中国科学: 信息科学 2019 年 第 49 卷 第 8 期: 988–1004
SCIENTIA SINICA Informationis
论文

一种高效的面向动态有向图的增量强连通分量算法

廖小飞[1,2,3,4], 陈意诚[1,2,3,4], 张宇[1,2,3,4*], 金海[1,2,3,4], 刘海坤[1,2,3,4], 赵进[1,2,3,4]

1. 华中科技大学大数据技术与系统国家地方联合工程研究中心, 武汉 430074
2. 华中科技大学服务计算技术与系统教育部重点实验室, 武汉 430074
3. 华中科技大学集群与网格计算湖北省重点实验室, 武汉 430074
4. 华中科技大学计算机科学与技术学院, 武汉 430074
* 通信作者. E-mail: zhyu@hust.edu.cn

收稿日期: 2018–05–16; 修回日期: 2019–01–24; 接受日期: 2019–04–18; 网络出版日期: 2019–08–07

国家重点研发计划 (批准号: 2018YFB1003500) 和国家自然科学基金 (批准号: 61832006, 61825202, 61702202) 资助项目

摘要 强连通分量 (strongly connected component, SCC) 算法可以将一个有向图缩略为有向无环图 (directed acyclic graph, DAG), 广泛应用于可达性查询等有向图分析应用. 尽管现有工作已经提出多种面向静态有向图的强连通分量算法, 但是它们需要高额的运行时开销来反复对整个图进行全量计算, 以响应现实世界中普遍存在的动态有向图结构的频繁变化. 其实, 在通常情况下, 动态有向图每次改变量极小 (少于 5%). 其允许我们以增量的方式对动态有向图进行强连通分量计算, 以缩短响应时间. 因此, 为解决此问题, 本文提出了一种高效的面向动态有向图的增量强连通分量算法 Incremental Strongly Connected Components Algorithm, 简称 Inc-SCC, 通过对不必要的计算进行裁剪以减少算法的数据访问量和计算量, 并利用 SCC 的不相交性进行并行处理以提升 SCC 计算效率. 其次, 提出了一种启发式优化方法进一步加快算法收敛速度. 实验结果显示, 本方法可以用于实时响应有向图持续性动态变化, 并且当整个有向图的边变化比例为 5% 时, 本方法相对于现有算法的加速比可达 2.8 到 12 倍, 当整个有向图的边变化比例为 0.5% 时, 本方法相对于现有算法的加速比可达 2.9 到 12 倍.

关键词 强连通分量, 动态有向图, 增量计算, 收敛, 有向无环图

1 引言

图作为一种常见的数据结构, 可以用来抽象表达现实事物间各种复杂的关联关系 [1]. 例如, 交通网络、社交网络、万维网和生物网络都采用图表示. 每个强连通分量 (strongly connected component, SCC) 是有向图中所有图顶点都相互可达的最大子图. 通过将每一个 SCC 压缩为一个图顶点, 强连通

引用格式: 廖小飞, 陈意诚, 张宇, 等. 一种高效的面向动态有向图的增量强连通分量算法. 中国科学: 信息科学, 2019, 49: 988–1004, doi: 10.1360/N112018-00125
Liao X F, Chen Y C, Zhang Y, et al. An efficient incremental strongly connected components algorithm for evolving directed graphs (in Chinese). Sci Sin Inform, 2019, 49: 988–1004, doi: 10.1360/N112018-00125

分量算法可以将一个有向图缩略为有向无环图 (directed acyclic graph, DAG) [2~6]. 大量的图应用, 例如拓扑排序 [7]、可达性查询 [8,9] 和图模式匹配 [10], 都采用强连通分量算法将有向图转换成 DAG, 以提高其执行效率, 被广泛应用于社交分析, 道路导航和软件分析等多种场景. 对于大规模的有向图, 强连通分量算法需要高额的数据访问开销和计算开销以遍历整个有向图寻找所有 SCCs. 目前已经出现大量面向静态有向图的强连通分量算法. 它们主要分为两类, 即串行方法和并行方法. 串行方法 (例如 Tarjan [3], Kosaraju [4] 算法) 采用深度优先搜索 (depth-first-search, DFS) 遍历有向图以获得全部的 SCCs, 其性能达到线性时间复杂度. 由于串行方法无法利用现在处理器的高并发性, 因此, 与处理器核数的发展相契合的并行方法逐渐成为主流. 并行方法 (例如 Coloring [5], FW-BW [11], FW-BW-TRIM [6] 算法) 主要通过并行地求解前向可达集和后向可达集的交集以获得各 SCC. 在现有的有向图数据集上, 其总计算时间远少于串行方法 [6].

在现实世界中, 有向图通常频繁发生变化, 例如社交好友关系变化和航班变化. 然而, 现有的强连通分量算法 [3~6,11~16] 主要面向静态有向图. 它们在处理动态有向图时, 把新获得的图镜像视为完全孤立的静态图, 采用强连通分量算法重新计算整个新图镜像中的 SCCs. 但是, 现有工作重新计算整个新图镜像的 SCCs 的效率很差, 无法满足动态有向图快速变化中的实时响应. 例如, Facebook 中的好友网络中包含 16 亿个节点和 1500 亿条边 [17]. 如此庞大的数据量使得其需要很长的时间以计算一个图镜像中的所有 SCCs, 导致计算结果失去时效性.

实践显示, 在变化过程中, 动态有向图的两个相邻图镜像之间的改变量通常少于 5% [18]. 同时, 我们观察到, 在两个相邻图镜像之间, SCC 具有如下性质: (1) 一个 SCC 内部结构[1)]的变化不会引起有向图中其他 SCC 的重新计算 (因为它不会对其他 SCC 内部结构产生影响, 我们称此特性为 SCC 的不相交性). (2) 只有当一个 SCC 内部发生边的删除时[2)], 才需要重新计算该 SCC. 现有并行算法在对动态有向图进行全量计算时无法感知到动态有向图变化过程中 SCC 的这两个特性, 因此面临大量冗余数据计算量和访问量.

为了解决上述问题, 本文提出了一种面向动态有向图的增量强连通分量算法 (incremental strongly connected components algorithm), 简称 Inc-SCC. 它利用上述观察, 通过对不必要的计算进行裁剪以减少算法的数据访问量和计算量. 具体来说, Inc-SCC 首先对每次变化过程中的增量进行分类. 对于增加边的情况, Inc-SCC 直接在 SCC 构成的 DAG (其图顶点是一个 SCC) 上计算 SCC. 对于删除边的情况, Inc-SCC 只对受到边删除影响的 SCCs 进行重新计算, 以避免那些不受影响的 SCC 的处理. 对于受影响的 SCCs, 它利用 SCC 的不相交性, 只在每个受影响的 SCC 内部进行并行的图顶点状态传播, 而不使其内部图顶点状态值传递出该 SCC, 减少它们对其他 SCC 的影响, 提升算法效率. 其次, 我们设计了一种启发式优化方法进一步加快算法收敛速度. 具体来说, 它会维护动态有向图中各 SCC 的根顶点 (某 SCC 的根顶点是此 SCC 中状态值未发生变化的图顶点), 然后选择每个发生变化的 SCC 的根顶点作为枢纽点, 通过并行 BFS 获得其前向可达集和后向可达集, 最后通过找出此前向可达集和后向可达集的交集 (即未发生状态值变化的图顶点) 获得新的 SCC. 在清除这个交集之后, 本方法在剩余图顶点中进行并行的图顶点状态值传播直到所有 SCC 被找到. 这样做可以减少每个 SCC 内部的冗余图顶点状态值传播量 (即减少图顶点更新次数), 从而能加快算法收敛速度. 实验结果验证了该启发式方法能够有效加快算法迭代, 提高算法执行效率.

1) 一个 SCC 的内部结构是指该 SCC 内部图顶点集合及这些图顶点之间的有向边集合.

2) 增加边有可能把原先的两个 SCC 变为一个更大的 SCC, 然而这不需要高额的 SCC 算法重新计算 SCC (我们只需要简单地在 DAG 图中合并这两个 SCC). 例如, 假设在 DAG 中 SCC_i 和 SCC_j 之间存在一条从 SCC_i 到 SCC_j 的有向边, 如果在 v_j 和 v_i 之间增加一条从 v_j 到 v_i 的有向边, 且 $v_i \in \mathrm{SCC}_i$, $v_j \in \mathrm{SCC}_j$, 那么 SCC_i 和 SCC_j 可以立即合并成为一个更大的 SCC.

本文的贡献可以归纳为如下几点:

(1) 观察并总结了在动态有向图变化过程中的 SCC 的特性, 提出了面向动态有向图的增量强连通分量算法 Inc-SCC 以裁剪冗余计算.

