分发挥位图在高效集合交集运算方面的优势，在以当前节点为根节点的子枚举树中重用该子位图。

BDS 方法的实现细节，我们通过回答以下两个关键问题的方式进行详细阐述：

(1) 如何创建和利用基于位图的计算子图？对于给定的节点 (L^*, R^*, C^*) ，我们利用位图从原始图 $G(U, V, E)$ 中创建计算子图 $CG_{bit}(U', V', E')$ 。为了构建 U' ，我们选择了所有 L^* 中的顶点。这是因为 $U \setminus L^*$ 中的顶点无法存在于 L' 中，不会对算法 1.1 (第 4、6、8 行) 中的中间结果产生影响。为了构建 V' ，我们选择了不包括 R^* 内顶点在内的所有与 L^* 中任意顶点相连的顶点，存储为 $\bigcup_{u \in L^*} N(u) - R^*$ 。这是因为未与 L^* 中任何顶点相连的顶点不能存在于 C' 或 $\Gamma(L')$ 中，也不能对算法 1.1 中的中间结果产生影响。我们从 V' 中排除 R^* 中的顶点，因为它们总是出现在所有子节点的 R' 和 $\Gamma(L')$ 中 (第 12 行)，这样可以减少不必要的计算。 E' 包含原图内 U' 和 V' 之间的所有边。当 U' 的大

小（即 $|L^*|$ ）小于某个给定的小的常数 τ 时，算法中的每个集合操作都可以通过按位与操作 ($\&$) 在 $O(\tau) = O(1)$ 的时间内完成。我们通过检查对于所有 V' 中的顶点 v 是否满足 $L' \cap N(v) = L'$ 来计算 $\Gamma(L')$ 。这一节点检查过程需要 $O(\tau \Delta(U)) = O(\Delta(U))$ 的时间，因为 V' 最多包含 $\tau \Delta(U)$ 个顶点，且每个顶点的集合交集运算只需 $O(1)$ 的时间。综上所述，创建和利用基于位图的计算子图使得能够加速极大二分团枚举算法中的集合运算过程，其中每个节点的计算时间为 $O(\Delta(U))$ 。

(2) 何时创建基于位图的计算子图？为了优化集合交集操作并提高计算效率，在考虑当前节点 (L^*, R^*, C^*) 的情况下，我们在满足以下条件时创建基于位图的计算子图：当 $|L^*|$ 小于或等于阈值 τ 且 C^* 不为空时。这些条件至关重要，因为 $|L^*|$ 直接影响集合交集所需的时间，而 C^* 影响位图在子节点中的重用次数。然而，选择适当的阈值 τ 是具有挑战性的。较大的 τ 会增加集合交集的时间（即 $O(\tau)$ ）和每个子图的内存使用。相反，较小的 τ 会导致创建更多小的子图，从而限制了在子节点中位图的重用机会。因此，在确定 τ 时，我们必须在集合交集计算的效率和位图的利用之间取得平衡。根据我们在 3.4.4 节的实验，我们建议将阈值 τ 设为 64，因为这种情况下每次操作只需要对两个 64 位长长整数（long long integer）进行高效的按位与操作。

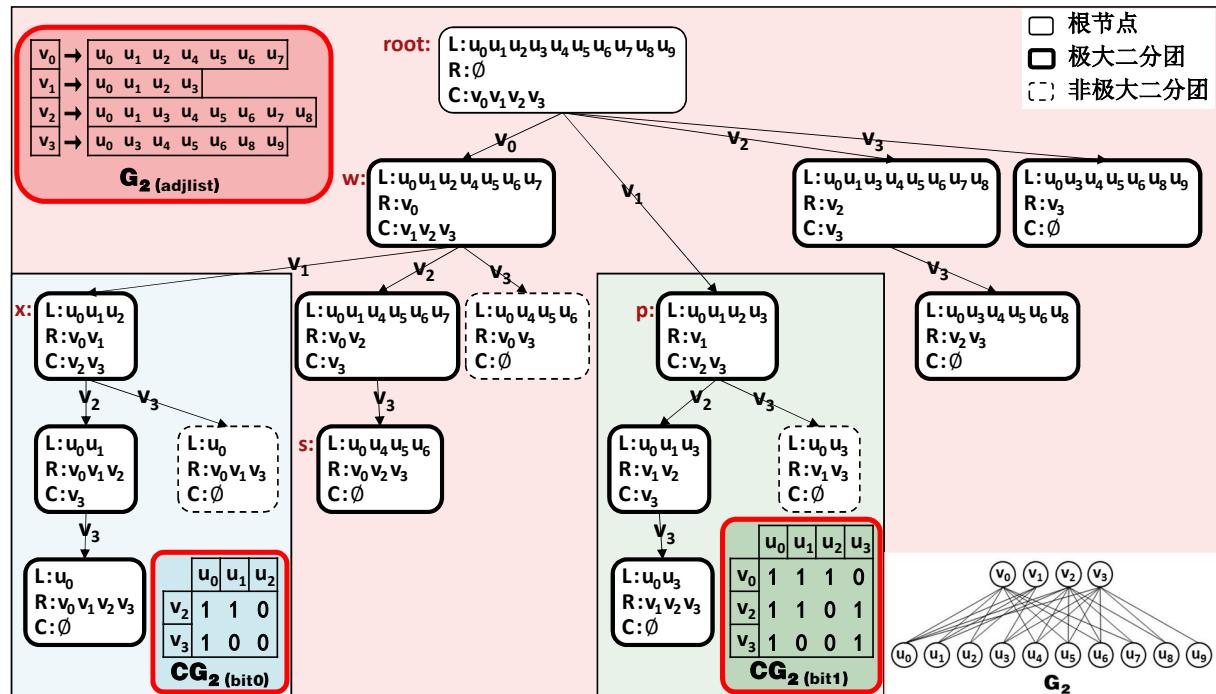


图 3.8 基于位图的动态子图方法示例

例 3.3. 图 3.8 展示了在二分图 G_2 上应用 BDS 方法进行极大二分团枚举的过程。在这个示例中，设定阈值 τ 为 4。初始时，图 G_2 采用邻接表作为默认存储方式（标记为 $G_{2(adjlist)}$ ）。当节点 w 遍历顶点 v_1 进入到节点 x 时，我们观察到 $|L_x| = 3 < \tau < |L_w| = 7$ ，且 C_x 非空，因此在节点 x 处生成基于位图的计算子图 $CG_{2(bit0)}$ 。在 $CG_{2(bit0)}(U_0, V_0, E_0)$ 中， U_0 包含 L_x 中的所有顶点，即 u_0 、 u_1 和 u_2 。 V_0 包含 v_2 和 v_3 ，因为它们与 L_x 中的某些顶点相连，但不包括 v_0 和 v_1 ，因为它们属于 R_x 。因此，利用 $CG_{2(bit0)}$ 中的位图，可以加速计算节点 x 的所有后继节点的集合交集运算。类似地，在节点 p 处，创建另一个基于位图的计算子图 $CG_{2(bit1)}$ ，以加速计算 p 节点所有后继节点的过程。特别地，在节点 s 处，尽管 $|L_s| = 4 = \tau$ ，但由于节点 s 的候选顶点集 C_s 为空，意味着若创建子位图，将没有子节点能够重复利用，因此在节点 s 处不生成该子位图。

算法 3.1 基于位图的动态子图方法运行过程

输入：当前枚举节点 (L_p, R_p, C_p) ，基于位图的计算子图 CG_{bit}

输出：所有以节点 (L_p, R_p, C_p) 为根节点的枚举子树中的极大二分团

```

1: procedure biclique_search_adambe_bit( $L_p, R_p, C_p, CG_{bit}$ ) :
2:   for  $v' \in C_p$  :
3:      $L_q \leftarrow L_p \& N_{bit}(v')$ ;
4:      $is\_maximal \leftarrow \text{True}$ ;
5:     for  $v'' \in U_{bit} \setminus (R_p \cup C_p)$  :
6:       if  $L_q = L_p \& N_{bit}(v'') \neq 0$  then
7:          $is\_maximal \leftarrow \text{False}$  ;
8:       end if
9:     end for
10:    if  $is\_maximal$  then
11:       $R_q \leftarrow R_p; C_q \leftarrow \emptyset$  ;
12:      for  $v_c \in C_p$  :
13:        if  $L_q \& N_{bit}(v_c) = L_q$  then
14:           $R_q \leftarrow R_q \cup \{v_c\}$  ;
15:        else if  $L_q \& N_{bit}(v_c) \neq 0$  then
16:           $C_q \leftarrow C_q \cup \{v_c\}$  ;
17:        end if
18:      end for
19:      输出极大二分团  $(L_q, R_q)$  ;
20:      biclique_search_adambe_bit( $L_q, R_q, C_q, CG_{bit}$ ) ;
21:    end if
22:     $C_p \leftarrow C_p \setminus \{v'\}$  ;
23:  end for
24: end procedure

```

算法 3.1 描述了基于位图的动态子图方法的运行过程。该过程的输入包括当前枚举

节点 (L_p, R_p, C_p) , 以及基于 BDS 方法实现细节 (1) 建立的基于位图的计算子图 CG_{bit} 。类似于算法 1.1, 该算法按顺序遍历 C_p 中的每个顶点 v' , 生成新的枚举节点 (第 2 行), 并递归调用 `biclique_search_adambe_bit` 过程 (第 20 行)。通过利用基于位图的计算子图 CG_{bit} , 该过程通过位运算操作 (`&`) 代替高开销的集合运算操作, 以高效实现节点检查 (第 4-9 行) 和节点生成 (第 3、11-18 行) 过程。我们用红色字体标记了位运算的使用。值得注意的是, 在递归过程中, 计算子图 CG_{bit} 会被反复重用 (第 1、20 行)。

BDS 方法的主要贡献包括以下两点:

1. **结合两种不同图存储结构的优势。**在选择图的存储结构时, 我们需要在计算效率和内存效率之间取得平衡。对于大规模二分图, 我们采用邻接表存储结构, 因为它具有较好的内存效率; 而对于小规模子图, 则使用位图存储结构, 因为它具有较好的计算效率。我们的 BDS 方法在原始图和较大的子图上继续沿用邻接表存储, 有效地控制了在极大二分团枚举过程中的内存使用。早期的极大二分团枚举算法研究在小规模图 $G(U, V, E)$ 上使用了位图存储^[15, 61]。然而, 即使在这些小规模图中, 如果 U 内的顶点数量不足够小, 每次按位集合交集操作的时间复杂度仍为 $O(|U|)$, 而非 $O(1)$ 。因此, 在许多情况下, 直接采用位图存储方式未必能达到理想的计算效率。相反, BDS 方法通过设置一个小小的阈值 τ 限制 U 的大小, 保证每次集合操作的完成时间为 $O(1)$, 从而保证了计算效率。
2. **充分利用二分图的特点。**相较于通用图, 基于位图的计算子图技术在二分图中有着特殊的优势。具体而言, 对于只有一个顶点集 V 的通用图 $G(V, E)$, 确定何时创建位图是一项挑战。具体地, 如果通用图顶点数 $|V|$ 很大且稀疏, 创建位图会导致内存使用效率低下; 相反, 如果 $|V|$ 很小, 则位图无法得到充分重用。最近的子图枚举算法的研究^[79]指出, 由于子位图技术的使用往往带来巨大的内存开销, 因此基于位图的计算子图优化技术仅应用在少数几项工作中^[75, 84, 97]。幸运的是, 在具有两个不相交顶点集 U 和 V 的二分图 $G(U, V, E)$ 中, 我们可以通过小的 $|U|$ 实现高效的内存使用, 并通过大的 $|V|$ 实现位图的有效利用。因此, BDS 方法通过充分利用二分图的独特特性, 显著改进了极大二分团枚举问题。

3.3.3 AdaMBE 算法设计

结合 LCG 方法和 BDS 方法，我们提出了一种自适应的极大二分团枚举算法，简称为 AdaMBE (Adaptive Maximal Biclique Enumeration)。该算法的核心思想是根据极大二分团枚举的不同阶段，分别采用这两种方法：在原图和大计算子图中采用 LCG 方法，利用动态缓存的计算子图提高计算效率；而在小型计算子图中采用 BDS 方法，通过位图加速其中的集合运算操作。LCG 方法和 BDS 方法能够在不同规模计算子图的场景下实现优势互补。此外，针对一个二分图 $G(U, V, E)$ ，AdaMBE 会根据顶点的邻居大小对集合 V 中的所有顶点进行初始排序。我们将在 3.4.4 节通过实验展示其高效性。

算法 3.2 AdaMBE 算法

输入：二分图 $G(U, V, E)$

输出：所有极大二分团

```

1: 将  $V$  中顶点按照邻居数量递增排序;
2: biclique_search_adambe( $U, \emptyset, V, G$ ) ;
3: procedure biclique_search_adambe( $L_p, R_p, C_p, CG_p$ ) :
4:   if  $|L_p| \leq \tau$  并且  $C_p$  不为空 then
5:     创建基于位图的计算子图  $CG_{bit}(U_{bit}, V_{bit}, E_{bit})$  ;
6:     biclique_search_adambe_bit( $L_p, R_p, C_p, CG_{bit}$ ) ;
7:     Return ;
8:   end if
9:   for  $v' \in C_p$  :
10:     $L_q \leftarrow N_p(v')$  ;  $R_q \leftarrow R_p$  ;  $C_q \leftarrow \emptyset$  ;  $CG_q \leftarrow \emptyset$  ;
11:    for  $v_c \in C_p$  :
12:       $N_q(v_c) \leftarrow L_q \cap N_p(v_c)$  ;
13:       $CG_q.insert(N_q(v_c))$  ;
14:      if  $N_p(v_c) = N_q(v_c)$  then
15:         $CG_q.delete(N_p(v_c))$  ;
16:      end if
17:      if  $N_q(v_c) = L_q$  then
18:         $R_q \leftarrow R_q \cup \{v_c\}$  ;
19:      else if  $N_q(v_c) \neq \emptyset$  then
20:         $C_q \leftarrow C_q \cup \{v_c\}$  ;
21:      end if
22:    end for
23:    if  $R_q = \Gamma(L_q)$  then
24:      输出极大二分团  $(L_q, R_q)$  ;
25:      biclique_search_adambe( $L_q, R_q, C_q, CG_q$ ) ;
26:    end if
27:    free( $CG_q$ ) ;  $C_p \leftarrow C_p \setminus \{v'\}$ ;
28:  end for
29: end procedure

```

算法 3.2 描述了 AdaMBE 算法的执行过程。对于二分图 $G(U, V, E)$, AdaMBE 选择将 V 中顶点按照邻居数量递增排序 (第 1 行)。随后, AdaMBE 从根节点 (U, \emptyset, V) 开始, 递归地调用 `biclique_search_adambe` 过程, 初始时默认当前计算子图为原图 G (第 2 行)。`biclique_search_adambe` 过程的输入包括当前枚举节点 (L_p, R_p, C_p) 以及节点 p 对应的计算子图 CG_p (第 3 行)。对于 $|L_p| > \tau$ 的较大的计算子图, AdaMBE 利用局部邻居信息加速枚举过程 (第 9-28 行)。这些局部邻居信息以蓝色字体标记。对于 $|L_p| \leq \tau$ 的较小的计算子图, AdaMBE 动态创建子位图, 并调用算法 3.1 中的 `biclique_search_adambe_bit` 过程 (第 4-8 行)。根据 3.4.4 节的实验结果, AdaMBE 默认将阈值 τ 设置为 64。

接下来, 我们从时间复杂度和空间复杂度两个方面对 AdaMBE 进行讨论。

时间复杂度: 与第 2 章介绍的 AMBEA 算法相似, AdaMBE 算法的时间复杂度由顶点排序时间和集合枚举树生成时间组成。其中, 按照顶点度数升序排列的时间复杂度为 $O(|V|\log(|V|))$, 在邻接表中每个节点的计算时间为 $O(|E|)$, 而在位图中每个节点的计算时间为 $O(\Delta(U))$ 。为了量化算法的计算时间, 我们用 β 表示枚举树中节点的数量, 包括输出极大二分团的节点和产生非极大二分团的节点。我们将 β 分解为两部分, 即 β_0 表示基于邻接表计算的枚举节点, 而 β_1 表示基于子位图计算的枚举节点。最终, 我们得到 AdaMBE 的计算复杂度为 $O(|E|\beta_0 + \Delta(U)\beta_1 + |V|\log(|V|))$ 。由于排序开销通常远小于生成集合枚举树的开销, 因此, AdaMBE 的时间复杂度可以简化为 $O(|E|\beta_0 + \Delta(U)\beta_1)$ 。在 3.4.3 节的技术点分解评估实验中, 我们观察到在真实数据集中, β_1 通常占据了大部分的计算时间, 因此相较于其他算法, AdaMBE 表现出明显的性能提升。

空间复杂度: AdaMBE 算法的空间复杂度可分为两部分: 输入二分图所需空间和极大二分团枚举过程的空间。输入二分图所占用的空间为 $O(|E|)$ 。在枚举过程中, LCG 方法会动态保存顶点的局部邻居, 构成原二分图的一个计算子图, 因此每个节点所需空间为 $O(|E|)$ 。由于枚举树的高度上限为 $O(\Delta(V))$, 因此 AdaMBE 的空间复杂度为 $O(|E| + |E|\Delta(V)) = O(|E|\Delta(V))$, 略高于当前最优算法的空间复杂度 $O(|E| + |V|\Delta(V))$ 。这是因为 AdaMBE 算法中的 LCG 方法需要额外空间以增加局部邻居的重用性, 从而提升计算性能。在 3.4.2 节的整体评估实验中, 我们观察到在测试的真实数据集上, AdaMBE 的最大内存使用仅约为 1GB, 并且在大多数情况下, AdaMBE 算法的内存占用仍低于先进算法 PMBE 和 ooMBEA, 因此相对于 LCG 方法带来的计算性能提升, 其空间开销是可以接受的。

并行扩展: 为了进一步提升性能, 我们在 AdaMBE 算法的基础上设计了其并行版本 ParAdaMBE。受到并行算法 ParMBE^[50] 的启发, ParAdaMBE 利用 Intel TBB 库^[98]的功能, 为二分图集合 V 中的每个顶点 v 分配一个线程, 负责处理由顶点 v 产生的子枚举树。这样, 多个枚举树可以无重叠地并发计算。此外, 只要存在额外线程可用, ParAdaMBE 将对子枚举树对应的任务进行进一步分解。为了进一步提升并行计算资源的利用, ParAdaMBE 通过循环展开技术有效并行化算法中的 for 循环, 最大程度地利用并行性缩短算法的运行时间。

总而言之, 与现有的极大二分团枚举方法忽视计算子图的特性不同, AdaMBE 利用这些特性加速顶点访问和集合操作。它根据处理图的大小灵活选择 LCG 和 BDS 技术, 优化了极大二分团枚举过程中最耗时的集合运算过程, 显著改善了计算效率。

3.4 实验评估

3.4.1 实验设置

实验环境设置: 本节的全部实验均在一台配备有 4 个 Intel Xeon (R) Gold 5318Y 2.10GHz CPU 和 128GB 内存的服务器上完成, 其中每个 CPU 拥有 24 个计算核心, 共计 96 个计算核心。本次实验环境的操作系统为 Linux kernel-5.4.0。在没有特殊说明的情况下, 算法默认使用单个计算核心串行执行。鉴于现有算法在大规模数据集上很难在短时间内完成, 我们将运行时间限制设置为 48 小时 (INF)。

比较算法: 在我们的实验中, 我们评估了串行和并行极大二分团枚举算法的性能。对于串行算法, 我们将 AdaMBE 与过去五年中的三种最先进的算法进行了比较: FMBE^[50], PMBE^[47] 和 ooMBEA^[48]。对于并行算法, 我们将 ParAdaMBE 与目前最先进的多线程并行算法 ParMBE^[50] 进行比较。默认情况下, 我们在同一台拥有 96 个核心的 CPU 的服务器上使用 96 个线程来运行 ParAdaMBE 和 ParMBE。此外, 为了深入评估本章提到的技术点, 我们实现了一些算法变体, 并在对应实验中详细描述。

