基于 KST 索引的最大连通 Steiner 分量查询算法

陈子阳[1),2)] 陈伟[2),3)] 贾勇[2)] 周军锋[4)]

[1)](上海立信会计金融学院 信息管理学院，上海 201620)

[2)](燕山大学 信息科学与工程学院，河北 秦皇岛 066004)

[3)](河北环境工程学院 信息工程系，河北 秦皇岛 066102)

[4)](东华大学 计算机科学与技术学院，上海 201620)

摘 要 查找图的连通分量在生物信息学领域有着重要应用价值，其中的关键问题之一是查询最大连通 Steiner 分量(SMCC). 针对已有最大连通 Steiner 分量查询方法中存在的查询效率低的问题，首先提出利用 *k*-edge 连通分量与(*k*+1)-edge 连通分量之间的包含关系建立顶点集合的分层索引 KST，和现有的专用索引相比，索引规模得到了缩减；然后提出基于 KST 索引的 SMCC 查询算法，以及具有顶点数量限制的 $SMCC_L$ 查询算法. 和已有方法中索引的是图中顶点不同，KST 索引中维护的是顶点集合的包含关系，其优点在于将已有方法在遍历过程中的一次一顶点的查询方式转换为更高效的一次一集合的查询方式，显著减少了需要访问的索引点数量，极大提升了查询处理的效率；最后，基于 15 个真实数据集进行实验测试，从不同角度验证了所提方法的的高效性.

关键词 无向图; *k*-edge 连通分量; 最大连通 Steiner 分量; 索引; 最大生成树

The KST Index based Querying Algorithm for Steiner Maximum-Connected Components

CHEN Zi-Yang[1),2)] CHEN Wei[2),3)] JIA Yong[2)] ZHOU Jun-Feng[4)]

[1)](School of Information and Management, Shanghai Lixin University of Accounting and Finance, Shanghai 201620)

[2)](School of Information Science and Engineering, Yanshan University, Qinhuangdao 066004)

[3)](Department of Information Engineering, Hebei University of Environmental Engineering, Qinhuangdao 066102)

[4)](School of Computer Science and Technology, Donghua University, Shanghai 201620)

Abstract Given a graph *G*, and a set of query nodes *q*, the Steiner Maximum-Connected Component (SMCC) is a connected subgraph of *G* with the maximum connectivity and the maximum number nodes which contains *q*. And $SMCC_L$ is the SMCC of *G* with the constraint of the number of nodes *L*. Finding SMCC or $SMCC_L$ is one of the fundamental operations in graph data processing, and is one of the hot issues, which has attracted much attention in the research field and can be applied in many applications, including bio-informatics, etc. Existing methods for SMCC and $SMCC_L$ query proceessing are mainly classified into the following two categories: (1)

本课题得到国家自然科学基金(No. 61472669, 61572421, 61873337)资助. 陈子阳,男, 1973年生，博士，教授/博导，上海立信会计金融学院特聘教授，计算机学会(CCF)会员，主要研究领域为数据库理论与技术. E-mail: zychen@ysu.edu.cn. 陈伟（**通讯作者**），女, 1980年生，博士生，副教授，计算机学会(CCF)会员，主要研究领域为图数据查询处理. E-mail: chenwei@ysu.edu.cn. 贾勇，男, 1990年生，硕士生，主要研究领域为图数据查询处理. E-mail: 1054699501@qq.com. 周军锋，男, 1977年生，博士，教授/博导，计算机学会(CCF)会员，主要研究领域为XML和半结构化数据查询技术. E-mail: zhoujf@dhu.edu.cn

Finding SMCC or SMCC$_L$ based on existing k-edge connected component algorithms which decrease the value of k from $|V|$ (the number of nodes in graph G) to 1 in turn, calculate all k-edge connected components in G, then the first k-edge connected component containing query q is SMCC of q, and the first k-edge connected component containing q whose number of nodes is greater than or equal to L is SMCC$_L$ of q. The shortcomings of such methods are that the search process needs to traverse graph G many times and the computation cost is high when the size of graph is large. (2) Finding SMCC or SMCC$_L$ based on special index which constructs the MST(Maximum Spanning Tree) index of G offline. When processing the query, it firstly calculates the connectivity of q, traverses the MST with any node in q as the start node, and only accesses the nodes corresponding to the edges satisfying specific conditions until all nodes in q are covered, then the visited nodes are the nodes in the SMCC of q. The shortcomings of such methods are that querying SMCC needs expensive traversal operations, each step of traversal can obtain at most one useful node (namely one-node-a-step). When the number of nodes in SMCC is large, the number of nodes that need to be accessed in the index tree will increase. Moreover, when query requests are executed frequently, the system load will increase rapidly. Considering that existing approaches on querying SMCC and SMCC$_L$ suffer from inefficiency, we first propose a KST index, which maintains the relationship between k-edge connected subgraph and (k+1)-edge connected subgraph. Compared with existing methods, the index size is largely reduced. Based on the KST index, we propose a new SMCC finding algorithm (namely SMCC-KST), and propose another algorithm to find SMCC with size constraint (namely SMCC$_L$-KST). Compared with existing index which maintains the relationship of nodes in the original graph, in our KST index, each index node represents a nodes set. The benefit is that in this way, we can improve existing approaches from one-node-a-step when traversing on the index to one-set-a-step, such that significantly reduces the number of visited index nodes, while at the same time improves the query efficiency. We conduct extensive experimental study on 15 real datasets. The experimental results show that our approaches perform much better than existing ones when querying SMCC and SMCC$_L$.

Key words undirected graph; k-edge connected components; steiner maximum-connected component; index; maximum spanning tree

1 引言

生物信息学综合运用数学、计算机科学和生物学的各种工具，通过对生物信息的获取、处理、存储、分析和解释等来阐明和理解大量数据所包含的生物学意义[1-3]. 蛋白质相互作用网络、基因调控网络、代谢路径等生物网络的飞速发展使得生物学数据库规模不断增长，也给数据挖掘技术带来了机遇和挑战. 由于这些数据量大且复杂度高，通常使用图数据结构来描述其中的复杂关系，从而生物学中的许多计算问题最后都是依赖于基于图的计算方式得以解决[4,5]. 其中的关键问题之一是查找连通分量，如在代谢工程中强连通量的查找可以提供生物体内代谢情况的大量信息[6]；利用聚类分析可以获得基因、蛋白质的重要性能和特征，而聚类过程通常可以转化为求图中连通分量的过程[7,8].

图中关于查找连通分量的研究主要包含如下两个方面:

(1) 查找 k-edge 连通分量[9-16]. 图 G 的一个 k-edge 连通分量是图 G 的一个最大顶点导出子图 g，并且 g 在删除任意(k-1)条边后仍连通，这里取最大值的 k 称为 g 的连通度. 例如，图 1 中的图 G 是 2-edge 连通分量，其连通度是 2，而子图 g_1 是一个连通度为 4 的 4-edge 连通分量.

(2) 查找查询相关的 k-edge 连通分量[17]. 即给定一个查询顶点集 q，求出图 G 中包含 q 的最大子图且该子图的连通度最大，即最大连通 Steiner 分量(Steiner Maximum-Connected Component, SMCC[17]). 如果限定结果子图的规模，即要求 SMCC 中包含的顶点数大于等于常数 L，则称这种最大连通的 Steiner 分量为 SMCC$_L$[17]. 类似这样包含查询顶点的最大连通的 k-edge 连通分量查询往往能揭示图中顶点间的特定关系，协助用户认识和定位查询相关的关系密切的顶点集. 例如，在图 1 所示的图 G 中，对查询 q={v_3,v_4,v_5}而言，g_1 是 q 的

SMCC, 连通度为 4. 如果 L=8, 则 q 的 SMCC$_L$ 是 $g_1 \cup g_2$, 连通度为 3. 这样，用户不但可以通过 SMCC 查询得知查询点所在的紧密子图中顶点之间的紧密连通关系，而且可知和给定查询点有同样紧密联系的其他顶点，如 v_1 和 v_2. 进而，当给定阈值 L 后，可进一步扩大返回子图的规模，获取更多顶点间以及这些顶点和查询点之间相对紧密的连通关系.

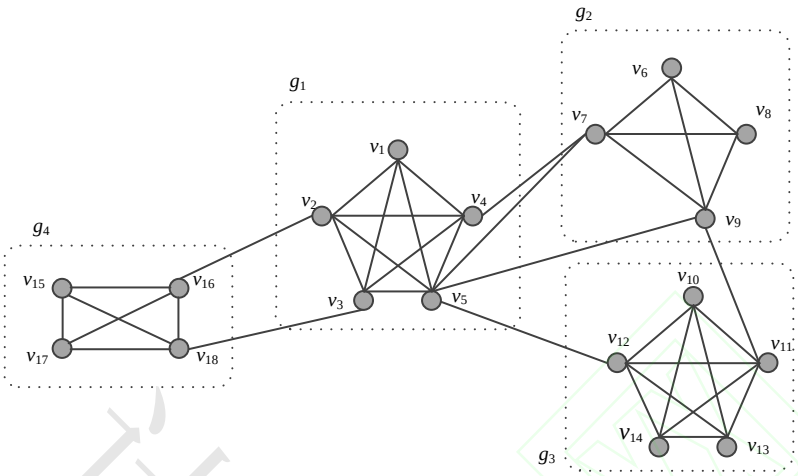


图 1　连通分量示意图 G

给定图 G 和查询 q, 求 q 的 SMCC 和 SMCC$_L$ 在计算生物学、社交网络、协同工作网络及电子商务网络中都有着广泛的应用，现有查询 SMCC 和 SMCC$_L$ 的方法主要分为两类:

(1) 基于现有的 k-edge 连通分量算法查询 SMCC 或 SMCC$_L$ 的方法. 该方法的基本思想是: 将 k 的值从$|V|$(图 G 中顶点数)到 1 依次减小，计算 G 中所有的 k-edge 连通分量，找到第一个包含查询 q 的 k-edge 连通分量，即为 q 的 SMCC; 找到第一个包含 q 且顶点数大于等于 L 的 k-edge 连通分量，即为 q 的 SMCC$_L$. 这类方法的问题是查找过程需要多次遍历图 G, 当图的规模较大时，计算代价高.