(2) 针对动态有向图变化过程中每个受增量影响的 SCC 内部图顶点状态的变化特性, 提出一种启发性方法来加快迭代收敛速度.

(3) 实验结果表明, Inc-SCC 适用于动态有向图持续性变化, 并且当整个有向图的边变化比例为 5% 时, 本方法相对于现有算法的加速比可达 2.8 到 12 倍, 当整个有向图的边变化比例为 0.5% 时, 本方法相对于现有算法的加速比可达 2.9 到 12 倍.

本文的其余部分组织如下: 第 2 节介绍相关工作; 第 3 节提出 Inc-SCC 的设计思路; 第 4 节给出 Inc-SCC 的实现; 第 5 节进行实验和结果分析; 第 6 节总结全文并展望未来工作.

2 相关工作

尽管现有工作已经尝试使用强连通分量算法解决有向图中的多种问题, 但是面向动态有向图的强连通分量算法仍然缺少高效支持. 这些研究驱动我们设计了面向动态有向图的增量强连通分量算法 Inc-SCC. 目前的强连通分量算法主要分为串行方法和并行方法.

Tarjan [3] 和 Kosaraju [4] 算法是两种经典的串行强连通分量算法. 两者均采用 DFS 方法遍历全图, 获得图顶点的拓扑顺序, 进而求得所有 SCCs. 它们都达到 $O(m+n)$ 的线性时间复杂度, 其中 m 为图的边数, n 为图的顶点数. 由于 Kosaraju 算法额外采用 DFS 对有向图进行一次反向扫描, 因此 Tarjan 算法效率通常更高. Zhang 等 [12] 利用 DFS 的弱序关系构建生成树, 在图的顺序遍历过程中快速提取 SCC. 并且, 它使用批处理方法以进一步降低磁盘 I/O 开销.

随着处理器中核数的增加, 由于串行算法无法利用处理器的高并发性使得其性能受核心频率限制, 因此并行强连通分量算法已经成为主流趋势. UFSCC [16] 将有向图分为多个划分块, 在各个划分块中执行 DFS 算法, 并且利用并查集和循环链表降低线程间通信开销. 然而, 该算法在不同划分块间存在共享数据问题, 导致其并行度不高. FW-BW 算法 [11] 是一种经典的并行强连通分量算法. 它首先以一个图顶点为开始通过前向后向扫描得到一个 SCC, 然后将剩余图顶点划分为 3 个不相交的集合, 接着在这 3 个不相交的划分块中反复采用上述操作直到找出所有 SCC. 由于真实世界图的幂律分布特性 [19], 有些 SCC 中的图顶点占到了整个图的绝大部分. 这使得 FW-BW 算法的并行度有限, 并且难以负载均衡, 在多核/众核处理器上性能差. FW-BW-TRIM 算法 [6] 基于 FW-BW 算法, 对有向图进行一次遍历以移除有向图中的孤立 SCC (即出度或者入度为 1 的图顶点) 从而减少处理量. OBF 算法 [13] 通过将有向图划分为多个独立子图, 在每个子图上分别执行 FW-BW 算法以提高性能. Hong 等 [14] 在 FW-BW-TRIM 算法的基础上, 进一步裁剪有向图中图顶点度为 2 的图顶点. 然而, 由于这些算法为任务级并行, 因而难以利用多核/众核处理的高并发性缩短其执行时间.

为了充分利用多核/众核处理器的高并行性, Coloring 算法 [5,15] 采用如下方法求解前向可达集和后向可达集的交集. 它为有向图中每个图顶点分配独特的初始状态值 (通常为图顶点 ID), 然后让每个图顶点并行地向其邻居图顶点传播自身的状态值直到收敛. 在迭代收敛时, 属于同一个 SCC 的图顶点具有相同的状态值. 其中状态值未发生变化的图顶点即为其所在 SCC 的根顶点. 然后, 此算法再从每个根顶点反向扫描以获得相应的 SCC. 上述过程反复进行, 直到所有 SCC 被找到. 相比 FW-BW 算法 [11], Coloring 算法为数据级并行, 能充分利用底层多核/众核处理器而获得更好的性能. 然而, Coloring 算法在迭代传播过程中存在大量冗余计算, 特别是对于动态有向图.

另外, 图计算系统可以用于加快强连通分量算法的迭代计算 [20,21]. FBSGraph [22] 是现有最好的异步图系统. 它提出了基于路径的图划分策略, 使得图顶点的状态沿着路径传播以加快异步迭代的传播速度. 其异步计算的性能达到了目前最先进水平. 除此之外, 还有很多新型体系结构的图系统. 暂存器内存为图计算提供高性能数据访问 [23]. 数据流模型 [24] 利用 FPGA 的可编程性以解决传统处理器中低指令级并行度问题. Foregraph [25] 利用 FPGA 片上存储资源的高带宽以降低迭代过程中图数据随机访问开销. 这些图系统提高了强连通分量算法的处理效率. 然而, 由于缺少对动态有向图的增量计算的支持, 因此在对动态有向图进行强连通分量计算时, 它们都存在大量的冗余计算开销. Version Traveler [26] 以增量方式在主存中维护不同图镜像间的差值, 进而融合增量与当前图镜像以达到多个图镜像间的快速切换. 但是该系统仍然无法解决新图镜像的冗余计算问题. Zhang 等 [27] 针对仅有模式图发生改变的应用场景, 建立后序增量匹配的索引以提高图模式匹配效率. 但是 Zhang 等没有对强连通分量算法进行优化. TimeReach 算法 [28] 在动态有向图场景下, 建立索引保存不同时刻的 SCC 信息, 提高历史可达性查询的效率. 但是 TimeReach 算法仍然在动态有向图场景中以全量方式计算强连通分量. 由于在这些动态图工作上, 全量计算的强连通分量算法已经成为性能瓶颈, 因此这些需求进一步促使我们提出面向动态有向图的增量强连通分量算法 Inc-SCC 以减少冗余计算开销.

3 Inc-SCC 算法设计

本节首先介绍算法中的相关概念, 然后提出 Inc-SCC 基本设计思想, 最后给出 Inc-SCC 算法.

3.1 相关概念定义

本小节给出 Inc-SCC 相关的基本概念和符号解释.

定义1 (动态有向图) 给定一个有向图 $G=(V,E)$, 其中 V 表示 G 中图顶点的集合, E 表示 G 中边的集合 [4]. 在现有动态图算法中, 由于连续的时间通常表示为离散的时间点, 因此我们将连续的时间表示为 $t_0,t_1,\ldots,t_i,t_{i+1},\ldots$, 其中 t_0 表示初始时间点. 则 t_i 时刻的图镜像可以记作 $G_{t_i}=(V_{t_i},E_{t_i})$. 因此, 时间段 $[t_i,t_j]$ 上的动态图可以表示为一组离散的有向图镜像的序列, 记作 $G_{t_i},G_{t_{i+1}},\ldots,G_{t_j}$. 在时间点 t_i 和 t_{i+1} 之间, 动态图在时间点 t_i 时的图镜像 G_{t_i} 会发生变化. 在时间点 t_i 和 t_{i+1} 这段时间内动态图的所有变化量的积累值表示为子图 $\Delta G_{t_{i+1}}=(\Delta V_{t_{i+1}},\Delta E_{t_{i+1}})$, 其中 $\Delta V_{t_{i+1}}$ 和 $\Delta E_{t_{i+1}}$ 分别表示在时间点 t_i 和 t_{i+1} 这段时间内动态图变化的图顶点集合和变化的边集合. 因此, 动态图在时间点 t_{i+1} 时的图镜像 $G_{t_{i+1}}=G_{t_i}\oplus\Delta G_{t_{i+1}}$, 其中 $\oplus$ 表示根据 $\Delta G_{t_{i+1}}$ 更新图镜像 G_{t_i}.

定义2 (强连通分量) 在有向图 $G=(V,E)$ 中, 如果存在至少一条有向路径, 使得 G 中的节点 v_i 可以到达 G 中的另一个节点 v_j, 则记作 $v_i\to v_j$. 当且仅当 $v_i\to v_j$ 和 $v_j\to v_i$ 成立时, v_i 和 v_j 相互可达, 记作 $v_i\leftrightarrow v_j$. 一个 SCC 是 V 的一个最大的子集 V_c, 对于每一对节点 v_i 和 v_j $(v_i,v_j\in V_c)$ 满足条件 $v_i\leftrightarrow v_j$; 对于任一节点 $v_i\in V_c$, 任一节点 $v_k\notin V_c$, $v_i\leftrightarrow v_k$ 都不成立. 在实际考虑中, 每个孤立图顶点均可以看作是单独的 SCC.