数据集: 为了准确评估本章提出的技术, 我们同时使用了真实数据集和合成数据集。为了进一步探索 AdaMBE 在包含超过 1 亿极大二分团的大数据集上的性能, 我们在上一章实验中的表 2.1 的基础上额外增加了大数据集, 如表 3.1 所示。其中, 真实数据集 CebWiki 和 TVTropes 来源于 KONECT 仓库^[89], TVTropes 数据集包含的极大二分团数

表 3.1 AdaMBE 增加的实验数据集统计信息

数据集	目录	类型	$ U(G) $	$ V(G) $	$ E(G) $	极大二分团数量
CebWiki (ceb)	作者关系	用户-编辑-文章	8,483,068	3,132	11,792,890	263,138,916
TVTropes (DBT)	特征关系	作品-拥有-特征	87,678	64,415	3,232,134	19,636,996,096
LJ10	归属关系	用户-属于-群组	2,301,031	1,421,088	11,227,130	7,430,705
LJ20	归属关系	用户-属于-群组	2,704,651	2,357,485	22,456,757	61,836,924
LJ30	归属关系	用户-属于-群组	3,163,966	2,889,804	33,686,334	343,257,225
LJ40	归属关系	用户-属于-群组	3,894,262	2,992,774	44,917,368	1,524,229,722
LJ50	归属关系	用户-属于-群组	4,572,628	3,057,410	56,150,150	6,387,845,280

量超过了 196 亿。合成数据集是在大数据集 LiveJournal ($|U|=7,489,073$, $|V|=3,201,203$, $|E|=112,307,385$) 上分别采样 10%、20%、30%、40% 和 50% 的边所生成, 分别记作 LJ10、LJ20、LJ30、LJ40 和 LJ50, 其中 LJ30、LJ40 和 LJ50 中的极大二分团数量均超过 1 亿。大数据集 LiveJournal 也来自于 KONECT 仓库。

3.4.2 整体评估

常规数据集上的性能评估: 为了验证我们提出的算法的整体效率, 我们在 12 个常规数据集上测量了所有竞争对手的运行时间和内存使用情况。运行时间不包括从磁盘加载图的时间, 内存使用表示相应进程的最大内存占用。

表 3.2 AdaMBE 在常规数据集上的整体运行时间评估 (单位: 秒)

数据集	AdaMBE	FMBE	PMBE	ooMBEA	ParAdaMBE	ParMBE
UL	0.00336	0.03059	0.01699	<u>0.00549</u>	0.00358	0.04363
UF	0.044	<u>0.189</u>	0.266	<u>0.215</u>	0.019	0.111
Mti	0.583	14.092	24.123	<u>2.871</u>	0.057	3.741
TM	0.978	7.932	19.165	<u>6.995</u>	0.447	1.371
AM	5.0	123.6	199.3	<u>18.6</u>	1.0	15.9
WC	2.7	<u>13.9</u>	49.1	<u>20.0</u>	0.7	2.9
YG	3.3	<u>38.3</u>	126.6	131.1	1.1	11.5
SO	80.3	3,705.7	10,933.2	<u>941.6</u>	5.0	691.3
Pa	8.4	<u>14.0</u>	23.4	<u>40.1</u>	4.0	10.0
IM	25.4	707.0	1,044.7	<u>214.5</u>	2.5	130.8
BX	186	<u>5,934</u>	13,234	<u>9,452</u>	46	892
GH	194	16,616	51,647	<u>9,639</u>	40	2,412

表 3.2 展示了在常规数据集上, AdaMBE、ParAdaMBE 以及主流极大二分团枚举算法的运行时间对比。为了方便比较, 我们使用下划线标记了每个数据集中性能最优的串行竞争对手。实验结果显示, 在不同的数据集下, 我们的串行算法 AdaMBE 在性能上始

终比最接近的串行竞争对手快 1.6 到 49.7 倍，而并行算法 ParAdaMBE 则比 ParMBE 算法快 2.5 到 137.8 倍。特别值得注意的是，在耗时最长的数据集 Github 上，AdaMBE 仅用时 194 秒完成任务，而 FMBE、PMBE 和 ooMBEA 算法分别需要 16,616 秒、51,647 秒和 9,639 秒。同样，ParAdaMBE 在 40 秒内完成任务，而 ParMBE 则需要超过 2,412 秒。这清楚地表明，与现有方法相比，AdaMBE 算法和 ParAdaMBE 算法在处理耗时较长的数据集时具有明显的性能优势，为大规模数据集上的极大二分团枚举任务提供了强大的技术支持。

表 3.3 AdaMBE 在常规数据集上的整体内存使用评估（单位：MB）

数据集	AdaMBE	FMBE	PMBE	ooMBEA	ParAdaMBE	ParMBE
UL	3.8	3.8	3.7	3.6	5.8	5.7
UF	3.6	4.1	4.3	4.1	6.5	15.6
Mti	7.5	5.8	10.4	12.1	48.7	98.8
TM	129	67	181	100	229	707
AM	85	45	119	175	426	869
WC	283	145	396	237	778	1,538
YG	23	14	33	37	332	182
SO	97	51	157	434	467	797
Pa	1,028	497	1,421	1,031	6,156	5,030
IM	204	104	294	525	1,029	2,507
BX	74	39	124	200	318	599
GH	32	18	55	105	271	431

表 3.3 展示了在常规数据集上，AdaMBE、ParAdaMBE 以及主流极大二分团枚举算法的内存使用对比。实验结果显示，与串行算法 PMBE 和 ooMBEA 相比，AdaMBE 的内存使用平均分别降低了 28.0% 和 29.8%。以数据集 Github 为例，AdaMBE 的内存使用是 32MB，而 PMBE 和 ooMBEA 的内存使用分别是 271MB 和 431MB。这是因为它们需要额外的内存来存储 CDAG 索引结构^[47] 和多个静态子图^[48]。与并行算法 ParMBE 相比，ParAdaMBE 的内存消耗平均降低了 29.7%。除此之外，尽管 FMBE 算法的内存使用量低于 AdaMBE，但其运行时间却远远落后于 AdaMBE。因此，我们可以得出结论，我们的方法在保持高内存效率的同时取得了显著的性能改进。

大规模数据集的评估：为了验证我们提出的算法在大规模二分图中的性能，我们在两个包含超过 2 亿个极大二分团的大规模数据集上测量了不同算法的运行时间。对于那些运行时间超过时间限制（48 小时）的算法，我们报告它们在截止 48 小时时所产生的极大二分团数量。

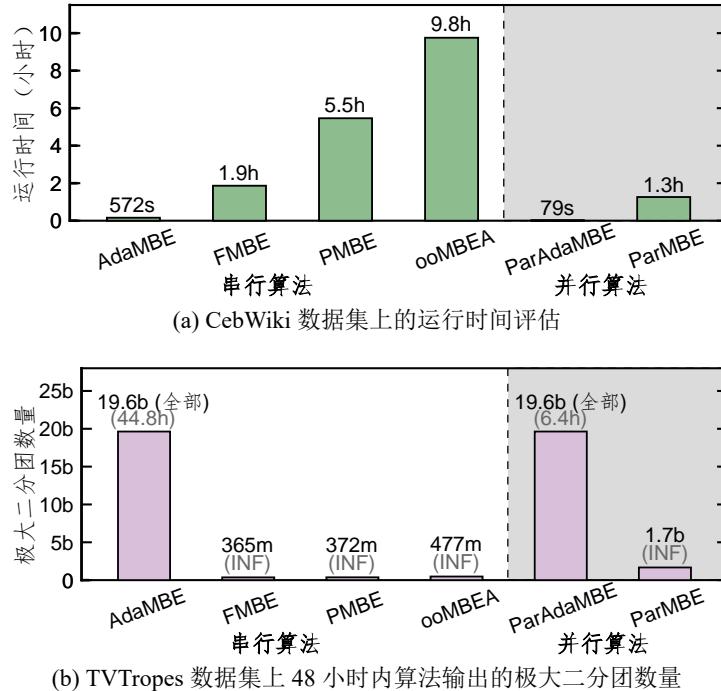


图 3.9 AdaMBE 在超大数据集上的整体评估

图 3.9a 展示了不同算法在 CebWiki 数据集上的运行情况。实验结果显示，AdaMBE 和 ParAdaMBE 的完成时间均小于 10 分钟，而其他现有算法的完成时间均超过 1 小时，最新的极大二分团枚举算法 ooMBEA 的运行时间甚至接近 10 小时。图 3.9b 展示了不同算法在 TVTropes 数据集上的运行情况。实验结果显示，只有 AdaMBE 和 ParAdaMBE 在 48 小时内完成了对全部 196 亿极大二分团的枚举任务，完成时间分别为 44.8 小时和 6.4 小时。而其他比较算法在 48 小时内所枚举的极大二分团数量均不超过总数量的 8.5%。这清楚地表明，由于性能限制，现有方法难以在大规模数据集上得到有效应用。相比之下，AdaMBE 算法通过优化数据结构，充分利用计算子图动态变化的特性，在大规模二分图场景中表现出更为明显的优势。

3.4.3 技术点分解评估

为验证本章提出的具体优化技术，我们设计了消融实验，沿用整体评估中的运行时间和内存使用指标。具体而言，我们在 8 个极大二分团数量较多的常规数据集上，对 3.3.1 提到的 LCG 方法和 3.3.2 节提到的 BDS 方法进行了分解评估。为了评估 LCG 和 BDS 方法的有效性，我们设计了三个变体：基准算法（禁用 LCG 和 BDS）、AdaMBE-LCG（启用 LCG 的基准算法）和 AdaMBE-BDS（启用 BDS 的基准算法）。我们比较了

AdaMBE 和这三个变体的运行时间和内存使用情况。随后，我们进行了单独的实验来深入分析 LCG 和 BDS 的性能提升原理。

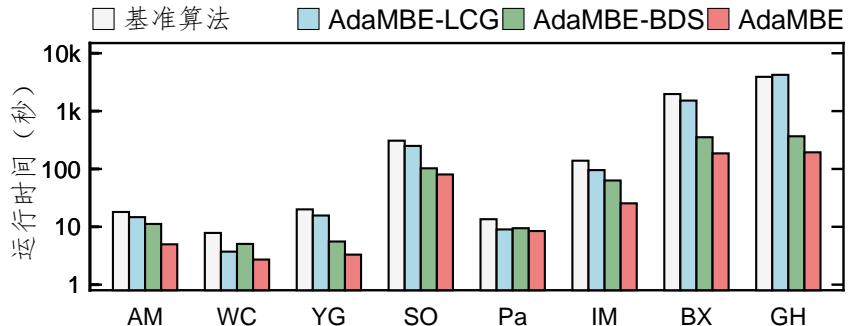


图 3.10 BDS、LCG 方法的运行时间分解评估（对数形式）

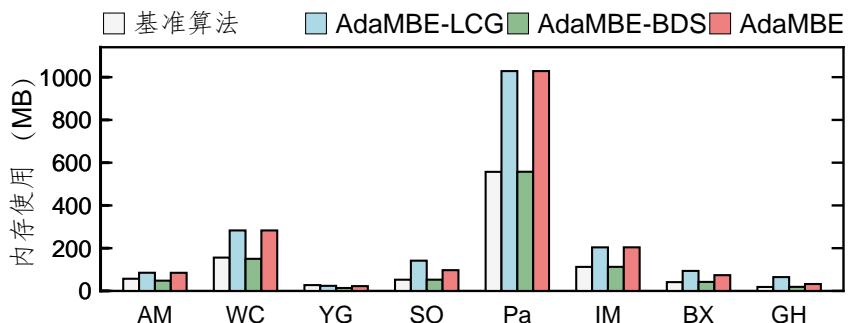


图 3.11 BDS、LCG 方法的内存使用分解评估（对数形式）

运行时间和内存使用评估：我们在包含更多极大二分团计数的 8 个常规数据集上评估了运行时间和内存使用情况。图 3.10 展示了 BDS 和 LCG 方法在运行时间上的分解评估。实验结果显示，AdaMBE 中的 LCG 和 BDS 方法均对基准算法起到了加速效果。在 8 个测试数据集上，AdaMBE-LCG 相比基准算法平均节约了 24% 的执行时间，表明 LCG 方法也具有一定的加速效果。AdaMBE 充分结合了这两种优化方法的优势，在所有测试数据集中性能都优于基准算法和其他变体。以 Github 数据集为例，AdaMBE 将基准算法的运行时间从 3,911 秒缩短至 194 秒。在 StackOverflow、BookCrossing 和 Github 等数据集上，相比基准算法，AdaMBE-BDS 分别加速了 3.0 倍、5.6 倍和 10.7 倍，显示出 BDS 方法在处理大数据集时的显著加速效果。图 3.11 展示了 LCG 和 BDS 方法在内存使用上的分解评估。实验结果显示，使用 LCG 方法的 AdaMBE-LCG 和 AdaMBE 算法的内存使用明显高于不使用 LCG 方法的基准算法和 AdaMBE-BDS，这表明 LCG 方法在运行过程中增加了内存使用，与 3.3.3 节中关于空间开销的讨论结果一致。然而，考虑到极大二分团枚举问题是一个计算密集型问题，内存需求相对较低，因此增加一定的内存使用

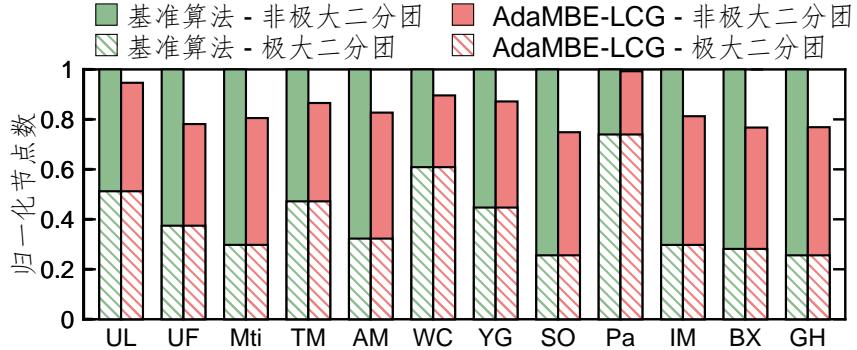


图 3.12 LCG 方法的节点剪枝效率

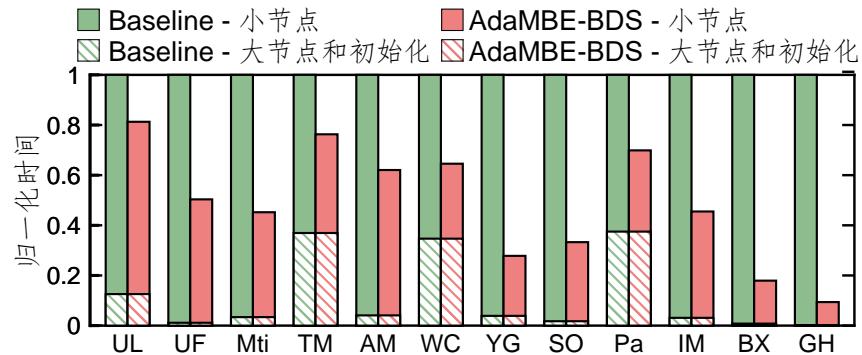


图 3.13 BDS 方法下的运行时间分解

来加速计算过程是一个明智的策略。具体来说，在 Github 数据集上，AdaMBE-BDS 和 AdaMBE 的运行时间为 367 秒和 194 秒，这表明 LCG 方法很好地补充了 BDS 方法。总体而言，在所有测试数据集上，AdaMBE 的内存使用都未超过 1.3 GB（在 CebWiki 数据集上），额外的内存使用是可以接受的。

LCG 方法的效果：为了更深入评估 3.3.1 节中的 LCG 方法，我们分别评估了该方法利用局部计算子图信息重新设计算法的核心操作的有效性。首先，LCG 方法消除了计算子图之外的所有顶点访问，并因此较少了许多数据集上超过 90% 的顶点访问，如 3.2.1 节的图 3.5 所示。其次，LCG 方法由于利用缓存的邻居信息，彻底避免了关于候选顶点局部邻居和节点集合 L 的重复集合交集计算，详见 3.3.1 节。最后，LCG 方法在图 3.12 中展现了节点剪枝的效率：该方法将 12 个常规数据集中对应非极大二分团的无效节点平均减少了 25%。与现有极大二分团仅关注枚举节点剪枝不同，LCG 方法同时关注并减少了无效的顶点访问、集合运算和枚举节点。

BDS 方法效果：为了评估 BDS 方法的加速效果，我们对基准算法和 AdaMBE-BDS 进行了运行时间分解分析。我们根据阈值 τ 将每个算法的时间分解为两个部分，详见图 3.13。具体而言，BDS 方法仅优化了 $|L|$ 等于或小于阈值 τ 的小节点 (L, R, C)，图中

标记了 BDS 方法无法优化的运行时间部分，包括重叠的大节点的计算时间和算法初始化时间。实验结果显示，BDS 方法可以优化的小节点运行时间部分通常超过了基准算法总时间的 60%，在 BookCrossing 和 Github 等大数据集上的占比更是超过了 99%。同时，在 BDS 方法可以优化的部分中，大多数数据集中小节点的运算时间缩短了 50% 以上，特别是在 Github 数据集上，BDS 方法实现了显著地缩短了 91% 的小节点运算时间，使得 AdaMBE-BDS 相对于基准算法加速了 10.7 倍。

3.4.4 敏感性测试

我们对 BDS 方法中的阈值 τ 选择、顶点的顺序选择以及算法的可扩展性进行了敏感性测试。

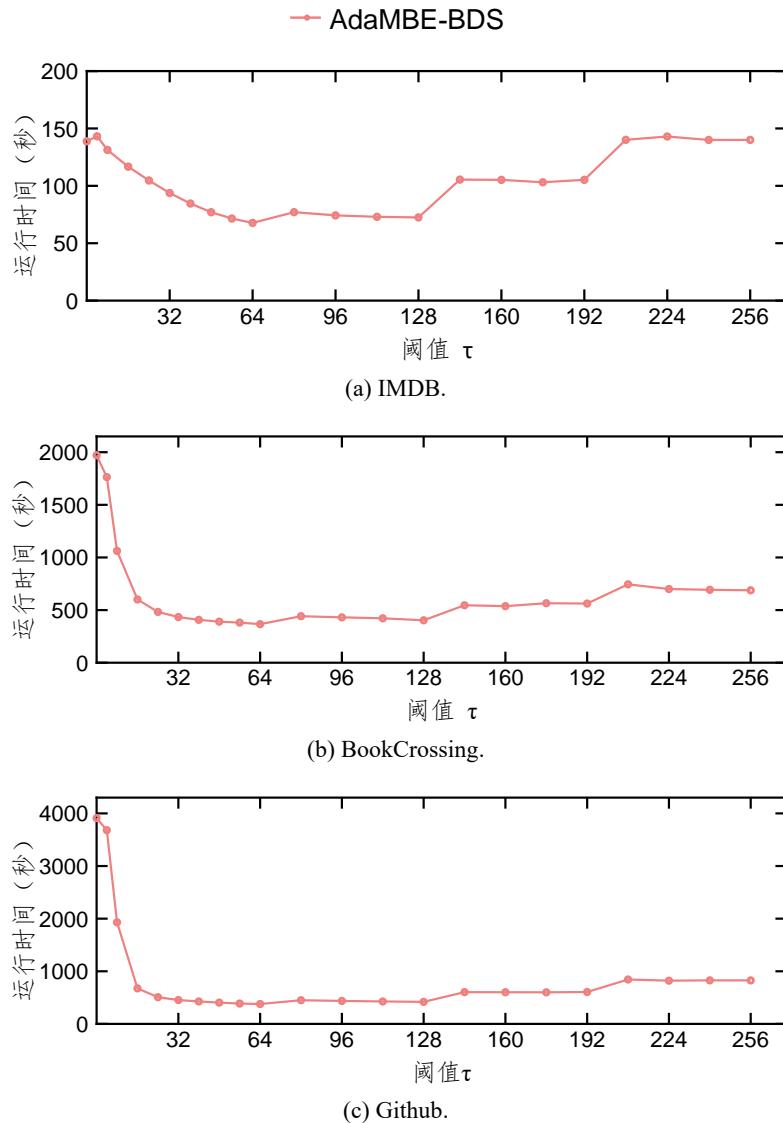


图 3.14 不同阈值 τ 下 BDS 方法性能评估

BDS 方法中阈值 τ 的影响: 为了确定 3.3.2 节中的阈值 τ , 我们在 AdaMBE-BDS 上进行了对不同 τ 值 (从 4 到 256) 的实验, 该实验仅使用 BDS 方法。图 3.14 展示了来自 3 个较大的常规数据集 (即 IMDB、BookCrossing 和 Github) 的实验结果。随着 τ 从 4 增加到 64, 运行时间持续减少。以 BookCrossing 数据集为例, 当 τ 为 4 时, 算法运行时间为 1,763 秒; 而当 τ 为 64 时, 算法运行时间仅为 366 秒。这是因为较大的 τ 导致创建的子图减少, 并且位图的重用增加。需要注意的是, 在 τ 不大于 64 时, 每次集合交集运算只需对两个 64 位长整数进行一次按位与 (&) 操作, 因此集合交集运算时间保持不变。然而, 当 τ 超过 64 时, 由于每次集合交集运算所需的时间增加, 运行时间变得更长。此外, 我们观察到当 τ 是 64 的倍数时, 运行时间较短, 因为这些情况增强了位图的重用, 并且每次集合交集运算所需时间相同。因此, AdaMBE 将 τ 的默认值设定为 64。

