计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2015.20150256
52(8):1768-1783,2015

# FSMBUS:一种基于 Spark 的大规模频繁子图挖掘算法

严玉良[1] 董一鸿[1] 何贤芒[1] 汪 卫[2]
[1](宁波大学信息科学与工程学院 浙江宁波 315211)
[2](复旦大学计算机科学技术学院 上海 200433)
(yyl8781697@live.com)

# FSMBUS: A Frequent Subgraph Mining Algorithm in Single Large-Scale Graph Using Spark

Yan Yuliang[1], Dong Yihong[1], He Xianmang[1], and Wang Wei[2]
[1](*Faculty of Electrical Engineering and Computer Science*, *Ningbo Univeristy*, *Ningbo*, *Zhejiang* 315211)
[2](*School of Computer Science*, *Fudan University*, *Shanghai* 200433)

**Abstract** Mining frequent subgraphs in a single large-scale graph is of huge demand with the rapid growth of the social networking. However, it is inefficient for the serial algorithms to mine frequent subgraphs in low support when mining for a single large-scale graph. Meanwhile, few existing distributed algorithms can't support the growth pattern mining, and the Hadoop framework they worked is not suitable for iterative running. In this paper, a distributed algorithm named FSMBUS for mining frequent subgraph in a single large-scale graph under Spark framework is proposed. It constructs the parallel computing candidate subgraphs by suboptimal CAM Tree, which returns all the frequent subgraphs for given user-defined minimum support. Additionally, infrequent patterns' test and searching order chosen is introduced to optimize the algorithm. Sorted-Greedy method is designed for data partition to balance the workload. Our experiments show that FSMBUS runs faster and more effective than the existing algorithms with real datasets, and even can run with the lower support threshold and the larger graph datasets as well. At the same time, FSMBUS runs 2~4 times faster on Spark framework than that on Hadoop framework.

**Key words** frequent subgraph; single large-scale graph; distribute mining; Spark; workload balance

**摘 要** 随着社交网络用户数的快速增加,大规模单图上频繁子图挖掘的需求越来越强烈.单机算法对大规模图的运行效率较低,难以支撑支持度较低的频繁子图的挖掘;现有的分布式环境下单图的频繁子图挖掘算法不支持子图增长模式的挖掘,它们所使用的 Hadoop 框架也不适合运行迭代式算法.提出了一种基于 Spark 的大规模单图频繁子图挖掘算法 FSMBUS,通过次优树构建并行计算的候选子图,在给定最小支持度时挖掘出所有的频繁子图,并利用非频繁检测和搜索顺序选择实现优化,还设计了一种名为 Sorted-Greedy 的轻量级数据划分方法.实验结果表明,FSMBUS 的效率要比现有单图上最新的算法快一个数量级,并支持更低最小支持度阈值以及更大规模图数据的挖掘,同时 FSMBUS 比其 Hadoop 的移植版要快 2~4 倍.

**关键词** 频繁子图;大规模单图;分布式挖掘;Spark;负载均衡

**中图法分类号** TP301

**收稿日期**:2015-03-26;**修回日期**:2015-05-28
**基金项目**:国家自然科学基金项目(61170006,61202007);宁波市自然科学基金项目(2013A610063,2013A610110)
**通信作者**:董一鸿(dongyihong@nbu.edu.cn)

频繁子图挖掘是从图集或单图中找出支持度大于某个阈值的所有子图的过程,是图挖掘算法研究中一项重要的研究工作,广泛应用于生物信息学、化学信息学、社交网络、WWW 万维网等领域中. 例如,在化学领域中,从分子判断中可以找出频繁出现的子结构进行药物的发现[1];在社交网络中,帮助辨别不同群体内部的关系,有助于理解他们的社交行为;此外在对图数据进行频繁子图挖掘之后,其结果还可以直接用于图数据的分类[2]、聚类[3]、索引的建立[4]等.

目前频繁子图的研究主要集中在图集的研究,但是像社交网络和万维网链接图等单图数据不适合被划分为图集再进行操作,这类大规模单图数据的顶点规模往往在百万级别以上,远远超过图集中的单个图,挖掘出来的频繁模式子图的规模也要比图集上的大许多,因此在进行子图同构测试时需要更多的时间,而且单图上挖掘候选子图时支持度的计算要比图集更为复杂.

针对单图频繁子图的挖掘已有文献[5-7],2014 年提出的 GRAMI 算法[7]是单图上效果最好的单机频繁子图挖掘算法,它将频繁子图挖掘问题转为解决限制约束问题(constraint satisfaction problem, CSP)模型,可以高效地进行千万级别顶点的单图挖掘. 但是 GRAMI 算法是一个单机算法,对大规模图的运行效率较低,难以支撑支持度较低的频繁子图的挖掘. 而现有分布式环境下单图的频繁子图挖掘算法[5-6]均是针对有向图并且需要指定子图的顶点个数 $k$,不支持子图增长模式的挖掘,其他针对图集的分布式频繁子图挖掘算法[8-9]大多是基于 MapReduce 框架[10],迭代计算时需要多次读写磁盘,造成大量的 IO 和序列化、反序列化开销,也无法移植到单图上进行挖掘. 基于上述缺陷,本文提出了一种分布式的大规模单图的频繁子图挖掘(frequent subgraph mining based on BFS under Spark, FSMBUS)算法,该算法利用次优树按广度优先方向进行候选子图的生长,在构建候选搜索数据域的基础上以单个候选子图为颗粒进行并行计算,提出非频繁检测和搜索顺序选择 2 种优化方式,它们可以显著地提升算法效率,并且还设计了一种轻量级的数据划分方法:Sorted-Greedy,最后在 Spark[11]分布式环境下的实验结果显示,相同的支持度下 FSMBUS 比 GRAMI 算法的运行效率提高了一个数量级,并且还支持更低最小支持度阈值以及更大规模图数据的挖掘,同时,FSMBUS 在 Spark 上运行时的性能普遍高于 Hadoop[12]平台 2~4 倍.

本文的主要贡献如下:1)与传统的深度优先搜索策略不同,本文运用广度优先搜索生成候选子图的方法将大规模单图并行化,提出了基于 Spark 的频繁子图挖掘架构;2)提出了分布式的大规模单图的频繁子图挖掘算法 FSMBUS,运用贪心的方法实现负载均衡,提出非频繁检测和搜索顺序选择进行优化,显著提升算法的效率;3)对比实验显示,FSMBUS 的运行效率高于已有算法,并支持更低最小支持度阈值以及更大规模的单图数据的挖掘.

## 1 相关工作

频繁子图挖掘主要是围绕子图的同构测试和候选子图的生长来进行开展的. 1994 年,Holder 等人[13]最早提出了同时支持单图和图集的频繁子图挖掘的 SUBDUE 算法. Kuramochi 等人[14-15]在 2004 年提出了基于子图折叠的挖掘 Grew 算法,2005 年又提出了基于 MIS 支持度策略的 Sigram 算法,该算法需要枚举出所有的同构子图,计算 MIS 支持度时开销很大,所以只能处理稀疏图. 2009 年 Jiang 等人[16]提出的 G-Miner 算法采用 G-Measure 支持度测量方式,将输入图分成若干个子区域,G-Miner 对每个子区域中的互不共享边的同构实例图进行计数. 2014 年 Elseidy 等人[7]提出的 GRAMI 算法将频繁子图挖掘转为 CSP 模型,并且在整个过程中不需要对历史的频繁模式图进行实例存储,在 MNI 支持度计算阶段使用了向下剪枝、懒搜索和唯一标签 3 种优化方法,可以高效地进行千万级别顶点的单图挖掘. 上述均为单机实现的算法,它们对大规模图的运行效率较低,难以支撑支持度较低的频繁子图的挖掘. 2009 年 Liu 等人[5]提出的 MRPF 和 2015 年 Shahrivari 等人[6]提出的 MRSUB 均为基于 MapReduce 分布式挖掘算法,它们主要是在单个有向复杂网络中针对指定 $k$ 个顶点的子图进行挖掘,但是都不支持子图增长模式的挖掘.

在研究工作更为丰富的图集方面,早期较为经典的有 AGM[17], FSM[18], GSAPN[19], FFSM[20]. 2005 年汪卫等人[21]结合图论和频集生成提出了 AMGM 和 SFP 算法,可以对简单图中连通频繁子图进行有效的挖掘. 2007 年李先通等人[22]在挖掘频繁子树的基础上提出了一种高效的频繁子图挖掘 GraphGen 算法. 2013 年的 FSM-H[8]和 MRFSM[9]是基于迭代 MapReduce 的分布式挖掘算法,FSM-

H 将整个过程分为数据划分阶段、准备阶段、挖掘阶段三大步骤,通过挖掘阶段的不断迭代可以进行百万级别的图集挖掘,MRFSM 则更加注重了在迭代时对负载均衡的考虑,但是 MapReduce 并不适合于迭代的图挖掘算法,使用它会造成许多额外的磁盘读写和序列化开销. 2014 年 Lin 等人[23]提出的 MRFSM 同 FSM-H,MRFSM 一样也是一个基于 MapReduce 的分布式图集挖掘算法,它没有使用迭代,而是将整个过程分为 filter 和 refinement 两个阶段,filter 阶段根据支持度的概率计算进行剪枝后输出局部频繁的子图,refinement 阶段巧妙地将 filter 阶段中的局部支持度转为了全局支持度,与 FSM-H 相比性能上得到了较大的提升,但只能挖掘图集数据. 这些图集上的挖掘算法虽然无法直接应用于单图挖掘,但是它们对后来频繁子图研究工作影响巨大,例如 GSAPN 的深度搜索编码 DFSCode、FFSM 候选子图生长的次优树、迭代的 MapReduce 分布式挖掘等.

## 2 基本概念

### 2.1 频繁子图

**定义 1.** 子图和子图同构. 给定一个图 $G=(V,E,L)$,它包含了一个顶点集合 $V$,一个边的集合 $E$ 以及一个关于顶点和边的标签映射函数 $L$. 现在有一个图 $S=(V_S,E_S,L_S)$,当且仅当 $V_S\subseteq V$,$E_S\subseteq E$,$L_S(v)=L(v)$,对于每个 $v\in V_S\cup E_S$ 都成立时,则称 $S$ 为 $G$ 的子图. $S$ 与 $G$ 的子图同构即存在一个映射函数:$f:V_S\rightarrow V$,函数满足:1)是 $\forall v\in V_S$,$L_S(v)=L(f(v))$;2) $\forall(u,v)\in E_S$;$(f(u),f(v))\in E$ 且 $L_S(u,v)=L(f(u),f(v))$,这个单映射函数 $f$ 被称为 $S$ 在 $G$ 中的一个嵌入.

**定义 2.** 频繁子图. 给定一个图 $G$ 以及一个最小支持度 $\tau$,$Sup(G,S)$表示子图 $S$ 在 $G$ 中同构图的计数,当 $Sup(G,S)\geqslant\tau$ 时 $S$ 为 $G$ 的一个频繁子图.