3.2 基本设计思想

本小节介绍 Inc-SCC 的基本设计思想. 动态有向图在变化时通常包含两种情况: 边的增加、边的删除. 注意, 增加或删除图顶点其实就是增加或删除与该图顶点相连接的边集合. 因此, 我们只需要考虑如何处理增加或删除边即可.

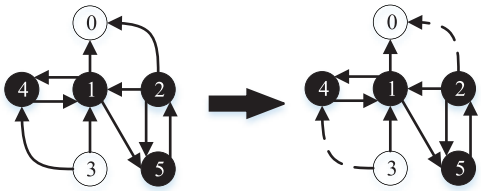

图 1　类型 1: SCC 内部结构未变化

Figure 1　The first type of deleted edges: no change in SCC structure

3.2.1　边的增加

对于增加边的情况, 我们可直接简单地在 SCC 构成的 DAG (其图顶点是一个 SCC) 上计算 SCC. 在动态有向图的变化过程中, 根据增加的边的两个图顶点在动态有向图中的不同位置, 可以分为以下两种情况: (1) 当一条增加的边的两个图顶点在同一个 SCC 中时, 这条增加的边的两个图顶点仍然属于同一个 SCC, 因此不会导致任何 SCC 的重新计算. (2) 当一条增加的边的两个图顶点在两个不同的 SCC 中时, 这条增加的边将成为 DAG 中两个不同图顶点之间的边. 在这种情况下, 动态有向图上的 SCC 计算问题转化为在 DAG 上的 SCC 计算问题. 增加边的开销确实是不可忽略的. 然而, 对于缩略有向图得到的 DAG, 其图顶点规模通常远小于原图的规模 [2]. 这意味着, 相对于现有方法 (增加每条边都需要对整个图重新计算 SCC) 对增加每条边进行全量计算开销来说, 在缩略后的 DAG 上操作的开销要低很多. 注意, 对于边增加的情况, 本方法相对于 Coloring 算法 [15] 的加速比:

$$S \geqslant (|V_O| \cdot |E_O|)/(|V_D| \cdot |E_D|), \tag{1}$$

其中 $|V_O|$ 表示原图中图顶点数量, $|E_O|$ 表示原图中边数量, 而 $|V_D|$ 表示缩略有向图得到的 DAG 中图顶点数量, $|E_D|$ 表示缩略有向图得到的 DAG 中边数量.

3.2.2　边的删除

考虑边的删除. 在动态有向图变化中被删除的每一条边, 在旧图镜像中都必定属于下列两种类型之一: 类型 1, 边的两个图顶点在两个不同的 SCC 中; 类型 2, 边的两个图顶点处于同一个 SCC 中. 不同类型的边的删除, 使得动态有向图产生不同的结构变化. 对这两种类型的情况分别进行说明.

类型 1: 边的两个图顶点在不同的 SCC 中. 通过将每个 SCC 缩略为一个图顶点, 得到有向图的 DAG 构造. 边被删除后, 导致以下两种情况之一: (1) 两 DAG 图顶点间仍存在一条边; (2) 两 DAG 图顶点间不存在任一条边. 但是, 这两种情况对于每个 DAG 图顶点所表示的 SCC 内部结构都没有产生影响, 不会导致相应的 SCC 被重新计算. 例如在图 1 中, 旧图镜像有 3 个 SCC, 分别为 $\{1, 2, 4, 5\}$, 0 和 3 两个孤立 SCC. 当边 (3, 4) 和 (2, 0) 被删除后, 新图镜像的每个 SCC 结构未发生变化, 对 SCC 计算过程不产生影响. 因此, 当类型 1 的边被删除时, 其所在的 SCC 不发生结构变化, 可以直接忽略处理.

类型 2: 边的两端点处于同一个 SCC 中. 类型 2 的边删除将导致不可预测的后果. 以下给出两个实例进行说明.

(1) 如图 2 所示, 边 (5, 2) 被删除后, 原本 SCC 内部 (黑色图顶点) 中的 3 个图顶点 1, 2, 5 之间的回路被打断; 图顶点 1 和 4 仍然保持相互可达的状态. 此时, 旧图镜像中的 SCC 的结构被破坏, 新图镜像中的 SCC 需要重新计算. (2) 如图 3 所示, 当边 (2, 5) 被删除后, 旧图镜像中的 SCC (黑色的图顶点) 仍然保持强连通的状态. 旧图镜像中的 SCC 未发生变化, 理论上不需要重新计算.

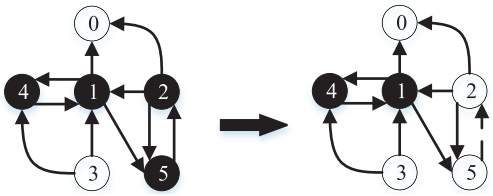

图 2 类型 2: SCC 内部需要重新计算

Figure 2 The second type of deleted edges: recomputation in SCC with structure changes

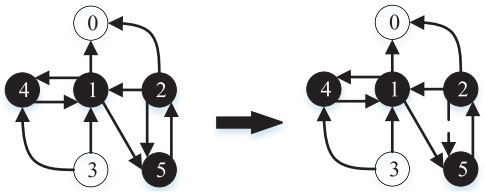

图 3 类型 2: SCC 内部不需要重新计算

Figure 3 The second type of deleted edges: no recomputation in SCC with structure changes

可以观察到, 在动态有向图的变化中, 上述两条类型 2 的边在 SCC 内部处于不同的拓扑位置, 导致 SCC 内部结构产生不同的变化. 实际应用中, 边的删除均为批量处理, 旧图镜像中的 SCC 内部可能发生多条边的删除. 因此, 无法通过识别边的拓扑位置对旧图镜像中 SCC 内部结构的变化进行预测. 在这种情况下, Inc-SCC 将对所有受到类型 2 的影响的 SCC 进行重新计算. 并且, 由于 SCC 内部结构变化不会对 SCC 外部的其他 SCC 的计算产生任何影响, 因此在每个 SCC 内部同时执行 Inc-SCC 以提高并行度.

由上可得 Inc-SCC 的核心思想: 利用增量的信息分析旧图镜像中的各个 SCC, 裁剪其中未发生内部结构变化的部分, 只对类型 2 中边所在的 SCC 进行重新计算以减少冗余计算数据. 同时, 分别在独立的 SCC 划分块中并行执行算法以加快图迭代收敛速度.

3.3 一种面向动态有向图的增量强连通分量算法 Inc-SCC

本小节介绍 Inc-SCC 算法. 如算法 1 所示, 它维护了动态有向图变化前的图镜像的 SCC 计算结果 SCCMAP($G_{t_{i-1}}$), 通过对当前图镜像 G_{t_i} 进行增量计算以获得 SCCMAP(G_{t_i}). SCCMAP(G_{t_i}) 中的数据为当前图镜像中每个图顶点的状态值 (即每个图顶点所在 SCC 的根顶点 ID). Inc-SCC 首先在预处理过程 (preprocess) 中分析动态有向图变化中的增量 increments 以获得需要重新计算的 SCC 集合 G_{new} (第 2 行), 然后在 LocalFBS 阶段和 LocalColoring 阶段中对 G_{new} 中的图数据进行计算并更新 SCCMAP(G_{t_i}) 直到所有需要重新计算的 SCC 都已经得到 (第 3 和 4 行).

Algorithm 1 Incremental SCC algorithm

Input: G_{t_i}, SCCMAP($G_{t_{i-1}}$), increments, $G_{t_{i-1}}$;
Output: SCCMAP(G_{t_i});
1: SCCMAP(G_{t_i}) $\Leftarrow \emptyset$;
2: $G_{\text{new}} \Leftarrow$ Preprocess(SCCMAP($G_{t_{i-1}}$), increments, $G_{t_{i-1}}$);
3: SCCMAP(G_{t_i}) $\Leftarrow$ SCCMAP(G_{t_i}) $\cup$ (G_{new});
4: SCCMAP(G_{t_i}) $\Leftarrow$ SCCMAP(G_{t_i}) $\cup$ LocalColoring(G_{new});