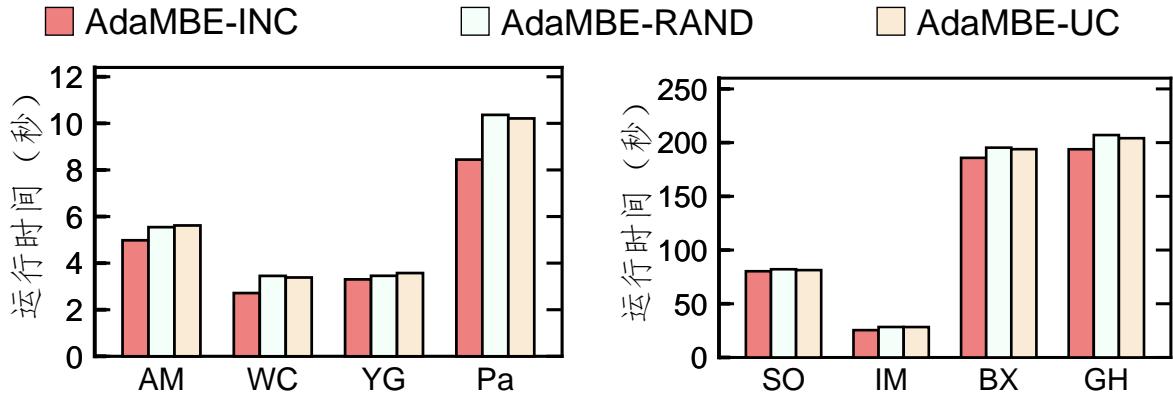


图 3.15 顶点顺序的性能评估

顶点排序的影响: 为了确定 AdaMBE 算法的顶点初始排序方法, 我们提出了三种变体: AdaMBE-INC、AdaMBE-RAND 和 AdaMBE-UC。这三种变体均以 AdaMBE 算法为基础, 它们的区别在于集合 V 中顶点的初始排序方式不同。具体而言, AdaMBE-INC 是根据邻居数量递增对顶点进行排序, AdaMBE-RAND 则以随机顺序对顶点排序, 而 AdaMBE-UC 采用了 ooMMEA 最新提出的单边顺序。在图 3.15 中可以看到, 在 8 个较大的常规数据集上, AdaMBE-INC 始终优于 AdaMBE-RAND 和 AdaMBE-UC, 特别是在包含大量顶点的数据集上表现更为显著。举例来说, 在 DBLP 数据集上, AdaMBE-INC、AdaMBE-RAND 和 AdaMBE-UC 的运行时间分别为 8.4 秒、10.4 秒和 10.2 秒。由此可见, 按照顶点邻居数量递增排序的方法对性能提升具有积极作用, 因此, AdaMBE 默认采用基于邻居数量递增的顶点排序方式。

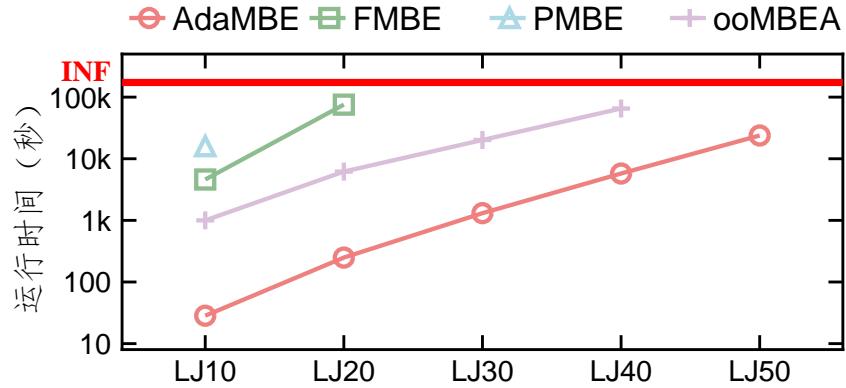


图 3.16 可扩展性评估（对数形式）

可扩展性评估：为了评估 AdaMBE 算法的可扩展性，我们利用从 LiveJournal 数据集中生成的合成数据集进行了实验，该数据集包含超过 1 亿条边。在本实验中，我们分别对 LiveJournal 数据集进行了 10%、20%、30%、40% 和 50% 的边抽样，并将它们分别命名为 LJ10、LJ20、LJ30、LJ40 和 LJ50。表 3.1 中总结了这些数据集的统计信息。如图 3.16 所示，实验结果显示，随着数据规模的增加，不同算法的运行时间呈指数增长。在这些大型合成数据集上，AdaMBE 的运行速度比所有串行竞争对手快 11.4 倍以上。具体而言，AdaMBE 仅需 6.6 小时就能完成 LJ50 数据集中包含超过 60 亿个极大二分团的枚举任务，而其他竞争对手则需要 48 小时才能完成。由此可知，AdaMBE 是处理大规模数据集的最佳选择。

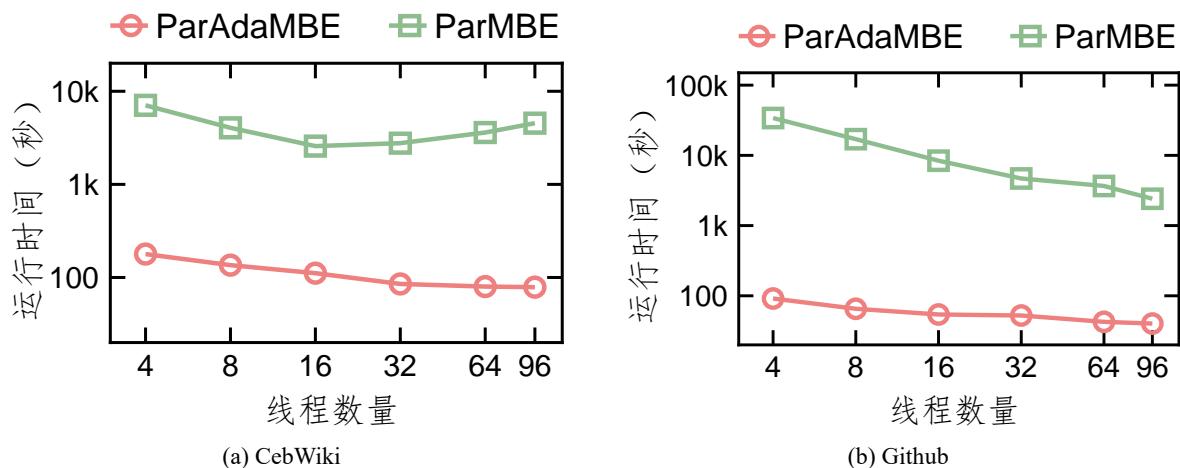


图 3.17 并行性评估（对数形式）

并行性评估：为了评估 AdaMBE 算法的并行性能，我们比较了并行版本 ParAdaMBE 和当前最优的并行算法 ParMBE 在相同线程数量下的执行效率。考虑到 ParMBE 在

TVTropes 数据集上的极大二分团枚举任务无法在 48 小时内完成，我们在其余数据集中选择了运行时间最长的两个数据集：CebWiki 和 Github 进行实验。我们分别使用 4、8、16、32、64、96 个线程在不同数据集上运行 ParAdaMBE 和 ParMBE 算法。考虑到实验服务器总共有 96 个计算核心，我们将线程数量上限设置为 96。如图 3.17 所示，相较于 ParMBE，ParAdaMBE 表现出明显的性能优势。此外，随着线程数量的增加，ParAdaMBE 的运行时间呈现线性下降的趋势，表明在各种线程配置下都能展现良好的性能。

3.5 本章小结

本章提出了局部计算子图的优化方法（LCG）和基于位图的动态子图（BDS）方法，并结合这两种方法形成高效的极大二分团枚举算法 AdaMBE，旨在解决极大二分团枚举问题中计算不规则、静态数据结构低效的挑战。首先，针对现有方法有算法使用静态图结构完成枚举，忽略枚举过程中子图动态变化的特性的问题，LCG 方法通过动态缓存枚举过程中的计算子图的方式优化算法的枚举过程，减少无效的顶点访问、集合运算和枚举节点。其次，针对邻接表上集合运算的高开销问题，BDS 方法在计算过程中动态生成子位图，以加速集合运算。通过利用位运算在子位图中进行集合运算，提升了效率。最后，将这两种技术整合，实现了 AdaMBE 算法及其并行版本 ParAdaMBE。实验结果充分证明了 AdaMBE 算法的高性能，以及本章中所有优化方法的具体作用。

4 基于 GPU 的极大二分团枚举算法

针对现有极大二分团枚举算法在并行扩展性方面受限于 CPU 计算核心数量的问题，本章引入了具有大量计算核心的 GPU 作为计算资源，以加速极大二分团枚举过程。本章介绍了现代 GPU 的硬件架构和软件编程模型。同时，本章指出在 GPU 上实现极大二分团枚举所面临的挑战，作为研究的动机。具体而言，简单地将现有算法映射到大量的 GPU 计算核心中会遇到内存短缺、线程分歧和负载不均等问题。为了解决这些问题，本章提出了以下优化方法：首先，针对内存短缺问题，本章提出了基于枚举节点重用的迭代方法。该方法可以重用枚举树根节点内存，避免为新枚举节点动态分配内存，从而减少内存开销。其次，针对线程分歧问题，本章提出了局部邻居数量感知的剪枝方法。该方法通过记录枚举过程中顶点局部邻居数量的变化对枚举空间进行剪枝，缓解了线程分歧问题。然后，针对负载不均问题，本章提出了负载感知的任务调度方法。该方法将每棵子枚举树对应的计算任务与一个线程束的计算资源进行绑定，在运行时动态预估子枚举树的大小并对较大的子枚举树进行进一步拆分，实现了细粒度的负载均衡。最后，本章结合上述三种方法，提出了基于 GPU 的高效极大二分团枚举解决方案 GMBE。实验证明，基于单个 NVIDIA A100 GPU 的 GMBE 相比现有基于 96 个 CPU 的并行算法 ParMBE 实现了 70.6 倍的性能提升。

4.1 现代 GPU 架构与编程模型

GPU，即图形处理单元（Graphics Processing Unit），最初是专门用于处理图形数据的处理器。随着计算需求的增加，现代 GPU 逐渐演变成了通用并行处理器，能够高效执行大规模数据并行计算任务，如科学计算、深度学习和密码学等。与传统的中央处理器（Central Processing Unit, CPU）相比，现代 GPU 拥有更多的计算核心，并且能够利用这些核心并行处理大量数据，因此成为加速图计算的有力工具，同时具备加速极大二分团枚举问题的潜力。在本节中，我们将深入探讨现代 GPU 的硬件架构、与硬件架构对应的主流的软件编程模型 CUDA（Compute Unified Device Architecture）以及针对 CUDA 的编程指南，作为本章的研究背景。

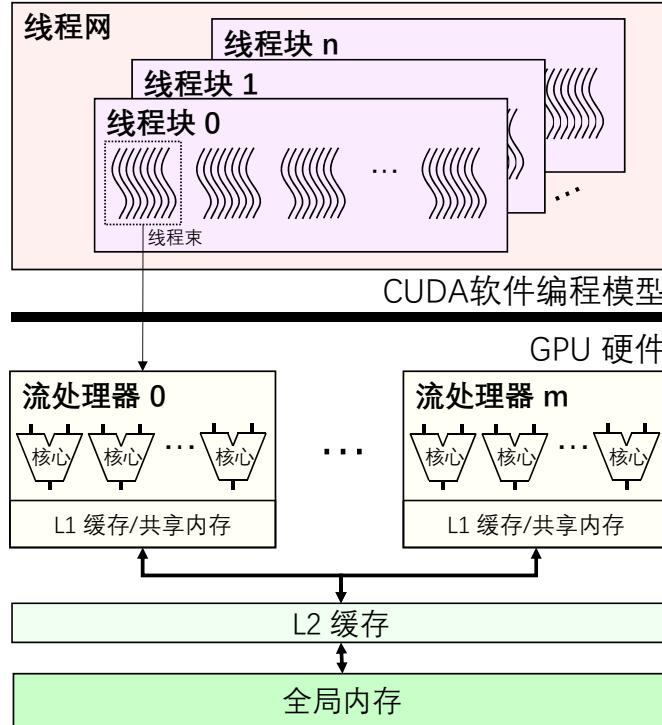


图 4.1 现代 GPU 硬件架构与软件编程模型

图4.1展示了现代 GPU 的硬件架构和软件编程模型。GPU 的硬件架构包括大量的计算核心，以及对应的层级存储结构。具体而言，一块现代 GPU 通常包括全局内存（Global Memory）、共享的 L2 缓存（Cache）以及大量的流处理器（Streaming Multiprocessor, SM）。每个流处理器包含单独的 L1 缓存、可编程的具有多分区（Multi-bank）的共享内存（Shared Memory），以及多个轻量级计算核心（Core）。主流的 GPU 可以配备上万个轻量级计算核心，提供了巨大的计算能力。然而，与丰富的计算资源相比，GPU 上的内存资源相对有限。例如，近年备受青睐的 NVIDIA A100^[99] 最多可以提供 6912 个核心，但只能提供最多 80GB 的全局内存。

与 GPU 的多计算核心的硬件架构相对应，主流的 CUDA 软件编程模型按照层级结构管理大量线程。具体而言，CUDA 编程模型提供了一个并行计算平台和一组 API^[100-101]，允许用户高效地利用 GPU 进行通用目的的处理。CUDA 采用 SIMT（Single Instruction, Multiple Threads）^[102] 执行模型来管理大量线程。它将 GPU 内核划分为多个线程网（Grid），每个线程网包含多个线程块（Block），每个线程块包括多个线程（Thread），并在执行期间分配给一个流处理器（SM）。流处理器将 32 个并行线程分组成一个线程束（Warp），并同时执行多个线程束。通过这种方式，成千上万个 GPU 核心可以高效地并行工作，实现高性能计算。

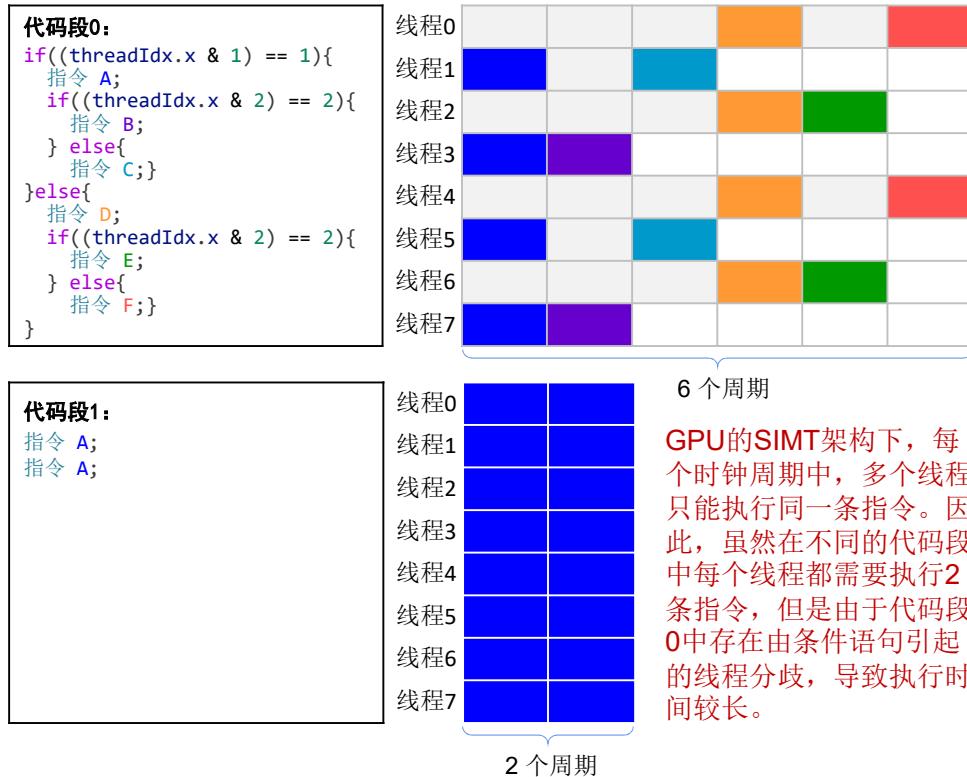


图 4.2 线程分歧示意图

为了提升 GPU 的执行性效率，我们针对 CUDA 编程模型总结了编程指南如下：

1. 减少动态内存分配。动态内存分配是指程序在运行过程中动态申请和释放内存。由于 GPU 允许大量的线程同时并发执行，大量线程可能会同时进行动态内存分配，从而引发线程争用、同步开销以及内存碎片化等问题，影响程序性能^[103]。为了避免这种情况的发生，建议采用静态内存分配方式，在程序开始前预估程序的内存使用情况并一次性分配一块足够大的内存，然后在运行时重复使用这块内存，避免频繁的动态内存分配和释放，以降低内存管理的开销并优化内存访问效率。
2. 减少线程分歧。线程分歧是指在 GPU 中同一线程束内的线程执行不同的代码路径。如图 4.2 所示，由于 GPU 使用 SIMT 的执行模式，即多个线程共享同一组指令，线程分歧会使 GPU 对不同执行路径进行串行化，导致部分线程处于空闲状态，从而严重影响执行性能^[101]。因此，在 CUDA 程序应设计简洁的算法和内核函数，尽量保持线程之间的执行路径相似，避免条件语句中大量的分支情况。可以考虑使用线程块内的协作和同步机制来避免线程分歧，尽量使每个线程束内的线程保持一致的执行路径，从而提高并行执行效率。

3. 平衡大量负载。平衡大量负载是指合理地分配计算任务和数据处理任务，确保每个处理单元的负载均衡，避免某些处理单元负载过重而导致性能瓶颈。由于 GPU 拥有大量轻量级计算核心，负载不平衡会导致数千个计算核心等待最慢的计算核心执行，造成严重的资源浪费^[101]。为了实现负载平衡，可以考虑将任务划分为较小的子任务，并合理分配给不同的处理单元，通过动态调整任务分配策略来实现计算任务在不同计算资源上的均匀分布，从而提高整体计算效率。此外，可以利用 CUDA 提供的性能分析工具来帮助识别和解决负载不平衡的问题^[104-106]，从而优化程序的性能表现。

