计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2024,33(4):162−170 [doi: 10.15888/j.cnki.csa.009462]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

利用图模型存储算法依赖关系的方法①

谢昌佐[1,2], 李子扬[1], 董裕民[1,2], 李雪松[1], 舒　展[1], 杨　光[1]

[1](中国科学院 空天信息创新研究院 中国科学院定量遥感信息技术重点实验室, 北京 100094)
[2](中国科学院大学 电子电气与通信工程学院, 北京 100094)
通信作者: 李子扬, E-mail: zyli@aircas.ac.cn

摘　要: 在大数据时代, 用于数据处理的算法数量呈爆发式增长, 当前对大量算法的管理方法通常是对算法分类、打标签或以任务为单位存储由算法构成的流程, 对任务集合中的算法间拓扑关系未能给予足够的重视. 随着领域知识与任务流程的积累, 算法间的依赖关系愈发重要. 本文基于巨量算法管理的需求, 提出了拆分有分支依赖关系为无分支依赖关系的管理方法, 通过免索引邻接图数据库的指针搜寻拓扑关系, 避免 Join 操作, 在管理算法依赖关系时具有先天优势. 另外为突出算法模块复用能力, 提出“连接点”的概念, 在图模型中用节点表示依赖关系边, 区分算法模块在不同任务流程的位置, 使被多个任务复用的算法模块在图中只需用一个算法模块节点表示. 最后, 基于具体项目验证了本文提出的算法关系管理方法, 证明本文算法关系管理方法在算法数量成规模且算法模块高复用的场景下具有明显优势.

关键词: 算法关系管理; 拓扑拆分; 模块复用; 图模型; 多任务

引用格式: 谢昌佐,李子扬,董裕民,李雪松,舒展,杨光.利用图模型存储算法依赖关系的方法.计算机系统应用,2024,33(4):162–170. http://www.c-s-a.org.cn/1003-3254/9462.html

Method of Using Graph Model to Store Algorithm Dependencies

XIE Chang-Zuo[1,2], LI Zi-Yang[1], DONG Yu-Min[1,2], LI Xue-Song[1], SHU Zhan[1], YANG Guang[1]

[1](Key Laboratory of Quantitative Remote Sensing Information Technology, Aerospace Information Research Institute, Chinese Academy of Sciences, Beijing 100094, China)
[2](School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100094, China)

Abstract: In the era of big data, the number of algorithms used for data processing is exploding. The current management method for a large number of algorithms is usually to classify and label the algorithms, or store task flows composed of algorithms on a task-by-task basis, while insufficient attention has been paid to the topological relationships between algorithms in the task set. With the accumulation of domain knowledge and task flows, the dependency between algorithms becomes increasingly important. Based on the requirement of massive algorithm management, this study proposes a management method for splitting branched dependencies into unbranched dependencies. By searching for topological relationships through pointers in an index-free adjacency graph database, it avoids Join operations and has innate advantages in managing algorithm dependencies. In addition, this study proposes connection points to highlight the reusability of algorithm modules, which are utilized to represent dependency edges in the graph model. The position of algorithm modules in different task flows can be distinguished, so that algorithm modules reused by multiple tasks only need to be represented by one algorithm module node in the graph. Finally, based on specific projects, the algorithm relationship management method proposed in this study is validated. It is proved that the algorithm relationship

① 基金项目: 国家重点研发计划 (2021YFC3000302)
收稿时间: 2023-10-07; 修改时间: 2023-11-09; 采用时间: 2023-12-05; csa 在线出版时间: 2024-03-04
CNKI 网络首发时间: 2024-03-08

management method has significant advantages in scenarios where the number of algorithms is large and the algorithm modules are highly reusable.

Key words: algorithm relationship management; topology splitting; module reuse; graph model; multitask

1 引言

在数据科学发展的过程中, 随着数据量的不断积累, 数据管理技术也得到了极大的促进. 当前, 数据管理技术[1]已经经历了人工管理阶段和文件系统阶段, 进入了数据库系统阶段, 并且涌现出了数据湖等新概念. 与此同时, 由于数据的爆炸式增长, 用于处理数据的各种算法的种类和数量也在飞速上涨. 在这样的背景下, 算法本身也可以被看作一种“新型数据”, 需要进行组织和管理.

为了有效地管理日益丰富多样的算法, 许多有益的实验和尝试已经进行过. 这些工作主要可分为 3 类: 第三方库[2]、计算机语言库和应用软件系统封装下的算法库[3].

以 GitHub 上的 The Algorithms 项目为例, 它号称是规模最大的开源算法库. 该项目使用排序算法、搜索算法、动态规划算法和数据结构算法等热门分类来管理算法, 并提供源代码供用户下载. 然而, 该算法库仅对算法进行了分类, 缺乏对算法之间关系的组织和管理.