由于动态有向图在变化过程中可能同时发生边的增加和删除, 因此, Inc-SCC 分别维护图镜像之间的两类图增量 (增加的边和删除的边), 然后对这两类图增量分别进行计算, 最后合并这两类计算的结果 (首先处理其中一类图增量, 然后维护相应的计算结果, 接着基于此计算结果处理另一类图增量). 需要注意, Inc-SCC 在算法 2 中首先处理增边情况, 然后处理删边情况 (这两步的顺序可以颠倒, 不影响算法正确性). 如算法 2 所示, Inc-SCC 在预处理过程中分别处理增加和删除的边. Inc-SCC 首先处理增加的边 (第 1∼7 行). 对于增加的边, Inc-SCC 算法直接在 SCC 构成的 DAG (其图顶点是一个 SCC) 上计算新的 SCC. 具体来说, Inc-SCC 首先根据 SCCMAP($G_{t_{i-1}}$) 将图快照 $G_{t_{i-1}}$ 中的图顶点状态值相同的图顶点缩略成一个图顶点以获得图快照 $G_{t_{i-1}}$ 的 DAG (第 1 行). 然后, Inc-SCC 并行检查每一条增加的边 (第 2 行), 当一条增加的边的两个图顶点在两个不同的 SCC 中时, 将这条增加的边加入到 DAG 中的相应的两个图顶点 (每个图顶点是一个 SCC) 之间 (第 3 和 4 行). 需要注意, 如果增加的边不在 DAG 的两个图顶点之间 (即在某 SCC 内部), 那么这条增加的边不会对 DAG 结构产生影响, 因此不需要将这条增加的边加入到 DAG 中. 所有增加的边检查完毕后, Inc-SCC 获得发生边增加后的 DAG (其中可能包含 SCC). Inc-SCC 采用 LocalColoring 方法对发生边增加后的 DAG 进行迭代图顶点状态值传播直到获得所有 SCC, 并且使用此 SCC 计算结果对 SCCMAP($G_{t_{i-1}}$) 进行更新以维护发生增加后的图镜像的 SCC 计算结果 (第 7 行).

Algorithm 2 Preprocess

Input: SCCMAP($G_{t_{i-1}}$); increments; $G_{t_{i-1}}$;
Output: G_{new};

```
1: DAG ⇐ Contract(SCCMAP(G_{t_{i-1}}), G_{t_{i-1}});
2: for each edge e ∈ increments which is added in parallel do
3:     if SCCMAP_{e.src} ≠ SCCMAP_{e.dst} then
4:         DAG ⇐ DAG ∪ e;
5:     end if
6: end for
7: SCCMAP(G_{t_{i-1}}) ⇐ LocalColoring(DAG);
8: G_new ⇐ ∅;
9: for each edge e ∈ increments which is deleted in parallel do
10:    if SCCMAP_{e.src} = SCCMAP_{e.dst} then
11:        OldSCC ⇐ SCCMAP_{e.src};
12:        G_new ⇐ G_new ∪ OldSCC;
13:    end if
14: end for
```

接着, Inc-SCC 处理删除的边 (第 8∼14 行). 对于删除的边, Inc-SCC 只对受到边删除影响的 SCC 进行重新计算以裁除没有发生内部结构变化的 SCC. 具体来说, Inc-SCC 首先将 G_{new} 初始化为空集 (第 8 行). 接着, Inc-SCC 并行地访问每一条被删除的边的源图顶点 (e.src) 和目的图顶点 (e.dst), 它比较这两个图顶点在发生删边前的图镜像的 SCCMAP 中的状态值是否相同 (第 9 和 10 行). 如果相同, 则表明这条边为基本设计思想中类型 2 的被删除的边, 它的两个端点在同一个 SCC 中, 需要对该 SCC 重新计算. 因此, 将该边所在 SCC 加入到 G_{new} 中 (第 11 和 12 行). 需要注意, 如果一条删除的边的两个端点不在同一个 SCC 中 (即类型 1 的被删除边), 这条边不会影响 SCC 内部结构, 因此不需要将 SCC 加入到 G_{new} 中. 当增量检查完毕时, G_{new} 中包含多个独立的数据块 OldSCC, 其中每个数据块 OldSCC 都是受到边删除影响的需要重新计算的一个 SCC. Inc-SCC 利用 SCC 的不相交性, 只在 G_{new} 中的每个 OldSCC 内部进行并行的图顶点状态传播, 而不使其内部图顶点状态值传递出该

OldSCC, 减少它们对其他 OldSCC 的影响, 提升算法效率.

我们发现 OldSCC 内部的图顶点具有以下特点: (1) 在各个 OldSCC 内部, 其根顶点本身的状态值 (各个 OldSCC 中其余图顶点的状态值在发生删边前的图镜像中通常都收敛到对应 OldSCC 在发生删边前的图镜像中的根顶点的状态值) 对整个 OldSCC 中图顶点状态传递速度通常起着最重要的作用; 另一方面, 在每个 OldSCC 内部通常只有少量的边删除 (这意味着此 OldSCC 中大量图顶点可能仍然与此 OldSCC 在发生删边前的图镜像中的根顶点具有相同的状态值). 因此, 我们提出了启发式优化方法, 它首先找出并清除每个 OldSCC 在发生删边前的图镜像中的根顶点所在 SCC 中的图顶点 (记作 LocalFBS 阶段), 然后对剩余图部分迭代进行图顶点状态值传播直至获得所有 SCC (记作 LocalColoring 阶段).

如算法 3 所示, 在 LocalFBS 阶段中, Inc-SCC 通过利用 OldSCC 内部图顶点的特点以减少每个 OldSCC 内部的冗余图顶点状态传播量 (即减少图顶点更新次数). 具体来说, 它首先将不需要重新计算的 SCC 中图顶点状态值加载到 SCCMAP(G_{t_i}) 中 (第 1 行), 然后在每个数据块 OldSCC 内部进行并行的图顶点状态传播. Inc-SCC 维护动态有向图变化过程中各 SCC 的根顶点, 然后选择每个根顶点作为各个数据块内的枢纽点 (第 3 行). 接着, Inc-SCC 采用并行宽度优先搜索 (breadth first search, BFS) 来寻找根顶点的前向可达集 D 和后向可达集 P. 其中, Inc-SCC 首先从根顶点出发沿着出边方向对数据块 OldSCC 内部进行一次前向扫描 (forward sweep, FS), 每个访问到的与根顶点状态值相同的图顶点都被加入到前向可达集中 (第 4 行); 然后, Inc-SCC 从根顶点出发沿着入边方向对前向可达集中的图顶点集合进行一次后向扫描 (backward sweep, BS) 以直接找出前向可达集和后向可达集的交集, 也就是动态有向图变化过程中未发生状态值变化的图顶点集合 (第 5∼7 行). Inc-SCC 将找到的 SCC 从 G_{new} 中移除 (第 8 行), 然后进入到 LocalColoring 阶段.

Algorithm 3 LocalFBS

Input: G_{new};
Output: SCCMAP(G_{t_i}); G_{new};
1: SCCMAP(G_{t_i}) $\Leftarrow$ SCCMAP($G_{t_{i-1}}$)\G_{new};
2: **for** all OldSCC in G_{new} in parallel **do**
3: Select root $\in$ OldSCC;
4: $D \Leftarrow$ FS(OldSCC, root);
5: $P \Leftarrow$ BS(OldSCC, root);
6: NewSCC(root) $\Leftarrow D \cap P$;
7: SCCMAP(G_{t_i}) $\Leftarrow$ SCCMAP(G_{t_i}) $\cup$ NewSCC(root);
8: $G_{\text{new}} \Leftarrow G_{\text{new}}$\NewSCC(root);
9: **end for**

如算法 4 所示, 在 LocalColoring 阶段中, Inc-SCC 在每个数据块 OldSCC 的剩余图顶点中并行地进行迭代图顶点状态值的传播以利用底层处理器的高并发性. 具体来说, Inc-SCC 首先初始化剩余图顶点的状态值集合 colors, 通常为图顶点本身 ID (第 2 行). 然后使每个图顶点向它的邻居顶点迭代传值 (第 4∼10 行). 在进行迭代传播时, Inc-SCC 根据用户定义规则来更新每个图顶点状态值 colors (第 5 和 6 行) 直至所有图顶点状态值不再变化. 此时, 数据块 OldSCC 内部被分为多个划分块, 每个划分块中图顶点具有相同的 colors 值. 每个相同 colors 的划分块中包含一个在迭代过程中的 colors 值未发生变化的图顶点 (即 root 图顶点) (第 11 行). 以 root 图顶点为起点进行反向扫描以找出相应 SCC (第 12 和 13 行). 然后, Inc-SCC 将找到的 SCC 从 G_{new} 中移除 (第 14 行), 反复进行上述操作直至 G_{new} 为空, 此时迭代停止, SCCMAP(G_{t_i}) 中的图顶点状态即为新图镜像的增量强连通分量算法的计算结果.