4.2 研究动机

尽管 GPU 具有强大的并行能力，在部分图挖掘问题上已经显示出加速效果，但在 GPU 上实现高效的极大二分团枚举仍然面临严峻的挑战。具体而言，这些挑战主要包括的内存短缺、线程分歧和负载不均等。本节将结合 GPU 加速图计算的相关工作，对上述挑战进行详细说明。为了方便表述，本章中总是以算法 1.1 在二分图 G_3 上的集合枚举树为例，如图 4.3 所示。

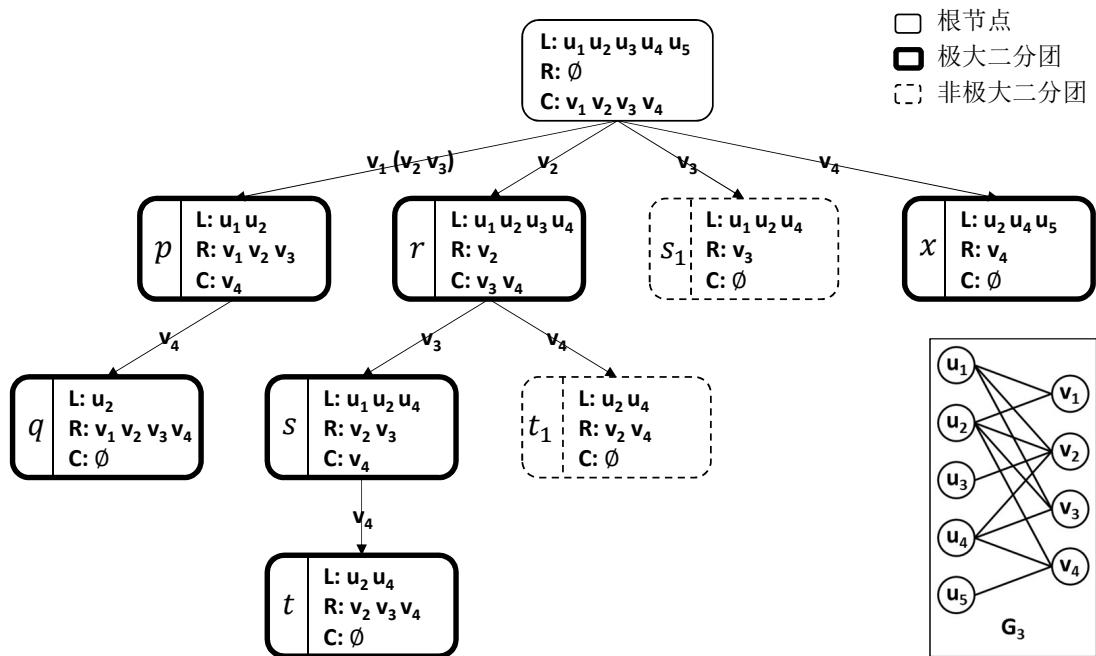


图 4.3 算法 1.1 在二分图 G_3 上的集合枚举树

4.2.1 内存短缺

根据 1.2.2.2 节的描述，我们注意到现有的极大二分团枚举算法（如算法 1.1 所示）在枚举过程中会动态的生成和释放枚举节点 (L, R, C)。因此，直接将这类算法迁移到 GPU 上会导致 4.1 节中提到的动态内存分配问题。为了实现高性能，现有的基于 GPU 的图挖掘算法（例如^[25, 75, 79, 84]）通常在执行之前就在 GPU 上预先分配足够的内存空间来容纳所有的枚举节点。参考这些算法的内存估计方法，我们可以得出在极大二分团枚举算法中，每个枚举节点需要使用 $O(|L| + |R| + |C|)$ 的内存，上限为 $O(\Delta(V) + \Delta_2(V))$ 。同时，在遍历过程中，每棵子树最多需要保存 $\Delta(V)$ 个活跃枚举节点用于回溯。因此，每棵枚举树的遍历过程需要预分配的内存总量为 $\Delta(V) \times (\Delta(V) + \Delta_2(V)) \times$ 顶点大小。举例来说，在使用 NVIDIA A100 GPU (40 GB 内存) 对真实世界的二分图 BookCrossing^[89] 进行极大二分团枚举时，每棵子树遍历过程的内存需求为 $13,601 \times (13,601 + 53,915) \times \text{sizeof(int)}$ $B = 3.67 \text{ GB}$ 。为了充分利用 NVIDIA A100 GPU 中的 108 个流处理器 (SM)，我们需要 $108 \times 3.67 \text{ GB} = 397 \text{ GB}$ 的内存，这超过了 GPU 中的内存空间 (40 GB)，因此面临严重的内存短缺问题。

4.2.2 线程分歧

由于图挖掘算法本身的不规则性^[46]，极大二分团枚举算法涉及大量的分支语句，这导致了 4.1 节所提到的线程分歧问题。在 GPU 上进行极大二分团枚举时，线程分歧主要来源于两个方面。首先，在极大二分团枚举过程中，同一线程束内的线程可能会同时访问不同的顶点以生成新节点，并使用不同顶点的邻居来计算不同的集合 L 、 R 和 C ，从而导致不同线程执行不同的控制流以访问不同内存区域。其次，如 2.1 节所述，现有的搜索空间优化方法通常会引入额外的判断语句来对搜索空间进行剪枝，导致不同线程执行路径差异，进一步加剧了线程分歧问题。举例来说，ooMBEA 算法^[48] 需要访问所有候选顶点的 2 跳邻居来识别批量枢纽 (Batch Pivots)，并剪掉不属于批量枢纽的候选顶点，以实现剪枝目的。然而，访问顶点的 2 跳邻居需要进行深度为 2 的深度优先搜索，这个过程会涉及到大量对不同顶点邻居边界的条件判断，因而导致更严重的线程分歧问题。因此，在优化枚举空间的同时减少线程分歧是一项具有挑战性的任务，开发适用于 GPU 的剪枝方法至关重要。

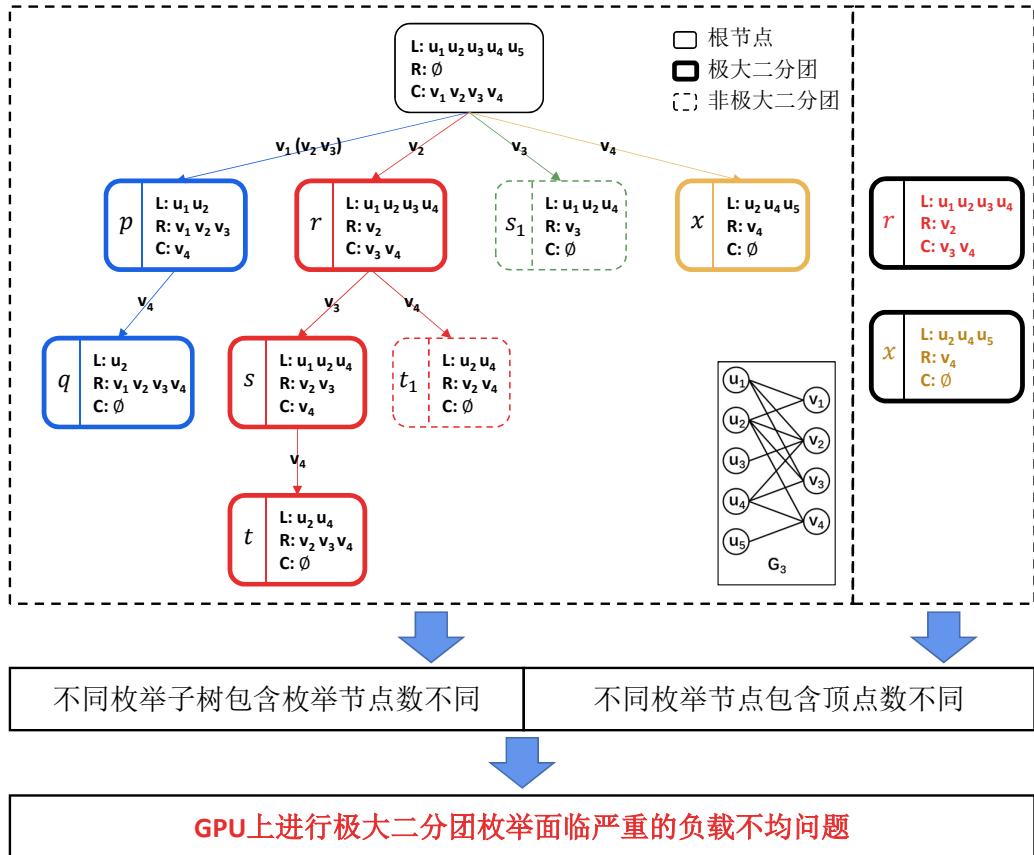


图 4.4 在 GPU 上进行极大二分团枚举中的负载不均问题说明

4.2.3 负载不均

如图 4.4 所示，极大二分团枚举问题在 GPU 上面临严重的负载不均问题，这是由于不同子枚举树和枚举节点之间的计算差异性所导致的。与其他 GPU 实现的图挖掘算法类似，我们将完整枚举树拆分成多个子枚举树，并尝试在不同计算单元之间平衡它们的负载分布^[75, 79, 84]。我们没有选择按照更细粒度的枚举节点进行任务划分，因为在此类图挖掘问题中，枚举节点往往数量庞大，细粒度划分会增加任务调度的开销，严重降低整体性能。目前，已有的图模式挖掘算法 G²Miner^[75] 通常将每个顶点生成的枚举子树分配给 GPU 上的一个线程束来独立执行。然而，我们的实验结果显示，这种粗粒度的 GPU 负载均衡策略在极大二分团枚举问题中效率较低。由于不同子枚举树包含的枚举节点数量差异很大，且每个节点内的顶点数量也不同，导致不同子枚举树的负载相差较大。这种负载不均会导致大量计算核心在等待最耗时的核心执行时浪费大量时间，造成资源的浪费。接下来，我们将通过一组实验结果详细说明这一问题。

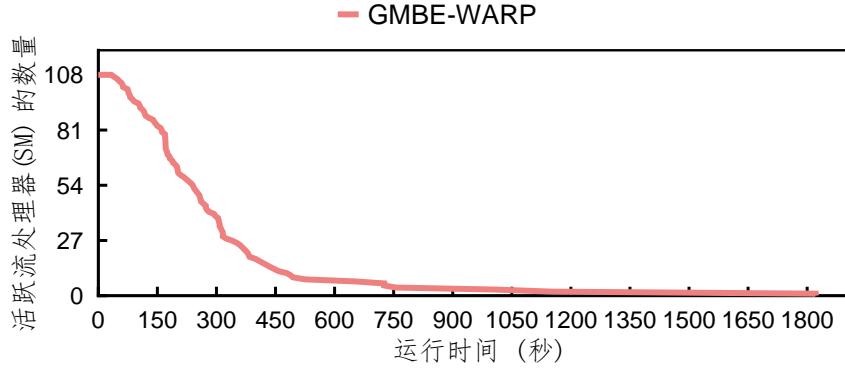


图 4.5 在 GPU 上进行极大二分团枚举的负载不均问题示例

例 4.1. 图 4.5 展示了在简单负载均衡方案下，在 GPU 上进行极大二分团枚举表现出的负载不均问题。具体而言，首先，在我们的最终解决方案 GMBE 的基础上，我们应用 G^2Miner 算法中将每个顶点生成的枚举子树分配给 GPU 上的一个线程束来独立执行的任务调度方案，形成变种 GMBE-WARP。随后，我们在 BookCrossing 数据集上运行 GMBE-WARP，并记录了 GPU 内活跃流处理器数量随着运行时间的变化图。实验结果显示，该算法的总运行时间为 1,822 秒。当程序运行至 20% 时（即 364 秒），活跃的流处理器仅有 22 个，占总数 108 个的 20%。这意味着在整个运行过程中，超过 80% 的流处理器（86 个 SMs / 共 108 个 SMs）将耗费 80% 的运行时间（1,458 秒 / 共 1,822 秒）等待最慢的一个子枚举树的计算。因此，现有的方法不能满足极大二分团枚举算法在 GPU 上的负载均衡需要，我们有必要实现更细粒度的负载均衡。

4.3 GMBE 算法

为了应对 GPU 上高效实现极大二分团枚举时所面临的内存短缺、线程分歧以及负载不均等挑战，本节提出了一系列对应的解决方案。这些解决方案包括基于枚举节点重用的迭代方法、局部邻居数量感知的剪枝方法以及负载感知的任务调度方法。最终，通过结合这些技术，我们提出了高效的 GPU 极大二分团枚举算法 GMBE。

4.3.1 基于枚举节点重用的迭代方法

为了减少内存使用，我们提出了一种基于枚举节点重用的迭代方法。该方法的核心思想在于仅保存子枚举树中的根节点 x 及其相关元数据，在迭代过程中反复重用节点 x 的存储空间，避免为大量子节点动态分配空间。在枚举过程中，后继节点总是能够

从节点 x 的存储空间中获取，因为我们发现任何子节点 c 中的顶点集 $L_c \cup R_c \cup C_c$ 始终是其父节点 x 的顶点集 $L_x \cup R_x \cup C_x$ 的子集。例如，在图 4.3 中，对于父节点 p 和子节点 q ，我们发现父节点 p 内的顶点集 $\{u_1, u_2, v_1, v_2, v_3, v_4\}$ 包含子节点 q 内的顶点集 $\{u_2, v_1, v_2, v_3, v_4\}$ ，这种父子节点间顶点集间普遍存在的包含关系为枚举节点重用技术提供了理论保证。接下来，我们在图 4.6 中给出了基本的根节点存储结构，并结合算法 4.1 详细描述了基于枚举节点重用的迭代方法。

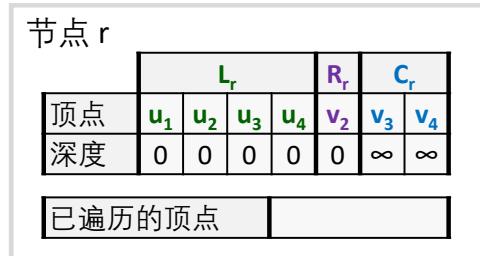


图 4.6 $node_buf$ 结构示意图

算法 4.1 基于枚举节点重用的极大二分团枚举迭代算法

输入: 二分图 $G(U, V, E)$

输出: 所有极大二分团

```

1: iteratively_search( $U, \emptyset, V$ );
2: procedure iteratively_search( $L_r, R_r, C_r$ ):
3:    $node\_buf$ .init_and_push(( $L_r, R_r, C_r$ ));
4:   while  $node\_buf$  非空 do
5:     ( $L_p, R_p, C_p$ )  $\leftarrow$   $node\_buf$ .pop();
6:     if  $C_p$  非空 then
7:        $v' \leftarrow C_p$  中编号最小的顶点;
8:        $node\_buf$ .push(( $L_p, R_p, C_p \setminus \{v'\}$ ));
9:        $L' \leftarrow L_p \cap N(v')$ ;  $R' \leftarrow R_p$ ;  $C' \leftarrow \emptyset$ ;
10:      for  $v_c \in C_p$  do
11:        if  $L' \cap N(v_c) = L'$  then
12:           $R' \leftarrow R' \cup \{v_c\}$ ;
13:        else if  $L' \cap N(v_c) \neq \emptyset$  then
14:           $C' \leftarrow C' \cup \{v_c\}$ ;
15:        end if
16:      end for
17:      if  $R' = \Gamma(L')$  then
18:        输出极大二分团  $(L', R')$ ;
19:         $node\_buf$ .push(( $L', R', C'$ ));
20:      end if
21:    end if
22:  end while
23: end procedure

```

具体而言，我们不再以递归的方式创建和释放枚举节点，而是采用类似栈的结构 *node_buf* 进行显式的迭代。如图 4.6 所示，每个 *node_buf* 包括根节点内的所有顶点、每个顶点的深度属性、以及从根节点到当前节点所遍历的所有顶点。每个顶点的深度根据当前节点的祖先节点数量确定。已遍历的顶点记录着从子枚举树根节点到当前节点所遍历的顶点，用于回溯搜索。在枚举过程中，我们只需要主动更新顶点的深度属性以及记录已遍历的顶点。通过这种方式，迭代过程可以重用 *node_buf* 对应的固定内存区域推导出所有的后继节点，进而极大程度地减少了 GPU 中的内存开销。算法 4.1 首先用根节点初始化 *node_buf*（第 3 行）。随后算法迭代地获取 *node_buf* 中的当前节点 (L_p, R_p, C_p) （第 5 行）。如果当前节点的候选顶点非空，则迭代地选择当前节点内其中的一个候选顶点产生新的枚举节点（第 6-21 行），否则弹出该节点，继续迭代流程。最终，算法与递归算法 1.1 得到相同的计算结果。随后，我们将详细阐述涉及枚举节点重用的关键函数：

- **init_and_push((L_r, R_r, C_r))**：该函数用于利用子树的根节点 (L_r, R_r, C_r) 创建并初始化 *node_buf*（第 3 行）。*node_buf* 存储了 $L_r \cup R_r \cup C_r$ 中的所有顶点，并记录每个顶点的深度以及已遍历的顶点。我们将 $L_r \cup R_r$ 中顶点的深度初始化为 0，将 C_r 中顶点的深度初始化为 ∞ 。
- **push((L', R', C'))**：该函数用于向 *node_buf* 中压入一个新的子节点 (L', R', C') （第 8,20 行）。首先，根据顶点深度的定义，我们知道当前节点的深度 D 总是比 *node_buf* 中已遍历的顶点多一个。当我们在深度为 D 处压入一个新节点 (L', R', C') 时，我们知道 $L' \subset L_r$, $R' \subset R_r \cup C_r$ 且 $C' \subset C_r$ 。我们将 L' 内顶点的深度更新为 D , R' 内深度为 ∞ 的顶点的深度更新为 D 。因此，通过访问顶点的深度属性，我们总是能够在原始的 (L_r, R_r, C_r) 对应的 *node_buf* 中找到新节点 (L', R', C') ，其中 L' 包含 L_r 内所有深度为 D 的顶点， R' 包含 $R_r \cup C_r$ 中所有深度不大于 D 的顶点， C' 包含 C_r 中深度为 ∞ 的顶点。最后，我们将用于生成节点 (L', R', C') 的已遍历顶点加入到已遍历的顶点集中。
- **pop()**：该函数用于得到当前栈顶节点 (L_p, R_p, C_p) ，并利用 *node_buf* 回溯到其父节点（第 5 行）。当我们在节点深度 D 处弹出节点 (L_p, R_p, C_p) 时，首先移除已遍历的顶点集中最新遍历的顶点，然后将 L_p 内顶点的深度更新为 $D-1$ ，将 C_p 内深度为 D 的顶点的深度更新为 ∞ 。

与传统的单一记录节点内顶点的枚举树节点结构相比，我们提出的 *node_buf* 在单个节点上额外记录了顶点深度以及已遍历的顶点，其内存开销的上限为 $3 \times \Delta(V) + 2 \times \Delta_2(V)$ 。但是与传统方法需要动态产生和释放枚举节点不同，*node_buf* 结构能够在枚举过程中被所有后继枚举节点重复使用，因此显著减少了遍历每棵枚举树的内存需求，为在 GPU 上并发运行数千个极大二分团枚举过程提供可能。例如，一块拥有 40 GB 内存的 A100 GPU 足以在 BookCrossing 数据集上并发运行超过 10,000 个子枚举树枚举过程，因为每个过程仅需要 $(3 \times 13,601 + 2 \times 53,915) \times \text{sizeof}(\text{int}) \text{ B} = 595 \text{ KB}$ 。与传统的方法相比，后者需要 $13601 \times (13,601 + 53,915) \times \text{sizeof}(\text{int}) \text{ B} = 3.67 \text{ GB}$ ，这种枚举节点重用方法在 BookCrossing 上的内存使用减少了 6,178 倍。

4.3.2 局部邻居数量感知的剪枝方法

现有的剪枝方法在 GPU 上由于线程分歧导致效率较低。为了解决这一问题，我们提出了一种新的剪枝方法，旨在减少枚举空间的同时减少线程分歧。具体而言，参照表 1.1，对于给定节点 (L, R, C) ，我们将顶点 $v \in V$ 的局部邻居定义为 $N_L(v)$ ，其中 $N_L(v)$ 等于 $N(v) \cap L$ 。随后，我们定义局部邻居数量为局部邻居的顶点个数。在算法 4.1 中，候选顶点的局部邻居数量总是作为中间结果出现（第 11、13 行），因此，我们能够在不引入额外计算的情况下得到候选顶点的局部邻居数量。基于这一现象，我们总结了如下定理：