(2) 基于专用索引查询 SMCC 或 SMCC$_L$ 的方法. 该方法的基本思想是: 离线构建 G 的 MST(Maximum Spanning Tree)索引. 查询时，首先计算 q 的连通度，然后以 q 中任意一个顶点为起始点，在 MST 上进行遍历，仅访问权值满足特定条件的边对应的顶点，直到覆盖 q 中所有的顶点. 此时，访问过的顶点即为 q 的 SMCC 中的顶点.

例如，图 2 是图 1 的 G 对应的 MST 索引. 给定查询 q={v_3,v_4,v_5}，首先通过计算得知 q 的连通度是 4, 然后任选 q 的一个顶点开始遍历. 假设以 q 中的 v_4 为起点进行遍历，且仅处理权值大于等于 4 的边对应的顶点，则 v_4、v_1、v_2、v_3、v_5 是其中一个遍历的顺序. 此时，由于已经找到所有查询点，且不存在权值大于或者等于 4 的边，因此无需遍历顶点 v_{16}、v_{18}、v_{12}、v_9、v_7，所以 q 的 SMCC 中包含的顶点就是 v_1、v_2、v_3、v_4、v_5.

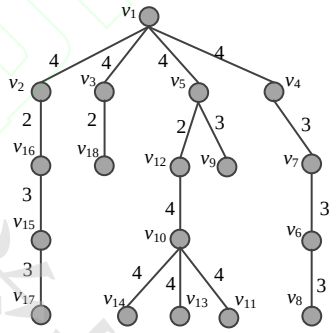


图 2　图 G 的 MST 索引

该类方法的问题是: 查询 SMCC 时，需要执行昂贵的遍历操作，每遍历一步，最多得到一个有用的顶点，可称为一次一顶点的方法. 当 SMCC 中包含的顶点数较多时，需要在索引树中访问的顶点数量也随之增加. 当查询请求执行的比较频繁时，系统负载迅速增大.

针对该问题，本文基于集合划分的思想，将图中顶点划分为较少的顶点集合，构建顶点集合的分层索引. 查询 SMCC 和 SMCC$_L$ 时，每一步遍历操作，可得到一个索引点对应的顶点集合，从而仅需访问少量的索引点即可得到查询结果，显著提升查询处理的效率，和已有方法相比，可称为一次一集合的方法. 本文的具体贡献如下.

(1) 提出一种 KST 索引结构，即利用图 G 中的每一个 k-edge 连通分量对图中顶点进行集合划分，并依据每一个 k-edge 连通分量与其所有(k+1)-edge

连通分量的包含关系建立分层索引，使得KST索引中以任意一个索引点为根的子树都对应图中的一个 k-edge 连通分量.

(2) 提出基于 KST 索引的 SMCC 和 SMCC$_L$ 查询算法. 该算法利用 KST 索引，仅需访问少量索引点即可快速得到满足条件的结果，避免了对图中顶点的多次随机访问，显著提升了算法的整体性能.

(3) 在15个真实数据集上通过实验对本文方法和已有方法进行了比较. 实验结果验证了本文提出方法的高效性.

本文的后续内容组织如下：第 2 节对基本知识及相关工作进行介绍，第3节介绍KST索引的基本思想和构建方法，第 4 节对 SMCC 和 SMCC$_L$ 的查询算法进行了详细描述，第 5 节通过实验对本文提出的算法进行验证，最后在第 6 节总结全文.

2 基本知识和相关工作

2.1 基本知识

无向图 G 用二元组表示，即 $G=(V,E)$，其中,是顶点的集合，E 是边的集合. 假定 $V_s \subseteq V$，则由 V_s 生成的顶点导出子图表示为 $G_s=(V_s,E_s)$，其中，V_s 是图 G_s 的顶点集合，E_s 是图 G_s 的边的集合且 $E_s \subseteq E$，即 $G_s=(V_s,\{(u,v)\in E|u,v\in V_s\})$. 为了叙述方便，在没有歧义的情况下，后续的介绍中将无向图简称为图.

定义 1. ***k*-edge 连通**[13]. 如果删除图 G 中的任意(k-1)条边后的剩余子图仍然是连通的，则称图 G 是 k-edge 连通的，最大的 k 值称为图 G 的边连通度，简称连通度.

例 1. 在图 1 所示的图 G 中，g_1 是 1-edge、2-edge、3-edge、4-edge 连通的，其中，g_1 最大是 4-edge 连通的，连通度为 4.

定义 2. ***k*-edge 连通分量**[13]. 假设 g 是图 G 的一个子图，如果 g 满足：(1) g 是 k-edge 连通的，(2) 在图 G 中，任何比 g 大的子图都不是 k-edge 连通的，则称 g 是图 G 的一个 k-edge 连通分量.

例 2. 如图 1 所示的图 G 是 2-edge 连通的，子图 g_1 是一个 4-edge 连通分量，g_4 是一个 3-edge 连通分量；而 g_2 不是一个 3-edge 连通分量，这是因为 G 中存在比 g_2 大的子图 $g_1 \cup g_2$ 是 3-edge 连通的，根据定义 2 可知，$g_1 \cup g_2$ 是一个 3-edge 连通分量. 这里，$g_1 \cup g_2$ 表示子图 g_1 和 g_2 构成的子图，包含 g_1 与 g_2 之间的边，如(v_4,v_7)、(v_5,v_7)、(v_5,v_9).

问题定义. 给定图 G 的顶点集 q，本文研究和 k-edge 连通分量密切相关的两个问题.

问题 1：返回包含 q 的具有最大连通度的 k-edge 连通分量，即 q 的 SMCC.

问题 2：给定子图规模 L，返回包含 q 的具有最大连通度且顶点数大于等于 L 的 k-edge 连通分量，即 q 的 SMCC$_L$.

例 3. 由定义 2 和问题定义可知，对于图 1 中的图 G 来说，如果 $q=\{v_1,v_4\}$，则 q 的 SMCC 是 g_1，用 sc 来表示 SMCC 的连通度，则 $sc(q)$=4；当 L=4 时，q 的 SMCC$_L$ 是 g_1，当 L=6 时，q 的 SMCC$_L$ 是 $g_1 \cup g_2$. 如果 $q=\{v_1,v_4,v_7\}$，则 q 的 SMCC 是 $g_1 \cup g_2$，$sc(q)$=3.

2.2 相关工作

如前所述，目前查询 q 的 SMCC 或 SMCC$_L$ 的方法主要分为两大类：一种是用 k-edge 连通分量算法查询 SMCC 或 SMCC$_L$，另一种是基于专用索引查询 SMCC 或 SMCC$_L$，分别介绍如下：

(1) 基于 k-edge 连通分量算法查询 SMCC 或 SMCC$_L$ 的方法

现有的计算 k-edge 连通分量的方法中能够用来查询 SMCC 和 SMCC$_L$ 的主要有基于切割的算法[9,11,12]、基于图分割的算法[13]和随机算法[14]. 由于基于图分割的算法和随机算法性能优于基于切割的算法[17]，这里只介绍基于图分割的算法和随机算法.

基于图分割的 k-edge 连通分量查询算法的主要思想是：图 G 是 k-edge 连通的当且仅当图的每一个分割都至少有 k 条边，从而将给定的图分解成不相交的连通子图来计算 k-edge 连通分量.

随机算法查询 k-edge 连通分量的主要思想是：在图中迭代查找小于 k 的切割，并依次将图进行分割，如果每个连通分量中都不存在小于 k 的切割点，则所得到的即为 k-edge 连通分量.

基于以上算法查询 SMCC 和 SMCC$_L$ 时，其性能主要取决于计算 k-edge 连通分量算法的高效性，本质上属于不依赖于特定索引的方法. 基于图分割算法的时间复杂度是 $O(h \times l \times |E|)$，其中 h 和 l 是常量因子；随机算法的时间复杂度是 $O(t \times |E|)$，其中 t 是迭代次数，$t=O(\log^2|V|)$. 采用基于图分割算法查询 SMCC 和 SMCC$_L$ 的时间复杂度为 $O(|V| \times h \times l \times |E|)$；采用随机算法查询 SMCC 和 SMCC$_L$ 的时间复杂度为 $O(|V| \times \log^2|V| \times |E|)$.

这类方法存在的共性问题是：查询处理时，需

要在线计算 k-edge 连通分量. 当图 G 变大时, 查询 q 的 SMCC 和 SMCC_L 的时间过长, 扩展性差.

(2) 基于专用索引查询 SMCC 或 SMCC_L 的方法

文献[17]第一个提出了 SMCC 和 SMCC_L 的查询问题, 并设计了专用索引来加速查询过程. 该索引的构建步骤如下:

首先为图中每条边添加基于 k-edge 连通分量的连通度的权值, 构建连通度图. 例如, 图 1 的连通度图如图 3 所示.

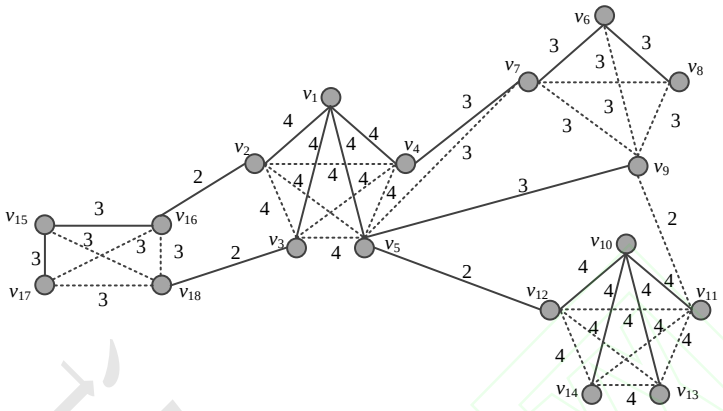


图 3 图 G 的连通度图

其次, 基于连通度图构建规模和图中顶点数量相同的索引结构 MST(该索引中边的权值为两顶点所在 SMCC 的连通度). 例如, 图 2 为图 1 对应的 MST 索引. 该索引和原图顶点规模相同.