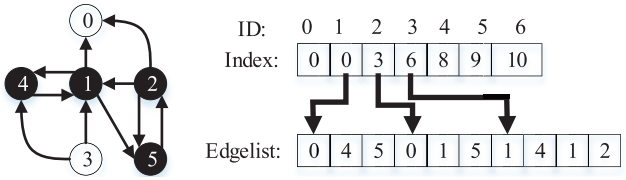

图 4 例图的 CSR 格式

Figure 4 The CSR format of an evolving graph

Algorithm 4 LocalColoring

Input: G_{new};
Output: SCCMAP(G_{t_i});

```
1: while G_new ≠ ∅ do
2:   for all OldSCC ∈ G_new in parallel do
3:     Init colors;
4:     while colors have changed do
5:       for edge e ∈ OldSCC in parallel do
6:         if colors[e.dst] > colors[e.src] then
7:           colors[e.dst] ⇐ colors[e.src];
8:         end if
9:       end for
10:    end while
11:    for all vertex root = colors[root] in parallel do
12:      NewSCC(root)⇐ BS(OldSCC, root);
13:      SCCMAP(G_{t_i}) ⇐ SCCMAP(G_{t_i}) ∪ NewSCC(root);
14:      G_new ⇐ G_new\NewSCC(root);
15:    end for
16:  end for
17: end while
```

4 Inc-SCC 算法的实现

本节介绍 Inc-SCC 算法的实现. 首先介绍 Inc-SCC 实现需要的数据结构, 其次说明算法并行的实现过程, 最后给出一种避免数据冲突的无锁方法.

4.1 数据结构

在 Inc-SCC 算法实现中, 本文使用了 3 种不同数据结构用于图数据表示和中间结果维护. 第 1 种数据结构为动态有向图的最新图镜像的表示. 本文使用压缩稀疏矩阵行 (compressed sparse row, CSR) 来构建动态有向图的最新图镜像. 其中, 边表 (edgelist) 紧凑地存储有向图中所有的边, 索引表 (index) 中的每一个表项都存储着相应编号的图顶点的邻接边在边表中的初始位置. 例如在图 4 中, 图顶点 1 的索引项为边表中的 0 位置, 顶点 2 的索引项为边表中的 3 位置, 因此顶点 1 的 3 个邻居顶点为 $\{0, 4, 5\}$. 由于 Inc-SCC 算法中需要同时进行前向扫描和后向扫描, 即分别沿着图顶点的邻接出边方向和邻接入边方向进行并行 BFS, 因此在使用 CSR 表示动态有向图时, Inc-SCC 分别建立了出边、入边的索引和边表以加快总体扫描速度. 对于一个有 $|V|$ 个图顶点和 $|E|$ 条边的动态有向图的最新图镜像, 需要至少 $2|V|+2$ 和 $2|E|$ 的空间来保存索引和边表. 在保证数据存储紧凑性的同时, CSR 格式还提供一定的空间局部性以加快数据访问. 第 2 种数据结构为 VISIT, 用于记录所有图顶点的访

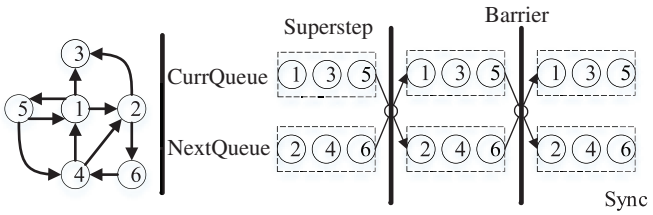

图 5 同步迭代

Figure 5 Synchronous iteration

问状态. 本文使用一个布尔型的数组对其进行实现. 在算法执行过程中, 将图顶点 v_i 的 VISIT 设置为 1 来表示该图顶点已经被访问. 在算法的不同执行过程中, VISIT 具有不同的访问模式. 例如, 在 LocalFBS 阶段中, Inc-SCC 从起点出发进行 BFS, 只对每个图顶点的 VISIT 进行一次更新操作. 然而在 LocalColoring 阶段中 Inc-SCC 反复迭代进行图顶点状态值传播, VISIT 被多次写访问. 第 3 种数据结构用于保存迭代过程的中间值, 实现为两个布尔型的数组: isCurr 和 isNext. 通过对所有图顶点进行标记, 它们分别索引当前迭代所需数据和下一轮迭代所需数据.

4.2 同步迭代模型

Inc-SCC 采用了同步并行计算模型 (bulk synchronous parallel, BSP). 如图 5 所示, 该迭代模型分为超步和路障两个阶段. 在超步阶段中, 各个线程独立执行各自的任务. 当所有线程都已经完成当前任务后, 进入到路障阶段. 在路障阶段中, 所有线程相互进行中间数据的整合和交换, 为下一轮迭代准备数据.

Inc-SCC 算法采用并行 BFS 来实现 LocalFBS 阶段和 LocalColoring 阶段中的状态值传播, 具体如下: 在初始状态时, 所有需要的数据结构被加载到内存中. 当前迭代的处理数据队列 CurrQueue 和下轮迭代的待处理数据队列 NextQueue 被清空, 相应的两个用于索引的数组 isCurr 和 isNext 也被清零. 选取并行 BFS 的起点 v_0, 将其加入到 CurrQueue 中, 将 v_0 的 isCurr 设置为 1. 同时将 v_0 的 VISIT 设置为 1 以避免该图顶点被重复访问. 迭代开始后, 首先进入到超步阶段. 每个线程根据图划分策略获取一个划分块中的数据, 并行地遍历所有邻接的图顶点. 算法只访问 VISIT 为 0 的图顶点, 被访问的新图顶点的 VISIT 被设置为 1. 并且设置该图顶点相应的 isNext 为 1, 将其加入到 NextQueue 中. 当 CurrQueue 中所有图顶点都已经处理完毕, 迭代过程进入到路障阶段. 清空 CurrQueue 和相应的索引 isCurr, 然后交换两个队列及其索引. 当下轮迭代到来时, NextQueue 将作为新的迭代过程的数据队列. 在最后一轮迭代开始时, CurrQueue 中已经没有数据可以访问. 此时, VISIT 中所有的数据项都已经被标记为 1, 迭代收敛. 同步迭代模型易于控制算法的并行过程, 能够解决算法执行中的数据相关问题以保证算法执行的正确性.

在同步迭代过程中, 经常出现一个图顶点同时被多个其他图顶点访问的情况. 如果不使用锁控制共享变量, 此类情况在图顶点状态传播中将造成严重的数据冲突. 例如, 在图 6 中, 图顶点 4 和 3 都向图顶点 0 写入各自的当前状态. 在并行中, 同一个图顶点被访问的先后顺序无法确定. 因此, 可能导致正确状态被错误状态覆盖, 影响到算法的正确性. 为了保证算法执行的正确性, 许多算法 [12,13] 使用锁控制并行. 但是锁的使用常常导致大量线程进入等待状态, 降低并行计算的性能. 因此, 本文使用无锁方法来避免数据冲突. 如图 6 所示, 当前迭代中的一个图顶点, 如果其至少有一个邻接的图顶点被其成功染色时, 那么将其加入到下轮迭代的数据队列中. 在新的迭代中, 该图顶点会再次检查其邻

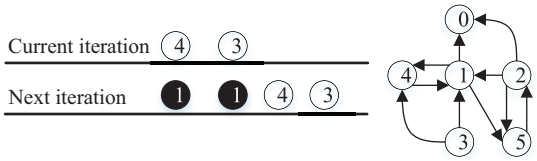

图 6　解决数据冲突的无锁方法

Figure 6　Lock-free solution for data conflicts

表 1　图数据集

Table 1　Graph datasets

Dataset	Nodes	Edges	SCCs	Nodes in the largest SCC
Soc-epinions1	75888	508837	42185	32223 (42.5%)
Email-euall	265214	420025	231000	34203 (12.9%)
Web-stanford	281903	2312497	29919	150527 (53.4%)
Web-notreDame	325729	1497134	203609	53968 (16.6%)
Web-Berstan	685230	7600595	109407	334856 (48.9%)
Wiki-Talk	2394385	5021410	2281879	111881 (4%)

接图顶点是否已经获得正确的值. 因为状态传播具有单调性, 所以这种方法能够避免数据冲突的发生. 无锁方法减少并行过程中的线程中断以提升算法的并行度.

5　实验与结果

本节对该增量算法 Inc-SCC 进行实验测试. Slota 等 [15] 提出的 Coloring 算法 (或称作 SCO 算法) 是目前最好的并行 SCC 算法, 其能够保证图顶点状态迭代传播的细粒度并行以加快 SCC 的计算效率. 在多核环境下时, 其计算 SCC 的性能远超过串行 Tarjan 算法. 为了验证增量算法 Inc-SCC 的性能提升, 本文选取 SCO 算法作为实验的基准算法.