在单图上进行频繁子图挖掘时支持度的选择非常重要. 为了有限时间内完成挖掘,支持度必须遵循反原子性原则,即 $g$ 为 $p$ 的一个子图,在给定输入图 $G$ 中则必定满足 $Sup(G,g)\geqslant Sup(G,p)$. 支持度计算策略同 GRAMI 算法[7]一样采用 MNI 支持度[24]的计算方式,因为其他方式是一个 NP 完全问题,需要使用其对应的替代方案计算,产生不必要的昂贵代价[7].

**定义 3.** MNI 支持度. $f$ 为子图 $S=(V_S,E_S,L_S)$在 $G$ 中的同构映射集合,$f$ 集合的长度为 $m$,同时 $F(v)=\{f_1(v),f_2(v),\cdots,f_m(v)\}$为 $f$ 集合上每一个 $v$ 去重之后的映射函数,其中 $v\in V_S$,则其 MNI 支持度计算方式表示为:$Sup(G,S)=\min\{t\mid t=|F(v)|,v\in V_S\}$.

图 1(a)是合作网络图,图 1(b)(c)(d)均为图 1(a)的子图,图 1(b)在图 1(a)中的同构映射集合为 $\{u_0,u_1,v_3\}$,$\{u_7,u_6,u_5\}$,$\{u_7,u_6,u_8\}$共 3 个,通过 $F(v)$ 函数去重之后得到结果:$F(v_1)=\{u_0,u_7\}$,$F(v_2)=\{u_1,u_6\}$,$F(v_3)=\{v_3,v_5,v_8\}$,支持度为 2,观察图 1(b)的支持度,共有 2 个同构映射:$\{u_{13},u_{12},u_{10}\}$,$\{u_{11},u_{10},u_{12}\}$,在用 $F(v)$ 进行去重映射之后得到:$F(v_1)=\{u_{13},u_{11}\}$,$F(v_2)=\{u_{12},u_{10}\}$,$F(v_3)=\{u_{10},u_{12}\}$,因此最终的支持度为 2.

在 GRAMI 中首次提出了利用 CSP 模型来进行子图同构判定,通过搜索可以查找出所有的同构

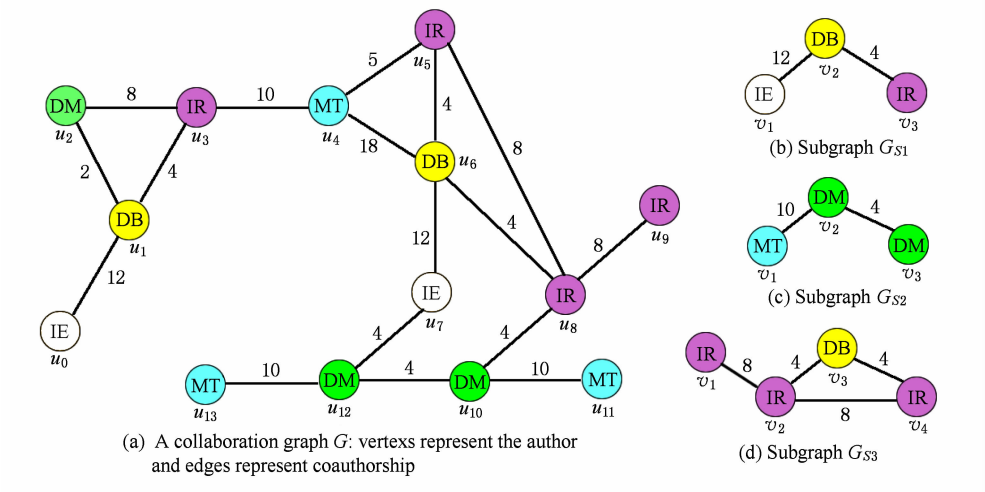



Fig. 1 The example of collaboration graph.

图 1 合作网络图样例

子图,该方法比最快的子图同构判定算法 GraphQL[25] 至少快 3 倍[7].

**定义 4.** CSP 模型. $S=(V_S,E_S,L_S)$ 为 $G=(V,E,L)$ 的一个子图,在 $G$ 中查找子图 $S$ 的同构用 $CSP(X,D,C)$ 模型来表示:

1) $X$ 所包含的变量 $x_i$ 表示 $V_S$ 中的各个顶点 $v$;

2) $D$ 是各个 $x_i\in X$ 数据域的集合,每个数据域都是 $V$ 的一个子集;

3) $C$ 表示有如下约束条件:

① $x_{i1}\neq x_{i2}$,表示同一个同构映射中同一个顶点最多只能被分配一次(顶点不重复性);

② $L(x_i)=L_S(v)$,表示分配的顶点的标签必须与子图的标签相对应(顶点一致性);

③ $L(x_{i1},x_{i2})=L_S(v_1,v_2)$,表示分配的顶点相互之间边的标签必须与子图的边标签对应(弧一致性).

**定理 1**[7]. 在子图 $S$ 对应 $G$ 的 CSP 模型中任何一个有效分配即为 $S$ 在 $G$ 中的一个子图同构.

有效分配是指在 CSP 模型中满足 $C$ 的约束,如图 1(b)在图 1(a)中的一个有效分配为 $(v_1,v_2,v_3)=\{u_0,u_1,u_3\}$,因为 $u_0,u_1,u_3$ 这 3 个顶点 $ID$ 互不相同,3 个顶点的标签(IE,DB,IR)与边的标签(12,4)也正好与图 1(b)相对应. 而图 1(a)中 $u_0,u_1,u_3$ 这 3 个顶点组成的的图恰好为图 1(b)的同构.

由定义 4 和定理 1 可知,频繁子图挖掘可以使用 CSP 模型根据候选子图的拓扑结构约束在真实图中找同构映射. 将图 1(b)在图 1(a)中进行频繁子图查找转为 CSP 模型为

$$\{X:(v_1,v_2,v_3),$$

$$D:\{\{u_0,..u_{13}\},\{u_0,..u_{13}\},\{u_0,..u_{13}\}\}$$

$$C:\{v_1\neq v_2\neq v_3,L(v_1)=\mathrm{IE},L(v_2)=\mathrm{DB},$$

$$L(v_3)=\mathrm{IR},L(v_1,v_2)=12,L(v_2,v_3)=4\}.$$

根据约束条件在数据域中找到的有效分配为 $\{u_0,u_1,u_3\},\{u_7,u_6,u_5\},\{u_7,u_6,u_8\}$,即为同构映射,然后计算 MNI 支持度来判断该候选子图是否为频繁子图.

## 2.2 候选子图的生长——次优树(suboptimal CAM tree)

次优树是 FFSM 算法[20] 提出的一种高效的候选子图生长策略,每个图使用邻接矩阵 $\boldsymbol{M}$ 存储,其中矩阵的对角线上存储顶点的标签,非对角线存储顶点与顶点相连的边的标签,0 表示两个顶点之间没有边相连.

**定义 5.** 标准邻接矩阵 CAM. 给定一个 $n\times n$ 的邻接矩阵 $\boldsymbol{M}$ 表示有 $n$ 个顶点的图 $G$,定义 $\boldsymbol{M}$ 的编码为该矩阵的下三角(含对角线)按顺序 $m_{1,1}m_{2,1}m_{2,2}\cdots m_{n,1}m_{n,2}\cdots m_{n,n-1}m_{n,n}$ 构成的序列,记作 $code(\boldsymbol{M})$,其中按字典序比较得到的 $\max(code(\boldsymbol{M}))$ 称为标准编码,其对应的表示矩阵称为标准邻接矩阵(canonical adjacency matrix, CAM).

**定义 6.** 次优树. 当图 $G$ 的矩阵 $\boldsymbol{M}$ 的极大子矩阵是另一个图的标准邻接矩阵时,该矩阵 $\boldsymbol{M}$ 称为次优邻接矩阵,记作次优 CAM,这里每一个 CAM 同

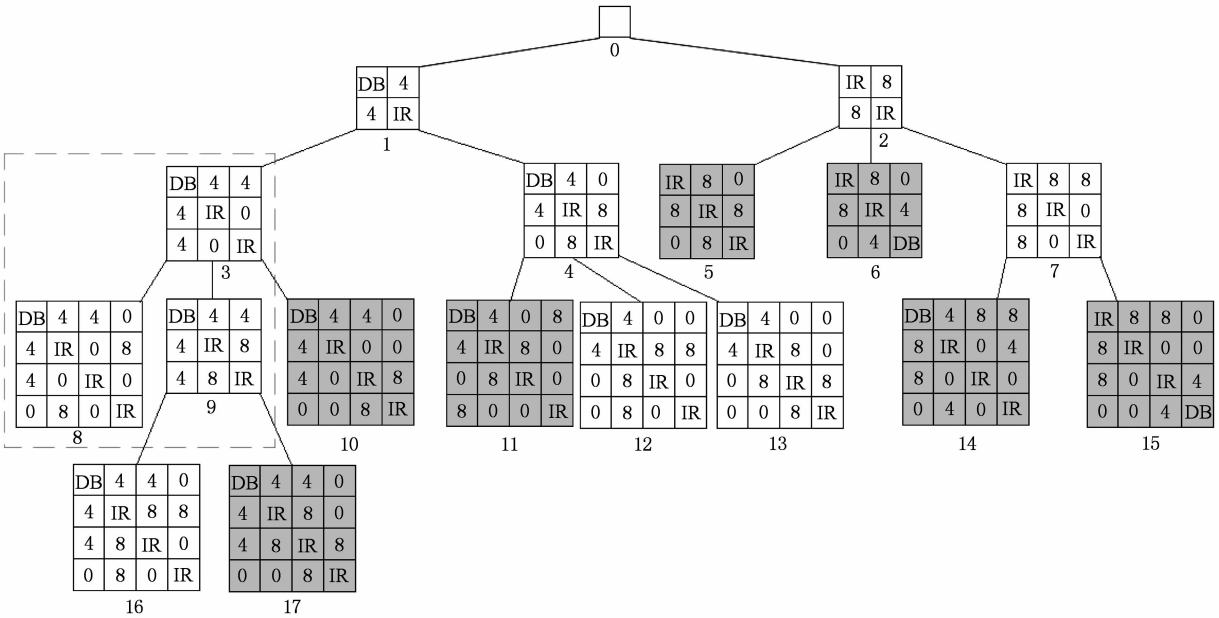



Fig. 2 Suboptimal CAM tree (DB greater than IR in dictionary order).

图 2 次优树(DB 的字典序大于 IR 的字典序)

时也是次优 CAM,由次优 CAM 构成的树叫作次优树(suboptimal CAM tree).

