基于 Spark 的图划分算法实现

李亮德[1], 王凌霄[1], 曹籽文[2], 侯诗铭[2], 路宏琳[2], 尉德利[3]

[1] 中国科学院自动化研究所

[2] 中国科学院信息工程研究所

[3] 中国科学院大学人工智能学院

2019 年 6 月 11 日

随着大数据时代的到来，数据的规模以前所未有的速度增长着，仅万维网的搜索引擎就可以抓取约一万亿的连接关系图。图作为一种由顶点和边构成的数据结构，能够简洁有力的表达事物之间的联系。对于一个大规模图的处理，必须进行图划分，降低分布式处理的各子图之间的耦合性，提高子图内部的连通性。最简单的随机划分算法利用哈希算法将数据划分到指定分区，谱方法将这一离散问题转换成计算矩阵的特征值以显示某种内在的连接关系，启发式算法通常将随机划分的结果作为初始划分，通过一些局部优化的方法减少交叉边的数量以达到降低通信代价的优化目的，层划分算法通过粗糙化、初始划分、细化三个阶段取得了较好的划分效果。Spark 作为一个大数据平台，提供了全面、统一的框架用于管理各种有着不同性质的数据集和数据源。本文依托 Spark 平台，利用其提供强大的存储与计算能力，研究并实现了上述四种图划分算法，并对其划分性能进行了测试。结果表明：

目录

1 研究背景 **3**

2 图划分算法 **3**

2.1 随机划分 4

2.1.1 哈希划分算法流程 4

2.2 谱方法 4

2.2.1 谱聚类算法流程 5

2.3 启发式算法 5

2.3.1 Kernighan-Lin 算法流程 5

2.4 多层划分算法 7

2.4.1 Metis 算法流程 7

2.4.2 粗化阶段 7

2.4.3 初始划分阶段 9

2.4.4 细化阶段 10

3 基于 Spark 的图划分 **10**

3.1 图数据实现 10

3.1.1 `Node` 类 10

3.1.2 `Graph` 类 10

3.2 哈希划分算法实现 12

3.3 谱聚类算法实现 12

3.4 Kernighan-Lin 算法实现 13

3.5 Metis 算法实现 13

4 总结 **16**

1 研究背景

随着大数据时代的到来, 数据的规模以前所未有的速度增长着, Facebook、Twitter、微博等社交媒体每天都产生大量的社交图数据。如何处理如此大规模的图数据成为目前研究的热点。其中, 图划分问题又是图数据处理领域中最为重要的问题之一, 图的搜索, 模式匹配等算法都需要图划分算法的支持。本文依托近些年来兴起的大数据平台, 利用其提供强大的存储与计算能力, 研究并实现了以大数据处理平台 Spark 作为处理引擎的图划分算法。

图作为一种由顶点和边构成的数据结构, 能够简洁有力的表达事物之间的联系。在今天这个信息化社会中，随着互联网用户的急剧增加，越来越多的网络数据问题摆在了我们面前。而在大数据时代下，大规模图数据处理问题便是一大热点。类似于网状图，如果将每个网络用户看作图中的节点，而将用户与用户之间的关系看作图中的边，那么整个网络就可看作一张网络图，大量的网络图组成的集合便是图数据。

图划分是经典的组合优化问题，它的应用背景极其广泛，包括软硬件协同设计、VLSI 设计、并行计算中的任务分配等众多领域。图划分是把一个图的顶点集分成 k 个不相交的子集，且满足子集之间的某些限制，即要找到一个图的划分，使得连接不同群组的边的权重尽可能小（意味着不同类中的点之间是不同的），而在群组内部的边有着很高的权重（意味着相同类中的点是彼此相似的）。近年来，计算机技术正以难以想象的速度发展，继而各个领域的问题变得日益复杂（比如软硬件划分、大规模集成电路设计、任务分配等），因此，人们迫切地需要对图划分进行深入而广泛的研究。

考虑到大量图数据的实时性需求，我们采用了 Spark 框架进行实验。Apache Spark 是一个围绕速度、易用性和复杂分析构建的大数据处理框架。Spark 最初在 2009 年由加州大学伯克利分校的 AMPLab 开发，并于 2010 年成为 Apache 的开源项目之一。与 Hadoop 和 Storm 等其他大数据和 MapReduce 技术相比，Spark 有如下优势：