基于 MST 索引, 查询 SMCC 的基本思想是: 根据查询 q 的 SMCC 连通度 k, 任意选取 q 的一个顶点为起点对 MST 索引进行遍历, 遍历时通过处理所有权值大于等于 k 的边来访问顶点, 即可得到 q 的 SMCC.

基于 MST 索引, 查询 SMCC_L 的基本思想是: 任意选取查询 q 的一个顶点为起点, 在 MST 索引上遍历时每次只访问权值大的边, 当访问过的顶点数与给定的 L 相等时, 记录当前访问过的边的最小权值 k; 继续遍历时仅通过处理所有权值大于等于 k 的边来访问顶点, 即可得到 q 的 SMCC_L.

可以看出, 该方法在查询 SMCC 和 SMCC_L 时, 需要执行昂贵的遍历操作, 每遍历一步, 最多得到一个有用的顶点. 针对该问题, 本文基于集合划分的思想, 将图中顶点划分为较少的顶点集合, 构建顶点集合的分层索引. 查询 SMCC 和 SMCC_L 时, 每一步遍历操作访问一个索引点, 对应一个顶点集合, 从而实现仅需访问少量的索引点即可得到查询结果的效果, 显著提升查询处理的效率.

3 KST 索引

3.1 基本思想

集合划分思想主要的依据是 k-edge 连通分量的定义, 即每一个 k-edge 连通分量完全被一个 (k-1)-edge 连通分量包含, 因此, 可以考虑利用 k-edge 连通分量对图的顶点进行划分: 每一个 k-edge 连通分量是一个顶点的集合, 用一个索引点表示. 利用(k-1)-edge 连通分量与 k-edge 连通分量之间包含关系构建顶点集的分层索引, 即构建 k-edge 连通分量生成树, 简称 KST 索引.

在如图 1 所示的图 G 中, g_1 是 4-edge 连通分量, g_3 是 4-edge 连通分量, g_4 是 3-edge 连通分量, $g_1 \cup g_2$ 是 3-edge 连通分量, $g_1 \cup g_2 \cup g_3 \cup g_4$ 是一个 2-edge 连通分量. 现将图 G 中的各 k-edge 连通分量分别看成对应的索引点, 可建立 KST 索引, 该索引如图 4 所示. 以索引点 2 为根的子树对应图 G, 即 $g_1 \cup g_2 \cup g_3 \cup g_4$, 它是一个 2-edge 连通分量; 索引点 2 的孩子为索引点 3、5、7, 以 3、5、7 为根的子树分别对应 G 的 3-edge 连通分量 $g_1 \cup g_2$、3-edge 连通分量 g_4 和 4-edge 连通分量 g_3; 索引点 3 的孩子是叶子索引点 6, 对应的是图 G 的一个 4-edge 连通分量 g_1. 依据图 G 中和 k-edge 连通分量和(k+1)-edge 连通分量的包含关系, 可知 KST 索引有以下性质:

性质 1. 以 KST 索引中任意一个点为根的子树都对应图 G 的一个 k-edge 连通分量.

由这个性质可知, 在KST索引中的索引点都是图中的顶点集合; 以任一索引点为根的子树表示的是图中的 k-edge 连通分量 $S(k)$; 以该索引点的孩子为根的子树 $S(k+n)$是 $S(k)$中的$(k+n)$-edge 连通分量$(n\geq 1)$. 也就是说, KST 索引中包含的索引点规模小于原图中的顶点规模, 当依据该索引进行 SMCC 和 SMCC_L 查询时, 无需遍历图中的所有顶点, 只对索引中的索引点进行遍历即可. 显然, 这种方法利用较小的索引, 避免了查询时随机访问 G 中顶点的代价.

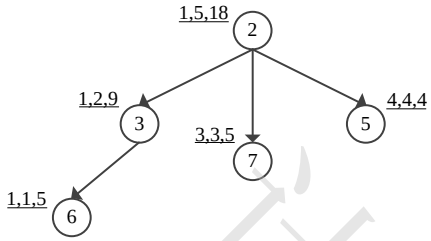


图 4 G 的 KST 索引

3.2 索引构建

KST 索引构建的基本步骤是: 求出图中所有 k 值对应的 k-edge 连通分量, 依据连通分量之间的包含关系形成基本的 KST 索引; 然后压缩基本 KST 索引中的冗余索引点; 最后对 KST 索引中的顶点进行编码.

3.2.1 基本 KST 索引

构建基本 KST 索引时需要计算图中的 k-edge 连通分量, 计算 k-edge 连通分量依据文献[13]的方法. 假设图 G 为一个连通图, 构建基本 KST 索引的方法是: 任何一个连通图本身就是 1-edge 连通的, 所以, 首先将图 G 看成一个索引点 *root*, 即基本 KST 索引的根, 并给图 G 中的所有顶点添加其所属的集合编号 *setID*, 因为 *root* 对应图中的所有顶点, 所以 *setID* 值均为 *root*; 接着求出 *root* 中的所有 2-edge 连通子图, 将每一个 2-edge 连通子图 $S(u)$中顶点的 *setID* 修改为 u, 并将 u 作为 *root* 的孩子索引点插入索引中; 然后对每一个 2-edge 连通子图 $S(u)$求出包含的所有 3-edge 连通子图. 依此类推, 直到每个索引点不能产生新的孩子为止, 如算法 1 所示. 注: 若图不连通, 算法 1 中设 k=0 即可.

算法 1. KST_basic

输入: 连通图 G

输出: 基本 KST 索引

```
BEGIN
1  [v1, v2, ···, v|V|].setID←root        //初始化图中所有顶点的集合编号为 root
2  root.k←1; S(root)←{G};                //S(v)是索引点 v 对应的图中顶点的集合, 用连通分量表示
3  enQueue(Q, root);                     //根索引点入队列
4  WHILE Q≠Ø DO
5      v←deQueue(Q);
6      S←ComputeKECCs(S (v),v.k+1);      // 计算 S(v)中的所有(k+1)-edge 连通分量, 并和相应的索引点对应起来
7      FOR EACH u∈S DO
8          insert u as a child node of v
9          set the setID of all nodes in S(u) as u
10         enQueue(Q, u);                //u 入队列
11     RETURN root
END
```

例 4. 考虑图 1 所示的图 G. 首先, 将图 G 看成索引点 1, 即根结点 $S(1)=G$. 然后初始化所有顶点的集合编号 *setID* 为 1(第 1 行). 在此基础上, 计算 2-edge 连通子图(第 6 行). 因为图 G 是 2-edge 连通分量, 所以将图 G 看成是索引点 2, 同时将所有顶点的 *setID* 修改为 2(第 9 行), 并将索引点 2 作为索引点 1 的孩子(第 8 行). 接下来求解 3-edge 连通分量, 因为 $g_1 \cup g_2$、g_3 和 g_4 都是 3-edge 连通子图(其中 $g_1 \cup g_2$ 和 g_4 是 3-edge 连通分量), 所以将 $g_1 \cup g_2$ 看成索引点3, g_3看成索引点4, g_4看成索引点5, 将$g_1 \cup g_2$中的顶点 v_1、v_2、……、v_9的 *setID* 修改为 3, 将 g_3中的顶点 v_{10}、v_{11}、……、v_{14}的 *setID* 修改为 4, 将 g_4中的顶点 v_{15}、v_{16}、……、v_{18}的 *setID* 修改为 5, 并将索引点 3、4 和 5 作为索引点 2 的孩子. 最后求解 4-edge 连通分量, g_1 和 g_4 都是 4-edge 连通分量, 分别看成索引点 6 和 7, 所以将 g_1 中的顶点 v_1、v_2、……、v_5的 *setID* 修改为 6, 顶点 v_{10}、v_{11}、……、v_{14} 的 *setID* 修改为 7. 由于顶点 v_1、v_2、……、v_5 原来的 *setID* 是 3, 因此, 将索引点 6 作为索引点 3 的孩子. 同理, 顶点 v_{10}、v_{11}、……、v_{14}原来的 *setID* 是 4, 因此, 索引点 7 即为索引点 4 的孩子. 由于 4-edge 连通分量是图 G 中连通度最大的 k-edge 连通分量, 所以, 算法结束. 图 G 的基本 KST 索引结构如图 5 所示, 索引点与图中顶点的对应关系如表 1 所示.