给定 2 个次优邻接矩阵 $\boldsymbol{M}$ 和 $\boldsymbol{N}$,通过 $\boldsymbol{M}$ 与 $\boldsymbol{N}$ 的连接操作可以生成新矩阵称为 FFSM-JOIN[20],$\boldsymbol{M}$ 或 $\boldsymbol{N}$ 扩展频繁边生成新矩阵称为 FFSM-Extension[20],现以空为根节点、频繁边为第 1 层叶子,频繁边可以通过这两大算子生成次优邻接矩阵作为第 2 层叶子节点,第 2 层叶子节点同样可以通过这两大算子生成第 3 层叶子,不断迭代此操作就可以生成次优树,即枚举出不重复的候选子图.

图 2 为图 1(d)生长过程中形成的次优树,树的根节点为空,其他节点都代表图 1(d)的一个子图,其中白色背景的为 CAM,灰色背景的为次优 CAM,每一个节点都由它的父节点通过 FFSM-JOIN 或者 FFSM-Extension 算子而生成.

## 3 FSMBUS

### 3.1 内存分布式计算框架——Spark

Spark[11]是伯克利大学在 2012 年提出的一个基于内存的分布式计算框架,其核心是弹性分布式数据集(resilient distributed datasets, RDD),它是对集群上并行处理数据的分布式内存的抽象,RDD 上主要有 Transformation 和 Action 两大类型的操作,并且当前 RDD 与上一个 RDD 会有依赖关系,Spark 的每个作业会将各个 RDD 串联起来形成有向无环图(directed acyclic graph, DAG),从而可以更加高效地执行,也因此 Spark 特别擅长运行迭代类型的算法,在处理迭代式作业时的速度超过 Hadoop[12]大约 20 倍. 其子项目 Graphx[26]是被认为 Pregel 和 GAS 模型的改进优化,其性能更是超过 GraphLab[27]和 Giraph[28],Graphx 提供子图查询、节点度的计算、邻居的收集等基本的图论算法操作,还可以使用 SparkCore 中的各种 Api 接口,因此在进行图计算框架选择时从运行效率和编程模型的角度来看 Spark 都是一个比较理想的选择.

### 3.2 频繁子图挖掘算法-FSMBUS

FSMBUS 算法是基于迭代的 RDD 设计的,以候选子图的边的数量作为 2 次迭代的增量,使用次优树作为候选子图的生长策略,在图 2 的次优树中发现:树的每一层的子图含有相同数量的边,在支持度的计算上互不影响,当前候选子图是否频繁将会直接影响孩子节点.

FSMBUS 的流程框架如图 3 所示,包括数据准备阶段和挖掘阶段. 挖掘阶段是算法的核心部分,将次优树按广度搜索进行生长,并行计算每一层中 CAM 所表示的候选子图是否频繁. 剪枝非频繁的候选子图,继续生长次优树,进入下一次迭代. 第 $i$ 次迭代生成边数为 $i+2$ 的频繁子图并且作为第 $i+1$ 次迭代的输入,直到候选子图全部为非频繁时停止. 算法主要解决了候选搜索数据域的构建以及支持度计算这两大难点.

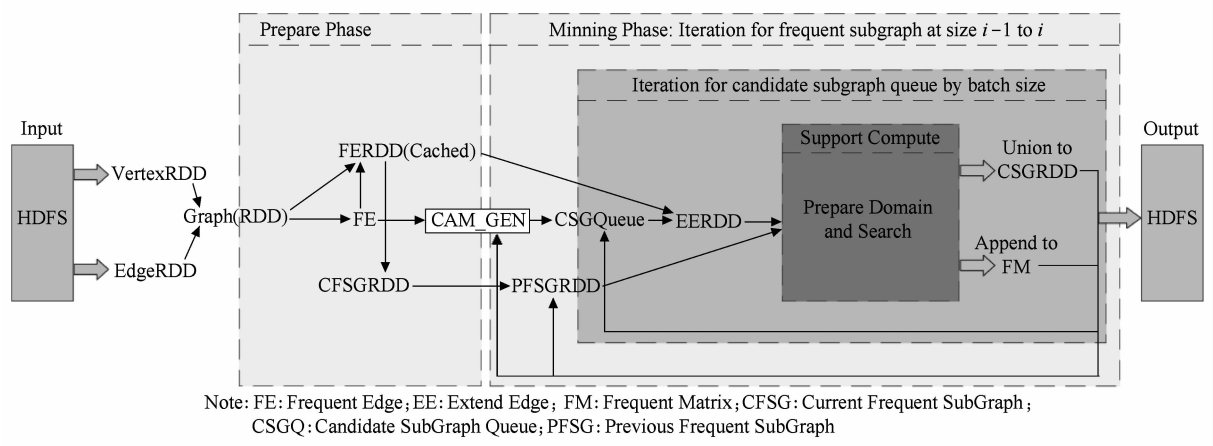



Fig. 3 The framework of FSMBUS algorithm.

图 3 FSMBUS 算法的流程框架图

#### 3.2.1 数据准备阶段

数据准备阶段主要完成频繁边的收集以及频繁子图数据集初始化的工作.

得到输入图之后,根据 MNI 支持度策略过滤非频繁边,将计算得到的频繁边按三元组(起始标签 *srcLabel*,边标签 *attr*,目标标签 *dstLabel*)的格式进行存储,这些频繁边即为次优树中的第 1 层的频繁子图,同时使用频繁边 RDD 存储频繁边 2 端所对应

的顶点映射,这些顶点映射为 CSP 模型中的有效分配,频繁边 RDD 在下文中将作为扩展边的缓存.

在图 3 的挖掘阶段中,每次迭代都需要父级频繁子图的 RDD. 为了能让迭代顺利地进行,在准备阶段需要初始化一个当前频繁子图 RDD 作为挖掘阶段的初始输入,在准备阶段中当前频繁子图数据集即为频繁边,由频繁边 RDD 转换而来,与之区别的是繁子图 RDD 以 Domain(CSP 模型中的 $D$)的方式进行存储,即它为一个数组,$D_i$(表示 $X$ 中索引为 $i$ 的数据域,下文均按此缩写表示)中存储对应变量上有效分配顶点,并且 $D_i$ 的各个顶点还会存储候选子图上相邻顶点的索引位置及其 $ID$. Domain 的结构如图 4(a)所示,它是一个含有 $k$ 个顶点的候选子图所表示的结构,$v_x$ 表示 $D_1$ 中的有效分配顶点,$\{k-2\rightarrow\{v_y\}\}$表示 $D_1$ 的 $v_x$ 与 $D_{k-2}$ 的 $v_y$ 相连,由于本算法是应用于无向图的挖掘,所以在 $D_{k-2}$ 上会对称的存在与 $D_1$ 相连的信息. 将 Domain 的结构对应图 4(b),可以得到 $u_1$ 和 $u_3$,$u_6$ 和 $u_5$,$u_6$ 和 $u_8$ 分别都有边相连,这种存储结构的优势是可以直接使用它来进行候选子图的支持度计算.

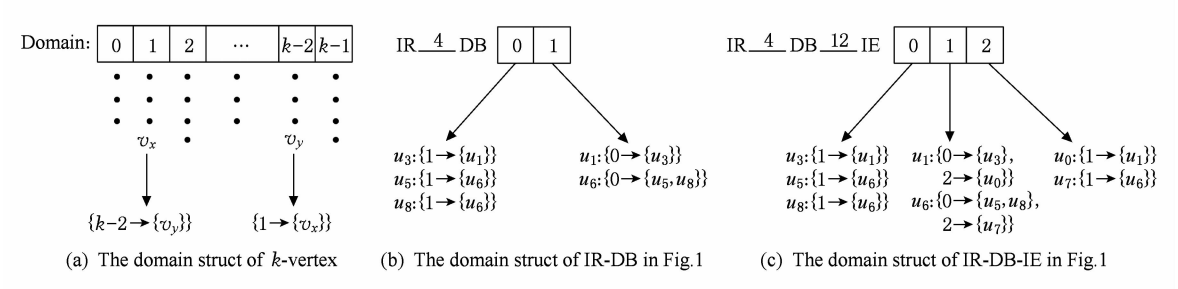



Fig. 4 The struct of domain.

图 4 Domain 存储结构

### 3.2.2 挖掘阶段

挖掘阶段通过迭代的方式将所有频繁子图计算出来,包括候选子图的生成、构建候选搜索数据域以及支持度的计算.

步骤 1. 候选子图的生成:第 $i$ 次迭代由第 $i-1$ 次迭代产生的频繁子图进行 FFSM-Join 和 FFSM-Extend 后产生候选子图,由于 FFSM-Join 和 FFSM-Extend 算子生成新的候选子图都是相当于在父级子图上添加了一条边,称为扩展边. 新产生的候选子图的数据结构除了自身的拓扑结构之后,还包含其父级子图的 $ID$ 号、该子图的扩展边,这 2 个数据可帮助该候选子图进行候选搜索数据域的构建.

**定义 7.** 外/内边. 在子图生长过程中,如果这条扩展边引入了一个新顶点,则称这条扩展边 E 为外边,否则为内边.

在图 5 中,$ID=8$ 的候选子图由 $ID=3$ 的图在索引为 1 的顶点上(这里索引从 0 开始计数)添加了一条外边而生成,其外边的目标点标签为 IR,外边的边标签为 8. $ID=9$ 的候选子图由 $ID=3$ 的图在索引为 1 的顶点上添加了一条内边而生成,其内边另一个顶点的索引为 2,内边的边标签为 8.

步骤 2. 构建候选搜索数据域:计算支持度之前需要取得当前候选子图对应顶点数据,称为候选搜索数据域(candidate domain, $CD$),使用 Domain 结构存储. FSMBUS 使用增量的方式来获取对应的 $CD$,每个候选子图都有父级子图 $ID$ 信息以及其扩展边信息,当前候选子图的 $CD$ 即为父级子图的有效分配数据连接扩展边对应的有效分配数据. 在图 4(c)中,IR-DB 的有效分配数据域为 $D_{\text{IR-DB}}=\{\{u_3,u_5,u_8\},\{u_1,u_6\}\}$,IR-DB-IE 是由 IR-DB 扩展了一条外边((1,2),(DB,12,IE))而来,其外边对应的有效分配数据为 $D_{\text{DB-IE}}=\{\{u_1,u_6\},\{u_0,u_7\}\}$,将 $D_{\text{DB-IE}}$ 连接到 $D_{\text{IR-DB}}$的索引位置为 1 上之后即可得到 IR-DB-IE 的 $CD=\{\{u_3,u_5,u_8\},\{u_1,u_6\},\{u_0,u_7\}\}$. 算法使用 2 个 Join 接口来完成上述操作,第 1 个 Join 使

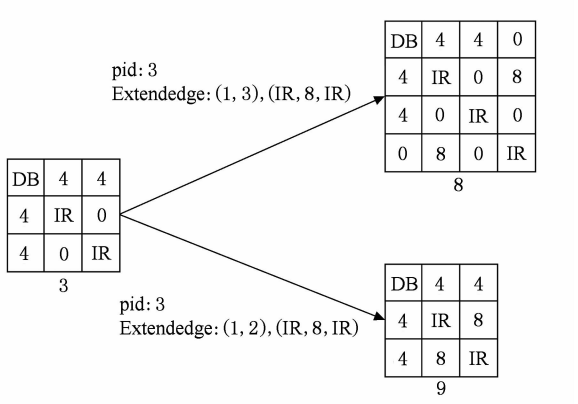



Fig. 5 The example of extend edge when growth the subgraph.