- Spark 提供了一个全面、统一的框架用于管理各种有着不同性质（文本数据、图表数据等）的数据集和数据源（批量数据或实时的流数据）的大数据处理的需求
- Spark 可以将 Hadoop 集群中的应用在内存中的运行速度提升 100 倍，甚至能够将应用在磁盘上的运行速度提升 10 倍

本文综合前人的研究，首先介绍图划分算法的研究现状，重点介绍几类典型的图划分算法。然后, 通过对比实验，分析、比较不同图划分算法的性能特点。最后，对图划分算法进行总结。

2 图划分算法

图数据划分问题是经典的 NP（Non-deterministic Polynomial）完全问题，通常很难在有限的时间内找到图划分的最优解。尽管其是难解问题，从 20 世纪 90 年代初期至今，国内外

研究者不断对图划分及其相关问题进行深入研究，提出了许多性能较好的图划分算法。现主要有随机划分法、谱方法、启发式方法、多层划分算法等。

2.1 随机划分

散列划分是最经典的随机图划分算法之一。每个节点都有唯一的 ID，散列划分通过一个 Hash 函数计算各 ID 的哈希值，然后将哈希值相同的节点分配到相同的分区中。散列划分的优势在于简单易实现，不需要额外的开销且划分均衡可控。但这种方法的缺陷也是非常明显的，它没有考虑到图的内部结构，各节点会被随机地划分到各分区中，结果就是区间边会非常多，对后续的操作造成很大不便，并产生巨大的通信开销。

2.1.1 哈希划分算法流程

Algorithm 2.1: Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization: $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$

Input: a policy π (if estimating q_π)

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$, a positive integer n;

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\lessapprox = 0$) All store and access operations (for S_t, A_t, and R_t) can take their index mod $n + 1$;

2.2 谱方法

谱聚类（Spectral Clustering, SC）是一种基于图论的聚类方法，可以追溯到 R.Leland 和 B.Hendrickson 在 1990 年提出的谱方法。谱聚类可以识别任意形状的样本空间，能够将带权无向图划分为两个或两个以上的最优子图，并保证结果收敛于全局最优解。

谱聚类利用样本数据的相似矩阵 L（即拉普拉斯矩阵）进行特征分解，然后对特征向量进行聚类。拉普拉斯矩阵反映了各节点间的连接信息，它可以通过图的度矩阵 D 和关联矩阵 A 求得：$L = D - A$ 构造关联矩阵 A 需要计算样本两两之间的相似度值，这里一般使用高斯核函数来计算相似度。度矩阵 D 是一个对角矩阵，对角元素 $D_{ii} = deg\ (i)$。

谱聚类使得子图内部尽量相似，子图间距离尽量较远。常见谱聚类的最优目标函数有两种：

1. 可以是 **Min Cut**，即割边最小分割——的 Smallest cut
2. 可以是 **Normalized Cut**，即分割规模差不多且割边最小的分割——Best cut

L 矩阵中的元素定义如下所示：

$$L_{ij} = \begin{cases} 1, & \text{if } e(i,j) \in E \\ -\deg(i), & \text{if } i = j \\ 0, & \text{other} \end{cases} \tag{1}$$

基于图谱理论的划分方法的主要思想是：如果图 G 是连通的，那么其第二小特征值 (最小的是 0) 也就是 Fiedler 特征值将给出图的连通性的一个度量。如果将每个节点按对应的 Fiedler 特征值进行排序，然后将排好的序列分成 k 个部分，每个部分的节点将构成一个分区，即可实现图划分。然而从计算量上来看，Fiedler 特征值的计算和排序都较为复杂。谱方法能为许多不同类的问题提供较好的划分，但是谱方法的计算量非常大。

2.2.1 谱聚类算法流程

Algorithm 2.2: 谱聚类算法流程
Input: a differentiable action-value function parameterization: $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy π (if estimating q_π)