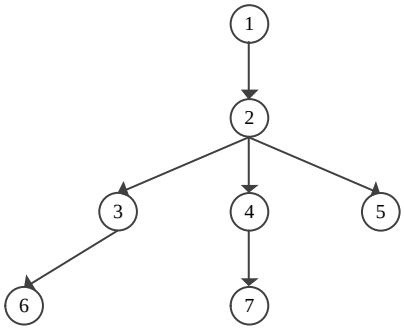


图 5　图 G 的基本 KST 索引

表 1　索引点与 G 中顶点的对应关系

索引点	G 中顶点	连通度
1	—	1
2	—	2
3	v_6 v_7 v_8 v_9	3
4	—	3
5	v_{15} v_{16} v_{17} v_{18}	3
6	v_1 v_2 v_3 v_4 v_5	4
7	v_{10} v_{11} v_{12} v_{13} v_{14}	4

从表 1 可以看出图 5 所示的基本 KST 索引中的索引点和 G 中的顶点对应关系：以索引点 1 为根的子树对应图 G 的 1-edge 连通子图，子树中包含的索引点为 1、2、3、4、5、6、7，对应图中全部顶点．以索引点 3 为根的子树对应图 G 的一个 3-edge 连通分量，包含的顶点有 v_1、v_2、v_3、v_4、v_5、v_6、v_7、v_8、v_9；叶子索引点 6 对应图 G 的一个 4-edge 连通分量，对应图中的顶点为 v_1、v_2、v_3、v_4、v_5．

3.2.2　KST 索引的压缩

在 KST 索引的构建过程中，k 值从 1 开始递增，增量为 1，所以对于连通度较大或者规模较大的图 G 而言，可能会有如下情况出现：图 G 或其中的子图，本身是一个(k+1)-edge 连通分量，但在索引构建过程中，该连通分量会首先被认为是 k-edge 连通的，并插入相应索引点．随着 k 的递增，索引中会插入多个表示相同连通分量的索引点．

基本的 KST 索引中可能存在冗余点的特征是：两个索引点连通度值相差为 1，但以这两个索引点分别为根的子树表示同一子图．如果基本 KST 索引中存在冗余索引点，就会使得基本 KST 索引不满足性质 1，所以要对基本 KST 索引进行压缩，以去除冗余索引点．

进行索引压缩的具体方法是：从基本 KST 索引的根开始逐层扫描，当发现有一个索引点 u 只含有一个孩子索引点 v，同时以 u 和 v 为根的子树对应 G 中相同的顶点集合，则删除索引点 u，保留索引点 v，并将 u 的父索引点作为 v 的父索引点．

例 5．以图 1 中的图 G 为例，图 G 对应的基本 KST 索引如图 5 所示，索引点对应的顶点如表 1 所示．对 KST 进行遍历，根据基本 KST 索引和表 1 可知，索引点 1 只有唯一一个孩子即索引点 2，且以索引点 1 和 2 为根的子树对应的顶点集合是相同的，都是图中的全部顶点，所以将索引点 1 删除，保留索引点 2．依此方法遍历 KST 中所有索引点，查找满足压缩条件的索引点进行删除，最终得到压缩后的 KST 索引如图 4 所示．

3.2.3　KST 索引的编码

为了进一步提高 KST 索引的查询效率，考虑为 KST 索引中的索引点添加标签，通过访问带标签的 KST 索引点，可快速确定以 KST 中任一索引点为根的子树对应的 k-edge 连通分量是否为满足条件的 SMCC 或者 SMCC$_L$．

为 KST 索引编码的基本思想是：给 KST 索引中的每一个索引点添加三元组<*sid*, *tid*, *size*>标签．其中，*tid* 值是这个索引点后序遍历的顺序号，*sid* 是这个索引点的孩子 *sid* 中的最小值．对叶子索引点而言，*sid*=*tid*．*size* 是以该索引点为根的子树所表示的 k-edge 连通分量对应图中的顶点数量．*sid* 和 *tid* 构成对应索引点的区间，其作用是：(1) 通过索引点区间[*sid*, *tid*]的包含关系来快速定位索引点的最低公共祖先(Lowest Common Ancestor, LCA)[18]；(2) 找到最低公共祖先后，通过该索引点的区间[*sid*, *tid*]可以找到 KST 索引中该最低公共祖先包含的索引点．标签 *size* 的作用是：在查询 SMCC$_L$ 时可以直接与 L 值进行比较，判断以当前索引点为根的子树是否对应满足条件的 SMCC$_L$．对 KST 索引进行编码的过程如算法 2 所示．

算法 2．　Label-KST

输入：压缩后的 KST 索引

输出：最终的 KST 索引

BEGIN

1　对压缩后的 KST 进行后序遍历，得到(u, n_u)；　//u 是 KST 中的索引点；n_u 是索引点 u 的后序遍历的顺序号

2　FOR EACH u DO　//按后序遍历的顺序依次进行判断和计算

3　　$u.size$←|$S(u)$|;　//$S(u)$表示索引点 u 对应的的图中顶点

集合

4 IF u 是 KST 中的叶子索引点

5 $u.sid \leftarrow n_u$;

6 $u.tid \leftarrow n_u$;

7 ELSE

8 u.$sid \leftarrow \infty$;

9 $u.tid \leftarrow n_u$;

10 FOR EACH child v of u DO

11 IF $u.sid > v.sid$

12 $u.sid \leftarrow v.sid$;

13 $u.size \leftarrow u.size + v.size$;

END

例 6. 通过对图 4 的 KST 索引执行算法 2 可以得到索引点的编码. 具体过程为: 索引点 6 是后序遍历时第 1 个访问的顶点, 其 *tid* 为 1. 因索引点 6 是叶子, 其 *sid* 等于 *tid* 值, 同时索引点 6 包含的顶点个数是 5, 所以其 *size* 是 5; 索引点 3 是后序遍历时第 2 个访问的顶点, 其 *tid* 为 2, 索引点 3 的孩子的 *sid* 中的最小值是 1, 所以索引点 3 的 *sid* 为 1, 以索引点 3 为根的子树中包含索引点 3 和 6, 其对应的 G 中顶点个数为 9, 所以索引点 3 的 *size* 为 9; 后续顶点处理过程类似. 图 1 对应的最终 KST 索引如图 4 所示.

3.3 复杂度分析

假设$|V|$表示图中顶点的数量, 算法 1 求 k-edge 连通分量的时间复杂度是 $O(|V|\times h\times l\times|E|)$[13], 其中 h 和 l 是常量因子. 其它构建操作的代价和 KST 索引中顶点数量成正比. 同时, 对索引进行压缩和编码的代价也和索引点数量成正比. 即给定连通分量后, 依据连通分量构建 KST 的时间复杂度为 $O(|N|)$, N 表示 KST 中的索引点集, $|N|$表示 KST 中的索引点数量, 而最坏情况下 KST 中索引点数量和原始图中顶点数量相同. 因此, 构建索引的时间复杂度仍为 $O(|V|\times h\times l\times|E|)$.

本文的 KST 索引是查询处理的关键索引. 除此之外, 还包括另外两种索引: (1) 顶点和索引点的对应关系. 由于每个顶点只对应到一个索引点上, 因此这部分索引的规模和图中顶点规模相同; (2) 每个索引点对应的顶点集合. 由于所有索引点对应的顶点集合之间没有公共顶点, 且所有索引点的顶点集的并集是整个图中的顶点集, 因此, 这部分索引的规模和图中顶点的规模相同. 由于 KST 索引中索引点数量的上限是图中顶点数量, 因此, 本文提出的索引的大小为 $O(|V|)$. 和 MST 索引相比, 本文的 KST 索引需要维护的信息更少.

4 基于 KST 索引的 k-edge 连通分量查询方法

4.1 SMCC查询算法

定理 1. 给定图 G 的一个查询 $q=\{v_0,v_1,\cdots,v_{|q|-1}\}$, 在 KST 索引中, q 对应的索引点集为 Γ, 假设 A 为 Γ 中所有点在 KST 上的最低公共祖先, 则以 A 为根的子树对应的 k-edge 连通分量为查询 q 的 SMCC.

证明. 由最低公共祖先[18]的定义可知, 以 A 为根的子树包含查询 q 中每一个查询点对应的 KST 索引点; 由性质 1 可知, 以 A 为根的子树对应的是图中的一个 k-edge 连通分量, 而以 A 的祖先索引点为根的子树对应的连通分量的连通度小于 k, 所以, 以 A 为根的子树对应的是包含 q 的连通度最大的 k-edge 连通分量, 即 q 的 SMCC.

证毕.

基于定理 1, 查询 SMCC 的基本思想是: 找到查询 q 在 KST 索引上对应索引点的最低公共祖先, 返回以该最低公共祖先为根的子树中索引点所对应的图中顶点即为 q 的 SMCC 中的顶点. 其中, 找到查询 q 在 KST 索引上对应索引点可以在 $O(1)$时间内完成. 具体做法是将顶点和索引点的对应关系用哈希表存储. 对于给定的查询点, 可以通过查询哈希表在常量时间确定对应的索引点. SMCC-KST 算法的具体描述如算法 3 所示.

算法 3. SMCC-KST

输入: KST 索引和查询 $q=\{v_0,v_1,\cdots,v_{|q|-1}\}$

输出: q 的 SMCC 所包含的顶点集

BEGIN

1 在 KST 中找到包含查询点的索引点集合 Γ;

2 $u \leftarrow LCA(\Gamma)$; //利用 KST 索引中索引点标签中的[*sid*, *tid*]区间找到 Γ 的最低公共祖先 u

3 $R \leftarrow$以 u 为根的子树中的所有索引点;

4 $V_q \leftarrow \cup_{v\in R}S(v)$;

5 RETURN V_q;

END

例 7.对于图 1 中的 G 而言, 假设给定的查询为 $q=\{v_4,v_{10},v_{13}\}$, v_4、v_{10}、v_{13} 对应 KST 索引中的索引点分别是 6、7(表 1), 在 KST 索引中, 索引点 6 和 7 的最低公共祖先为索引点 2, 以索引点 2 为根的子

树包含索引点 2、3、5、6、7, 因此, 可知查询 q 的 SMCC 所包含的顶点即为图 G 中的所有顶点. 这其中, 索引点 6、7 的最低公共祖先是离索引点 6 和 7 最近的, 且区间能够覆盖索引点 6 和 7 的区间. 而索引点 6 和 7 的区间分别为[1,1]、[3,3], 所以求其最低公共祖先的具体方法是: 取 *tid* 较大的索引点 7 作为起始点, 经判断, 其区间不能同时覆盖索引点 6 和 7 的区间, 然后向上访问索引点 7 的父亲索引点 2, 其区间为[1,5], 能覆盖索引点 6 和 7 的区间, 所以索引点 6 和 7 的最低公共祖先是索引点 2, 其 *sid* 和 *tid* 分别是 1、5, 所以 *tid* 值在区间[1,5]的索引点即为以 2 为根的子树包含的索引点. 由图 4 的 KST 索引可知, 其对应的索引点为 6、3、7、5、2, 这些索引点对应的 G 中所有的顶点, 即 G 中所有的顶点为查询 q 的 SMCC 包含的顶点(第 3~5 行).