5.1　实验环境

实验在单机环境下进行. 该单机节点的主存容量为 8 GB, 并且具有一个主频为 3.3 GHz 的 Intel Xeon E3-1230 v2 的 4 核处理器. 其操作系统为 Linux, 相应的内核版本为 3.19.0-15-generic. 在该单机节点上最多可分配 8 个线程. 该增量算法 Inc-SCC 及基准算法均以编程语言 C++ 实现, 使用的编译器版本为 g++ 4.9.2. 实验选取了 6 个有向图用于测试. 这些图可以在 SNAP [29] 上获得. 表 1 中给出了每个图的几项主要信息: 图顶点数, 有向边数, SCC 个数, 最大 SCC 中图顶点个数及最大 SCC 占有向图的比例. 上述静态有向图都作为动态有向图变化中初始的图镜像. 需要注意, 本文实验主要测试删边情况, 以评估本方法相对于现有方法至少能提升多少性能. 由于本方法对于增加边的情况会如式 (1) 那样相对删边情况获得更好的加速比 (因为本方法对于删边情况像现有方法一样需要对原图进行处理, 虽然本方法可以利用动态有向图特性减少冗余数据访问和处理). 实验以相同的概率对整个图镜像中的所有边进行随机删除, 通过设置参数控制每次变化的增量边占整个图镜像的比例, 即增量的尺寸. 在以下各个实验中, 每个实验都能保证正确的计算结果. 并且, 对于每一组数据, 都进行了 30 次的测试, 求取其平均值.

表 2 增量占图镜像总边数的 0.5% 时, Inc-SCC 各执行部分的时间开销

Table 2 Time cost of each part in Inc-SCC when the ratio of changed edges of the entire graph snapshot is 0.5%

Dataset	Preprocess (ms)	Process (ms)	LocalFBS (ms)	LocalColoring (ms)
Soc-epinions1	1.135	5.708	5.448	0.260
Email-euall	2.628	4.502	4.261	0.241
Web-stanford	2.628	4.502	4.261	0.241
Web-notredame	3.117	11.438	10.380	1.058
Web-Berstan	6.946	55.542	48.880	6.662
Wiki-Talk	19.199	68.166	64.032	3.134

表 3 Inc-SCC 相对基准算法的加速比

Table 3 Speedup of Inc-SCC over baseline

Dataset	SCO (ms)	Inc-SCC (ms)	Speedup
Soc-epinions1	46.746	5.708	8.190
Email-euall	28.100	4.502	6.242
Web-stanford	120.951	35.729	3.385
Web-notredame	51.147	11.438	4.472
Web-Berstan	160.622	55.542	2.897
Wiki-Talk	825.124	68.166	12.104

5.2 实验结果及分析

实验 1 测试 Inc-SCC 中各部分的时间开销. 由于 Inc-SCC 中预处理过程可以与动态有向图的数据更新过程相互重叠从而忽略其时间开销, 因此本文不考虑预处理过程的开销对算法的影响. 将动态有向图变化时的增量的尺寸设置为占图镜像中边数总量的 0.5%. 表 2 展示了实验结果. 数据显示, 对于参与实验的 6 个不同的数据集, 在 Inc-SCC 实际执行计算的两个阶段中, LocalFBS 阶段的平均时长占总运行时长的 95% 以上, LocalColoring 阶段只占时间开销中极小的一部分. 实验 1 表明, 受到影响的 SCCs 内部的大部分图顶点仍然与根顶点属于同一个 SCC, 在 LocalFBS 阶段中可提取出这些图顶点. 剩余的少量图顶点经过图迭代染色后, 算法很快就收敛结束. 这符合本文对受到影响的 SCCs 内部结构变化的分析, 显示出所提出的启发性算法能加快图迭代的收敛, 提升算法性能.

实验 2 测试了与基准算法相比, Inc-SCC 的性能提升. 此实验仍然将增量的尺寸设置为占图镜像中边数总量的 0.5%. 表 3 中展示了实验结果. 结果显示, 对于参与实验的 6 个不同的数据集, Inc-SCC 相对基准算法均有明显的性能提升. 其中, 在动态有向图 Wiki-Talk 中达到了 12 倍的最大加速比. 而且, 所有 6 个数据集的加速比也在 2.9 倍以上. 此实验验证了 Inc-SCC 的基本设计思想的可行性. 通过裁剪冗余数据量这一策略, Inc-SCC 能够显著减少动态有向图变化过程中的冗余计算, 缩短强连通分量算法的运行时间.

在实验 3 中, 选取了 4 个动态有向图来测试 Inc-SCC 的性能提升与增量的尺寸的关系. 增量占图镜像中边数总量的比例被设置为 7 个数值, 分别为 0.05%, 0.1%, 0.25%, 0.5%, 1%, 2.5% 和 5%. 图 7 展示了各个数据集上 Inc-SCC 的性能提升随着增量的尺寸变化的趋势图. 数据显示, 当增量占图镜像边数总量的比例小于的 0.5% 时, Inc-SCC 的性能提升能够维持在高水平. 但是, 随着增量尺寸的不断增大, Inc-SCC 的性能缓慢下降, 最终趋于稳定. Inc-SCC 的性能提升随着增量尺寸而变化的原

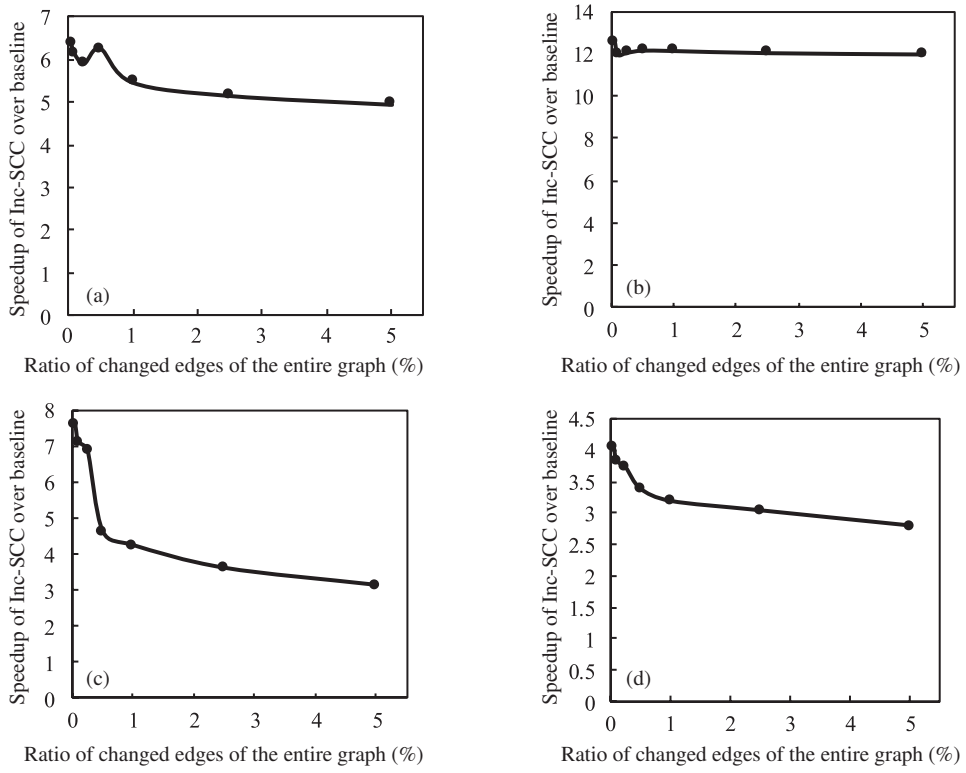

图 7　在不同数据集中, Inc-SCC 随增量占有向图总边数的比例的变化的性能曲线

Figure 7 Performance curves of Inc-SCC when the ratio of changed edges of the entire graph is different on four different datasets. (a) Email-EuAll; (b) Wiki-Talk; (c) Web-NotreDame; (d) Web-Stanford

因如下: (1) 当增量占图镜像的比例达到一定程度时, 受影响的 SCC 的数量趋于饱和. 因此, 裁剪策略的性能提升达到瓶颈. 但是 Inc-SCC 的性能提升仍然能维持在 2.8 倍以上. (2) 由于被删除的边增加, SCC 内部结构不再表现出幂律分布特性. 更多的图数据进入 LocalColoring 阶段, 使得迭代过程中的数据冲突增加, 降低迭代效率.

我们对具有不同最大 SCC 占比的数据集进行了测试, 其结果如图 8 所示, 其中图中横轴表示最大 SCC 中图顶点占整个有向图中图顶点的比例, 纵轴表示 Inc-SCC 相比基准算法的加速比. 实验结果显示, 在通常情况下本方法性能上限会随着最大 SCC 拥有的图顶点占整个图中所有图顶点的比例的增加, 而逐渐减少. 因为 Inc-SCC 算法主要通过裁除无需重新计算的 SCCs, 以减少冗余图数据加载和处理. 如果有向图中的最大 SCC 拥有的图顶点数量占整个图中所有图顶点数据的比例等于 1, 那么本文所提算法的性能提升将最小.