定理 4.1. 假设当前节点为枚举树中的节点 s ，在选择候选顶点 v_r 时，如果对于当前节点 s 中顶点 v_r 的局部邻居数量与任一子节点 t 中顶点 v_r 的局部邻居数量相等，我们可以安全地裁剪由节点 s 选择顶点 v_r 所产生的子节点。

证明. 为了证明该定理，我们假设节点 s 已经遍历顶点 v_t 产生节点 t ，并且节点 s 将遍历顶点 v_r 产生新的节点 r 。进一步地，我们假设节点 s 中顶点 v_r 的局部邻居数量与子节点 t 中顶点 v_r 的局部邻居数量相等，即 $N(v_r) \cap L_s = N(v_r) \cap L_t$ 。由于 $L_t = L_s \cap N(v_t)$ ，我们知道 $N(v_r) \cap L_s = N(v_r) \cap L_t = N(v_r) \cap L_s \cap N(v_t)$ ，因此 $N(v_r) \cap L_s \subseteq N(v_t)$ 。由于 $L_r = N(v_r) \cap L_s$ ，我们知道 $L_r \subseteq N(v_t)$ ，即 v_t 与 L_r 中的所有顶点相连。由于我们已经遍历 v_t 产生节点 t ，因此节点 r 无法使用 v_t 扩展 R_r ，进而产生非极大二分团。因此我们可以安全地裁剪节点 r 。

□

根据定理 4.1，我们观察到可以通过比较父子节点中候选顶点局部邻居数量是否变化进行剪枝。因此，在算法 4.1 中，我们通过在 *node_buf* 中额外维护所有候选顶点的局部邻居数量来进一步优化。具体而言，在 *node_buf* 执行弹出操作 *pop()* 弹出节点 *r* 时，会临时保存栈顶节点候选顶点的局部邻居数量。当恢复到节点 *r* 的父节点 *p* 时，我们将对节点 *p* 和节点 *r* 中候选顶点的局部邻居数量进行批量比较。如果发现部分候选顶点的局部邻居数量未发生变化，我们可以安全地裁剪这些无用顶点。相比现有剪枝方法中涉及大量判断语句所引入的线程分歧问题，这种剪枝方法在 GPU 上不会额外引入线程分歧问题，因为剪枝的过程同一线程束中的线程总是批量比较相同候选集中的元素。我们通过一个示例详细展示了基于栈的迭代算法，并利用局部邻居数量进行剪枝。

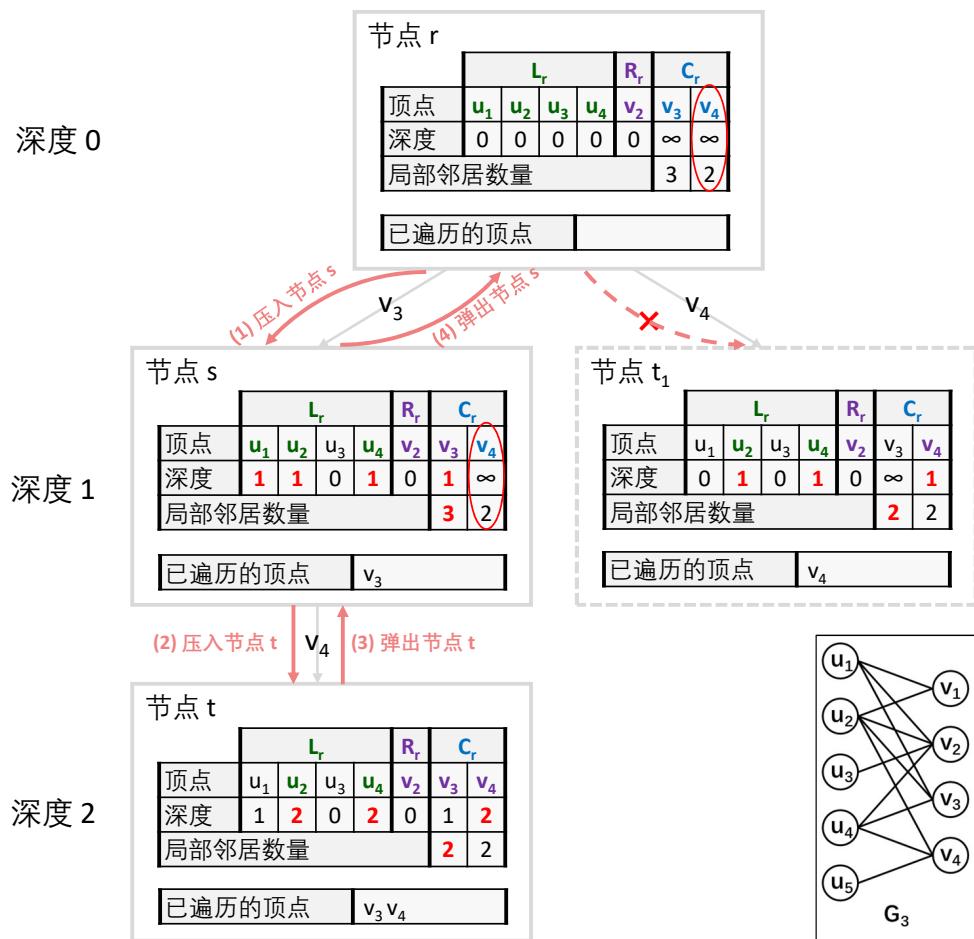


图 4.7 局部邻居数量感知的剪枝方法示意图

例 4.2. 如图 4.7 所示，在图 4.3 中以节点 *r* 为根节点的子树中，算法利用 *node_buf* 中固定大小的内存，迭代地枚举了其中的所有极大二分团。具体而言，我们首先使用节

点 $r (L_r, R_r, C_r)$ 初始化 $node_buf$, 将 $L_r \cup R_r$ 中的顶点深度初始化为 0, 并将 C_r 中的顶点深度初始化为 ∞ 。对于 C_r 中的顶点, $node_buf$ 根据定义初始化了其局部邻居数量 (即 $|N_L|$)。例如, 在节点 r 处, 我们初始化 v_3 的局部邻居数量 $|N_L(v_3)|$ 为 $|N(v_3) \cap L_r| = |\{u_1, u_2, u_4\} \cap \{u_1, u_2, u_3, u_4\}| = 3$ 。

接下来, 我们通过遍历 v_3 生成了节点 s 。我们知道 $L_s = L_r \cap N(v_3) = \{u_1, u_2, u_4\}$ 。节点 s 的深度为 1, $node_buf$ 将 $L_s \cup R_s$ 中顶点 u_1, u_2, u_4 , 和 v_3 的深度更新为 1。其他顶点的深度保持不变。随后, 我们利用 L_s 更新了 v_3 和 v_4 的局部邻居数量。通过计算, 我们知道 $|N_L(v_3)| = |N(v_3) \cap L_{(s)}| = |\{u_1, u_2, u_4\} \cap \{u_1, u_2, u_4\}| = 3$ 。 $|N_L(v_4)| = |N(v_4) \cap L_{(s)}| = |\{u_2, u_4, u_5\} \cap \{u_1, u_2, u_4\}| = 2$ 。我们可以类似地生成其他节点。

在处理完以节点 s 为根节点的子树后, $node_buf$ 弹出节点 s 并重置顶点的深度和局部邻居数量, 用于恢复节点 r 。对于 $node_buf$ 内深度与节点 s 深度相同的顶点, 我们将 u_1, u_2 和 u_4 的深度重置为 0, 并将 v_3 的深度重置为 ∞ 。由于弹出节点 s 后, v_4 的局部邻居数量始终是 2, 我们主动地在节点 r 处通过移除无用的候选顶点 v_4 来进行剪枝。

4.3.3 负载感知的任务调度方法

为了进一步探索极大二分团枚举任务在 GPU 上的大规模并行性, 我们提出了负载感知的任务调度方法。在图挖掘算法的 GPU 上任务调度方面, 一种简单的方法是为每个子枚举树分配一个任务。如算法 4.2 所示, 我们用集合 V 中的不同顶点 v_s 产生不同的枚举子树, 分别作为一个任务。

算法 4.2 GPU 任务调度的简单方法

```

1: for  $v_s \in V$  do
2:    $L_s \leftarrow N(v_s); R_s \leftarrow \{v_s\}; C_s \leftarrow \emptyset;$ 
3:   for  $v_c \in N_2(v_s)$  do
4:     if  $L_s \cap N(v_c) = L_s$  then
5:        $R_s \leftarrow R_s \cup \{v_c\};$ 
6:     else if  $v_c$  的索引在  $v_s$  之后 then
7:        $C_s \leftarrow C_s \cup \{v_c\};$ 
8:     end if
9:   end for
10:  if  $v_s$  是  $R_s$  中索引最小的顶点 then
11:     $iteratively\_search(L_s, R_s, C_s);$ 
12:  end if
13: end for

```

然后，我们可以参考已有的图挖掘算法，将每个任务映射到 GPU 的不同的线程束^[75]或者不同线程块^[84]上。我们将两种方案分别称为基于线程束的调度方法（Warp-centric Scheme）和基于线程块的调度方法（Block-centric Scheme）。然而，我们的研究表明，由于算法 4.2 在第 11 行创建的并行任务运行时间差异很大，对于极大二分团问题，这两种简单的调度方法不足以平衡 GPU 中不同流处理器（SM）上的任务负载。具体而言，在一个线程束或一个线程块中运行的最慢任务经常阻塞其他任务，导致 EuAll 数据集上存在 97.8% 的性能下降。4.4.3 节将展示更多任务调度相关的实验结果。

为了更好地平衡 GPU 中的任务负载，我们提出了一种针对极大二分团枚举的负载感知的基于任务的调度方法（Task-centric Scheme）。该方法基于 GPU 编程模式中的持久线程编程模型（Persistent Thread, PT）^[107]。具体而言，我们根据 GPU 中的流处理器数量创建对应数量的线程组，将每个线程组固定地映射到一个 GPU 的流处理器上。随后，为每个线程组设置固定数量的线程束，将每个流处理器上线程束的数量用 WarpPerSM 表示，并在后文讨论其对系统性能的影响。

我们设计了一个可以创建负载感知任务的 GPU 核函数（Kernel Function），该函数会将处理较大枚举树的任务递归地分解为更小的任务，并将这些负载感知任务添加到每个流处理器的全局结构 *SM_task_queue* 中。当一个任务完成时，PT 的软件调度器会从 *SM_task_queue* 中出队一个任务，并在相应的 SM 上迭代执行 *iteratively_search()*。

为实现负载感知，关键问题在于高效地检测出哪些负载的任务较重，并及时对高负载任务进行分解。对此，我们根据子枚举树根节点 (L, R, C) 的信息来估计枚举树的高度和节点数量。具体而言，我们估计枚举树的高度为 $\min\{|L|, |C|\}$ 。我们估计枚举树中节点数量为 $\min\{|L|, |C|\} \times |C|$ ，因为 $|C|$ 代表了每个节点能够产生子节点的最大数量。我们经验性地设置了两个阈值 *bound_height* 和 *bound_size*，只有当 $\min\{|L|, |C|\}$ 大于 *bound_height* 且 $\min\{|L|, |C|\} \times |C|$ 大于 *bound_size* 时，我们将该任务分解成多个子任务，以实现更好地负载平衡。

算法 4.3 描述了针对极大二分团枚举问题的 GPU 负载感知的基于任务的调度算法。当 *SM_task_queue* 不为空时，它从队列里获取一个节点（第 6 行），如果我们估计该节点对应的负载较小时，我们直接启动一个 GPU 任务（第 37 行）；反之，如果对应的负载超过阈值时（第 23 行），我们将该任务拆分成小任务并将小任务对应的根节点入队进行枚举（第 23-35 行）。这些小任务对应的节点将重新参与 *SM_task_queue* 的调度。如果

SM_task_queue 为空时，我们将从 $processing_v$ 获取需要处理的当前顶点 v_s 。然后用顶点 v_s 产生一个枚举任务，并运行在 GPU 上（第 8-20 行）。

算法 4.3 负载感知的基于任务的调度方法

```

1:  $processing\_v$  是一个全局变量，初始化为 0;
2:  $SM\_task\_queue$  是用于负载均衡的全局并发队列;
3: procedure warp_kernel :
4:   while true do
5:     if  $SM\_task\_queue$  非空 then
6:        $(L, R, C) \leftarrow SM\_task\_queue.dequeue()$  ;
7:     else
8:        $v_s = atomicInc(processing\_v)$ 
9:       if  $v_s \in V$  then
10:         $L \leftarrow N(v_s); R \leftarrow \{v_s\}; C \leftarrow \emptyset;$ 
11:        for  $v_c \in N_2(v_s)$  do
12:          if  $L \cap N(v_c) = L$  then
13:             $R \leftarrow R \cup \{v_c\};$ 
14:          else if  $v_c$  的索引在  $v_s$  之后 then
15:             $C \leftarrow C \cup \{v_c\};$ 
16:          end if
17:        end for
18:      else
19:        return;
20:      end if
21:    end if
22:    if  $R = \Gamma(L)$  then
23:      if  $\min\{|L|, |C|\} \times |C| >$  数量边界，并且  $\min\{|L|, |C|\} >$  高度边界 then
24:        for  $v_t \in C$  do
25:           $L_t \leftarrow N(v_t); R_t \leftarrow R; C_t \leftarrow \emptyset;$ 
26:          for  $v_c \in C$  do
27:            if  $L_t \cap N(v_c) = L_t$  then
28:               $R_t \leftarrow R_t \cup \{v_c\};$ 
29:            else if  $L_t \cap N(v_c)$  非空 then
30:               $C_t \leftarrow C_t \cup \{v_c\};$ 
31:            end if
32:          end for
33:           $SM\_task\_queue.enqueue((L_t, R_t, C_t))$ 
34:           $C \leftarrow C \setminus \{v_t\}$ 
35:        end for
36:      else
37:        iteratively_search( $L, R, C$ );
38:      end if
39:    end if
40:  end while
41: end procedure

```

4.3.4 GMBE 算法设计

基于以上所述技术要点，我们提出了基于 GPU 的高效极大二分团枚举算法 GMBE。除了前文提及的主要技术要点之外，接下来将详细描述 GMBE 算法的具体实现细节。

二分图预处理：我们将输入的二分图 G 以压缩稀疏行（Compressed Sparse Row, CSR）格式进行表示。首先将图 G 载入 CPU 内存，并快速提取图 G 的重要特征，如 $|U|$, $|V|$, $|E|$, $\Delta(V)$, 和 $\Delta_2(V)$ 。由于二分图中 U 和 V 是对称的，在本文中我们总是选择顶点数较少的集合作为 V ，类似于 ooMMEA^[48]。然后，我们通过按照顶点邻居数量递增的顺序对 V 中的所有顶点进行排序，类似于 MineLMBC^[6] 和 iMMEA^[15]，并对每个顶点的所有邻居列表按照顶点 ID 递增的顺序进行排序，类似于大多数相关工作^[75, 84]。最后，我们将整个二分图 G 传输到 GPU 的全局内存，并在 GPU 上枚举所有极大二分团，而无需从主机传输任何额外数据。

无锁任务队列：为了减少同步开销，我们使用 CUDA 中的 atomicCAS 原语以无锁方式管理任务队列。我们实现了一个两级任务排队机制以进一步改善负载平衡。具体来说，我们为每个线程块实现了一个局部任务队列，以便线程块中的所有线程束可以通过访问局部任务队列来平衡工作负载。此外，我们实现了一个全局任务队列来平衡不同线程块之间的工作负载。每个线程块只允许一个代理线程束来管理局部任务队列和全局任务队列之间的任务。我们使用共享内存实现局部任务队列，并在全局内存中实现全局任务队列，因为共享内存上的原子操作比全局内存上的原子操作更快。

基于相交路径的集合并集操作：为了降低算法 4.3 第 11 行计算 2 跳邻居的开销，我们在 GPU 上实现了基于相交路径的集合并集操作。这一方法受到了文献^[92]的启发。具体而言，为了并行化集合并集操作，每个线程使用滑动窗口遍历一个相交路径，并在当前窗口内独立查找部分相交路径。最后，线程束中的所有线程同步部分结果，生成完整的相交路径。

多 GPU 上的 GMBE 算法实现：高性能计算机可能由多个 GPU 组成，以加速应用执行性能。为了支持这种情况，我们可以轻松地将 GMBE 算法扩展到多 GPU 计算机。其基本思想是在所有 GPU 设备上共享算法 4.3 中的全局变量 $processing_v$ ，并将第 8 行中的 atomicInc 原语替换为 atomicInc_system^[101]。因此，MBE 问题被划分为多个独立的子问题，并每个 GPU 独立处理这些子问题。整体运行时间由运行时间最长的 GPU 决

定。实验结果显示，GMBE 在多个 GPU 上是高效的，因为多个 GPU 上的每个线程束可以使用原子原语自动平衡工作负载，几乎没有同步开销。从理论上讲，GMBE 也可以扩展到分布式计算环境，其中多台机器（每台机器都有一个或多个 GPU）通过网络连接。由于本文侧重于单机环境，我们将探索将 GMBE 应用于分布式多机集群作为未来工作。

4.4 实验评估

本节将利用真实数据集，通过对现有串行和并行方法的对比，全面评估 GMBE 算法的性能表现。首先，介绍实验环境设置，包括实验平台、数据集、比较对象和测试方法。随后，通过对已有算法在真实数据上的执行情况进行分析，从运行时间的角度对 GMBE 进行整体评估。接着，通过消融实验，对本章提出的枚举节点重用技术、剪枝技术和负载均衡技术进行详细评估。最后，对涉及的参数、GMBE 在不同 GPU 上的适用性以及在多 GPU 上的可扩展性进行敏感性测试。

4.4.1 实验设置

实验环境设置：本节的主要实验在一台配备有 1 个 NVIDIA A100 GPU^[99] 和 4 个 Intel Xeon (R) Gold 5318Y 2.10GHz CPU 的 Linux 服务器上进行。其中每个 NVIDIA A100 GPU 包括 108 个流多处理器 (SMs) 和 40 GB 全局内存，每个 Intel Xeon (R) Gold 5318Y 2.10GHz CPU 拥有 24 个计算核心，总计 96 个计算核心。操作系统为 Linux 内核-5.4.0。在默认情况下，GMBE 算法及相关变种在单个 A100 GPU 上执行，其他现有的基于 CPU 的比较算法均在 CPU 上执行。

数据集：本节使用 12 个真实世界数据集来验证 GMBE 的性能，如表 4.1 所示。对于允许两个顶点之间存在多条边的数据集，例如 MovieLens (Mti)，我们仅保留每对顶点之间的一个唯一边用于极大二分团枚举分析。这些唯一边的数量用 $|E|$ 表示。由于在二分图中 U 和 V 是对称的，我们总是将顶点集合中拥有较少顶点的集合设置为 V ，即 $|U| > |V|$ 。我们记录了顶点集的最大度数以及最大二跳度数，用于分析 GMBE 在特定数据集上的内存使用。我们从 SNAP 仓库^[108] 获取 Amazon 和 EuAll 数据集，并从 KONECT 仓库^[89] 获取其他数据集。由于极大二分团枚举算法的运行时间主要取决于数据集的极大二分团数量，我们将所有数据集按其极大二分团计数的升序排序。