图 5 子图生长扩展边样例(与图 2 中的虚线部分对应)

用三元组边作为键($key$),与频繁边 RDD 进行 Join 之后即可得到扩展边的数据集扩展边 RDD,扩展边 RDD 以父级子图 $ID$ 作为 $key$ 即可与父级频繁子图 RDD 进行 Join 操作,这里的 Join 操作仅仅是将扩展边数据域与父子图的有效分配数据域存储在同一个 RDD 中,此时需要按上面 $D_{\text{IR-DB}}$ 连接 $D_{\text{DB-IE}}$ 的步骤形成最终的 $CD$.

步骤 3. 支持度计算:支持度计算是频繁子图挖掘的核心,需要进行子图的同构测试,而子图同构测试是 NP 完全问题,FSMBUS 通过对候选搜索数据域进行搜索计算支持度,采用启发式方法进行搜索,其伪代码如下:

**算法 1.** 支持度计算 $IsFrequent(D,\tau)$.

输入:$D$ 表示当前候选子图的候选搜索数据域,$\tau$ 表示最小支持度;

输出:如果该候选子图的支持度大于 $\tau$,返回 ture,否则返回 false.

① val $solDomain$, $nonCandidateMap$; /* 分别存储有效分配实例和无效分配项 */
② val $searchIndexOrder=getOrder(D)$; /* 获取搜索顺序 */
③ for each $i$ in $searchIndexOrder$
④ if $CheckFrequent(solDomain)$ return true;
⑤ for each $v$ in $D(i)$
⑥ if $v$ 包含在 $solDomain(i)$中
go to next $v$;
⑦ $instance=SearchBackTracking(v)$;
/* 使用回朔进行有效分配搜索 */
⑧ if $instance$=empty
⑨ $nonCandidateMap(i).add(v)$;
⑩ if $D(i).size\text{-}nonCandidateMap(i).size<\tau$ return false;
⑪ else
⑫ $solDomain.add(instance)$
⑬ if $solDomain(i).size\geqslant\tau$ go to next $i$;
⑭ if $solDomain(i).size\leqslant\tau$ return false;
⑮ if $checkFrequent(solDomain)$ return true;
⑯ return true.

支持度计算时逐个遍历变量 $X$ 的数据域,构建深度优先搜索路径进行有效分配的搜索,直至 MNI 支持度大于 $\tau$ 或者当前变量的有效分配与剩余候选项个数之和小于 $\tau$ 时停止. 在算法 1 中,行③~⑤遍历数据域,行⑦搜索有效分配,行⑧~⑮是频繁/非频繁条件判断的剪枝. 计算过程完成后,如果得到的结果是频繁子图,整个搜索数据域将会根据 $nonCandidateMap$ 进行剪枝,以减少生成候选子图的搜索空间,同时减少 RDD 的存储空间.

在图 6(a)中,$G_1$ 为输入图,$S_1$ 为候选图,图 6(a)的右侧为计算支持度时的迭代过程($\tau=2$),其中第 1,2 次迭代是以 $v_1$ 为起始变量,虽然 2 次迭代都搜索到了有效实例,但是在有效分配中存在重复的分配项,所计算的支持度仍小于 2,由于此时 $v_1$ 中的候选项全部搜索完毕,下一次迭代以 $v_2$ 为起始变量,在第 3 次迭代搜索到有效实例($u_2,u_3,u_1,u_4$)之后,计算 $solDomain$ 的支持度为 2,完成了 $S_1$ 在 $G_1$ 中的支持度计算.

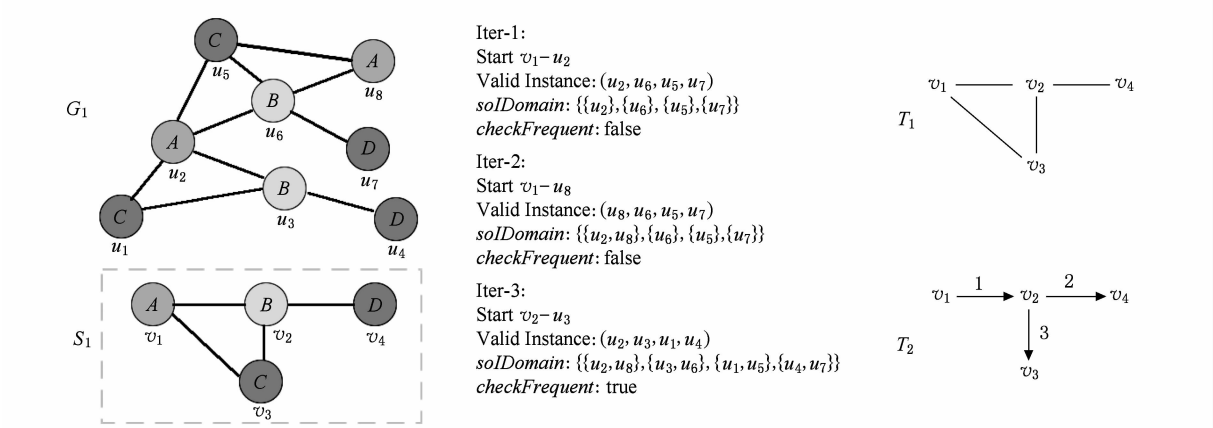



Fig. 6 The example of search.

图 6 搜索样例图

支持度计算的核心是有效分配搜索,候选项的有效性是根据是否满足 CSP 模型的约束进行判断:顶点不重复性以及弧一致性(顶点一致性的约束在构建 $CD$ 时已经进行了强制满足). 以当前搜索变量

为起点,形成深度搜索路径,该路径上的每个点会记录当前点对应的变量索引以及其父变量的索引,用于获取下一个变量的候选项,上下 2 个变量将会满足弧一致性的约束.在图 6(b)中每个数字代表每个顶点索引,$T_1$ 为候选子图的拓扑结构,以变量 $v_1$ 为起始点,形成的深度搜索路径如 $T_2$ 所示为 $(v_1,-1)\rightarrow(v_2,0)\rightarrow(v_4,1)\rightarrow(v_3,1)$,在确定 $v_1$ 有效项 $u_2$ 的情况下在 $v_2$ 中继续进行有效实例的搜索,$v_2$ 中的搜索域是根据 $v_1-u_2$ 在 Domain 中的相邻点得到的,所以 $v_2$ 会与 $v_1$ 自动满足弧一致性.如果约束边在搜索路径中存在,不需要额外进行弧一致性的检测,被搜索时会像 $v_2-v_1$ 一样自动满足弧一致性检测;当搜索到 $v_3$ 发现该变量与 $v_1$ 有相连边,其边 $v_3-v_1$ 不在搜索路径中,此时才需要额外检测弧的一致性约束.如果所有点都分配到了有效的项,说明找到了有效分配实例,返回该实例;否则 $u_2$ 为无效项,返回 empty.

### 3.3 搜索优化

支持度计算时的有效分配搜索是一个 NP 完全问题,将消耗整个算法的大部分时间.这里提出 2 个优化方法:非频繁检测和搜索顺序的选择,前者在搜索前提前判断非频繁图,后者是搜索过程中对候选子图提前剪枝.

**剪枝策略 1.** 非频繁检测.根据 MNI 支持度的定义,如果数据域中存在候选项数量小于 $\tau$ 时,则该候选子图必定为非频繁子图.

**定义 8.** 局部中心区域.$v$ 为候选子图中的任意一个顶点,$v_i$ 为顶点 $v$ 的相邻顶点集合,这里将当前顶点 $v$ 与其相邻顶点 $v_i$ 组成的区域称为局部中心区域,用 $\Gamma(v)$ 来表示.如图 7 中 $\Gamma(v_2)=(v_2,\{v_1,v_3\})$ 即为一个局部中心区域.

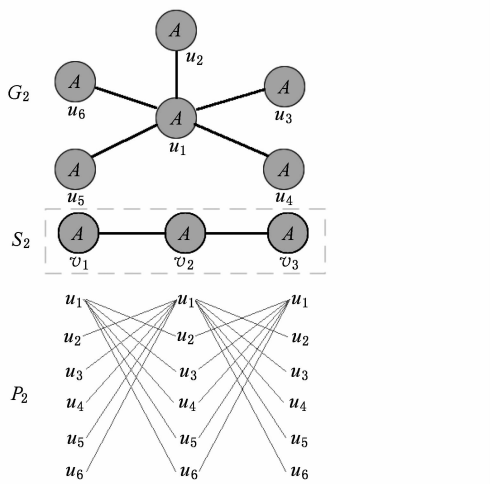



Fig. 7 The example of optimize search.

图 7 搜索优化样例

**定理 2.** 在一个候选搜索数据域的局部中心区域 $\Gamma(v)$ 中,如果其相邻顶点 $v_i$ 不满足顶点不重复性约束,则该 $\Gamma(v)$ 也无法满足顶点不重复性的约束,此时有效分配搜索时搜索路径如果经过该 $\Gamma(v)$ 的中心顶点 $v$,该搜索路径上的分配一定无法满足候选子图的约束.

证明. 搜索数据域是根据扩展边构建,每条边上的 2 个点满足顶点不重复性要求的(一条边上的 2 个顶点肯定是不重复的),因此在 $\Gamma(v)$ 的中心顶点与它任意相邻顶点都是满足顶点不重复性约束,在检测 $\Gamma(v)$ 上是否满足顶点不重复性约束时只需要检测相邻顶点.到了有效分配搜索阶段,在搜索时假如搜索路径上含有无效 $\Gamma(v)$ 的中心顶点时,继续搜索该路径时一定会经过该 $\Gamma(v)$ 的相邻顶点,此时当前搜索路径上的分配将无法满足顶点不重复性的约束. 证毕.

根据定理 2,在进行有效分配搜索之前,当 $\Gamma(v)$ 中心顶点的度大于等于 2 时,算法对顶点进行不重复性的约束检测,如果 $\Gamma(v)$ 不满足该约束,将其 $\Gamma(v)$ 的中心顶点移除.检测结束,如果当前变量的候选项数量小于 $\tau$,则该候选子图一定不频繁,不再搜索.