2.3 启发式算法

启发式算法通常是在对图已经进行过划分的结果上对其进行优化调整的。启发式算法的划分结果很大程度的依赖初始划分的结果，为获得较好划分结果，需要与能够产生好较优划分的算法结合。为了提高算法的执行效率，启发式算法通常将随机划分的结果作为初始划分，然后通过一些局部优化的方法减少交叉边的数量以达到降低通信代价的优化目的。

W.Kernighan 和 S. Lin 提出了 Kernighan-Lin 算法 [2]，它是一种比较典型的基于启发式规则的求解策略。KL 算法是一种贪婪算法，其初始划分选取了递归二分法。首先将图随机划分成两个子图且要求子图规模尽量一致即保证负载均衡，接着随机交换划分后两个子图的任意两个顶点并对其进行标记，依次选取未被标记的顶点进行交换，当所有标记过的顶点都经过交换之后，结束一轮迭代过程。记录每次迭代的过程中记录的交叉边个数最少，即划分质量最优的时顶点的划分状态，并将这种状态作为下次迭代的初始化分，以此类推。当下次迭代对划分质量无明显改善时，停止迭代。KL 算法流程图如下图所示。

2.3.1 Kernighan-Lin 算法流程

KL 算法的时间复杂度为 $O(TNN)$ ，其中 n 为节点数，t 为迭代次数。所以当图的数据很大时，执行时间过长，因此不适用于大规模图。此外，KL 算法是初始解敏感的算法，即产生一个较好初始社区的条件是事先得知社区的个数或平均规模，若初始解较差，则会造成收敛速度缓慢，最终解较差。由于现实世界中的社区无法事先预知，故 KL 算法的实用价值不大。

Algorithm 2.3: Kernighan-Lin 算法流程

Input: 未分区图数据 $G(V, E)$

Output: 已分区图数据 $G'(V, E, P)$

对 $G(V, E)$ 进行随机划分，保证各个子图规模相等或者相似，得到图 $G_r(V, E, P_{rand})$

repeat

  compute D values for all a in A and b in B

  let g_v, a_v, and b_v be empty lists

  for $n := 1$ *to* $|V|/2$ **do**

    find a from A and b from B, such that $g = D[a] + D[b] - 2 * c(a, b)$ is maximal

    remove a and b from further consideration in this pass

    add g to gv, a to av, and b to bv

    update D values for the elements of $A = A\ a$ and $B = B\ b$

  end

  find k which maximizes g_{max}, the sum of $gv[1], ..., gv[k]$

  if $g_{max} > 0$ **then**

    Exchange av[1],av[2],...,av[k] with bv[1],bv[2],...,bv[k]

  end

until $g_{max} \leq 0$;