4.2 $SMCC_L$查询算法

根据图 G 的 KST 索引和 k-edge 连通分量的性质可知, 只要在 KST 中找到一个索引点, 以该点为根的子树所对应的 k-edge 连通分量的连通度最大、包含 q 且顶点数大于等于 L, 这个 k-edge 连通分量为 q 的 $SMCC_L$. 因此, 给出利用 KST 索引结合最低公共祖先查询 $SMCC_L$ 的相关定理如下.

定理 2. 给定图 G 的一个查询 $q=\{v_0,v_1,\cdots,v_{|q|-1}\}$ 和常数 L, 假设 KST 索引中 q 对应的索引点集为 Γ, A 为 Γ 中所有点在 KST 上的最低公共祖先. 如果以 A 为根的子树对应的 k-edge 连通分量满足其中包含的顶点数大于等于 L, 则该 k 边连通分量即为查询 q 的 $SMCC_L$; 否则, 设 PA 为 A 的祖先索引点, 若 PA 是满足以 PA 为根的子树对应的 k 边连通分量中包含的顶点数大于等于 L 这一条件时距离 A 最近的祖先索引点, 则以 PA 为根的子树对应的 k 边连通分量即为查询 q 的 $SMCC_L$.

证明. 由定理 1 可知, 以 A 为根的子树对应的是包含 q 且连通度最大的 k-edge 连通分量. 对于 q 的 $SMCC_L$ 而言, 其查询结果中的顶点数量有限制, 因此需要对其中查询到的连通分量中包含的顶点个数进行判断: 若其中的顶点数大于等于 L, 那么, 这个 k 边连通分量即为 q 的 $SMCC_L$; 否则, 由最低公共祖先的定义和性质 1 可知, 以 A 的祖先索引点 PA 为根的子树对应的 k 边连通分量必然包含 q, 且索引点 PA 离 A 越远, k 值越小, 即: PA 必须离 A 最近, 才能保证连通度最大. 此时, 如果以 PA 为根的子树对应的 k 边连通分量满足其中的顶点数大于等于 L, 则这个 k 边连通分量为 q 的 $SMCC_L$.

证毕.

因此, 基于 KST 索引查询 $SMCC_L$ 的基本思想是: 首先求出 q 在 KST 索引上的最低公共祖先的索引点, 然后判断以该索引点为根的子树对应的连通分量中包含的顶点数与 L 的关系, 若大于等于 L, 则以该索引点为根的子树对应的连通分量即为 q 的 $SMCC_L$, 以该索引点为根的子树中包含的顶点即为所求的 $SMCC_L$ 中的顶点; 否则, 递归访问该索引点的父索引点, 若以其父索引点为根的子树中包含图中的顶点数大于等于 L, 则该子树中所有索引点包含的顶点即为 q 的 $SMCC_L$ 中的顶点. $SMCC_L$-KST 算法具体描述如算法 4 所示.

算法 4. $SMCC_L$-KST

输入: KST 索引、查询 $q=\{v_0,v_1,\ldots v_{|q|-1}\}$和数值 L

输出: q 的 $SMCC_L$ 所包含的顶点集

```
BEGIN
1  在 KST 中找到包含所有查询点的索引点集 Γ;
2  u←LCA(Γ);          //利用 KST 中索引点标签中的[sid, tid]区间找到 Γ 的最低公共祖先 u
3  WHILE u.size < L DO
4      u←u.parent;
5  R←以 u 为根的子树中的所有索引点;
6  V_qL←∪_{v∈R} S(v);
7  RETURN V_qL;
END
```

例 8. 对图 1 的 G 而言, 假设查询 $q=\{v_1,v_4,v_5\}$, $L=6$. v_1、v_4、v_5 对应的索引点都是 6(表 1), 所以 v_1、v_4、v_5 在 KST 索引上对应索引点的最低公共祖先就是索引点 6(第 1~2 行). 又因为索引点 6 的 *size* 为 5, $5<L=6$(图 4), 接着继续访问索引点 6 的父亲(第 3~4 行), 即索引点 3, 索引点 3 的 *size* 值是 9, $9>L=6$, 满足条件. 由于索引点 3 的区间是[1,2], 所以 *tid* 为 1 和 2 的索引点是以索引点 3 为根的子树包含的索引点, 由图 4 可知, *tid* 值 1 和 2 对应的索引点为 6 和 3. 所以查询 q 的 $SMCC_L$ 包含的顶点是索引点 6 和 3 对应的 G 中顶点, 即顶点 v_1、v_2、v_3、v_4、v_5、v_6、v_7、v_8、v_9(表 1).

4.3 复杂度分析

SMCC-KST 和 $SMCC_L$-KST 算法在执行时, 对 KST 索引的处理异常简单. 首先确定查询点的最低公共祖先, 这一步可在 $O(h\times|q|)$代价内完成, 其中 h 是 KST 索引的树高, q 是给定的查询顶点集合. 然后查询满足条件的索引点, 这一步的代价是 $O(|N|)$,

这里|*N*|是 KST 索引中的索引点数量. 最后是输出索引点对应的 *G* 中顶点集合. 因此, 算法 3 和算法 4 的时间复杂度为 O(*h*×|*q*|+|*N*|+*M*), 这里 *M* 是 SMCC 和 SMCC$_L$ 中包含的顶点数量. 由于 *h*×|*q*|+|*N*| 通常情况下远远小于 *M*, 因此, 该时间复杂度可以进一步表示为 *O*(|SMCC|) 和 *O*(|SMCC$_L$|), 这里 |SMCC|(|SMCC$_L$|)表示相应的 SMCC(SMCC$_L$)中的顶点数量. 和文献[17]相比, 文献[17]查询和输出时访问 MST 索引是随机访问模式, 而 KST 索引的每个索引点对应的图中顶点可以集中存放, 可以做到顺序访问模式. 因而可以充分利用 cache 预取功能提升 cache 命中率, 获得时间复杂度相同情况下执行效率的大幅度提升. 另一方面, 查询 SMCC 和 SMCC$_L$ 时, 本文提出的算法仅需访问少量顶点即可得到结果, 因而算法 3 和算法 4 的时间复杂度分别是 *O*(|SMCC|)和 *O*(|SMCC$_L$|).

5 实验

5.1 实验环境

本文实验所用机器的基本配置为 AMD Athlon(tm) II X2 250, 3000MHz CPU, 16 GB 内存, 操作系统为 Ubuntu 12.04.4 LTS. 所有实验用 C++ 实现, 并用 GCC4.6.3 进行编译.

根据文献[17]的结果显示, 基于 MST 索引的 SMCC 和 SMCC$_L$ 查询算法比基于 *k*-edge 分量的查询算法高效得多, 因此实验中只比较基于 MST 的算法 (MST) 和本文提出基于 KST 的算法 (SMCC-KST 和 SMCC$_L$-KST). 基于 MST 的算法源代码由原作者提供. 以下讨论中, 当处理 SMCC 和 SMCC$_L$ 时, KST 分别代表 SMCC-KST 和 SMCC$_L$-KST.

5.2 数据集

本文所用数据集为 15 个真实数据集, 如表 2 所示. 其中 Human、Agrocyc、Ecoo、Vchocyc 和 Mtbrv 来自 Ecocyc①, 是描述基因组、转录调控、转运蛋白和代谢途径的生物信息数据集; 其中 Ecoo 已经作为黄金标准数据集用于基因功能的预测、转录起始点的预测、调节网络重构、功能性和直接蛋白—蛋白的交互作用和功能性预测. Go② 为基因本体数据集, 定义了用于描述基因功能的概念类, 以及这些概念之间的关系. 本文提出的算法可以加快 SMCC 和 SMCC$_L$ 的查询速度, 对于研究分子功能、基因产物的分子活性、细胞成分、基因产物活跃的团等方面有极大的实际应用价值.

同时, 本文的算法也可应用于社交网络, 用来进行社区发现. 为此, 在实验中也使用了社交数据集用于测试算法的性能. 这些社交数据集包括 ca-GrQc、ca-CondMat、email-EuAll、soc-Epinions1、amazon0601、web-Goole、wiki-Talk、as-Skitterh 和 LiveJournal. 这些数据集都从 Standford SNAP library③下载.

表 2 数据集统计信息

数据集	|*V*|	|*E*|	顶点的平均度
Human	38,811	39,576	1.02
Agrocyc	12,684	13,408	1.06
Ecoo	12,620	13,350	1.06
Vchocyc	9,491	10,143	1.07
Mtbrv	9,602	10,245	1.07
Go	6,793	13,361	1.97
ca-GrQc	4,185	13,422	6.46
ca-CondMat	21,363	91,286	8.55
email-EuAll	224,832	339,925	3.02
soc-Epinions1	75,877	405,739	10.69
amazon0601	403,364	2,443,311	12.11
web-Goole	665,957	3,074,322	9.23
wiki-Talk	2,388,953	4,656,682	3.90
as-Skitter	1,694,616	11,094,209	13.09
LiveJournal	4,843,953	42,845,684	17.69

5.3 性能比较及分析

本文实验的评价指标包括: (1) 构建索引的时间、索引大小及索引的相关统计信息; (2) 查询 SMCC 和 SMCC$_L$ 的时间; (3) 构建 KST 索引时冗余索引点的压缩效果. 实验中分别用 5 个查询组对 SMCC 和 SMCC$_L$ 进行测试, 每个查询组由 1,000 个随机产生的大小固定的查询 *q* 组成, 5 个查询组中查询 *q* 的大小分别为 2、5、10、20 和 30. SMCC$_L$ 中的常数 *L* 为 10~1,000 之间随机选取的数值. 对于每一个查询组, 统计的是针对其中 1,000 个查询 *q* 分别执行 3 次的平均时间.