例如在图 7 中,$G_2$ 为输入图,$S_2$ 为候选子图($\tau=2$),$P_2$ 为 $S_2$ 的候选搜索域,如果对其 $P_2$ 直接进行搜索,最终将会得到 $solDomain=\{\{2,3,4,5,6\},\{1\},\{2,3,4,5,6\}\}$,其 MNI 支持度为 1,可以判定 $S_2$ 为非频繁子图.假如进行非频繁检测,将会对 $(v_2,\{v_1,v_3\})$ 中的结构进行检测,结果可以发现 $(2,\{1,1\})$,$(3,\{1,1\})$,$(4,\{1,1\})$,$(5,\{1,1\})$,$(6,\{1,1\})$ 均为无效 $\Gamma(v)$,所以 $D_1$ 中 2,3,4,5,6 这些 $ID$ 的顶点被移除,此时 $D_1$ 只有一个 $ID$ 为 1 的候选项,则可以直接判断 $S_2$ 肯定为非频繁子图.

**剪枝策略 2.** 搜索顺序的选择.在有效分配搜索时,需要对深度搜索路径上的候选项进行全部遍历,大量的时间花在无效项的搜索上.在算法 1 的第⑨⑩行中,每次进行有效分配搜索时如果结果为 empty,就将起始项作为无效项进行记录,如果 $D$ 中对应变量里的有效分配与剩余的候选项个数之和小于 $\tau$,则该候选子图为非频繁子图,提前结束搜索.因此在搜索顺序上进行优化,按照 $D$ 中候选项个数的变量升序排序,从而提前判定候选子图为非频繁子图.

### 3.4 负载均衡

分布式算法的耗时符合木桶效应,FSMBUS 的运算时间取决于运算最久的那台 Worker. 构建候选搜索域时 RDD 之间根据父频繁子图 *ID* 进行 Join,在 Join 之后 Spark 是针对候选子图的父频繁子图 *ID* 通过默认的 HashPartitioner 进行划分,但是每个父频繁子图所拥有的候选搜索数据域大小不同,在计算支持度时所需的计算力也差异较大,此时使用默认的数据划分可能会造成数据倾斜,从而增加算法的运算时间. 本文运用贪心的方法 Sorted-Geedy 进行数据划分.

在图 8 中,Join 之后的数据分布如图 8(a)所示,$m_i$ 表示父频繁子图的 *ID* 号,上方的数字表示该父频繁子图所含有的数据域大小 $v_i$,现在将这些数据划分到 3 个数据分区中 $p_i$,假如按照 HashPartitioner 进行划分将会得到图 8(b)中的结果,此时会发现分区 $p_1$ 与分区 $p_3$ 中的数据存在着较大的不均衡,最终算法整体的运行耗时将会与 $p_3$ 相同,而本文的 Sorted-Geedy 划分方法期望是达到图 8(c)中的效果,分区 $p_1$,$p_2$,$p_3$ 中所含的数据大小相等或者尽量相近,这样图 8(c)相对图 8(b)的划分将会大大利用集群的资源,加快算法的运算速度.

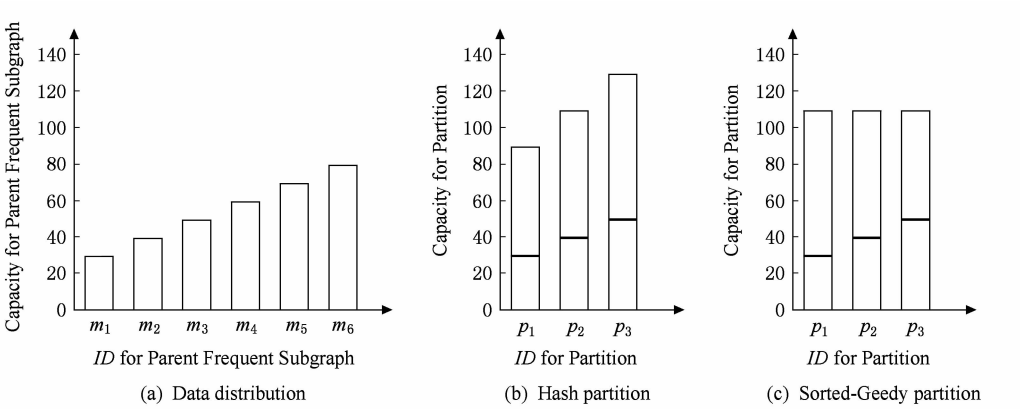



Fig. 8 Data partition.

图 8 数据划分图

为了达到图 8(c)中的划分效果,Sorted-Geedy 划分设计如下:

1) 将待计算的候选子图按照候选数据域大小进行降序排序,同时将分区中的空间置 0;

2) 逐个取候选子图,将其数据域放入已存储数据域空间最小的分区中;

3) 不断进行第 2 步的迭代,直至将所有的候选子图遍历.

FSMBUS 算法的伪代码:

**算法 2.** FSMBUS($v\_path$,$e\_path$,$\tau$).

输入:$v\_path$,$e\_path$ 分别表示输入图顶点和边的文件存储路径,$\tau$ 表示最小支持度;

输出:$S$ 为返回的频繁子图的集合.

```
① val G=Graph(v_path,e_path);
② val FE=G.groupby.filter.map.filter.filter.collect; /*检测输入图中的频繁边*/
③ val FERDD=(G,FE).filter.flatMap.groupby.filter.map.cache; /*获取频繁边的信息*/
④ val CFSGRDD=FERDD.map.cache;
   /*初始化频繁边数据集*/
⑤ val matrixMap=Cam_Init(ConnectSplit(FE)); /*将频繁边分割并转为矩阵*/
⑥ while matrixMap.size>0
⑦    S.union(matrixMap.matrix); /*添加频繁子图*/
⑧    matrixMap=Cam_Gener(matrixMap); /*使用 CamCode 进行连接和扩展*/
⑨    val candidateSubGQueue=matrixMap.enqueue; /*形成候选子图队列*/
⑩    val PFSGRDD=CFSGRDD;
⑪    while candidateSubGQueue.size>0
⑫       val batchFreqSubGRDD=sc.parallelize(candidateSubGQueue.dequeue(batchSize))
⑬       .join(FERDD).map.join(PFSGRDD).map{
⑭          D=Domain_Construct(); /*候选子图候选数据域构建*/
⑮          InFrequent_detect(); /*非频繁检测*/
```

⑯ *IsFrequent*($D$,$\tau$);} /* 支持度计算(算法 1) */

⑰ if *batchFreqSubGRDD*.*count*>0

⑱ *matrixMap*. *add*(*batchFreqSubGRDD*.*matrix*);

⑲ CFSGRDD. *union*(*batchFreqSubGRDD*);

⑳ return $S$.

行①~⑤数据准备阶段,行⑥~⑲为最核心的挖掘部分,非频繁检测优化在行⑮,搜索顺序选择在算法 1 中的行②.在行⑨~⑫可以了解 FSMBUS 并不是将所有的候选子图同时进行并行计算,而是先将候选子图压入队列,再批量从队列中取出进行计算,这是因为存储候选子图的 Domain 数据需要消耗不少的内存空间,如果同时将所有候选子图的 Domain 数据进行存储很容易造成集群内存溢出,所以 FSMBUS 以队列的形式进行批处理操作,通过控制批处理参数 *BatchSize* 可以有效地避免内存溢出的异常,这也说明了 FSMBUS 拥有良好的鲁棒性.

### 3.5 复杂度分析

现设 $N$ 为输入图的顶点数量,$n$ 为候选子图的顶点数量,在搜索时候选项是有效分配的概率为 $\eta$,$\theta$ 为不满足顶点不重复性约束的概率,$\tau$ 为频繁子图的最小支持度阈值.次优树算法通过 2 个候选子图之间进行连接或者单个候选子图与频繁边进行扩展来生成新子图的,所以候选子图的生成需要 $O((N/\tau)^2)$.本文采用二分图的最大匹配来进行非频繁检测,计算最大匹配时使用了匈利亚算法,故非频繁检测一共消耗 $O((n-2)\times N^3)$,在有效分配搜索时,每列都需要进行 $\tau/\eta$ 次搜索,由于在非频繁检测优化时会将不满足顶点不重复的候选项移除,支持度计算所需的复杂度为 $O(n\times\tau/\eta\times((1-\theta)\times N)^{n-1})$.最终 FSMBUS 在挖掘阶段每个候选子图的复杂度上界为 $O((n-2)\times N^3+n\times\tau/\eta\times((1-\theta)\times N)^{n-1})$,而 GRAMI 算法的复杂度是 $O(n\times\tau/\eta\times N^{n-1})$,当候选子图的顶点数量 $n>4$ 时,非频繁检测用的时间对于有效分配搜索是微不足道的,同时 FSMBUS 算法的复杂度要小于 GRAMI.

## 4 实验结果与分析

实验用到普通 PC 机和工作站,机器配置方案如表 1 所示.FSMBUS 使用 Scala2.10.4 进行开发,集群环境为 Spark1.2.0,运行的模式为 Spark On Yarn,其中 Yarn 对应的 Hadoop 版本为 2.2.0,所有节点均使用普通 PC 机,运行时的通用参数 driver-memory 为 6 GB,executor-memory 为 6 GB,使用 executors 的数量和并行度的值 parallelism 会根据不同的实验有不同的设置,在具体的实验中会有说明.

**Table 1 The Configuration of Machines Used by Experiment**

**表 1 实验所用的机器配置方案**

| No. | Type | Number | Operation System | CPU Frenquence/GHz | Memory Size/GB | JRE | Disk Size/TB | Total Memory Size/GB |
|---|---|---|---|---|---|---|---|---|
| 1 | PC | 1 | Centos 6.5 | 4 Core 2.4 | 16 | v1.7.0 | 1 | 16 |
| 2 | PC | 2 | Centos 6.5 | 4 Core 2.4 | 8 | v1.7.0 | 1 | 16 |
| 3 | PC | 12 | Centos 6.5 | 4 Core 2.4 | 8 | v1.7.0 | 1 | 96 |
| 4 | Workstation | 1 | Centos 6.5 | 4 Core 2.6 | 96 | v1.7.0 | 2 | 96 |

实验一共使用了表 2 中的 4 个真实数据集,其描述和相关处理如下:

1) DBLP:该数据集是 DBLP 数据库中论文作者合作的信息,其顶点表示论文的作者,顶点的标签为该作者所在的学术领域,边表示 2 个作者之间存在合作关系,边的标签为 2 个作者之间合作的次数,该数据集的原始数据为幂律分布,本文为了测试 FSMBUS 在稀疏数据上的性能,这里将度为 10 以上的顶点从 DBLP 中进行了移除,形成了新的稀疏数据集.

2) MiCo:该数据集为微软的合作者信息描述,其顶点和边的含义参考 DBLP,MiCo 数据集是 Elseidy[7]从 academic.research.microsoft.com 中抓取了计算机科学的信息而得到的.