2.4 多层划分算法

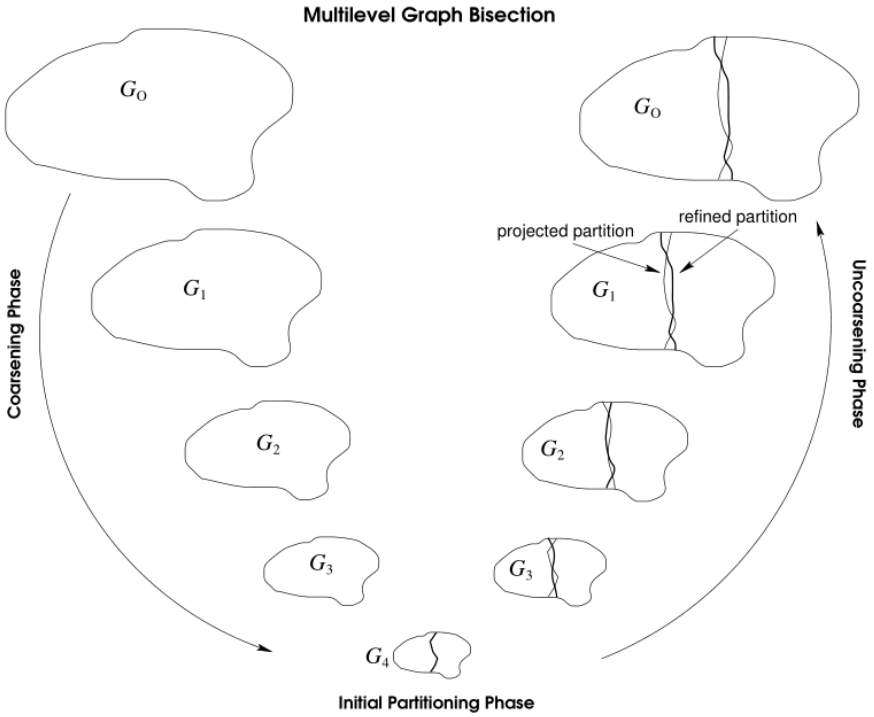

图 1

为了处理规模较大的图，文献 [3] 提出了多层的图划分框架 METIS。多层划分算法包括三个阶段：粗糙化 (coarsening) 阶段、初始划分、细化阶段 (uncoarsening)。第一阶段通过粗糙化技术将大图 $G = (V, E)$ 约化为可接受的小图；第二阶段将第一阶段获得的小图进行随机划分，并进行优化；第三阶段通过细化技术以及优化技术将小图的划分还原为原图的划分。该算法广泛地应用在各类大图的划分，对于百万规模以内的图，通常具有较好的实际效果。

2.4.1 Metis 算法流程

2.4.2 粗化阶段

在粗化阶段，一些联系较强的部分节点会聚集形成一个局部整体，这个局部整体会以一个带节点权重的超级节点 (Super node) 的形式呈现出来，而构成该局部整体的节点以及它们之间的边会暂时的从图中隐藏掉，同时对外只呈现出一个权重节点。为了得到这些局部整体，需要将部分图顶点进行融合。顶点融合的最终目的是为了减小原始图的规模，因此顶点的匹配应该最大化。最大匹配的定义如下：如果一个匹配在不使两条边指向同一个顶点的情况下无法再增加新的边，这个匹配叫做图的最大匹配。（由于匹配的计算方法不同，最大匹配的结果可能会有差异）。

经常用到的寻找匹配的策略主要有两种：随机策略 (Random Matching，RM) 和权重边策略 (Heavy edge matching，HEM)。

Algorithm 2.4: Metis 算法流程

Input: 未分区图数据 $G_o(V_o, E_o)$

Input: 分区数量 k

Output: 已分区图数据 $G'(V, E)$

if n_t 足够小 **then**

  直接在图 $G_t = (V_t, E_t)$ 上进行划分，得到对应的划分模式 P_t

  返回 P_t

else

  粗化图 G_i，得到一个规模较小的相似图 G_{i+1}

  迭代函数 $P_{i+1} = METIS(G_{i+1})$，以找到一系列的粗化图，直到图的规模足够小，即找到图 G_t

end

$i = t$; //用以记录细化阶段的层

repeat

  $G_(i-1) = recover(G_i, P_i)$ //划分模式的投射

  $P_(i-1)* = refine(G_(i-1), P_(i-1))$ //改善划分质量

  $i = i - 1$

until $i == 0$;

随机策略 随机策略算法顾名思义，即随机选取边来组成一个匹配，其时间复杂度为 $O(m)$。用随机算法可以快速有效地生成一个匹配随即最大匹配算法，按照如下步骤工作：先按照随机的顺序遍历原始图中所有的顶点，如果一个顶点 u 还没有被匹配，算法会随机地选择其尚未匹配的相邻顶点。如果存在这样的相邻顶点 v，就将边 $(u\ v)$ 加入匹配中。如果已经不存在未配的相邻顶点，则顶点 u 标记为未匹配顶点。这种策略虽然可以有效地找出一个极大匹配，但却没有考虑边和节点的权重值等信息。

权重边策略 权重边匹配即在用重边策略寻找匹配时要尽可能地选择边权重值最大的边，是一种简单有效的求取最大匹配的方法。在粗化过程中生成的各图的节点数在不断减少，但各图中节点的权重之和始终保持不变，但各图中的边权重之和却在持续减小。对于粗化过程中生成的两个连续的图 $G_i = (V_i, E_i)$ 和 $G_(i+!) = (V_(i+1), E_(i+1))$ 以及从 G_i 生成 $G_(i+1)$ 过程中的一个匹配 $M_i \in E_i$。如果 A 是边的集合，定义 $W(A)$ 为 A 中边的权值的总和。显然可以得到：$W(E_(i+1)) = W(E_i) - W(M_i)$ 其中，$W(E_i)$ 和 $W(E_(i+1))$ 分别表示图 G_i 和 $G_(i+1)$ 中的边的权重值之和，而 $W(M_i)$ 则表示匹配 M_i 中的所有边的权重值之和。