① ecocyc.org

② www.geneontology.org

③ snap.standford.edu/data/

5.3.1 索引构建时间及索引大小

表 3 展示的索引构建时间, 索引大小和 KST 索引的统计信息, 包括 KST 索引的索引点数量、KST 索引的叶子索引点数量, 以及 KST 索引点数量与 MST 索引点数量的比值.

在索引构建时间方面, 如表 3 所示, 在实验的 15 个数据集上, 构建 KST 索引比构建 MST 索引稍快. 在规模略小的数据集上 KST 索引的构建时间比 MST 索引的构建时间快 38%～54%, 但随数据集规模变大, KST 和 MST 索引时间差别越不明显. 主要原因是: 这两种索引的构建效率都取决于以求解图的 *k*-edge 连通分量的时间效率, 求解图的 *k*-edge 连通分量在索引构建过程中占绝大部分, 如表 3 的第 4 列所示, 如果求 *k*-edge 连通分量的时间忽略不计, 则 KST 索引的构建速度会明显优于 MST 索引的构建速度.

在索引大小方面, 如表 3 所示, 在实验的 15 个数据集上, MST 所占内存是 KST 的 2~4 倍. 原因在于, 如表 3 的 7~9 列所示, KST 中索引点数量远小于 MST 中索引点数.

表 3 索引构建时间、索引大小和 KST 索引的统计信息

数据集	索引构建时间/ms			索引大小/MB		KST 索引统计信息		
	MST	KST	计算 *k*-edge 连通分量时间	MST	KST	索引点数量	叶子数量	KST 索引点数量/MST 索引数量
Human	16.063	8.303	7.823	1.09	0.31	4	2	0.00010
Agrocyc	4.708	2.158	2.008	0.36	0.10	4	2	0.00032
Ecoo	4.611	2.541	2.011	0.35	0.10	4	2	0.00032
vchocyc	3.167	1.937	1.517	0.27	0.08	6	3	0.00063
Mtbrv	3.356	1.976	1.556	0.27	0.08	5	2	0.00052
Go	16.802	15.112	14.802	0.19	0.08	6	3	0.00088
ca-GrQc	41.319	40.419	40.319	0.11	0.04	112	87	0.02676
ca-CondMat	692.024	670.324	670.024	0.59	0.17	240	200	0.01123
email-EuAll	1,333	1,229	1,227	6.29	1.76	39	3	0.00017
soc-Epinions1	8,192	8,098	8,097	2.12	0.60	182	109	0.00240
amazon0601	30,853	30,215	30,209	11.29	3.23	2,248	1,824	0.00557
web-Goole	59,340	57,994	57,898	23.96	6.27	9,417	6,445	0.01414
wiki-Talk	60,210	58,603	58,576	66.89	18.67	139	9	0.00006
as-Skitter	180,357	177,914	177,894	47.44	13.26	803	538	0.00047
LiveJournal	2,082,409	2,053,306	2,053,237	135.63	38.29	13,670	6,976	0.00282

5.3.2 查询 SMCC 的时间

图 6 和图 7 分别展示的是在较小规模数据集(Go 和 vchocyc)以及在较大规模数据集上(Wiki-Talk 和 as-Skitter)SMCC 的查询时间对比折线图. 可以看到, 当数据集规模增大后, 查询效率提升更明显. 例如, 在数据集 wiki-Talk 和 as-Skitter 上, KST 比 MST 快 4 到 5 个数量级. 限于篇幅, 其他数据集的对比结果在表 4 展示. 如表 4 所示, 在求不同查询的 SMCC 时, KST 算法在所有数据集上比 MST 至少快 1 个数量级. 主要原因有两点: 首先, 如表 3 所示, KST 索引远小于 MST 索引. 其次, 查询过程中 KST 访问的索引点数量远少于 MST (如表 5 所示).

表 5 展示的是每个查询组中查询时访问索引点的数量之和, 之所以有很多相同的数值, 原因在于, 如表 3 所示, 不同数据集上构建的 KST 索引和 MST 索引比起来, 规模很小. 由于索引点数量很少, 因而个别索引点对应相当一部分图中顶点, 极端情况下甚至趋近于图中顶点的总数. 在此情况下, 很多查询点都映射到了相同的索引点, 导致大量查询访问了 KST 索引上相同的索引点.

我们以大小不同, 但访问索引点数量相同的三个数据集(Human, email-EuAll, wiki-Talk)为例进行说明. 如表 5 所示, 在这三个数据集上, 处理 1,000 个查询需要访问的索引点为(|*q*|+1)×1,000, 即当查

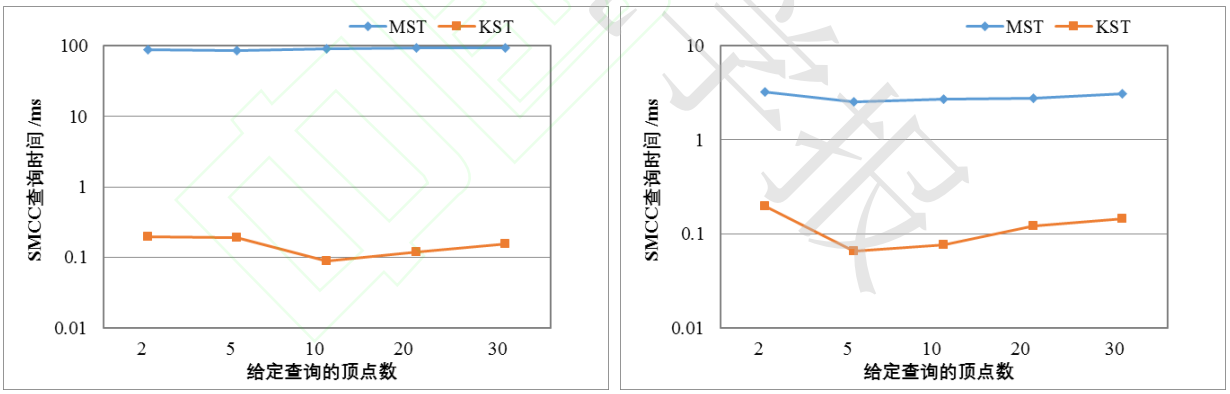
询包含 i 个查询点时，需要访问 i+1 个索引点来得到查询结果. 具体原因如下.

首先，这三个数据集对应的KST索引如下面的图 8 所示，均包含一条最左边的比较长的路径，同时第二层还有多个其他索引点. 在每个 KST 索引中，顶点内的数字为索引点的编号，顶点左边的数字是该索引点对应的图中顶点集的顶点数量. KST 索引的索引点编号是后序遍历的顺序号，可以看出根索引点的编号最大，对应图中的顶点集合也最大. 对 Human 来说，根索引点包含 98%的图顶点；对 email-EuAll 来说，根索引点包含 84%的图顶点；对 wiki-Talk 来说，根索引点包含 74%的图顶点. 同时，除了最左边的路径，第二层其他索引点包含的图顶点很少. 由于本文实验中查询点是随机给定的，这导致绝大多数情况下，给定查询中的查询点对应的所有索引点都在最左边的路径上.

其次，对于给定的查询，当得到所有的索引点后，本文求最低公共祖先的方法是首先定位编号最大的索引点 v，然后从 v 开始向上遍历，在遍历过程中确定从 v 开始的第一个公共祖先点. 对以上三个数据集来说，由于每个查询对应的索引点恰好都在最左边的路径上，同时最靠近根的索引点的编号最大，且恰好编号最大的索引点又是最低公共祖先索引点. 因而，对每个查询来说，如果其包含 i 个查询点，当得到结果时，总共需要访问的索引点数量都是 i+1 个. 因此会出现在不同数据集上处理不同查询时，访问相同数量索引点的情况. 注意本文所说的访问索引点数量，并非不同索引点的数量，而是访问索引点的次数.

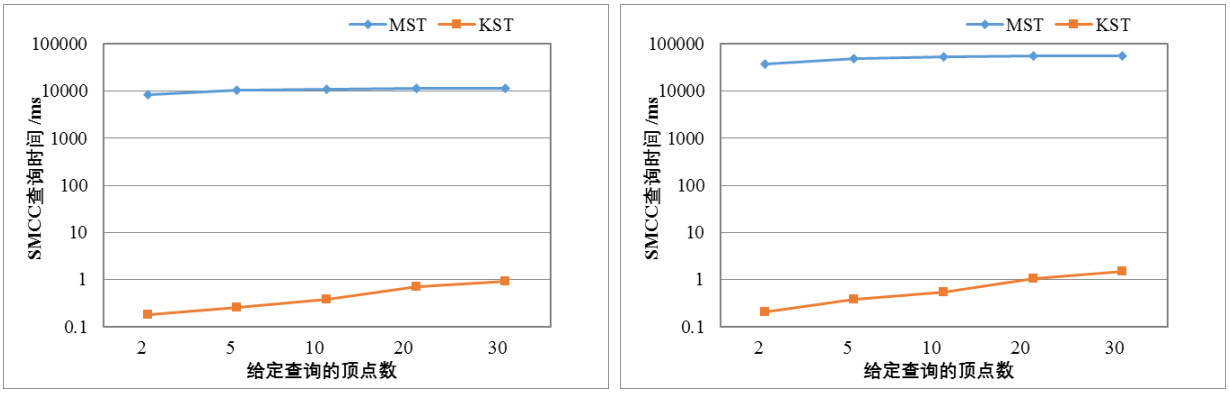
最后，对于这三个数据集的索引而言，即使给定查询的索引点不都位于最左边的路径，也有访问索引点数量相同的结论. 这是由索引本身的形状造成的.