3) Twitter:该图模型为 Twitter 上面的社交网络信息,其中顶点为 Twitter 中的用户,边表示 2 个用户之间存在联系,由于原始数据集上不存在顶点和边的标签,本文在进行实验时随机的对顶点添加了标签,标签的数量为 100 个以及他们服从高斯分布,边的标签都使用 1 来进行填充.

4) PldWeb:该数据集是 2012 年曼海姆大学在

互联网上使用爬虫获取的顶级域名之间的链接关系模型,其顶点代表各自的顶级域名,边则表示2个域名之间至少存在一条链接,由于该数据集自身并不包含标签,这里也是用了Twitter数据集上的标签添加方法对PldWeb进行处理,同时还移除了度为1 000以上的顶点.

**Table 2 The Description of Dataset**

**表2 实验数据集描述**

| Datasets | Type | Vertex Number | Vertex Label Number | Edge Number | Edge Label Number |
|---|---|---|---|---|---|
| DBLP | Sparse | 151 574 | 7 | 191 840 | 17 |
| MiCo | Power Law | 100 000 | 29 | 1 080 298 | 106 |
| Twitter | Power Law | 11 316 811 | 100 | 85 331 846 | 1 |
| PldWeb | Sparse | 35 024 175 | 100 | 303 194 717 | 1 |

### 4.1 FSMBUS性能评估

为了评估FSMBUS的性能,实验部分选择与下列算法进行对比:

1) GRAMI:文献[7]提出的算法,代码是作者提供的GRAMI_UNDIRECTED_SUBGRAPHS版本,它为单机环境下的算法.

2) FSMBUS-H:本文提出的FSMBUS算法在Hadoop2.2.0环境下的实现,FSMBUS-H算法使用迭代的MapReduce进行挖掘,利用HDFS进行数据共享.

3) NAVFSM:由于未见分布式环境下单图的处理方法,这是本文提出的朴素的单图上的分布式频繁子图挖掘算法,在MNI支持度计算阶段没有使用启发式搜索,而是直接使用枚举的方式来进行暴力计算,也是基于Spark平台运行.

FSMBUS与各个算法的对比实验结果在图9

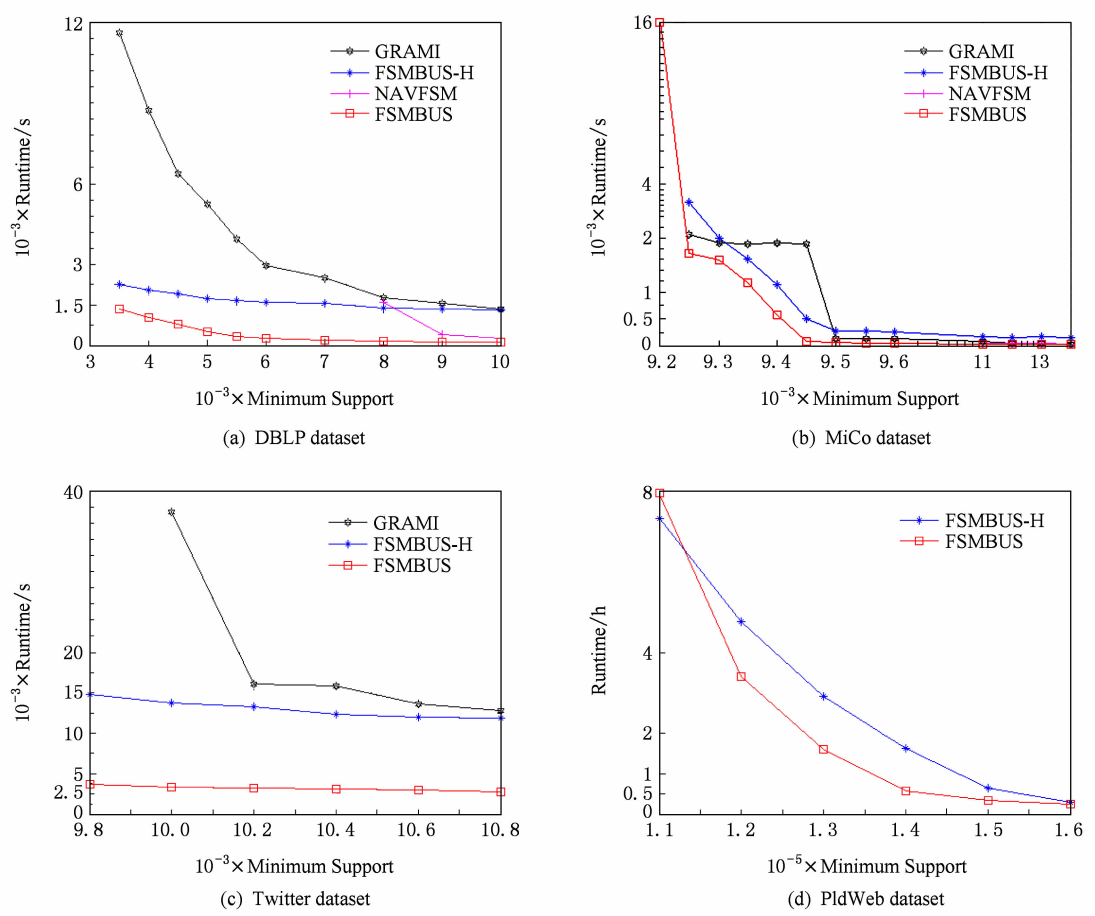



Fig. 9 Performance comparison with other algorithms.

图9 FSMBUS性能对比图

中给出.由于 GRAMI 为单机算法,而其余皆为分布式运行的算法,为了保证公平性,实验以总内存一致为前提展开,其中 DBLP 和 MiCo 数据集上 GRAMI 算法运行环境是表 1 的方案 1,FSMBUS 算法运行环境是方案 2,其额外的参数是 *num-executors*=1、*executor-cores*=1 和并行度 *parallelism*=1,Twitter 和 PldWeb 数据集上 GRAMI 使用了方案 4,FSMBUS 运行环境为方案 3,它的 *num-executors*=11(因为还有一台机器为 Driver),Twitter 的并行度 *parallelism*=11,PldWeb 的 *parallelism*=22,FSMBUS-H 和 NAVFSM 算法运行的配置与 FSMBUS 一致.图 9 显示 FSMBUS 算法性能最高,在相同的最小支持度阈值下,运行时间明显少于其他算法,最快比 GRAMI 快 20 倍;FSMBUS-H 性能次之,它在运行时需要多次从 HDFS 上面读写大量数据,实验显示其性能要普遍低于 FSMBUS,但是在 PldWeb(d)数据集上支持度到 110 000 时 FSMBUS-H 性能却超过了 FSMBUS,这是由于数据量太大、内存不足,Spark 会将数据溢写到磁盘,同时有会有大量的垃圾回收造成的;GRAMI 算法仅仅排在第 3,在 MiCo(图 9(b))数据集上支持度从 9 500 降到 9 450 时,其运行时间突然增加近 20 倍,这是因为此时 GRAMI 算法的 3 种优化并没有及时起到效果导致的,GRAMI 在 MiCo(图 9(b))数据集支持度为 9 200和 Twitter(图 9(c))数据集支持度为 9 800 时无法在 24 h 之内得到结果,在超大规模的数据集 PldWeb(图 9(d))上更是无法正常运行(OOM 异常);性能最差的是朴素算法 NAVFSM,由于在支持度计算是使用的枚举的暴力方法,时间复杂度和空间复杂度都非常高,实验显示 NAVFSM 只能运行在 DBLP(图 9(a))和 MiCo(图 9(b))这 2 个较小的数据集上,并且允许支撑的最小支持度也非常有限.但实验中发现 NAVFSM 在可运行的最小支持度下性能并不是最差,因为 NAVFSM 是基于 Spark 的,其外层框架与 FSMBUS 一致,所以当候选子图顶点数量较少时也有不错的性能.

根据上述实验结果分析可以得出结论:

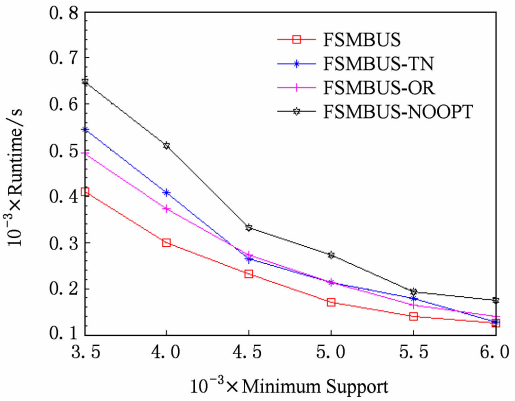
1) 与 GRAMI 相比,FSMBUS 支持更低支持度阈值和更大规模数据集,运算速度要高于 GRAMI 2~20 倍.FSMBUS 虽然同 GRAMI 一样使用了启发式算法进行搜索来计算 MNI 支持度,但是在支持度计算之前 FSMBUS 使用的候选子图的非频繁检测优化可以避免大量的非频繁子图进入搜索过程,减少搜索的空间;运用搜索顺序优化提前结束非频繁子图的搜索,比 GRAMI 的懒搜索和唯一标签而言更为简单高效.搜索时 CSP 约束的判断上使用了本文提出的数据结构从而减少判断次数,支持更低支持度阈值的计算.FSMBUS 是一个运行在 Spark 上分布式算法,将最耗时的支持度运算以单个候选子图为颗粒进行并行运算,同时在处理超大规模数据集时遇到存储内存不足的情况下会将数据溢写到本地磁盘上,运行效率要比 GRAMI 高数倍,并支持更大规模的数据集.

2) 相比于 Hadoop 环境,FSMBUS 算法更加适合运行在迭代能力强的 Spark 平台上.FSMBUS 算法在运行时需要大量的迭代,而 Hadoop 在迭代运算时需要频繁读写 HDFS,会造成许多额外的磁盘 IO 和序列化开销,Spark 通过 DAG 的作业执行计划和 RDD 之间的相互依赖可以高效地处理迭代需求,并且 Spark 是基于内存运算,除了输入输出,Spark 在内存足够的情况下不会产生额外的磁盘读写开销.

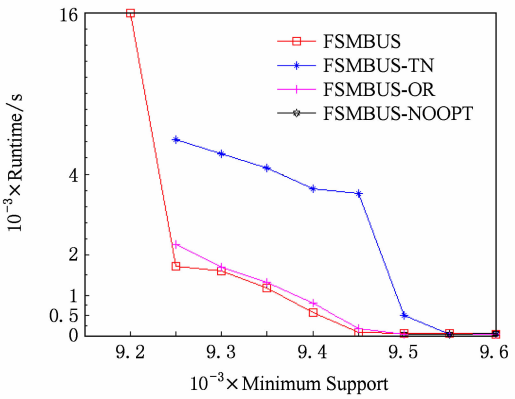
### 4.2 算法的优化性能影响

本节中进行的实验主要是用于演示算法优化上的相关性能影响,除了非频繁检测和顺序优化之外,还包括负载均衡、参数 *BatchSize*、并行度的优化.