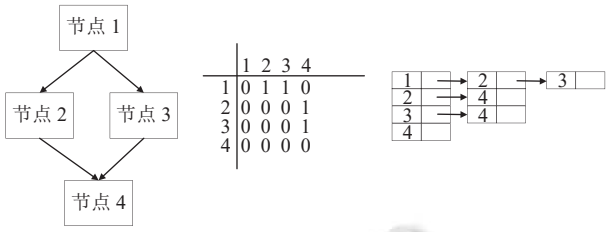
在计算机语言领域, 诸如 C++的 STL[4]等计算机语言链接库为开发者提供了多种算法调用接口. 然而, 这些库中的算法往往被天然地打包在一起, 算法之间的依赖关系没有得到明确的组织. 类似地, Python 等语言也有相应的算法库存在.

此外, 应用软件系统离不开算法的调用, 为提高软件代码的复用能力[5]和方便针对性的优化系统, 一些行业应用软件对算法进行了统一的封装. 举例来说, 遥感图像处理平台 ENVI 允许用户在软件中直接调用一个或多个算法来构建解决方案[6,7]. 然而, 当前这些算法库中的算法关系仅体现在类别上, 缺乏对算法之间依赖关系的刻意组织和管理.

实际上, 算法经过多年的囤积, 算法之间存在着错综复杂的类别和依赖关系[8,9], 形成了一个复杂的网络结构. 为了有效管理这种网络关系结构下的算法, 本文提出了一种基于图模型的存储管理算法关系的方法. 这种方法具有通用性, 能够帮助研究人员更好地组织、管理和发现不同领域的大量算法, 并加强算法在不同任务下的依赖关系表示, 从而提高数据科学工作的效率和成果质量.

2 算法关系存储技术现状

相比于管理算法本身, 管理算法关系更为重要, 算法依赖关系构成拓扑关系[10], 如图 1(a), 节点表示算法, 有向边表示算法间的依赖关系, 以往常使用邻接矩阵[11,12]、邻接表等数据结构存储, 如图 1(b) 和图 1(c). 邻接矩阵第 n 行、第 m 列的数字为 1, 表示存在节点 n 至节点 m 的有向关系, 邻接表约定表头存储的节点与单向链表存储的其他节点均为相邻节点.



(a) 有向图 (b) 邻接矩阵 (c) 邻接表

图 1 邻接矩阵与邻接表存储有向图

随着算法数量的增加, 算法关系存储技术对存储和管理效率提出了更高的要求, 通常情况下, 可以在关系型数据库设计表结构存储两种数据结构的信息, 如表 1 所示, 表 1 中权重可默认置为 1.

表 1 关系型数据库存储有向图

源节点	目标节点	权重
1	2	1
1	3	1
2	4	1
3	4	1

为每个任务算法关系单独建表会导致表单数量和任务数量同步增长, 并且, 在这种情况下, 从某个算法模块出发, 寻找后续多层算法模块需要 Join 操作, 在算法量达到一定规模时, 效率很难提升.

目前, 突出的算法关系存储模型是基于免索引邻接的图数据库. 这种模型利用图模型清晰展现算法之间的依赖关系, 充分利用指针搜寻关系的优势, 使得存

储和管理复杂的算法依赖关系变得更加高效. 因此, 基于免索引邻接的图数据库模型为当前和未来的算法关系存储提供了全新的技术路径和解决方案.

3 算法关系建模

3.1 算法关系概念集

算法关系的组织管理会涉及算法模块所属类别、算法模块被应用到的任务或者项目、算法模块具体充当哪些任务处理流的节点, 因此, 算法关系组织涉及的概念元素[13]有 C、P、W 和 O, 其中 C 为算法类别名称, P 为任务或项目名称, W 为处理流名称, O 为算法模块名称. 这些概念元素之间存在的关系有分类、实例、作用、依赖、关联、包含等.

算法类别可进行严格的层级分类关系设计, 例如算法类别 $C11$ 是算法类别 $C1$ 的子类别, 该分类关系表示为 ($C1$, R_{classify}, $C11$).

算法模块是算法原子的具体实例, 例如算法模块 $O1$ 是算法类别 $C211$ 下的某个实例, 该实例关系表示为 ($O1$, R_{instance}, $C211$).

通常在表征大型项目处理流程时, 需要记录算法模块之间的依赖关系. 例如项目的处理流 $W1$ 中, 算法模块 $O2$ 的执行依赖于算法模块 $O1$ 的顺利执行, 该算法模块依赖关系表示为 ($O2$, R_{dependw1}, $O1$), 该关系由处理流 $W1$ 作用.

项目通常对应多个解决方案, 每个解决方案有完整处理流程, 因此一个项目对应若干处理流. 本文将项目与处理流的关系定义为关联, 例如项目 $P1$ 关联处理流 $W1$, 该关联关系表示为 ($P1$, R_{relate}, $W1$).