由此可见，粗化后的图中的边权重之和与粗化过程中的匹配有很大的关系。如果匹配中的边权重之和越大，则粗化后的图中边权重之和越小，反之亦然。所以，HEM 使得匹配中的边权重之和最大化，即在每一次的粗化过程中尽可能多的减少边权重，从而在粗化过程结束时得到的图中的边权重之和最小。

2.4.3 初始划分阶段

Metis 第二步是对粗化后的图进行 k 路划分。对粗化图 $G_m = (V_m, E_m)$ 计算划分 P_m 使得划分后的每部分大致均匀地含有原图的 $|V|/k$ 个顶点。

一种产生 k 路初始划分的办法就是不断地对原图进行粗化操作，直到粗化图只剩下 k 个顶点。这个含有 k 个顶点的粗化图可以作为原始图的 k 路初始划分。在这个过程中会产生两个问题：

1. 对于很多图来说，在进行了几次粗化之后，每一次的粗化过程所能减少的图的规模过小，因此粗化耗费的资源会很大。

2. 即使将原图粗化到了仅剩个顶点，这些顶点的权值也极有可能差异很大，最终导致初始划分的平衡度大大降低。

多级划分算法是算法的基本思想，就是在进行了顶点融合算法之后，原图的规模已经减小到了比较容易处理的地步。经过顶点粗化的处理以后，顶点数和边数大大减少的新图将更有利于初始划分的进行。在多级划分中，初始划分就是现将图进行二路划分，并在划分的同时考虑负载的平衡。初始划分常用的算法是几何划分和谱划分等。由于概化图规模一般较小，执行以上算法通常耗时很少。

2.4.4 细化阶段

如前所述，细化阶段是粗化阶段的反过程，其实就是还原过程。在细化过程中，在粗化过程中被隐藏的边和节点将会逐步重新呈现出来，

在这个步骤中，粗化图的划分，会通过回溯每一级的粗化图 G_m 的划分 P_m 还原成原图。虽然 P_{i+1} 是 G_{i+1} 的局部最小划分，但是细化后的划分 P_i 可能不再是 G_i 的局部最小划分。由于 G_i 更加精细，所以会有更大的自由度优化 P_i 以减少边割。因此，仍然可以使用局部细化启发式算法例如 KL 算法来优化划分 G_{i+1} 的划分。每进行一次细化，算法会对细化后的划分使用优化算法。划分优化算法的最基本思想是在划分后的两个部分中选择两个顶点集合进行互换，如果得到的新划分有更小的边割，则采用新的划分。

3 基于 Spark 的图划分

Apache Spark 是一个开源集群运算框架，最初是由加州大学柏克莱分校 AMPLab 所开发。相对于 Hadoop 的 MapReduce 会在运行完工作后将中介数据存放到磁盘中，Spark 使用了存储器内运算技术，能在数据尚未写入硬盘时即在存储器内分析运算。Spark 在存储器内运行程序的运算速度能做到比 Hadoop MapReduce 的运算速度快上 100 倍，即便是运行程序于硬盘时，Spark 也能快上 10 倍速度。

Apache Spark 项目包含下列几项：弹性分布式数据集（RDDs）、Spark SQL、Spark Streaming、MLlib 和 GraphX。Spark 提供了分布式任务调度，调度和基本的 I/O 功能。Spark 的基础程序抽象是弹性分布式数据集（RDDs），RDD 一个可以并行操作、有容错机制的数据集合。RDDs 可以透过引用外部存储系统的数据集创建（例如：共享文件系统、HDFS、HBase 或其他 Hadoop 数据格式的数据源），或者是透过在现有 RDDs 的转换而创建（比如：map、filter、reduce、join 等等）。