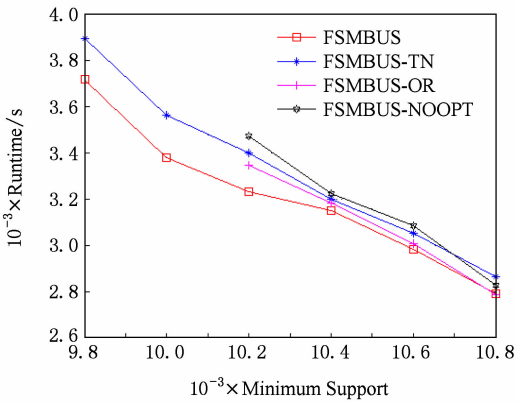
图 10 中显示的结果是非频繁检测和顺序优化对 FSMBUS 算法的影响,图 10 中标注的 FSMBUS 表示带所有优化的算法,FSMBUS-TN 表示只带非频繁检测优化的算法,FSMBUS-OR 表示只带顺序优化的算法,FSMBUS-NOOPT 表示未带任何优化的算法,运行环境与 4.1 节中的一致.实验中显示不同的优化在不同的数据集下对算法的影响不同,但是 2 个优化一起使用对 FSMBUS 的算法性能提升比较大.在图 10(a)的 DBLP 数据集中 FSMBUS-TN,FSMBUS-OR 均有 30% 左右的性能提升,而 FSMBUS 可以使性能提升 60%.图 10 的 MiCo(图 10(b)),Twitter(图 10(c))数据集中这两者的优化性能差异较大,在图 10(b)中 FSMBUS-NOOPT 仅仅支持的最小支持度阈值为 9500,FSMBUS-TN 虽然可以支持更低的支持度,但是其时间开销相当巨大的,而 FSMBUS-OR 在性能上却得到极大的提升,几乎与 FSMBUS 保持一致.支持度为 9 200 时只有 FSMBUS 才能得到算法结果,其图 10(c)中却与图 10(b)正好相反,FSMBUS-NOOPT 和 FSMBUS-OR 均只能支持 10 020 的支持度,而 FSMBUS-TN 的性能却非常接近 FSMBUS.PldWeb 数据集下只有 FSMBUS 才能正常运行,因而未显示在图 10 中,这也体现了这 2 个优化对性能的提升之大.




(a) DBLP dataset




(b) MiCo dataset

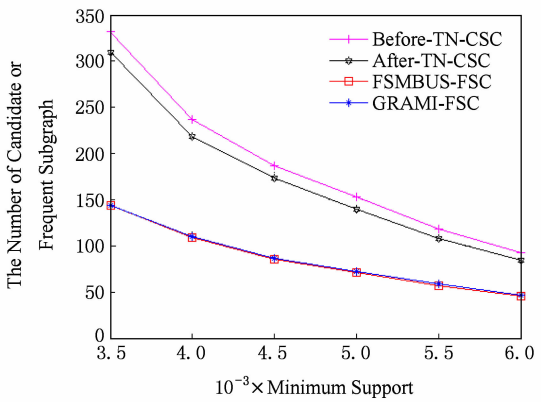



(c) Twitter dataset

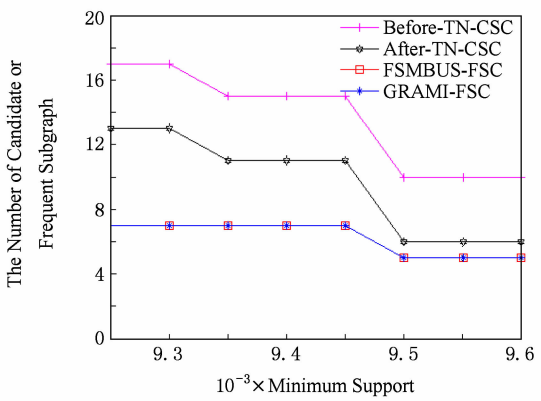
Fig. 10 Performance comparison of infrequent patterns' test and search order chosen.

图 10 非频繁检测和顺序优化对比实验结果

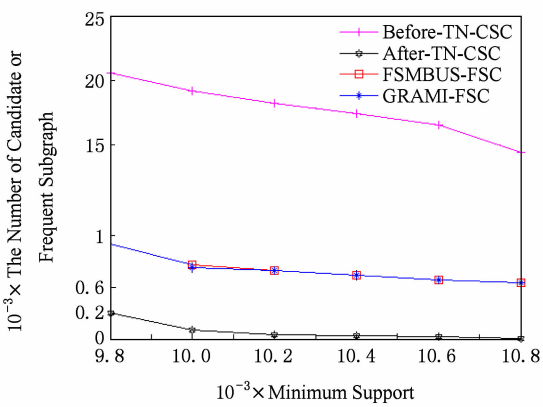
图 11 统计非频繁检测前后候选子图的数量以及最终 FSMBUS,GRAMI 计算的频繁子图数量的实验结果,其中 Before-TN-CSC,After-TN-CSC 分别表示非频繁检测前和非频繁检测后的候选子图的数量,FSMBUS-FSC,GRAMI-FSC 分别表示 FSMBUS,GRAMI 算法各自计算的最终频繁子图的数量. 图 11(a) 和图 11(b)显示在 DBLP 和 MiCo 数据集上进行非频繁检测后大概只剪枝掉了 20%～30%的候选子图,相比图 11(c)中 Twitter 数据上的 99%剪枝数量少了很多,因此在图 11(a) 和图 11(b)中 FSMBUS-TN 的提升效果并不是很显著. 结合图 11(c)对比图 10(c)中 FSMBUS-TN 与 FSMBUS-OR 的提升效果,因为 $\tau \geqslant 10\,200$ 时处理的大部分候选子图的顶点数量都是小于 4,根据 3.4 节可知非频繁检测的复杂度大于支持度计算,在 $\tau < 10\,200$ 之后,非频繁检测在支持度计算前是微不足道的,非频繁检测时除了剪枝掉大量的非频繁候选子图同时还大大




(a) DBLP dataset




(b) MiCo dataset



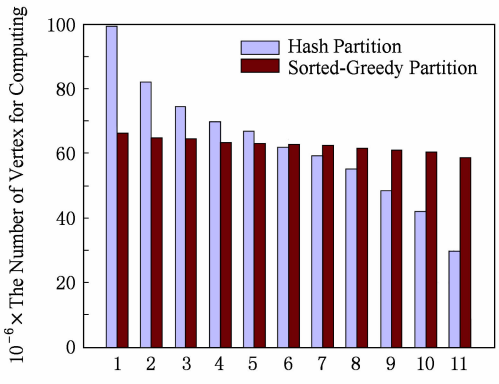

(c) Twitter dataset

Fig. 11 Comparison of candidate of frequent subgraph.

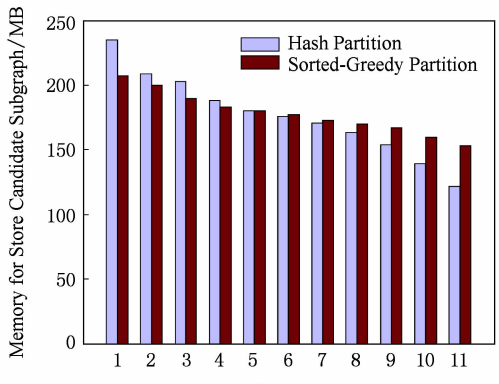
图 11 候选/频繁子图对比

减小支持度计算的搜索空间. 图 11 中 FSMBUS 和 GRAMI 这 2 个算法所计算的频繁子图的数量基本全部吻合,这也验证了本文 FSMBUS 算法的正确性.

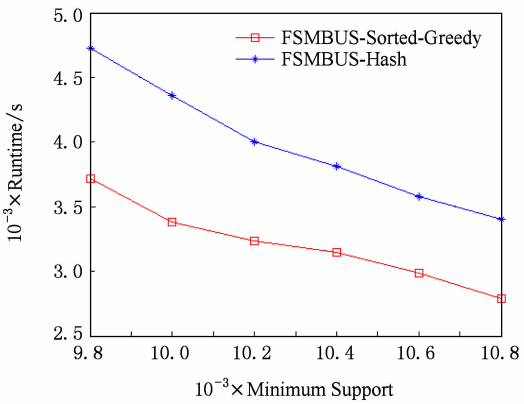
图 12 是负载均衡效果对比实验. FSMBUS 运算时间主要与候选子图搜索数据域中待搜索顶点数量有关,所以本实验的 Sorted-Greedy 划分是以候选子图的待搜索顶点数量作为划分的指标,图 12(a)和图 12(b)分别显示的是单次迭代各个 *executor* 需要处理的顶点数量和存储这些顶点所需的内存,




(a) Comparison of vertex number




(b) Comparison of store memory




(c) Comparison of partition method

Fig. 12 Comparison of load balance on Twitter dataset.

图 12 Twitter 数据集上负载均衡对比

可以发现 FSMBUS 使用 Hash 划分时 *executor* 上的数据存在较为明显的倾斜,但是使用了 Sorted-Greedy 划分方法后 *executor* 上数据几乎为均匀分布了,因此 Sorted-Greedy 划分可以有效地加速 FSMBUS 的运算,在图 12(c)中也验证了这一点:基于 Sorted-Greedy 的 FSMBUS 算法要比默认的 Hash 划分在运算速度上提升了 25%左右.

为了避免集群使用内存暴增的情况出现,FSMBUS 添加了一个参数 *BatchSize*,并行处理 *BatchSize* 个候选子图. 图 13 的实验中显示了 Twitter 数据集中不同的 *BatchSize* 对性能的影响,环境配置与 4.1 节一致,测试的最小支持度阈值为 10 000. 发现 *BatchSize*=20 000 时性能最优,这是因为 *BatchSize*<20 000 时需要更多的迭代次数才能完成 *k-size* 候选子图支持度的计算,也就是相当于会有更多的作业产生以及还有一小部分的重计算的开销,所以总的耗时会大于 *BatchSize*=20 000,但是当设定的 *BatchSize*>20 000 时,由于内存有限,每次迭代都需要处理更多的候选子图,频繁产生垃圾回收,导致性能差于 *BatchSize*=20 000. *BatchSize* 在不同的数据集和不同的支持度下将会有不同的最优值.