表 4.1 GMBE 实验数据集统计信息

数据集	$ U $	$ V $	$ E $	$\Delta(U)$	$\Delta_2(U)$	$\Delta(V)$	$\Delta_2(V)$	极大二分团数量
MovieLens (Mti)	16,528	7,601	71,154	640	5,817	146	3,217	140,266
Amazon (WA)	265,934	264,148	925,873	168	635	546	903	461,274
Teams (TM)	901,130	34,461	1,366,466	17	18,516	2,671	2,838	517,943
ActorMovies (AM)	383,640	127,823	1,470,404	646	3,956	294	7,798	1,075,444
Wikipedia (WC)	1,853,493	182,947	3,795,796	54	47,190	11,593	4,629	1,677,522
YouTube (YG)	94,238	30,087	293,360	1,035	37,513	7,591	7,356	1,826,587
StackOverflow (SO)	545,195	96,680	1,301,942	4,917	146,089	6,119	31,636	3,320,824
DBLP (Pa)	5,624,219	1,953,085	12,282,059	287	7,519	1,386	2,119	4,899,032
IMDB (IM)	896,302	303,617	3,782,463	1,590	15,451	1,334	15,233	5,160,061
EuAll (EE)	225,409	74,661	420,046	930	135,045	7,631	23,844	12,306,755
BookCrossing (BX)	340,523	105,278	1,149,739	2,502	151,645	13,601	53,915	54,458,953
Github (GH)	120,867	59,519	440,237	3,675	29,649	884	15,994	55,346,398

比较算法：由于目前没有现有的极大二分团枚举算法能在 GPU 上运行，我们将 GMBE 与面向 CPU 的极大二分团枚举算法进行比较，包括最近的串行版本，即 MBEA^[15]、iMBEA^[15]、PMBE^[47] 和 ooMBEA^[48]，以及最前沿的并行极大二分团枚举算法 ParMBE^[50]。为了公平比较，我们从作者处获取所有竞争对手的经过优化的代码，并在相同平台上运行它们。由于我们的服务器中仅有 96 个 CPU 计算核心，我们将 ParMBE 设置为 96 个线程运行。

测量方法：我们测量每个算法的运行时间，不包括从磁盘读取图所花费的时间。在没有特别说明的情况下，GMBE 使用枚举节点重用迭代枚举所有极大二分团，通过局部邻居数量裁剪无用节点，并应用负载感知的任务中心方案以实现负载平衡。默认情况下，GMBE 将 *bound_height* 和 *bound_size* 的阈值分别设置为 20 和 1,500，将 WarpPerSM 设置为 16，并在枚举前根据顶点度按升序对 V 中顶点进行排序。我们还实现了其他变体来评估本文提出的技术，并将在相应的实验中详细介绍这些变体。

4.4.2 整体评估

表 4.2 展示了 GMBE 在真实数据集上与最先进的极大二分团枚举算法的运行时间对比结果。实验结果表明，由于 GMBE 能够高效利用 GPU 上的大量计算资源，因此在所有测试数据集上，GMBE 在 CPU 上比任何其他竞争对手快 3.5 倍–69.8 倍。具体而言，在单个 A100 GPU 上，GMBE 在 ActorMovies 数据集上的表现超过了 96 核 CPU 上最先进的并行极大二分团枚举算法 ParMBE 达到 70.6 倍。与所有在 Github 上花费超过

表 4.2 GMBE 整体运行时间评估（单位：秒）

数据集	GMBE	MBEA	iMBEA	PMBE	ooMBEA	ParMBE	加速比
Mti	0.075	14.983	5.528	6.505	<u>2.432</u>	2.555	32.6
WA	0.009	0.838	1.055	0.922	<u>1.412</u>	<u>0.536</u>	62.8
TM	0.107	87.377	18.930	68.180	4.087	<u>0.949</u>	8.9
AM	0.190	131.873	39.050	887.641	<u>11.796</u>	13.436	62.0
WC	0.630	117.220	29.828	1143.291	<u>12.896</u>	2.225	3.5
YG	0.814	1214.139	314.321	141.712	112.834	<u>9.439</u>	11.6
SO	25	10814	4533	10755	758	<u>691</u>	27.2
Pa	0.11	46.72	18.66	53944.64	19.48	<u>7.14</u>	63.7
IM	2	953	420	9018	147	<u>123</u>	69.8
EE	13	7105	2386	2370	1255	<u>396</u>	30.8
BX	50	74471	32040	9184	8367	<u>892</u>	17.8
GH	132	58199	30800	7321	8525	<u>2412</u>	18.2

40 分钟枚举所有极大二分团的现有极大二分团枚举算法相比，GMBE 仅需 132 秒，因此对实际大数据集上的极大二分团枚举是非常有帮助的。此外，我们使用 NVIDIA Nsight Compute 软件^[104] 对 GMBE 进行性能分析。分析结果显示，所有真实数据集上的平均线程束执行效率为 64%，内存利用率为 12%。这些结果可以归因于极大二分团枚举问题中固有的不规则性^[46]。

4.4.3 技术点分解评估

我们设计了消融实验，分别对 GMBE 中的枚举节点重用方法、剪枝方法和调度方法等技术点进行了分解评估。

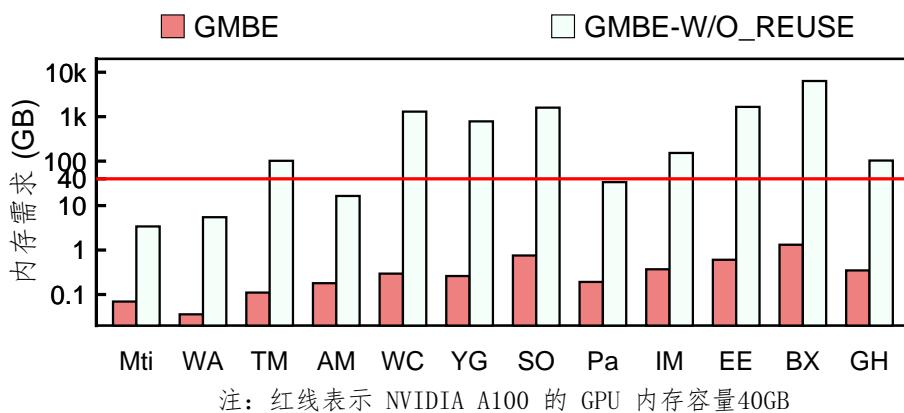


图 4.8 枚举节点重用方法的分解评估（对数形式）

枚举节点重用方法的效果：为了研究 4.3.1 节中枚举节点重用方法的效果，我们设

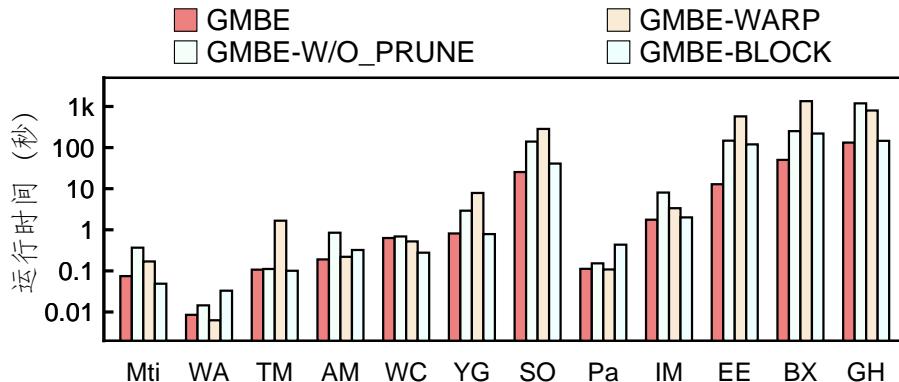


图 4.9 剪枝方法和任务调度方法的分解评估（对数形式）

表 4.3 剪枝方法使用前后生成的非极大二分团与极大二分团比值比较 δ/α

数据集	Mti	WA	TM	AM	WC	YG	SO	Pa	IM	EE	BX	GH
GMBE	9.04	0.734	1.63	12.9	0.71	2.11	89.4	0.362	15.5	4.04	3.40	11.1
GMBE-w/o_PRUNE	66.0	3.68	3.88	53.0	2.89	20.1	174	1.43	74.4	56.0	27.3	51.4

计了变体 GMBE-w/o_REUSE，根据 4.3.1 节的要求在 GPU 上预先分配内存。我们分别估计了 GMBE 在是否开启枚举节点重用优化方法的情况下使用 `cudaMalloc` 原语分配的内存需求。这些内存需求包括用于输入二分图和运行时子树的预分配内存。图 4.8 显示，枚举节点重用方法在所有测试数据集上将内存需求显著减少了 49 倍–4,819 倍。而 GMBE-w/o_REUSE 在多个数据集上内存需求超出了 A100 GPU 的内存容量上限，因此导致程序无法运行。

剪枝方法的效果：为了研究 4.2.2 节中局部邻居数量感知的剪枝方法的效果，我们设计了一个变体 GMBE-w/o_PRUNE，仅禁用了 GMBE 的剪枝功能。如图 4.9 所示，GMBE 总是优于 GMBE-w/o_PRUNE。这是因为局部邻居数量感知的剪枝方法通过批量比较局部邻居，在控制线程分歧的同时裁剪了大量极大二分团枚举问题的搜索空间。同时，剪枝方法通过比较候选顶点集中相同顶点的局部邻居数量来增强内存访问的连续性。为了进一步探索剪枝的效率，我们使用 α 表示极大二分团的数量，使用 δ 表示通过节点检查（算法 4.1 的第 18 行）生成的被剪枝的非极大二分团的数量。由于 α 对于每个数据集都保持不变，我们使用比值 δ/α 来表示 GMBE 和 GMBE-w/o_PRUNE 的剪枝效率，如表 4.3 所示。通过比较 GMBE 和 GMBE-w/o_PRUNE 的 δ/α 比值，我们观察到所提出的剪枝方法可以在所有测试数据集中避免 48.7%–92.8% 的非极大二分团检查。由于极大二分团枚举任务的枚举空间随着极大二分团数量的增加而增长，剪枝技术在更大的数据集

中尤为重要。具体而言，局部邻居数量感知的剪枝方法将运行时间从 Github 上的 1,191 秒显著减少到 132 秒。

负载感知的任务调度方法的效果：为研究 4.3.3 节中负载感知的任务调度方法的效果，我们设计了两个变体 GMBE-WARP 和 GMBE-BLOCK，分别采用基于线程束和基于线程块的方案。如图 4.9 所示，在包括 EuALL、Github、BookCrossing、StackOverflow 和 IMDB 在内的大规模实验数据集上，GMBE 明显比 GMBE-WARP 和 GMBE-BLOCK 更快，并在其他数据集上花费不到一秒的时间。在大规模实验数据集上，GMBE 的性能更好，因为它动态检测和分区具有重负载的任务，并管理无锁任务队列以在更细粒度上重新平衡工作负载。实验结果显示，在 EuALL 数据集上，GMBE、GMBE-WARP 和 GMBE-BLOCK 的运行时间为 13 秒、573 秒和 119 秒。由此可见，相较于其他任务调度方法，负载感知的任务调度方法能够显著提升计算性能，尤其在负载不均匀的情况下表现更为突出。

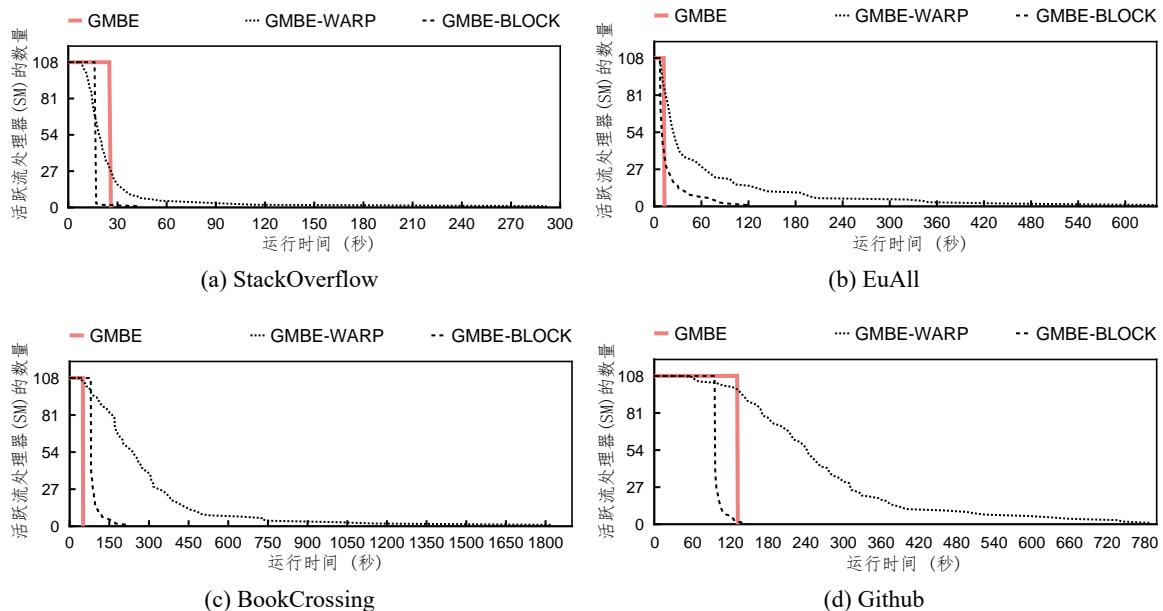


图 4.10 不同调度方法中 SM 上的运行时负载比较

为了更深入地探究 GPU 上进行极大二分团枚举面临的负载不平衡问题，我们记录了在运行 GMBE、GMBE-WARP 和 GMBE-BLOCK 时 GPU 内活跃流处理器数量随着运行时间的变化情况。图 4.10 对比展示了在 StackOverflow、EuALL、BookCrossing 和 Github 数据集上 SM 的运行负载。由于存在工作负载不平衡，一些负载较轻的 SM 可能会提前完成并等待那些负载较重的 SM，这将增加整体运行时间。由于负载不平衡问题

的存在，GMBE-WARP 中活跃的 SM 数量迅速减少，因而表现最差。相比之下，GMBE-BLOCK 获得了更好的性能，因为它可以为每个工作负载分配更多资源（即一个线程块而非一个线程束），从而减少了等待具有最重负载的 SM 的时间。然而，GMBE-BLOCK 仍然存在不足，因为极大二分团枚举问题的工作负载可能严重不平衡。例如，在 EuALL 数据集上，超过 80% 的 SM (86 个 SM / 108 个 SM) 浪费了超过 80% 的运行时间 (98 秒 / 118 秒) 在等待最慢的 SM。相比之下，GMBE 总是能够实现最佳性能，因为它能够在最细粒度上工作，使得每个 SM 大致同时完成其工作。在 BookCrossing 数据集上，甚至在 GMBE-BLOCK 的活跃 SM 数量开始减少之前，GMBE 就已经完成了运算，因为它在每个 SM 中激活了所有的线程束，而相比之下，GMBE-BLOCK 在运行时可能只使用了每个 SM 中的一小部分线程束。

4.4.4 敏感性测试

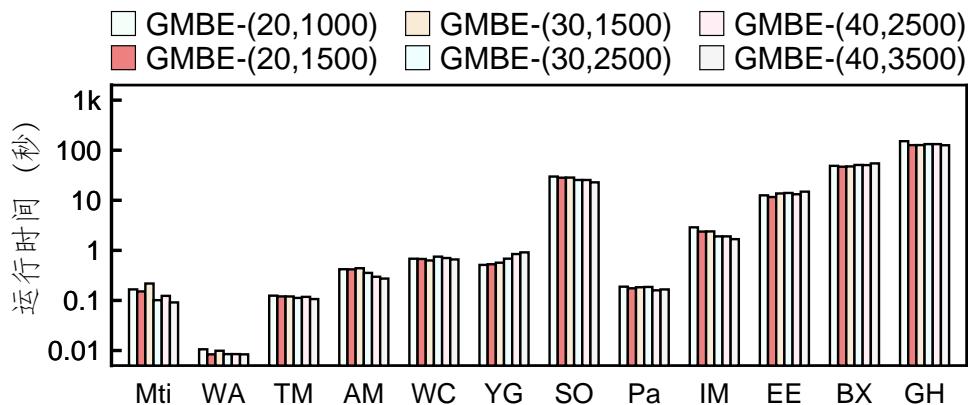


图 4.11 负载感知任务调度下阈值设置（对数形式）

负载感知任务调度阈值对性能的影响：为了探究在 4.3.3 节中阈值 $bound_height$ 和 $bound_size$ 的有效配置，我们设计了多个变种 GMBE- (m, n) ，其中 m 和 n 分别代表 $bound_height$ 和 $bound_size$ 。我们总是设置 m 大于 n^2 ，因为我们知道 $|L| \times |C|$ 始终大于或等于 $(\min\{|L|, |C|\})^2$ 。阈值的选择是并行粒度和同步开销之间的权衡。我们需要更小的阈值来在更细的粒度上平衡工作负载。然而，阈值不应该过小。否则，我们将不得不处理更多带有巨大同步开销的任务。图 4.11 表明，在大多数情况下，变种 GMBE- $(20, 1500)$ 在运行时间上优于其他变种。因此，GMBE 默认应用这种配置。

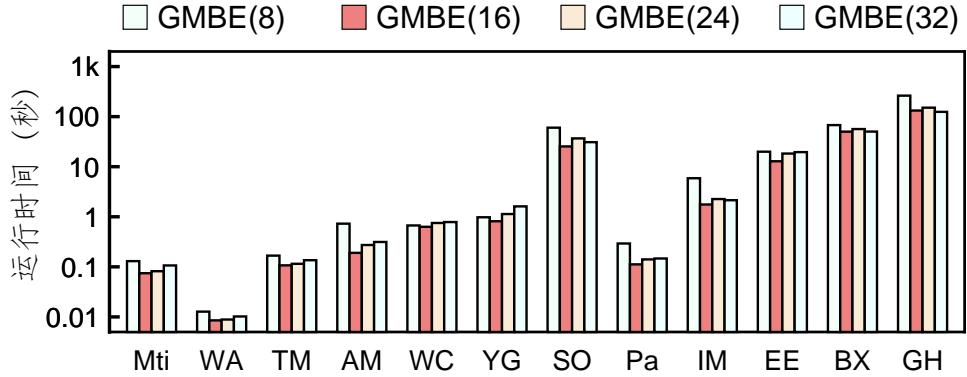


图 4.12 参数 WarpPerSM 设置 (对数形式)

每个流处理器中线程束数量的影响：为确定 4.3.3 节中 PT 模型中参数 WarpPerSM，我们设计了将 WarpPerSM 设置为 8、16、24 和 32 的不同变种。参数 WarpPerSM 的选择是并行性和每个线程束资源之间的权衡。直觉上，我们希望 WarpPerSM 更大，以便可以并行运行更多的极大二分团枚举任务。然而，WarpPerSM 不应该过大，因为每个流多处理器中的计算资源（例如寄存器）是有限的。较大的 WarpPerSM 可能会降低 GMBE 的性能，因为每个线程束将拥有更少的资源来运行极大二分团枚举任务。图 4.12 显示，在大多数大规模数据集（如 BookCrossing、StackOverflow、IMDB、DBLP 和 EuAll）中，变种 GMBE-16 的性能优于其他变种 3.83 倍。此外，由于其广泛的枚举空间需要更多线程束并行枚举极大二分团，GMBE-16 在 Github 上比 GMBE-32 慢 0.94 倍。考虑到在大多数情况下的效率，GMBE 默认将 WarpPerSM 设置为 16。

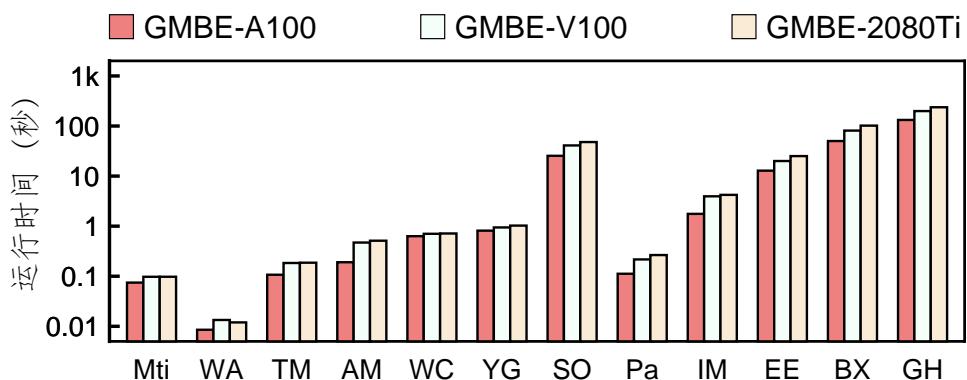


图 4.13 GMBE 在不同型号 GPU 上的适用性 (对数形式)