3.1 图数据实现

为了在 Spark 上进行图划分，本文将先介绍 `Node` 和 `Graph` 这两个类的基本情况。

3.1.1 `Node` 类

`Node` 类是图数据处理的基础。

3.1.2 `Graph` 类

`Graph` 类是将节点和边组合起来。

`edgeRDD` 的每项由以下四个部分组成：起点节点 Id 值、终点节点 Id 值、边权重、是否已经被匹配。

表 1: Node 类主要属性

属性	类型	定义
idx	String	节点的唯一 Id
neighbour	Map[String,Double]	节点的所有近邻节点
E	Double	外部权重，即节点与其他子图内的节点的连接权重和
I	Double	内部权重，即节点与本子图内的节点的连接权重和
partition	Int	节点所在子图的 Id
chosen	Boolean	节点是否与其他子图的节点交换过
composition	List[Node]	该节点的组成节点列表（仅用于节点聚合/拆分过程）
composLevel	Int	该节点的聚合程度（仅用于节点聚合/拆分过程）
isMark	Boolean	节点是否与其他节点匹配过
weight	Double	节点的权重

表 2: Graph 类主要属性

属性	类型	定义
nodeNum	Long	图内节点的个数
edgeRDD	RDD[(Str, Str, Double, Bool)]	图内所有边数据的 RDD 形式
nodeRDD	RDD[Node]	图内所有节点的 RDD 形式

3.2 哈希划分算法实现

```
import org.apache.spark.{HashPartitioner, TaskContext}
import util.Graph

object HashGraphPartition {
    def partition(graph: Graph, partitions: Int): Graph = {
        val assigenment = graph.nodeRDD.map(x => (x.getIdx, 0)).partitionBy(
            new HashPartitioner(partitions)).map(x => (x._1,
                TaskContext.getPartitionId))
        graph.buildPartitionGraph(assigenment)
    }
}
```

3.3 谱聚类算法实现

在 Spark 中实现谱聚类算法时，可以调用 MLlib 类库中的 Power iteration clustering 算法。

幂迭代聚类（Power iteration clustering）由 Frank Lin 和 William W.Cohen 提出，最早发表于 ICML 2010。在数据归一化的逐对相似矩阵上，使用截断的幂迭代，PIC 寻找数据集的一个超低维嵌入（低纬空间投影，embedding ）。这种嵌入恰好是很有效的聚类指标，使它在真实数据集上总是好于广泛使用的谱聚类方法（比如说 NCut）。

Algorithm 3.1: PIC 算法流程

Input: 按行归一化的关联矩阵 W

Input: 期望聚类数 k

随机选取一个非零初始向量 v^0

repeat

  $v^{(t+1)} = \frac{Wv^{(t)}}{\left|Wv^{(t)}\right|_1}$

  $\delta^{(t+1)} = \left|v^{(t+1)} - v^{(t)}\right|$ 增加 t 值

until $|\delta^t - \delta^{t-1}| \simeq 0$;

使用 k-means 算法对向量 v^t 中的点进行聚类 **Output:** 类 $C_1, C_2, \cdots, C_K$

```
def partition(graph: Graph, partitions: Int, maxIter: Int, weightNorm:Boolean):
    Graph = {

    var weight:scala.collection.Map[String,Double]=null
```

```
    if(weightNorm==true)
        weight = graph.nodeRDD.map(x=>(x.getIdx,x.getWeight)).collectAsMap()

    def weight_norm(sim:Double,x:String,y:String):Double={
        if(weightNorm) sim/math.pow(weight(x)*weight(y), 2)
        else sim
    }

    val simEdge = graph.edgeRDD.map(
        x => (x._1.toString.toLong,
               x._2.toString.toLong,
               gaussianSimilarity(weight_norm(x._3,x._1,x._2), 1)))

    val modelPIC = new PowerIterationClustering()
            .setK(partitions) // k : 期望聚类数
            .setMaxIterations(maxIter) //幂迭代最大次数
            .setInitializationMode("degree") //使用度初始化
            .run(simEdge)

    val assign = modelPIC.assignments.map(x => (x.id.toString, x.cluster))
    graph.buildPartitionGraph(assign)
    graph
}
```

```
def gaussianSimilarity(dist: Double, sigma: Double): Double =
    math.exp(-dist / (2 * sigma * sigma))
```