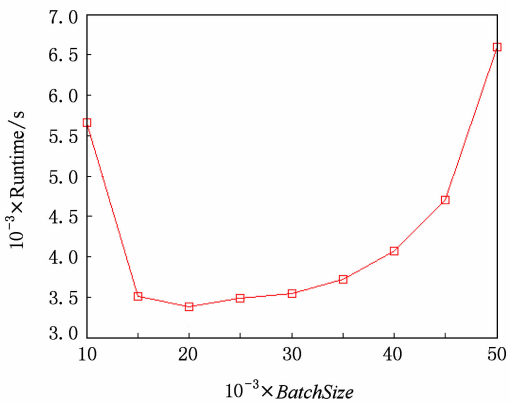
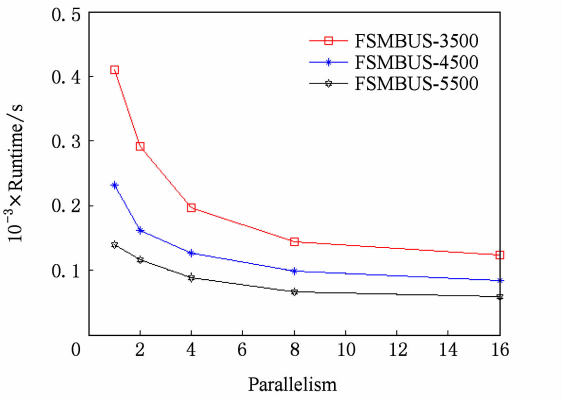



Fig. 13 Comparison of *BatchSize* on Twitter dataset.

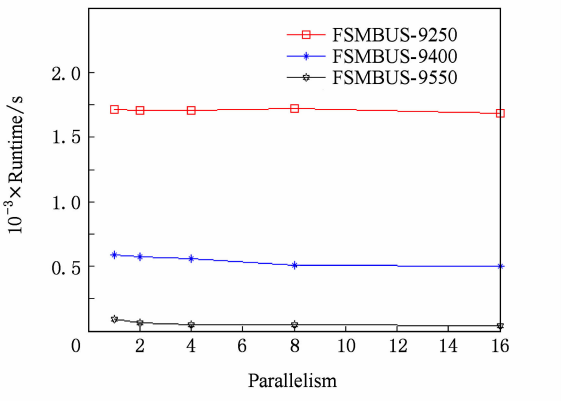
图 13 Twitter 数据集上 *BatchSize* 对比结果

图 14 演示的是不同支持度下并行度参数对性能的影响. 实验配置是 *num-executors*=11、*executor-cores*=4,横轴为并行度参数. 在图 14(a)和图 14(c)中可以发现在不同支持度下随着并行度的增加 FSMBUS 的性能也在不断地提升,同时造成 CPU 的满载运行、任务切换成本和 IO 成本不断增加;并行度到了一定的阈值后继续增加并行度对性能的影响也微乎其微了. 图 14(b)中观察到随着支持度的降低,并行度的增加对运行时间的影响越来越小,这

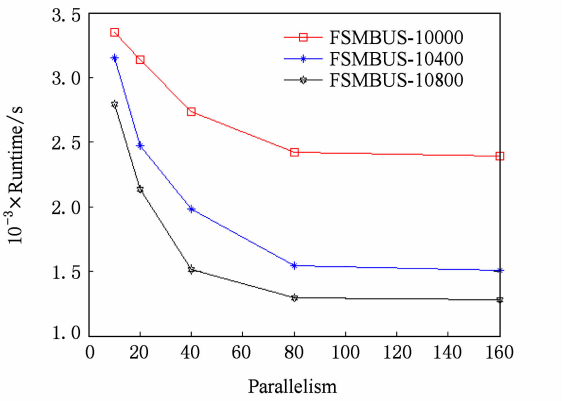
是因为 MiCo 数据集上存在度极端大的顶点,含有这个顶点的候选子图在计算支持度时耗时特别久,根据木桶原理,总运行时间几乎与该候选子图的计算时间一致,所以在这种情况下增加并行度对运行时间是几乎不会有影响的.




(a) DBLP dataset




(b) MiCo dataset




(c) Twitter dataset

Fig. 14 Comparison of parallelism.

图 14 并行度对比结果

## 5 总 结

本文提出了一种基于 Spark 的大规模单图的频繁子图挖掘算法——FSMBUS,该算法使用次优树按广度优先方向进行候选子图的生长,并行计算候选子图的支持度,通过迭代的 RDD 对大规模单图进行频繁子图的挖掘. 提出了非频繁检测和搜索顺序选择 2 种优化,为避免图划分时的数据倾斜,还设计了 Sorted-Greedy 的数据划分方法实现负载均衡. 实验显示 FSMBUS 的性能高于 GRAMI 一个数量级,支持更低的支持度以及更大规模的数据集,同时,FSMBUS 在 Spark 上运行时的性能普遍高于 Hadoop 平台 2~4 倍. 降低频繁子图支持度计算阶段的复杂度,提升频繁子图的挖掘速度是进一步的研究内容.

## 参 考 文 献

[1] Borgelt C, Berthold M R, Patterson D E. Molecular fragment mining for drug discovery [G] //Symbolic and Quantitative Approaches to Reasoning with Uncertainty. Berlin: Springer, 2005: 1002-1013

[2] Wang Guijuan, Yin Jian, Zhan Weixu. A novel graph classification approach based on embedding sets [J]. Journal of Computer Research and Development, 2012, 49(11): 2311-2319 (in Chinese)
(王桂娟, 印鉴, 詹卫许. 一种新的基于嵌入集的图分类方法 [J]. 计算机研究与发展, 2012, 49(11): 2311-2319)

[3] Guralnik V, Karypis G. A scalable algorithm for clustering sequential data [C] //Proc of the 1st IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2001: 179-186

[4] Yan X, Yu P S, Han J. Graph indexing: A frequent structure-based approach [C] //Proc of the 17th ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2004: 335-346

[5] Liu Y, Jiang X, Chen H, et al. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network [G] //Advanced Parallel Processing Technologies. Berlin: Springer, 2009: 341-355

[6] Shahrivari S, Jalili S. Distributed discovery of frequent subgraphs of a network using MapReduce [OL]. [2015-03-25]. http://link.springer.com/article/10.1007/s00607-015-0446-9

[7] Elseidy M, Abdelhamid E, Skiadopoulos S, et al. GRAMI: Frequent subgraph and pattern mining in a single large graph [C] //Proc of the 40th Int Conf on Very Large Data Bases. Berlin: Springer, 2014: 517-528

[8] Bhuiyan M A, Al Hasan M. An iterative MapReduce based frequent subgraph mining algorithm [J]. IEEE Trans on Knowledge and Data Engineering, 2013, 27(3): 608-620

[9] Lu W, Chen G, Tung A K H, et al. Efficiently extracting frequent subgraphs using mapreduce [C] //Proc of the 1st IEEE Int Conf on Big Data. Piscataway, NJ: IEEE, 2013: 639-647

[10] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters [J]. Communications of the ACM, 2008, 51(1): 107-113

[11] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [C] //Proc of the 9th USENIX Conf on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2012: 141-146

[12] Apache. Hadoop [OL]. [2009-03-06]. http://hadoop.apache.org

[13] Holder L B, Cook D J, Djoko S. Substucture discovery in the SUBDUE system [C] //Proc of the 1st the Conf on Knowledge Discovery and Data Mining. New York: ACM, 1994: 169-180

[14] Kuramochi M, Karypis G. Grew-a scalable frequent subgraph discovery algorithm [C] //Proc of the 4th IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2004: 439-442

[15] Kuramochi M, Karypis G. Finding frequent patterns in a large sparse graph * [J]. Data Mining and Knowledge Discovery, 2005, 11(3): 243-271

[16] Jiang X, Xiong H, Wang C, et al. Mining globally distributed frequent subgraphs in a single labeled graph [J]. Data & Knowledge Engineering, 2009, 68(10): 1034-1058

[17] Inokuchi A, Washio T, Motoda H. An apriori-based algorithm for mining frequent substructures from graph data [G] //Principles of Data Mining and Knowledge Discovery. Berlin: Springer, 2000: 13-23

[18] Kuramochi M, Karypis G. Frequent subgraph discovery [C] //Proc of the 1st IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2001: 313-320

[19] Yan X, Han J. Gspan: Graph-based substructure pattern mining [C] //Proc of the 2nd IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2002: 721-724

[20] Huan Jun, Wang Wei, Prins J. Efficient mining of frequent subgraphs in the presence of isomorphism [C] //Proc of the 3rd IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2003: 549-552

[21] Wang Wei, Zhou Haofeng, Yuan Qingqing, et al. Minging frequent patterns based on graph theory [J]. Journal of Computer Research and Development, 2005, 42(2): 230-235 (in Chinese)
(汪卫, 周皓峰, 袁晴晴, 等. 基于图论的频繁模式挖掘[J]. 计算机研究与发展, 2005, 42(2): 230-235)

[22] Li Xiantong, Li Jianzhong, Gao Hong. An efficient frequent subgraph mining algorithm [J]. Journal of Software, 2007, 18(10): 2469-2480 (in Chinese)
(李先通, 李建中, 高宏. 一种高效频繁子图挖掘算法[J]. 软件学报, 2007, 18(10): 2469-2480)

[23] Lin W, Xiao X, Ghinita G. Large-scale frequent subgraph mining in mapreduce [C] //Proc of the 30th IEEE Int Conf on Data Engineering. Piscataway, NJ: IEEE, 2014: 844-855

[24] Bringmann B, Nijssen S. What is frequent in a single graph? [G] //Advances in Knowledge Discovery and Data Mining. Berlin: Springer, 2008: 858-863

[25] He H, Singh A K. Graphs-at-a-time: Query language and access methods for graph databases [C] //Proc of the 27th ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2008: 405-418

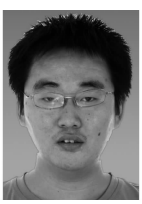
[26] Xin R S, Gonzalez J E, Franklin M J, et al. Graphx: A resilient distributed graph system on spark [C] //Proc of the 1st Int Workshop on Graph Data Management Experiences and Systems. New York: ACM, 2013: Article No. 2

[27] Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab: A framework for machine learning and data mining in the cloud [J]. Proceedings of the VLDB Endowment, 2012, 5(8): 716-727

[28] Ching A. Giraph: Large-scale graph processing infrastruction on Hadoop [C/OL] //Proc of the 2nd Hadoop Summit. 2011 [2015-03-25]. http://giraph.apache.org

**Yan Yuliang**, born in 1990. Master candidate in the Faculty of Electrical Engineering and Computer Science, Ningbo University. His main research interests include graph mining, cloud computing and machine learning.

**Dong Yihong**, born in 1969. PhD. Professor and master instructor in the Faculty of Electrical Engineering and Computer Science, Ningbo University. His main research interests include big data, data mining and artificial intelligence.



**He Xianmang**, born in 1981. PhD. Lecturer in the Faculty of Electrical Engineering and Computer Science, Ningbo University. His main research interests include big graph, privacy preserving (hexianmang@nbu.edu.cn).

**Wang Wei**, born in 1970. Professor and PhD supervisor of Fudan University. His main research interests include data mining, privacy preserving, and complex structure data management (wangweil@fudan.edu.cn).