需要注意的是，如果索引形状发生了变化，编号最大的索引点不一定离根最近，这时以编号最大的索引点为起始点向上查找最低公共祖先可能会访问较多的索引点. 显然，理论上最好的方法是记录每个索引点的层数，然后根据层数选择离根最近的索引点为起点向上遍历，这样可以访问最少的索引点. 本文选择以编号最大的索引点为起始点的原因是：对大部分数据集来说，我们注意到 KST 索引都有左偏性，因而直接选择编号最大的索引点为起始点可以用更少的判断在获得最优结果的同时获得更快的响应速度；其次，我们的方法无需为每个索引点维护层数信息，索引更简洁.



(a) Go 数据集 (b) vchocyc 数据集

图 6 在较小规模图上 SMCC 的查询时间对比



(a) Wiki-Talk 数据集　　(b) as-Skitter 数据集

图 7　在较大规模图上 SMCC 的查询时间对比

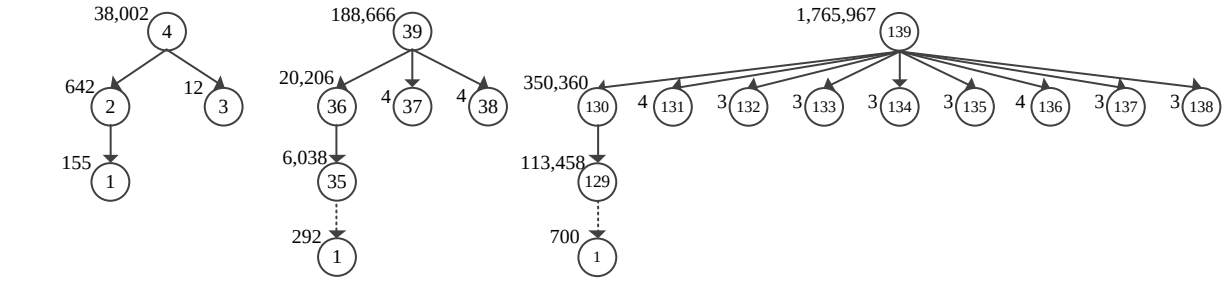
表 4　SMCC 查询时间比较　　/ms

数据集	|q|=2		|q|=5		|q|=10		|q|=20		|q|=30	
	MST	KST	MST	KST	MST	KST	MST	KST	MST	KST
Human	4.56	0.055	3.66	0.066	3.69	0.086	3.77	0.116	3.96	0.158
Agrocyc	3.42	0.049	10.39	0.061	2.67	0.074	2.73	0.116	2.30	0.159
Ecoo	3.53	0.06	3.33	0.083	3.29	0.075	3.36	0.115	3.63	0.153
Mtbrv	4.23	0.19	10.88	0.061	3.21	0.095	3.27	0.116	3.54	0.148
ca-GrQc	2.03	0.055	47.41	0.239	1.84	0.083	1.88	0.117	3.03	0.146
ca-CondMat	30.56	0.059	427.18	0.103	471.51	0.084	481.68	0.143	528.28	0.18
email-EuAll	204.56	0.085	302.46	0.147	337.32	0.123	344.60	0.165	356.97	0.204
soc-Epinions1	834.90	0.08	1082	0.083	1,237	0.11	1,264	0.125	1,409	0.179
amazon0601	283,562	0.167	33,261	0.307	32,582	0.452	33,285	0.821	36,761	1.173
web-Goole	24,837	0.218	31,811	0.402	34,941	0.681	35,694	1.257	41,734	1.798
LiveJournal	278,887	0.214	370,728	0.473	398,600	0.875	407,194	1.659	441,253	2.472

表 5　查询 SMCC 时访问的索引点数量

数据集	|q|=2		|q|=5		|q|=10		|q|=20		|q|=30	
	MST	KST	MST	KST	MST	KST	MST	KST	MST	KST
Human	155,000	3,000	155,000	6,000	155,000	11,000	155,000	21,000	155,000	31,000
Agrocyc	114,001	3,000	114,000	6,000	114,000	11,000	114,000	21,000	114,000	31,000
Ecoo	142,020	3,000	142,000	6,000	142,000	11,000	142,000	21,000	142,000	31,000
vchocyc	108,832	3,000	109,787	6,000	116,000	11,000	116,000	21,000	116,000	31,000
Mtbrv	120,360	3,000	121,680	6,000	129,000	11,000	129,000	21,000	129,000	31,000
Go	233,345,010	3,000	2,461,676	6,000	2,407,077	11,000	2,474,000	21,000	2,487,350	31,000
ca-GrQc	998,788	3,148	1,173,008	6,049	1,317,765	11,004	1,412,709	21,000	1,467,921	31,000
ca-CondMat	3,129,826	3,061	396,668	6,020	4,297,169	11,011	4,630,060	21,001	4,633,256	31,000
email-EuAll	2,053,014	3,000	2,576,886	6,000	2,806,441	11,000	2,839,988	21,000	2,914,880	24,880
soc-Epinions1	5,065,038	3,001	5,935,574	6,000	6,920,713	11,000	6,909,644	21,000	6,992,008	31,000
amazon0601	167,102,398	3,044	187,293,510	6,032	193,091,704	11,014	198,369,493	21,002	202,526,788	31,003
web-Goole	109,449,029	3,251	136,018,866	6,136	150,352,586	11,084	168,519,135	21,018	176,088,923	31,011
wiki-Talk	32,340,935	3,000	40,891,570	6,000	42,940,307	11,000	46,716,698	21,000	45,347,810	31,000

as-Skitter	245,545,833	3,000	302,982,278	60,003	333,609,651	11,027	347,138,451	21,000	343,034,851	31,005
LiveJournal	773,577,325	3,702	987,110,288	6,590	1,046,228,635	11,132	1,079,496,837	21,040	1,134,490,698	31,098

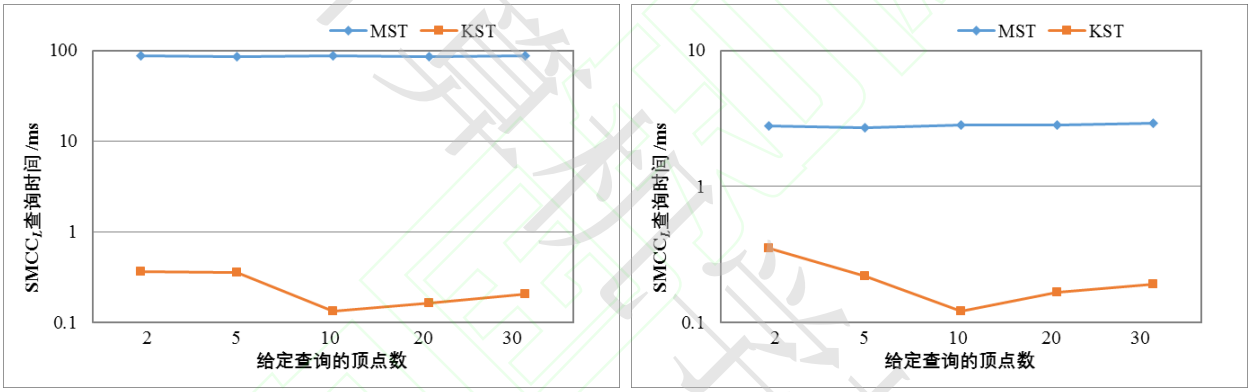


(a)Human (b) email-EuAll (c) wiki-Talk

图 8 KST 索引示例

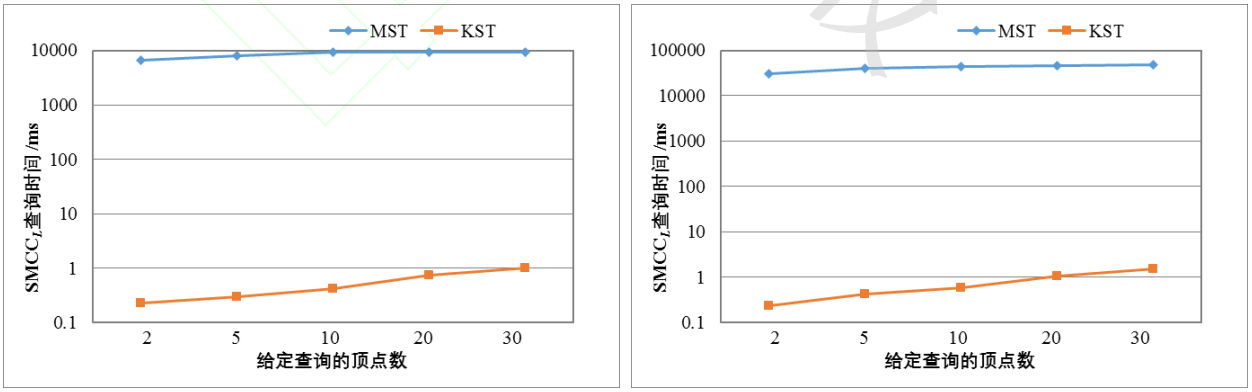
5.3.3 查询 $SMCC_L$ 的时间

图 9 和图 10 分别展示的是在较小规模数据集(Go 和 vchocyc)以及较大规模数据集(Wiki-Talk 和 as-Skitter)$SMCC_L$ 的查询时间对比折线图. 和 SMCC 相同, 其他数据集的对比结果以表格方式在表 6 展示. 和 SMCC 类似, 在求不同查询的 $SMCC_L$ 时, KST 算法比 MST 算法至少要快 1 个数量级, 且数据集规模越大, 效率提高越明显. 主要原因也是在查询 $SMCC_L$ 时, KST 算法比 MST 算法访问的索引点数量都要少.