在不同 GPU 上的适用性：为了探究 GMBE 在不同 GPU 上的适用性，我们分别在 NVIDIA A100 GPU (108 个流多处理器，40 GB 全局内存)、NVIDIA V100 GPU (80 个

流多处理器，32 GB 全局内存)^[109] 和 NVIDIA 2080Ti GPU (68 个流多处理器，11 GB 全局内存)^[110] 上对 GMBE 进行评估。图 4.13 显示 GMBE 在所有三种 GPU 上都表现出了适用性。GMBE-A100 稍微快于 GMBE-V100 和 GMBE-2080Ti，因为 A100 GPU 包含比其他 GPU 更多的计算资源。

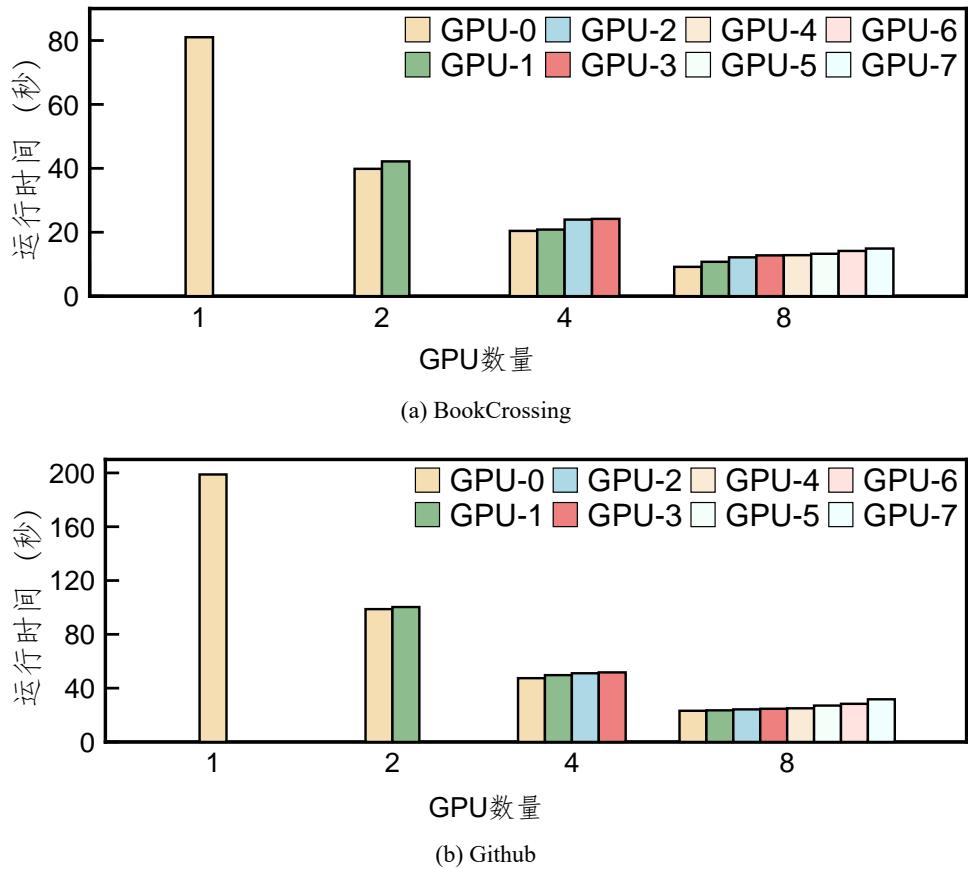


图 4.14 GMBE 在多 GPU 环境下的可扩展性

多 GPU 环境下的可扩展性：为了探究 GMBE 在多 GPU 环境下的可扩展性，我们在一台配备了 8 个 NVIDIA V100 GPU 的机器上进行了实验。为了优化 GMBE 以适应多 GPU 配置，我们将问题分解为多个独立的子问题，总运行时间由运行时间最长的子问题决定。图 4.14 显示，随着 GPU 数量的增加，GMBE 在 Github 和 BookCrossing 数据集上呈线性扩展，因为每个 GPU 几乎同时完成其执行。在多个 GPU 的帮助下，GMBE 能够在 31 秒内枚举出 Github 数据集中超过 5500 万个极大二分团，相较于 96 核 CPU 机器上最先进的并行极大二分团枚举算法 ParMBE (即 2411 秒)，性能提升了 77 倍。

4.5 本章小结

本章介绍了针对极大二分团枚举问题的高效 GPU 解决方案 GMBE。首先，针对 GPU 上运行极大二分团枚举算法所带来的内存短缺问题，我们提出了基于枚举节点重用的迭代方法，通过重用根节点内存，减小动态内存分配带来的内存开销。其次，针对现有剪枝方法引入的线程分歧问题，本章提出了局部邻居数量感知的剪枝方法，通过对中间结果的批量比较，在实现高效剪枝性能的同时缓解了线程分歧问题。再次，针对现有方法难以实现极大二分团枚举任务负载均衡的问题，本章提出了负载感知的任务调度方法，动态识别并拆分具有高负载的任务，实现细粒度的任务调度。最后，我们结合上述技术，实现了 GMBE 算法。实验结果充分证明了 GMBE 算法在 GPU 上的高性能以及本章中所有方法的具体作用。

5 总结与展望

本章对全文的研究内容和主要创新点做出总结，并对极大二分团枚举问题的未来可以继续探索的研究问题和方向进行展望。

5.1 工作总结

随着信息技术的迅猛发展，大规模数据的生成和积累已成为常态。为了充分挖掘和利用这些数据中的有效关联信息，二分图结构被广泛应用于表示两个不同群体之间的联系，比如在电子商务场景中用户和商品之间的购买关系。极大二分团是二分图中的一类稠密子图，代表着数据集中那些紧密连接的群体，能够有效揭示二分图中存在的某种规律或共同特征，为探究群体行为内部的脉络和联系提供帮助。极大二分团枚举问题的目标是高效地识别并枚举给定二分图中的全部极大二分团，该问题具有广泛的应用场景，并受到学术界和工业界的广泛关注。本文的主要工作是研究高效的极大二分团枚举方法。针对极大二分团枚举问题中搜索空间庞大、使用静态数据结构以及并行扩展性差等问题，本文从搜索空间剪枝方法、自适应的数据结构和基于 GPU 的并行算法三个方面分别提出了三种独立的极大二分团枚举方法。通过大量实验证明，这三种方法相较于现有的极大二分团枚举方法均表现出明显的性能优势。本文的主要研究内容和贡献如下：

(1) 主动的极大二分团枚举算法。为了优化极大二分团枚举问题的搜索空间，现有工作设计了多种优化方法。然而，现有工作的搜索空间仍然非常庞大，在枚举过程中仍然会生成大量的非极大二分团，消除非极大二分团的过程带来高昂的节点检查开销。对此，我们提出了主动的极大二分团枚举算法 AMBEA，主要包括主动的集合枚举树 (ASE) 和主动的顶点合并剪枝方法 (AMP) 两个核心技术点。主动的集合枚举树打破了现有枚举树仅允许使用部分顶点进行节点生成的结构限制，总是使用全部顶点将每个二分团扩展为极大二分团，裁剪了大量产生非极大二分团的分枝。主动的顶点合并剪枝方法打破了现有剪枝方法由特定顶点触发的限制，通过改变节点的生成过程，主动地合并具有相同局部邻居的顶点，提升剪枝效率。实验结果表明，AMBEA 对比最新的 ooMBEA 等算法压缩了 2.4-9.0 倍的搜索空间，缩短了 1.2-5.3 倍的运行时间。

(2) 自适应的极大二分团枚举算法。现有的极大二分团枚举问题研究往往注重于算法优化，却忽略了数据结构表示所带来的固有低效性问题，导致现有方法仅适用于小数据集，其中包含不超过 1 亿个极大二分团。针对这一问题，我们提出了自适应的极大二分团枚举算法 AdaMBE，主要包括基于局部计算子图的优化方法（LCG）和基于位图的动态子图方法（BDS）两个核心技术点。为解决现有方法忽略了枚举过程中子图动态变化特性而导致的大量无效内存访问问题，基于局部计算子图的优化方法通过动态缓存枚举过程中的计算子图，优化算法的枚举过程，从而减少无效的顶点访问、集合运算和枚举节点。考虑到现有方法在邻接表上进行大量集合运算会带来高计算开销，基于位图的动态子图方法在枚举过程中自适应地生成子位图，并通过位图上的位运算加速集合运算。该方法充分结合了邻接表与位图两种不同表示方法的优势，同时也考虑到二分图中两个顶点集大小不同的特性。实验结果显示，AdaMBE 相比最新的 ooMBEA 等算法，运行时间缩短了 1.6 至 49.7 倍，并能够应用于包含超过 190 亿个极大二分团的数据集。

(3) 基于 GPU 的极大二分团枚举算法。目前，现有的极大二分团枚举算法均采用 CPU 实现，因此其并行扩展性受到 CPU 计算核心数量的限制。为解决这一问题，我们引入了具有大量计算核心的 GPU 作为计算资源，用于加速极大二分团枚举过程。我们提出了基于 GPU 的高效极大二分团枚举解决方案 GMBE，主要包括基于枚举节点重用的迭代方法、局部邻居数量感知的剪枝方法以及负载感知的任务调度方法，分别用于解决内存短缺、线程分歧和负载不均等问题。基于枚举节点重用的迭代方法仅存储子枚举树根节点以及对应的元数据，通过重用枚举树根节点的内存，避免为新枚举节点动态分配内存，从而减少内存开销。局部邻居数量感知的剪枝方法通过记录枚举过程中顶点局部邻居数量的变化来对枚举空间进行剪枝，并通过批量比较局部邻居的方法，在剪枝的同时最小化线程分歧问题。负载感知的任务调度方法将每棵子枚举树对应的计算任务与一个线程束的计算资源进行绑定，在运行时动态预估子枚举树的大小并对较大的子枚举树进行进一步拆分，实现了细粒度的负载均衡。实验结果显示，基于 GPU 的 GMBE 方法比现有基于 96 个 CPU 的并行算法 ParMBE 实现了 70.6 倍的性能提升。

综上所述，本文聚焦于极大二分团枚举问题。从剪枝能力、数据结构和并行实现三个方面进行探索，提出了三种独立的高性能极大二分团枚举算法和多个通用的核心技术点。这三种算法对极大二分团枚举问题的性能提升方式进行了全面探索，将枚举介质扩展至 GPU，并实现对包含超过百亿个极大二分团规模的二分图进行极大二分团枚举。

5.2 研究展望

极大二分团枚举问题作为图数据挖掘和图论领域中的经典难题，在广泛应用的同时，与图模式挖掘、极大团枚举等问题密切相关。尽管本文已经就极大二分团枚举问题展开了多方面的探索和实践，但相关领域仍有许多问题值得深入探究和发掘。未来的研究可以在以下几个方面展开：

(1) 极大二分团枚举问题深度优化。首先，从技术路线的宏观层面来看，目前本文提出的三条技术路线相互独立。对枚举性能的深入挖掘可以从融合不同的技术路线入手。在结合 AMBEA 和 AdaMBE 的技术路线时，可以利用 AdaMBE 中的基于位图的动态子图方法加速枚举节点的计算过程，从而在 AMBEA 的基础上形成基于 CPU 的高效极大二分团枚举算法。在结合 AMBEA 和 GMBE 的技术路线时，可以利用 AMBEA 的剪枝方法优化 GMBE 的枚举空间，有望提升算法效率。然而，合并 AMBEA 和 GMBE 需要克服 AMBEA 剪枝方法可能导致的线程分歧问题。在结合 AdaMBE 和 GMBE 的技术路线时，可以将 AdaMBE 中的基于位图的动态子图方法直接迁移到 GPU 中，从而在 GMBE 的基础上形成基于 GPU 的高效极大二分团枚举算法。其次，从枚举过程中节点计算的中观层面来看，我们研究发现节点检查在枚举过程中占用大量的枚举时间。虽然目前的研究方法已经提出大量的剪枝优化方法，但高开销的节点检查过程仍无法避免。考虑到现有方法都是针对算法本身设计的启发式方法，未来的研究工作可以尝试引入机器学习方法，通过深度网络模型对枚举节点进行快速批量检查，进而优化枚举性能。最后，从集合运算和顶点访问的微观层面来看，虽然 AdaMBE 方法注意到了计算过程中的细粒度冗余问题，即无效顶点访问问题，但冗余计算仍然大量存在。未来的研究工作可以从冗余的角度进行深入优化，例如从系统层面同时处理多个枚举节点并实时动态减少冗余，进一步提升极大二分团枚举的性能。

(2) 相关图挖掘问题的深度优化。在本文的研究过程中，我们发现极大二分团枚举问题中的技术挑战在相关图挖掘问题中同样存在。这导致目前的极大团枚举问题的 GPU 实现方案性能不如传统的 CPU 实现，在图模式挖掘问题中，目标子图内的顶点数量通常较少（不超过 10 个）。为了解决这一问题，未来的研究工作可以考虑以下几个方面的优化：首先，可以将 AMBEA 中的剪枝优化方法迁移到相关问题中，以突破相关问题中枚举树的结构限制。通过引入更高效的剪枝策略，可以减少枚举过程中的无效计

算，从而提升相关图挖掘问题的性能。其次，可以将 AdaMBE 中混合不同图表示的思路迁移到相关问题中，加速问题求解过程中大量的集合交集运算。通过选择合适的图表示方法，并结合适当的数据结构和算法设计，可以有效降低集合操作的计算复杂度，从而提高相关图挖掘问题的效率。最后，可以将 GMBE 中枚举节点重用与任务感知的细粒度任务调度方法迁移到相关问题中，进一步拓展图模式挖掘问题中的目标子图规模。通过合理地利用枚举节点重用技术和任务感知的调度策略，可以优化多任务之间的调度效率，提高相关图挖掘问题的可扩展性和并行性。通过以上的优化措施，我们有望在相关图挖掘问题中实现更高效的算法设计与实现，进而推动相关图挖掘领域的进步。

(3) 极大二分团枚举问题的应用领域扩展。近年来，研究者们在不同类型的二分图上定义了各种形式的二分团，从而拓宽了二分图数据挖掘的研究内容和应用领域。考虑到极大二分团枚举问题在这类挖掘中的基础地位，我们的研究工作将在相关衍生问题中扮演着基础性的角色。例如，在最大边二分团搜索问题中，我们可以基于任一极大二分团枚举算法，并结合现有的最大边二分团搜索问题的剪枝策略，提出新的高效实现方案。同样地，我们的研究成果也可应用于优化类似的二分团枚举问题，如相似二分团枚举、 (p, q) 二分团枚举、公平极大二分团枚举以及二分团渗透社区等问题的求解。这为未来更多自定义二分团挖掘算法的优化提供了多方面的思路与启示。

参考文献

- [1] Maier C, Simovici D. Bipartite Graphs and Recommendation Systems[J]. Journal of Advances in Information Technology-in print, 2022.
- [2] Lyu B, Qin L, Lin X, et al. Maximum Biclique Search at Billion Scale[J]. Proceedings of the VLDB Endowment (PVLDB), 2020.
- [3] Li Z, Hui P, Zhang P, et al. What Happens Behind the Scene? Towards Fraud Community Detection in E-Commerce from Online to Offline[C]//Companion Proceedings of the Web Conference. 2021: 105-113.
- [4] Lyu B, Qin L, Lin X, et al. Maximum and Top-k Diversified Biclique Search at Scale[J]. The VLDB Journal (VLDBJ), 2022, 31(6): 1365-1389.
- [5] 李爽. 多值属性二部图选择最优组: 天际线二部图的研究与实现[D]. 广州大学, 2023.
- [6] Liu G, Sim K, Li J. Efficient Mining of Large Maximal Bicliques[C]//Proceedings of the International Conference of Data Warehousing and Knowledge Discovery (DaWaK). 2006: 437-448.
- [7] 邹凌君, 陈峻, 戴彩艳. 基于广义后缀树的二分网络社区挖掘算法[J]. 计算机科学, 2017, 44(07): 221-226.
- [8] Alzahrani T, Horadam K. Finding Maximal Bicliques in Bipartite Networks Using Node Similarity[J]. Applied Network Science (ANS), 2019, 4(1): 1-25.
- [9] Gibson D, Kumar R, Tomkins A. Discovering Large Dense Subgraphs in Massive Graphs[C]//Proceedings of the VLDB Endowment (PVLDB). 2005: 721-732.
- [10] Schweiger R, Linial M, Linial N. Generative Probabilistic Models for Protein–Protein Interaction Networks—The Biclique Perspective[J]. Bioinformatics, 2011, 27(13): I142-I148.
- [11] Xiang Y, Payne P R, Huang K. Transactional Database Transformation and Its Application in Prioritizing Human Disease Genes[J]. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), 2011, 9(1): 294-304.
- [12] Driskell A C, Ané C, Burleigh J G, et al. Prospects for Building the Tree of Life from Large Sequence Databases[J]. Science, 2004, 306(5699): 1172-1174.
- [13] Kaytoue M, Kuznetsov S O, Napoli A, et al. Mining Gene Expression Data with Pattern Structures in Formal Concept Analysis[J]. Information Sciences, 2011, 181(10): 1989-2001.
- [14] Liu Y. Computational Methods for Identifying MicroRNA-Gene Regulatory Modules[G]//Handbook of Statistical Bioinformatics. Springer, 2022: 187-208.
- [15] Zhang Y, Phillips C A, Rogers G L, et al. On Finding Bicliques in Bipartite Graphs: A Novel Algorithm and Its Application to the Integration of Diverse Biological Data Types[J]. BMC Bioinformatics, 2014, 15: 1-18.
- [16] Siswantining T, Bustamam A, Swasti O, et al. Analysis and Prediction of Protein Interactions Between HIV-1 Protein and Human Protein Using LCM-MBC Algorithm Combined with Association Rule Mining[J]. Communications in Mathematical Biology and Neuroscience, 2021, 2021: 64.
- [17] Yang J, Peng Y, Zhang W. (P, Q)-Biclique Counting and Enumeration for Large Sparse Bipartite Graphs[J]. Proceedings of the VLDB Endowment (PVLDB), 2021, 15(2): 141-153.
- [18] Yang J, Peng Y, Ouyang D, et al. (P, Q)-Biclique Counting and Enumeration for Large Sparse Bipartite Graphs[J]. The VLDB Journal (VLDBJ), 2023: 1-25.
- [19] Ye X, Li R H, Dai Q, et al. Efficient Biclique Counting in Large Bipartite Graphs[J]. Proceedings of the ACM on Management of Data, 2023, 1(1): 1-26.
- [20] Hermelin D, Manoussakis G. Efficient Enumeration of Maximal Induced Bicliques[J]. Discrete Applied Mathematics (DAM), 2021, 303: 253-261.
- [21] Gély A, Nourine L, Sadi B. Enumeration Aspects of Maximal Cliques and Bicliques[J]. Discrete Applied Mathematics (DAM), 2009, 157(7): 1447-1459.
- [22] 何琨, 邹晨昊, 周建荣. 大规模稀疏图的极大团枚举算法[J]. 华中科技大学学报 (自然科学版), 2017, 45(12): 1-6.
- [23] Blanuša J, Stoica R, Ienne P, et al. Manycore Clique Enumeration with Fast Set Intersections[J]. Proceedings of the VLDB Endowment (PVLDB), 2020, 13(12): 2676-2690.