实验 4 测试增量尺寸占图镜像边数总量的比例为 1% 时, 有向图 Wiki-Talk 和 Web-NotreDame 的连续变化下的性能, 图 9 中横轴表示动态有向图连续变化的次数. 实验结果显示 Inc-SCC 在 Wiki-Talk 上的表现较为稳定, 在 Web-NotreDame 上却出现了性能的波动. 这是因为在处理 Web-NotreDame 时, Inc-SCC 算法在 LocalFBS 阶段中找出并清除了少量图顶点, 大量图顶点进入 LocalColoring 阶段, 影响了 Inc-SCC 的性能稳定性. 但是, 在该数据集上 Inc-SCC 的平均性能提升仍然达到 4 倍.

实验 5 测试增量算法 Inc-SCC 的可扩展性. 实验的线程数设置为 1 至 8 个, 增量尺寸占图镜像边数总量的比例为 0.5%. 在图 10 中, 3 条折线分别代表性能提升理想值 Ideal, 基准 SCO 算法加速

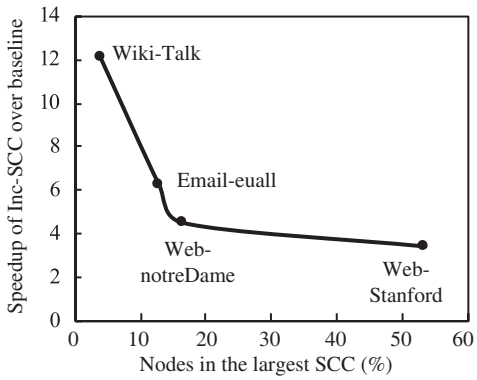

图 8 Inc-SCC 随最大 SCC 中图顶点占整个有向图中图顶点比例的变化的性能曲线

Figure 8 Performance curves of Inc-SCC when the ratio of nodes in the largest SCC of the entire graph is different

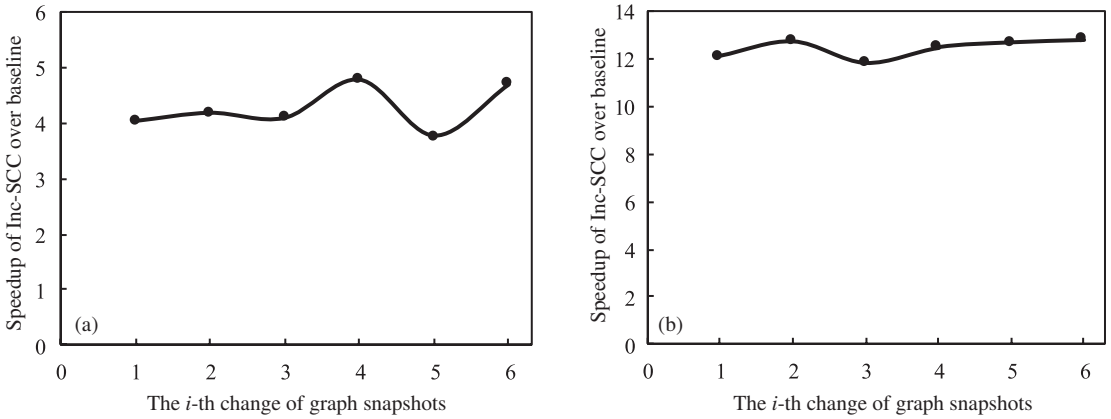

图 9 在不同数据集中, 动态有向图的连续变化时 Inc-SCC 的性能曲线

Figure 9 Performance curves of Inc-SCC with graphs continuing changing on two different datasets. (a) Web-NotreDame; (b) Wiki-Talk

比, 增量算法 Inc-SCC 加速比. 加速比表示算法在多线程时相对于单线程的性能提升. 因为实验中处理器为 4 核, 所以理想加速比的上限为 4 倍. 在这 6 个有向图中, Inc-SCC 与基准 SCO 算法的可扩展性都低于理想值. 在 Web-notreDame, Email-EuAll 和 Wiki-Talk 这 3 个最大 SCC 占比较小的图中, Inc-SCC 与基准 SCO 算法的可扩展性基本一致. 而在其他 3 个最大 SCC 占比较大的图中, Inc-SCC 的可扩展性要差于基准 SCO 算法. 其原因如下: (1) 在 SCC 内部进行扫描和状态传播中, 线程数量增加, 致使冗余计算增加, 降低算法并行度. (2) Inc-SCC 使用 hash 方式对数据进行随机划分. SCC 内部数据局部性不佳, 导致额外数据访问开销.

6 结论

本文提出了面向动态有向图的增量强连通分量算法 Inc-SCC. 它通过裁剪策略, 减少动态有向图处理过程中的冗余数据, 降低计算开销; 提出了启发性优化算法以提高迭代收敛速度. 实验结果显示, Inc-SCC 可以用于实时响应有向图持续性动态变化. 并且当整个有向图的边变化比例为 5% 时, 本方法相对于现有算法的加速比可达 2.8 到 12 倍, 当整个有向图的边变化比例为 0.5% 时, 本方法相对于现

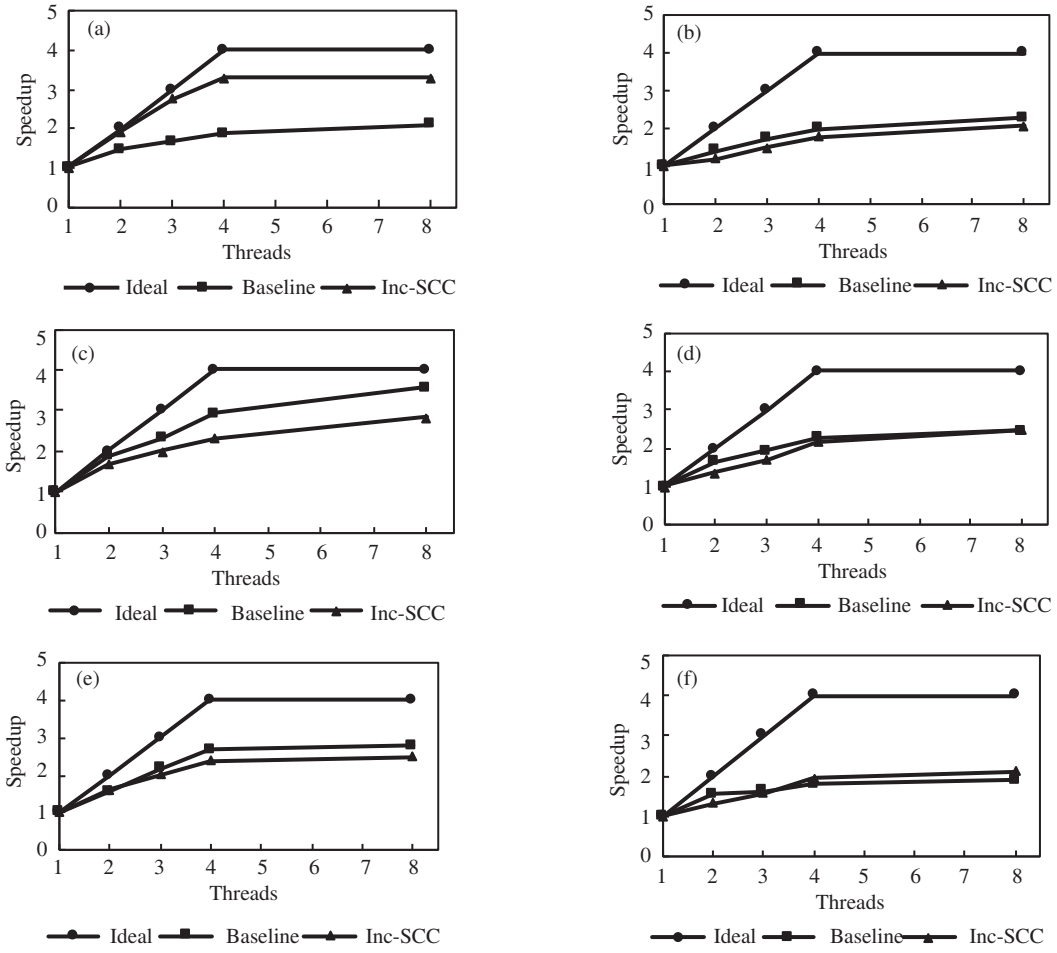

图 10　在不同数据集中, Inc-SCC 的扩展性

Figure 10 The scalability of Inc-SCC on six different datasets. (a) Email-EuAll; (b) Soc-Epinions1; (c) Web-Berstan; (d) Web-NotreDame; (e) Web-Stanford; (f) Wiki-Talk

有算法的加速比可达 2.9 到 12 倍. 在未来的工作中, 面向动态有向图的增量强连通分量算法 Inc-SCC 仍然存在很多优化空间: (1) 需要新型数据结构和动态图划分策略以支持动态有向图的快速更新并保证良好的数据局部性. (2) 研究表明, 异步迭代方法的收敛速度要比同步迭代方法更快. 由于 Inc-SCC 使用同步迭代方法, 因此可以采用异步方法对 Inc-SCC 进行优化. (3) 将该裁剪策略与现有的一些辅助方法 (例如 TRIM [6] 算法) 结合以进一步优化 Inc-SCC 的效率.