3.4 Kernighan-Lin 算法实现

3.5 Metis 算法实现

```
def partition(graph: Graph, c:Int, mode:String,weightNorm:Boolean): Graph={
    var partitionedGraph = graph
    var level:Int=0

    // 1.coarsening phase
```

```
    val coarsenRes = coarsen(graph, c,mode)
    partitionedGraph = coarsenRes._1 // coarsen Graph
    level = coarsenRes._2 //coarsen level

    // 2.partitioning phase
    partitionedGraph = initialPartition(partitionedGraph, weightNorm:Boolean)
    partitionedGraph.printGraph()

    // 3.un-coarsening phase
    partitionedGraph = refine(partitionedGraph,level) //need level to decide refine
        order

    partitionedGraph
}
```

```
private def coarsen(graph: Graph, c: Int,mode:String): (Graph,Int)={
    /** Step 1: Coarsening Phase (via Maximal Matching Algorithm)
        * input: origin graph G_o,
        *        coarsening parameter c,
        *        mode
        * output: coarsen graph G_c
    * */
    var coarsenedGraph = graph
    var level = 0
    while(coarsenedGraph.nodeNum > c*k){
        level += 1
        coarsenedGraph = maximalMatching(coarsenedGraph, mode,level)
        coarsenedGraph = refreshMarkedFlag(coarsenedGraph)
    }
    (coarsenedGraph,level)
}
```

```
private def initialPartition(graph: Graph, useWeightNorm:Boolean): Graph={
    /** Step 2: Partitioning Phase (via Spectral Clustering)
        * input: graph - coarsened graph g
        *        useWeightNorm - whether to normalize the weights.
        * output: splitGraph - initial partition graph
        * */
```

```
    val splitGraph = SpectralClustering.partition(graph, k, 40, useWeightNorm)
    splitGraph
}
```

```
private def refine(graph: Graph,level:Int): Graph={
    /** Step 3: Refining Phase (via Kernighan-Lin Algorithm)
        * input: graph - initial partition graph
        *       level - refine level
        * output: refined graph
        * */
    var refinedGraph = graph
    var refineLevel = level
    while(refineLevel!=0){
        /** Step 1: Split coarsen node to refined nodes. */

        // 1.1 Find coarsen nodes and refine them (new node)
        var refinedNodeRDD = refinedGraph.nodeRDD.
                        filter(x=>x.getComposLevel==refineLevel)
        refinedNodeRDD = refinedNodeRDD.map(_.setCompositionPartition())

        // 1.2 Save the nodes which don't need to refine
        val nodeRDD = refinedGraph.nodeRDD.filter(x=>x.getComposLevel!=refineLevel)
        refinedNodeRDD = refinedNodeRDD.flatMap(x=>x.getComposition)

        // 1.3 Union two parts of refined nodes.
        refinedGraph.nodeRDD = refinedNodeRDD.union(nodeRDD)

        /** Step 2: Partitioning */
        val assignment = refinedGraph.nodeRDD.map(x=>(x.getIdx,x.getPartition))
        refinedGraph = KernighanLin.partition(refinedGraph, assignment, needMaxGain
            = true)
        refinedGraph.nodeRDD = refinedGraph.nodeRDD.map(_.setChosen(false))
        refineLevel-=1 //refine Level decrease 1, up refine
    }

    refinedGraph.nodeNum = refinedGraph.nodeRDD.count()
    refinedGraph
}
```

4 总结

图划分问题普遍存在较高的计算复杂度，但图数据在现实中具有重要的价值，这也是人们不断追求更简易的图划分算法的原因。上文提到的四种图划分算法在处理方法与结构上各具特色，也各有优劣。

传统图划分算法可以处理节点数和边数较少的图，包括局部改进图划分算法和全局图划分算法，其中局部改进图划分算法中比较经典的是 KL 算法，全局图划分算法中比较经典的是谱划分法和多层图划分 METIS 算法。然而这些算法具有较高的时间复杂度，无法处理节点数为百万级以上的图。所以接下来我们可以对分布式图算法进行进一步的研究，从而处理大数据时代中大规模的图划分问题。