- [24] 杜明, 钟鹏, 周军锋. 一种面向不确定极大团枚举的高效验证算法[J]. 智能计算机与应用, 2020, 10(03): 14-20.
- [25] Wei Y W, Chen W M, Tsai H H. Accelerating the Bron-Kerbosch Algorithm for Maximal Clique Enumeration Using GPUs[J]. IEEE Transactions on Parallel and Distributed Systems (TPDS), 2021, 32(9): 2352-2366.
- [26] 王恒. 概率图上 Top-K 极大团枚举问题研究[D]. 东华大学, 2021.
- [27] Dai Q, Li R H, Liao M, et al. Fast Maximal Clique Enumeration on Uncertain Graphs: A Pivot-Based Approach[C]//Proceedings of the International Conference on Management of Data (SIGMOD). 2022: 2034-2047.
- [28] 许绍显, 廖小飞, 邵志远, 等. 图数据中极大团枚举问题的求解: 研究现状与挑战[J]. 中国科学: 信息科学, 2022, 52(05): 784-803.
- [29] Zaki M J, Ogiara M. Theoretical Foundations of Association Rules[C]//3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery. 1998: 71-78.
- [30] Martin-Prin J, Dlala I O, Travers N, et al. A Distributed SAT-Based Framework for Closed Frequent Itemset Mining[C]//International Conference on Advanced Data Mining and Applications (ADMA). 2022: 419-433.
- [31] Konecny J, Krajča P. Systematic Categorization and Evaluation of CbO-Based Algorithms in FCA[J]. Information Sciences, 2021, 575: 265-288.
- [32] Janostik R, Konecny J, Krajča P. LCM from FCA Point of View: A CbO-Style Algorithm with Speed-Up Features[J]. International Journal of Approximate Reasoning (IJAR), 2022, 142: 64-80.
- [33] Chen L, Liu C, Zhou R, et al. Efficient Exact Algorithms for Maximum Balanced Biclique Search in Bipartite Graphs[C]//Proceedings of the International Conference on Management of Data (SIGMOD). 2021: 248-260.
- [34] Yao K, Chang L, Yu J X. Identifying Similar-Bicliques in Bipartite Graphs[J]. Proceedings of the VLDB Endowment (PVLDB), 2022, 15(11): 3085-3097.
- [35] 蒋意浩. 大规模二分图上 (m,n) 二分团计数问题研究[D]. 东华大学, 2022.
- [36] Yin Z, Zhang Q, Zhang W, et al. Fairness-aware Maximal Biclique Enumeration on Bipartite Graphs[J]. IEEE 39th International Conference on Data Engineering (ICDE), 2023: 1665-1677.
- [37] Chen Z, Zhao Y, Yuan L, et al. Index-Based Biclique Percolation Communities Search on Bipartite Graphs[C]//IEEE 39th International Conference on Data Engineering (ICDE). 2023: 2699-2712.
- [38] Wang J, Yang J, Ma Z, et al. Efficient Maximal Biclique Enumeration on Large Uncertain Bipartite Graphs[J]. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2023.
- [39] Sun R, Wu Y, Chen C, et al. Maximal Balanced Signed Biclique Enumeration in Signed Bipartite Graphs[C]//IEEE 38th International Conference on Data Engineering (ICDE). 2022: 1887-1899.
- [40] Sun R, Chen C, Wang X, et al. Efficient Maximum Signed Biclique Identification[C]//IEEE 39th International Conference on Data Engineering (ICDE). 2023: 1313-1325.
- [41] Wang J, Yang J, Zhang C, et al. Efficient Maximum Edge-Weighted Biclique Search on Large Bipartite Graphs[J]. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2022.
- [42] Zhao Y, Chen Z, Chen C, et al. Finding the Maximum K -Balanced Biclique on Weighted Bipartite Graphs[J]. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2022.
- [43] Ma Z, Liu Y, Hu Y, et al. Efficient Maintenance for Maximal Bicliques in Bipartite Graph Streams[J]. World Wide Web (WWW), 2022, 25(2): 857-877.
- [44] Alexe G, Alexe S, Crama Y, et al. Consensus Algorithms for the Generation of All Maximal Bicliques[J]. Discrete Applied Mathematics (DAM), 2004, 145(1): 11-21.
- [45] 中华人民共和国商务部. 2022 年中国电子商务发展总报告[EB/OL]. 2022. <http://images.mofcom.gov.cn/dzswws/202306/20230609104929992.pdf>.
- [46] Burtscher M, Nasre R, Pingali K. A Quantitative Study of Irregular Programs on GPUs[C]//2012 IEEE International Symposium on Workload Characterization (IISWC). 2012: 141-151.
- [47] Abidi A, Zhou R, Chen L, et al. Pivot-Based Maximal Biclique Enumeration[C]//Proceedings of the 29th International Conference on International Joint Conferences on Artificial Intelligence (IJCAI). 2020: 3558-3564.
- [48] Chen L, Liu C, Zhou R, et al. Efficient Maximal Biclique Enumeration for Large Sparse Bipartite Graphs[J]. Proceedings of the VLDB Endowment (PVLDB), 2022, 15(8): 1559-1571.

- [49] Mukherjee A P, Tirthapura S. Enumerating Maximal Bicliques from a Large Graph Using MapReduce[J]. IEEE Transactions on Services Computing (TSC), 2016, 10(5): 771-784.
- [50] Das A, Tirthapura S. Shared-Memory Parallel Maximal Biclique Enumeration[C]//IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC). 2019: 34-43.
- [51] Rymon R. Search Through Systematic Set Enumeration[C]//Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR). 1992: 539-550.
- [52] Malgrange Y. Recherche des Sous-Matrices Premières d'une Matrice à Coefficients Binaires. Applications à Certains Problèmes de Graphe[C]//Proceedings of the Deuxième Congrès de l'AFCALTI. 1962: 231-242.
- [53] Sanderson M J, Driskell A C, Ree R H, et al. Obtaining Maximal Concatenated Phylogenetic Data Sets from Large Sequence Databases[J]. Molecular Biology and Evolution, 2003, 20(7): 1036-1042.
- [54] Mushlin R A, Kershenbaum A, Gallagher S T, et al. A Graph-Theoretical Approach for Pattern Discovery in Epidemiological Research[J]. IBM Systems Journal, 2007, 46(1): 135-149.
- [55] Makino K, Uno T. New Algorithms for Enumerating All Maximal Cliques[C]//Algorithm Theory-SWAT. 2004: 260-272.
- [56] Das A, Sanei-Mehri S V, Tirthapura S. Shared-memory Parallel Maximal Clique Enumeration from Static and Dynamic Graphs[J]. ACM Transactions on Parallel Computing (TOPC), 2020, 7(1): 1-28.
- [57] Jin Y, Xiong B, He K, et al. On Fast Enumeration of Maximal Cliques in Large Graphs[J]. Expert Systems with Applications (ESWA), 2022, 187: 115915.
- [58] Li J, Li H, Soh D, et al. A Correspondence Between Maximal Complete Bipartite Subgraphs and Closed Patterns[C]//Proceedings of the 9th European Conference on European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD). 2005: 146-156.
- [59] Grahne G, Zhu J. Reducing the Main Memory Consumptions of FPmax* and FPclose[C]//Proceedings of the Workshop on Frequent Item Set Mining Implementations (FIMI). 2004: 75.
- [60] Uno T, Kiyomi M, Arimura H, et al. LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed-/Maximal Itemsets[C]//Proceedings of the Workshop on Frequent Item Set Mining Implementations (FIMI): vol. 126. 2004.
- [61] Li J, Liu G, Li H, et al. Maximal Biclique Subgraphs and Closed Pattern Pairs of the Adjacency Matrix: A One-to-One Correspondence and Mining Algorithms[J]. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2007, 19(12): 1625-1637.
- [62] Gaume B, Navarro E, Prade H. A Parallel between Extended Formal Concept Analysis and Bipartite Graphs Analysis[C]//Proceedings of the Computational Intelligence for Knowledge-Based Systems Design. 2010: 270-280.
- [63] Bhatnagar R, Kumar L. An Efficient Map-Reduce Algorithm for Computing Formal Concepts from Binary Data[C]//IEEE International Conference on Big Data (Big Data). 2015: 1519-1528.
- [64] He Y, Li R, Mao R. An Optimized MBE Algorithm on Sparse Bipartite Graphs[C]//International Conference on Smart Computing and Communication (ICSCC). 2018: 206-216.
- [65] 何宇. 面向大规模二分图的团枚举算法研究[D]. 深圳大学, 2019.
- [66] Qin C, Liao M, Liang Y, et al. Efficient Algorithm for Maximal Biclique Enumeration on Bipartite Graphs[C]//Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD): vol. 1075. 2020: 3-13.
- [67] Damaschke P. Enumerating Maximal Bicliques in Bipartite Graphs with Favorable Degree Sequences[J]. Information Processing Letters (IPL), 2014, 114(6): 317-321.
- [68] Shaham E, Yu H, Li X L. On Finding the Maximum Edge Biclique in a Bipartite Graph: A Subspace Clustering Approach[C]//Proceedings of the 2016 SIAM International Conference on Data Mining (SDM). 2016: 315-323.
- [69] Depth-First Search[EB/OL]. https://en.wikipedia.org/wiki/Depth-first_search.
- [70] Breadth-First Search[EB/OL]. https://en.wikipedia.org/wiki/Breadth-first_search.
- [71] Minimum Spanning Tree[EB/OL]. https://en.wikipedia.org/wiki/Minimum_spanning_tree.
- [72] Shortest Path Problem[EB/OL]. https://en.wikipedia.org/wiki/Shortest_path_problem#Single-source_shortest_paths.

- [73] Jamshidi K, Mahadasa R, Vora K. Peregrine: A Pattern-Aware Graph Mining System[C]//Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys). 2020: 1-16.
- [74] Chen X, Dathathri R, Gill G, et al. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU[J]. Proceedings of the VLDB Endowment (VLDB), 2020, 13(8): 1190-1205.
- [75] Chen X, et al. Efficient and Scalable Graph Pattern Mining on GPUs[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2022: 857-877.
- [76] Chen J, Qian X. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2022: 47-61.
- [77] Chen J, Qian X. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2023: 413-426.
- [78] Hu L, Zou L, Özsu M T. GAMMA: A Graph Pattern Mining Framework for Large Graphs on GPU[C]//IEEE 39th International Conference on Data Engineering (ICDE). 2023: 273-286.
- [79] Shi T, Zhai J, Wang H, et al. GraphSet: High Performance Graph Mining through Equivalent Set Transformations[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). Denver, CO, USA, 2023.
- [80] Hu Y, Liu H, Huang H H. Tricore: Parallel Triangle Counting on GPUs[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2018: 171-182.
- [81] Alusaifeer T, Ramanna S, Henry C J, et al. GPU Implementation of MCE Approach to Finding Near Neighbourhoods[C]//International Conference on Rough Sets and Knowledge Technology. 2013: 251-262.
- [82] Lessley B, Perciano T, Mathai M, et al. Maximal Clique Enumeration with Data-Parallel Primitives[C]//2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV). 2017: 16-25.
- [83] Guo W, Li Y, Tan K L. Exploiting Reuse for GPU Subgraph Enumeration[J]. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2020, 34(9): 4231-4244.
- [84] Almasri M, Hajj I E, Nagi R, et al. Parallel k-Clique Counting on GPUs[C]//Proceedings of the 36th ACM International Conference on Supercomputing (ICS). 2022: 1-14.
- [85] Wei Y, Jiang P. STMatch: Accelerating Graph Pattern Matching on GPU with Stack-Based Loop Optimizations[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2022: 53:1-53:13.
- [86] Eppstein D, Löffler M, Strash D. Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time[C]//Proceedings of the International Symposium on Algorithms and Computation (ISAAC). 2010: 403-414.
- [87] Chang L. Efficient Maximum Clique Computation and Enumeration over Large Sparse Graphs[J]. The VLDB Journal (VLDBJ), 2020, 29(5): 999-1022.
- [88] Yao K, Chang L, Qin L. Computing Maximum Structural Balanced Cliques in Signed Graphs[C]//IEEE 38th International Conference on Data Engineering (ICDE). 2022: 1004-1016.
- [89] Kunegis J. Konect: The Koblenz Network Collection[C]//Proceedings of the 22nd International Conference on World Wide Web (WWW). 2013: 1343-1350.
- [90] Pagh R, Rodler F F. Cuckoo Hashing[J]. Journal of Algorithms, 2004, 51(2): 122-144.
- [91] MurmurHash[Z]. <https://en.wikipedia.org/wiki/MurmurHash>. 2024.
- [92] Green O, Yalamanchili P, Munguía L M. Fast Triangle Counting on the GPU[C]//Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms. 2014: 1-8.
- [93] Green O, Fox J, Watkins A, et al. Logarithmic Radix Binning and Vectorized Triangle Counting[C]//IEEE High Performance Extreme Computing Conference (HPEC). 2018: 1-7.
- [94] Fox J, Green O, Gabert K, et al. Fast and Adaptive List Intersections on the GPU[C]//IEEE High Performance Extreme Computing Conference (HPEC). 2018: 1-7.
- [95] Shi T, Zhai M, Xu Y, et al. Graphpi: High Performance Graph Pattern Matching Through Effective Redundancy Elimination[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2020: 1-14.

- [96] Gui C, Liao X, Zheng L, et al. Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow[C] //USENIX Annual Technical Conference (ATC). 2023: 71-85.
- [97] Chen X, Dathathri R, Gill G, et al. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining[C]//Proceedings of the ACM International Conference on Supercomputing (ICS). 2021: 378-391.
- [98] Intel. OneAPI Threading Building Blocks[Z]. <https://github.com/oneapi-src/oneTBB>. 2024.
- [99] NVIDIA A100 Tensor Core GPU[Z]. <https://www.nvidia.com/en-gb/data-center/a100/>. 2024.
- [100] CUDA from Wikipedia[Z]. <https://en.wikipedia.org/wiki/CUDA>. 2024.
- [101] CUDA C++ programming guide[Z]. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. 2024.
- [102] Single Instruction, Multiple Threads (SIMT) from Wikipedia[Z]. https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads. 2024.
- [103] Winter M, Parger M, Mlakar D, et al. Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks[C]//Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). 2021: 219-233.
- [104] NVIDIA Nsight Compute[Z]. <https://developer.nvidia.com/nsight-compute/>. 2024.
- [105] NVIDIA Nsight Systems[Z]. <https://developer.nvidia.com/nsight-systems/>. 2024.
- [106] CUDA Toolkit[Z]. <https://developer.nvidia.com/cuda-toolkit/>. 2024.
- [107] Gupta K, Stuart J A, Owens J D. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads[C]//Innovative Parallel Computing (InPar). 2012: 1-14.
- [108] Leskovec J, Krevl A. SNAP Datasets: Stanford Large Network Dataset Collection[Z]. <http://snap.stanford.edu/data>. 2014.
- [109] NVIDIA V100 Tensor Core GPU[Z]. <https://www.nvidia.com/en-gb/data-center/v100/>. 2024.
- [110] GeForce RTX 20 Series[Z]. <https://www.nvidia.com/en-gb/geforce/20-series/>. 2024.

作者简历

个人信息

姓名：潘哲

性别：男

出生日期：1996 年 1 月 7 日

籍贯：山西省阳泉市

教育经历

2014 年 9 月至 2018 年 6 月，就读于浙江大学计算机科学与技术专业，获得工学学士学位。

2018 年 9 月至 2021 年 3 月，就读于浙江大学计算机科学与技术专业，攻读硕士学位，导师为姜晓红副教授。期间于 2021 年 3 月转为博士研究生。

2021 年 3 月至 2024 年 6 月，就读于浙江大学计算机科学与技术专业，攻读博士学位，导师为何水兵研究员。

攻读博士学位期间取得的科研成果

已发表的学术论文

[1] **Zhe Pan**, Shuibing He, Xu Li, Xuechen Zhang, Yanlong Yin, Rui Wang, Lidan Shou, Mingli Song, Xian-He Sun, Gang Chen. “Enumeration of Billions of Maximal Bicliques in Bipartite Graphs without Using GPUs”. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2024. Accept. (对应第三章)

[2] **Zhe Pan**, Shuibing He, Xu Li, Xuechen Zhang, Rui Wang, Gang Chen. “Efficient Maximal Biclique Enumeration on GPUs”. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2023. No. 16, pp. 1-13. (对应第四章)

[3] **Zhe Pan**, Zonghua Gu, Xiaohong Jiang, Guoquan Zhu, De Ma. “A Modular Approximation Methodology for Efficient Fixed-Point Hardware Implementation of the Sigmoid Function”. IEEE Transactions on Industrial Electronics (TIE). 2022. Vol. 69, No. 10, pp. 10694-10703.

[4] **Zhe Pan**, Yuruo Jin, Xiaohong Jiang, Jian Wu. “An FPGA-Optimized Architecture of Real-time Farneback Optical Flow”. IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). 2020. pp. 223.

[5] **Zhe Pan**, Xiaohong Jiang, Jian Wu, Xiang Li. “Hybrid XML Parser Based on Software and Hardware Co-design”. IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). 2019. pp. 325.

在投的学术论文

[1] **Zhe Pan**, Xu Li, Shuibing He, Xuechen Zhang, Rui Wang, Yunjun Gao, Gang Chen, Xian-He Sun. “AMBEA: Aggressive Maximal Biclique Enumeration in Large Bipartite Graph Computing ”. IEEE Transactions on Computers (TC). 2024. Under Review (Major revision).

(对应第二章)

[2] **Zhe Pan**, Shuibing He, Xu Li, Xuechen Zhang, Rui Wang, Yanlong Yin, Gang Chen. “Advanced Maximal Biclique Enumeration on GPUs Using Bitmaps”. IEEE Transactions on Parallel and Distributed Systems (TPDS). 2024. Under Review. (对应第四章)

已授权的发明专利

[1] 姜晓红, **潘哲**, 吴健, 尹建伟, 邓水光, 李莹, 吴朝晖. 基于 FPGA 的 XML 解析器、可重构计算系统. 中国发明专利. 专利号: 202010061906.X. 授权公告日: 2023.07.04.

[2] 姜晓红, **潘哲**, 吴健. 基于 FPGA 的稠密光流计算系统及方法. 中国发明专利. 专利号: 201811600605.9. 授权公告日: 2023.05.23.

[3] 姜晓红, **潘哲**, 马德, 朱国权, 郝康利. 基于牛顿迭代法的非线性激活函数计算装置. 中国发明专利. 专利号: 202011090563.6. 授权公告日: 2022.06.21.

在申请的发明专利

[1] 何水兵, **潘哲**, 李旭. 一种基于候选顶点合并技术的极大二分团枚举方法. 中国发明专利. 专利号: 202311387848.X. 专利公开日: 2024.01.05. (对应第二章)

[2] 何水兵, **潘哲**, 李旭. 一种基于混合存储的极大二分团枚举方法. 中国发明专利. 专利号: 202311385166.5. 专利公开日: 2024.01.05. (对应第三章)

[3] 孙贤和, 李旭, 何水兵, **潘哲**, 陈刚. 一种 GPU 负载均衡的极大二分团枚举方法. 中国发明专利. 专利号: 202311563374.X. 专利公开日: 2024.02.06. (对应第四章)

论文答辩委员会对论文的评语和表决结果：

该论文研究了在大规模二分图场景下的极大二分图枚举问题，具有重要的研究意义和应用价值。论文主要贡献如下：

(1) 针对剪枝能力受限的问题，提出一种主动的极大二分图枚举算法 AMBEA，提升了对无效枚举节点的剪枝能力。

(2) 针对静态数据结构的低效性问题，提出一种自适应的极大二分图枚举算法 Adaptive AMBEA，扩展了二分图算法的枚举能力。

(3) 针对并行扩展性受限于 CPU 核心数量的问题，提出一种基于 GPU 的极大二分图枚举算法 GMBE，克服了在 GPU 实现过程中面临的内存短缺、线程分歧和负载不均等问题。

论文研究方法和技术路线正确，写作符合学术规范，结构完整，条理清晰，研究工作具有良好的创新性。答辩过程中，该生思路清晰，回答正确，表明该生已掌握扎实的学科和专业知识，具备独立从事科研工作的能力。

经答辩委员会投票表决，一致同意潘哲同学通过博士学位论文答辩，并同意授予该同学工学博士学位。

答辩委员会主席签字：丁伟华

2024 年 6 月 4 日