参考文献

1 Leskovec J, Kleinberg J, Faloutsos C. Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, 2005. 177–187

2 Zhang Y, Liao X, Shi X, et al. Efficient disk-based directed graph processing: a strongly connected component approach. IEEE Trans Parallel Distrib Syst, 2018, 29: 830–842

3 Tarjan R. Depth-first search and linear graph algorithms. SIAM J Comput, 1972, 1: 146–160

4 Aho A V, Hopcroft J E, Ullman J. Data Structures and Algorithms. Boston: Addison-Wesley, 1983. 1–983

5 Orzan S M. On distributed verification and verified distribution. Dissertation for Ph.D. Degree. Amsterdam: Free University of Amsterdam, 2004

6 McLendon Iii W, Hendrickson B, Plimpton S J, et al. Finding strongly connected components in distributed graphs. J Parallel Distrib Comput, 2005, 65: 901–910

7 Hellings J, Fletcher G H L, Haverkort H. Efficient external-memory bisimulation on DAGs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, 2012. 553–564

8 Sarwat M, Sun Y. Answering location-aware graph reachability queries on geosocial data. In: Proceedings of the 33rd International Conference on Data Engineering, San Diego, 2017. 207–210

9 Yildirim H, Chaoji V, Zaki M J. Grail: scalable reachability index for large graphs. Proc VLDB Endow, 2010, 3: 276–284

10 Fan W, Li J, Ma S, et al. Graph homomorphism revisited for graph matching. Proc VLDB Endow, 2010, 3: 1161–1172

11 Fleischer L K, Hendrickson B, Pinar A. On identifying strongly connected components in parallel. In: Prceedings of the 14th International Parallel and Distributed Processing Symposium, Cancun, 2000. 505–511

12 Zhang Z W, Yu J X, Qin L, et al. I/O efficient: computing SCCs in massive graphs. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, 2013. 181–192

13 Barnat J, Chaloupka J, van de Pol J. Improved distributed algorithms for SCC decomposition. Electron Notes Theor Comput Sci, 2008, 198: 63–77

14 Hong S, Rodia N C, Olukotun K. On fast parallel detection of strongly connected components in small-world graphs. In: Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, 2013. 1–11

15 Slota G M, Rajamanickam S, Madduri K. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In: Proceedings of the 28th IEEE Parallel and Distributed Processing Symposium, Phoenix, 2014. 550–559

16 Bloemen V, Laarman A, van de Pol J. Multi-core on-the-fly SCC decomposition. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Barcelona, 2016. 1–12

17 Ding L L, Li Z D, Ji W T, et al. Reachability query of large scale dynamic graph based on improved Huffman coding. Acta Electron Sin, 2017, 45: 359–367 [丁琳琳, 李正道, 纪婉婷, 等. 基于改进哈夫曼编码的大规模动态图可达查询方法. 电子学报, 2017, 45: 359–367]

18 Ntoulas A, Cho J, Olston C. What's new on the web? the evolution of the web from a search engine perspective. In: Proceedings of the 13th International Conference on World Wide Web, New York, 2004. 1–12

19 Broder A, Kumar R, Maghoul F, et al. Graph structure in the Web. Comput Netw, 2000, 33: 309–320

20 Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM, 2008, 51: 107–113

21 Wang J, Zhang L, Wang P Y, et al. Memory system optimization for graph processing: a survey. Sci Sin Inform, 2019, 49: 295–313 [王靖, 张路, 王鹏宇, 等. 面向图计算的内存系统优化技术综述. 中国科学: 信息科学, 2019, 49: 295–313]

22 Zhang Y, Liao X, Jin H, et al. FBSGraph: accelerating asynchronous graph processing via forward and backward sweeping. IEEE Trans Knowl Data Eng, 2018, 30: 895–907

23 Wang L, Yang X J, Dai H D. Scratchpad memory allocation for arrays in permutation graphs. Sci China Inf Sci, 2013, 56: 052119

24 Jin H, Yao P C, Liao X F. Towards dataflow based graph processing. Sci China Inf Sci, 2017, 60: 126102

25 Dai G, Huang T, Chi Y, et al. Foregraph: exploring large-scale graph processing on multi-FPGA architecture. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, 2017. 217–226

26 Ju X, Williams D, Jamjoom H, et al. Version traveler: fast and memory-efficient version switching in graph processing systems. In: Proceedings of the 2016 USENIX Annual Technical Conference, Denver, 2016. 523–536

27 Zhang L X, Wang W P, Gao J L, et al. Pattern graph change oriented incremental graph pattern matching. J Softw, 2015, 26: 2964–2980 [张丽霞, 王伟平, 高建良, 等. 面向模式图变化的增量图模式匹配. 软件学报, 2015, 26: 2964–2980]

28 Semertzidis K, Pitoura E, Lillis K. TimeReach: historical reachability queries on evolving graphs. In: Proceedings of the 18th International Conference on Extending Database Technology, Brussels, 2015. 121–132

29 Jure L, Andrej K. SNAP Datasets: Stanford Large Network Dataset Collection. 2014. http://snap.stanford.edu/data

An efficient incremental strongly connected components algorithm for evolving directed graphs

Xiaofei LIAO[1,2,3,4], Yicheng CHEN[1,2,3,4], Yu ZHANG[1,2,3,4*], Hai JIN[1,2,3,4], Haikun LIU[1,2,3,4] & Jin ZHAO[1,2,3,4]

1. *National Engineering Research Center for Big Data Technology and System, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
2. *Service Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
3. *Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
4. *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074*, China*

* Corresponding author. E-mail: zhyu@hust.edu.cn

Abstract The strongly connected components (SCC) algorithm can contract a directed graph into a directed acyclic graph and is widely used in directed graph analysis applications, such as reachability queries. A variety of SCC algorithms for static directed graphs have been proposed but such algorithms require non-negligible runtime overheads to repeatedly perform computations on an entire graph in response to the frequent changes in the evolving directed graphs that are ubiquitous in the real world. In general, evolving directed graphs are often evolving with minor changes (less than 5%). It allows us to compute SCC in an evolving directed graph on the basis of incremental computations in order to reduce the response time. This paper proposes Inc-SCC, an efficient incremental SCC algorithm for evolving directed graphs, reducing the data access and computation overhead of the algorithm by eliminating unnecessary computations, and using the disjoint feature of SCC for parallel processing to improve the performance of the SCC algorithm. We propose a heuristic optimization method to further speed up the convergence of Inc-SCC. Experiments show that Inc-SCC can be used to enhance the timeliness for evolving directed graphs. When the number of the changed edges of the entire directed graph is 5%, the speedup of Inc-SCC over the existing algorithm is from 2.8 to 12 times. When the number of the changed edges of an entire directed graph is 0.5%, the speedup of Inc-SCC over the existing algorithm is from 2.9 to 12 times.

Keywords strongly connected components, evolving directed graph, incremental computation, convergence, directed acyclic graph

Xiaofei LIAO received his Ph.D. degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is currently a professor at the School of Computer Science and Engineering at HUST. His research interests include system virtualization, system software, and cloud computing.

Yicheng CHEN received his B.S. degree in computer science from the Huazhong University of Science and Technology (HUST), China, in 2017. He is currently a graduate student at the School of Computer Science at HUST. His specialty is system software and architecture.

Yu ZHANG received his Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST) in 2016. He is currently a postdoctor at the School of Computer Science at HUST. His research interests include big data processing and system software and architecture. His current research mainly focuses on application-driven big data processing and optimizations.

Hai JIN is a Cheung Kung Scholars chair professor of computer science and engineering at the Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. degree in computer engineering from HUST in 1994. He worked at the University of Hong Kong between 1998 and 2000, and he was a visiting scholar at the University of Southern California between 1999 and 2000. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.