(a) Go 数据集 (b) vchocyc 数据集

图 9 在较小规模图上 $SMCC_L$ 的查询时间对比



(a) Wiki-Talk 数据集 (b) as-Skitter 数据集

图 10 在较大规模图上 $SMCC_L$ 的查询时间对比

表 6 $SMCC_L$ 查询时间比较 /ms

数据集	|q|=2		|q|=5		|q|=10		|q|=20		|q|=30	
	MST	KST	MST	KST	MST	KST	MST	KST	MST	KST

Human	3.75	0.108	3.75	0.119	3.80	0.135	3.82	0.175	3.82	0.202
Agrocyc	2.75	0.092	2.80	0.104	2.76	0.123	2.82	0.158	2.84	0.202
Ecoo	3.26	0.102	3.27	0.125	3.33	0.119	3.34	0.157	3.45	0.224
Mtbrv	3.21	0.361	3.30	0.232	3.25	0.137	3.35	0.16	3.33	0.199
ca-GrQc	15.70	0.107	15.18	0.291	12.72	0.13	4.87	0.167	2.41	0.194
ca-CondMat	228.43	0.105	298.41	0.149	292.66	0.131	286.74	0.185	279.27	0.225
email-EuAll	137.18	0.139	182.28	0.201	226.95	0.163	239.83	0.205	227.58	0.251
soc-Epinions1	750	0.121	938	0.124	1,077	0.155	1,101	0.177	1159	0.227
amazon0601	29,467	0.232	33,135	0.372	32,374	0.498	33,717	0.855	33,923	1.217
web-Goole	26,525	0.273	33,310	0.457	33,952	0.736	34,286	1.3	35,982	1.838
LiveJournal	245,958	0.264	308,621	0.523	367,292	0.923	365,118	1.705	381,123	2.523

5.3.4 KST 索引的压缩效果

表 7 展示的是 KST 索引压缩前后的对比结果. 从表 7 可以看出, 当索引点数量增大后, KST 索引经过压缩, 可以显著减少索引点的数量.

表 7　KST 索引的压缩效果对比

数据集	压缩前索引点数量	压缩后索引点数量
Human	4	4
Agrocyc	4	4
Ecoo	4	4
Vchocyc	6	6
Mtbrv	5	5
Go	6	6
ca-GrQc	309	112
ca-CondMat	371	240
email-EuAll	39	39
soc-Epinions1	197	182
amazon0601	3,725	2,248
web-Goole	20,060	9,417
wiki-Talk	139	139
as-Skitter	849	803
LiveJournal	28,593	13,670

6 结论

针对现有方法在查询 q 的 SMCC 和 $SMCC_L$ 时存在的效率低下问题, 我们提出了一种维护顶点集关系的 KST 索引, 以及基于 KST 索引的高效查询算法. 和已有方法在索引中维护原始图中顶点间的关系不同, 本文提出的KST索引维护的是顶点集合的包含关系. 同时, KST 满足以每个索引顶点为根的子树必是一个 SMCC 的特性. 基于该索引, 查询处理就实现了从索引遍历时的一次一顶点到 KST 索引上的一次一集合的高效转变, 显著减少了需要访问的索引点数量, 提高了算法的效率. 基于 15 个真实数据集的实验结果表明, 本文提出的方法不仅所需的索引构建时间短、索引规模小, 而且查询处理时, 拥有异常高效的查询性能, 数据集规模越大, 效果越明显.

参 考 文 献

[1] Fan W, Li J, Ma S, Wang H, Wu Y. Graph homomorphism revisited for graph matching[J]. The Proceedings of the VLDB Endowment (PVLDB), 2010, 3(1/2): 1161-1172

[2] Zhao Y, Gu R, Du S. The research status and development tendency of bioinformatics[J]. Journal of medical informatics, 2012, 33(5): 2-6 (in Chinese)
(赵屹, 谷瑞升, 杜生明. 生物信息学研究现状及发展趋势 [J]. 医学信息学杂志, 2012, 33(5): 2-6

[3] Lu D, Xu J, Xiang C, Xie J. Survey of clustering methods for big data in biology[J]. Journal of Shanghai University (Natural Science), 2016,22(1):45-57 (in Chinese)
(路东方, 许俊富, 项超娟 ,谢江. 生物大数据中的聚类方法分析[J].上海大学学报(自然科学版),2016,22(1):45-57)

[4] Zhang Y, Liu Y B, Xiong G, Jia Y, Liu P, Guo L. Survey on succinct representation of graph data. Journal of Software, 2014, 25(9): 1937-1952 (in Chinese)
(张宇, 刘燕兵, 熊刚, 贾焰, 刘萍, 郭莉. 图数据表示与压缩技术综述. 软件学报, 2014, 25(9): 1937-1952)

[5] Ding Y, Zhang Y, Li Z H, Wang Y. Research and advances on graph data mining. Journal of Computer Applications, 2012, 32(1): 182-190 (in Chinese)
(丁悦, 张阳, 李战怀, 王勇. 图数据挖掘技术的研究与进展. 计算

机应用, 2012, 32(1): 182-190)

[6] Zhou H, Zheng H, Li Y, Li H. Estimating stability of 13C MFA calculating model through the concept of strong connected component[J]. Beijing Biomedical Engineering, 2009,28(1):34-38 (in Chinese)
(周宏, 郑浩然, 李毅, 李恒. 基于强连通分量的 13C MFA计算模型稳定性判断[J]. 北京生物医学工程, 2009,28(1):34-38)

[7] Liu H, Liao B, Peng H. Reseach of clustering algorithm in protein-protein interaction network[J]. Computer engineering and aoolications, 2009, 44(30): 142-144 (in Chinese)
(刘昊, 廖波, 彭利红. 基于蛋白质相互作用网络的聚类算法研究[J]. 计算机工程与应用, 2009, 44(30): 142-144)

[8] Zhou H, Zhou Y, Zhang X, Tan S. A clustering algorithm for categorical variables based on connected components[J]. Control and Decision, 2015,30(1): 39-45 (in Chinese)
(周红芳, 周扬, 张晓鹏, 谈姝辰. 基于连通分量的分类变量聚类算法 [J]. 控制与决策, 2015,30(1): 39-45)

[9] Zhou R, Liu C, Yu J X, Liang W, Chen B, Li J. Finding maximal k-edge-connected subgraphs from a large graph[C]// Proceedings of International Conference on Extending Database Technology, Beijing, China, 2012: 480-491

[10] Hartuv E, Shamir R. A clustering algorithm based on graph connectivity[J]. Information Processing Letters, 2000, 76(4): 175-181

[11] Yan X, Zhou X J, Han J. Mining closed relational graphs with connectivity constraints[C] //Proceedings of International Council for Open and Distance Education, Tokyo, Japan, 2005: 357-358

[12] Papadopoulos A N, Lyritsis A, Manolopoulos Y. Skygraph: an algorithm for important subgraph discovery in relational graphs[C] //Proceedings of The 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, USA. 2008:57-76

[13] Chang L, Yu J X, Qin L, Liu C, Lin X, Liang W. Efficiently computing k-edge connected component via graph decomposition[C] //Proceedings of the ACM SIGMOD International Conference on Management of Data, New York, USA, 2013:205-216

[14] Akiba T, Iwata Y, Yoshida Y. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction [C]// Proceedings of 22nd ACM International Conference on Information and Knowledge Management, San Francisco, USA, 2013:909 -918

[15] Cosgaya-Lozano A, Zeh N. A heuristic strong connectivity algorithm for large graph[C]//Proceedings of International Symposium on Experimental Algorithms, New York, USA, 2009: 113-124

[16] Zhang Z, Qin L, Yu J X. Constract & Expand: I/O efficient SCCs computing[C]//Proceedings of IEEE 24th International Conference on Data Engineering, Chicago, USA, 2014: 208-219

[17] Chang L, Lin X, Qin L, Yu J X, Zhang W. Index-based optimal algorithms for computing steiner components with maximum connectivity[C]//Proceedings of the ACM SIGMOD International Conference on Management of Data. Melbourne, Victoria, Australia, 2015:459-474

[18] Aho A V, Hopcroft J E, and Ullman J D. On finding lowest common ancestors in trees[C]//Proceedings of ACM Symposium on Theory of Computing, San Francisco, USA, 1973: 253-265

CHEN Zi-Yang, born in 1973, Ph.D., professor, Ph.D. supervisor, Distinguished Professor of Shanghai Lixin University of Accounting and Finance. His research interests include database theory and techniques.

CHEN Wei, born in 1980, Ph.D. candidate, associate professor. Her research interests include query processing on graph data.

JIA Yong, born in 1990, M.S. candidate. His research interests include query processing on graph data.

ZHOU Jun-Feng, born in 1977, Ph.D., professor, Ph.D. supervisor. His research interests include query processing on XML and semi-structured data.

Background

Given a graph G, and a set of query nodes q, the Steiner Maximum-Connected Component (SMCC) is a subgraph of G containing q, which has the maximum connectivity and the maximum number nodes, and SMCC_L requires that the number of nodes in the returned result is greater than or equal to L. Finding SMCC/SMCC_L has attracted much attention in the research field, which can be applied in many applications, such as bio-informatics, social networks, collaborative work

networks, electronic commerce networks, and so on. Due to the increase of the size of graph, it is still a challenging task to answer SMCC queries faster with less index size and index construction time.

Considering that existing approaches on querying SMCC suffer from inefficiency, we proposed two optimized algorithms, namely SMCC-KST and SMCC$_L$-KST, to accelerate the query processing of SMCC and SMCC$_L$. We first proposed a KST index, which maintains the relationship between k-edge connected subgraph and (k+1)-edge connected subgraph. Each node of KST index represents a set of nodes in G and the subtree rooted at it corresponds to a k-edge connected component in G. Compared with existing methods, the index size is largely reduced. Based on the KST index, we proposed a new SMCC finding algorithm, and proposed another algorithm to find SMCC with size constraint. Compared with existing index which maintains the relationship of nodes in the original graph, in our KST index, each index node represents a set of nodes. The benefit is that in this way, we can improve existing approaches from one-node-a-step to one-set-a-step, such that to significantly reduce the number of visited index nodes to improves the query efficiency. We conduct rich experiment on 15 real datasets, the experimental results verified that our approaches perform much better than existing ones for SMCC queries.

This research was partially supported by the grants from National Science and Technology Major Project (No. 61472339, 61572421, 61873337).