考虑到任务可能存在复杂拓扑关系的处理流程, 以及某些任务的处理流可能成为其他任务的子处理流, 存储时有必要把处理流划分为子处理流.

本文算法关系概念集涉及的关系如图 2 所示, 其中, 依赖关系是研究的重点.

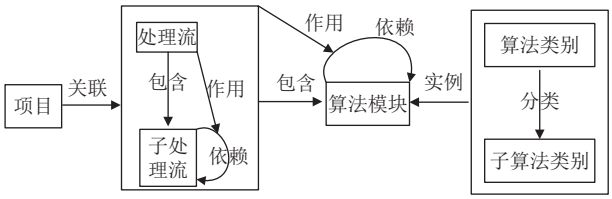


图 2 算法关系概念集模型

3.2 算法处理流分析与拆分

项目解决方案通常由多个处理流组成, 处理流由算法模块和依赖关系构成, 总体呈现为有向无环图, 如图 3 所示.

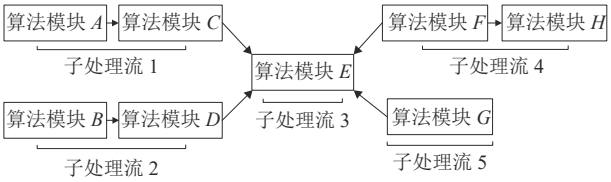


图 3 处理流图

显然, 当处理流中算法模块的出度、入度都小于等于 1 时, 也就是说任务的算法模块成单链状, 更容易存储. 因此, 可以先将处理流划分为无分支子处理流, 然后存储原处理流和无分支子处理流关系, 再存储无分支子处理流和算法模块的关系. 并且, 处理流的进一步拆分, 既考虑子处理流之间的潜在语义关系, 又考虑到子处理流未来的复用需求.

一种处理流拆分规则是如果算法模块的入度大于 1, 则该算法模块与其前置的算法模块划分到不同子处理流, 如果算法模块的出度大于 1, 则该算法模块与其后置模块划分到不同子处理流. 图 3 处理流中算法模块 E 入度、出度均为 2, 大于 1, 因此算法模块 E 与其依赖节点和后置节点被划分到不同子处理流, 即算法模块 E 单独作为无分支子处理流, 最后图 3 处理流总共划分为 5 个无分支子处理流.

本文处理流拆分算法具体如算法 1 所示, 任务的处理流最初存储在二维哈希表里, 类似于邻接矩阵, 以图 3 为例, 节点 C 依赖于节点 A, 则 $depend[C][A]=1$, 节点 A 后接节点 C, 则 $next[A][C]=1$, 对于节点 E, $depend[E].size()$ 代表 E 的入度, $next[E].size()$ 代表 E 的出度, *depend* 和 *next* 可以相互推导. 任务算法节点集合 $X\{A, B, C, D, \cdots\}$、*depend* 和 *next* 两个二维哈希表经算法 1 处理后, 输出拆分后的子工作流集合 Sub_w、子工作流序号集合 Sub_{seq}、子工作流之间的依赖关系二维哈希表 *subdepend*, 其中 $Sub_w[i]$对应的子工作流序号就是 $Sub_{seq}[i]$.

算法 1. 处理流→无分支子处理流拆分算法

输入: 算法节点集合 $X\{A, B, C, D, \cdots\}$、*depend* 和 *next* 两个二维哈希表

1) 初始化子处理流集合 Sub_w, 初始化子处理流序号集合 Sub_{seq}, 初始化子处理流起始序号 *subid*, 初始化哈希数组 *nodesub* 记录节点所属的子处理流, 初始化二维哈希数组 *subdepend* 记录子处理流的依赖关系

```
2) for x in X:
    if(depend[x].size()>1)
      for key, value in depend[x]:
        next[key][x]=0; //断开依赖节点
    if(next[x].size()>1)
      for key, value in next[x]:
        next[x][key]=0; //断开后置节点
3) for x in X:
    sign[x]=1; //初始化哈希数组 sign, 标记 x
4) for x in X:
    if(sign[x]==1)
      //创建一维动态数组 sub;
      subid++; //子处理流 id
      sign[x]=0;
      sub.push(x); //节点 x 加入 sub
      nodesub[x]= subid;
      while(next[x].size()>0)
        for key, value in next[x]:
          if(next[x][key]==0)
            break; //发现断开位置跳出 for 循环
            break; //发现断开位置跳出 while 循环
          x=key;
          sign[x]=0;
          sub.push(x);
          nodesub[x]= subid;
      Sub_w.push(sub); //把 sub 存入子处理流集合
      Sub_seq.push(subid);
5) for sub in Sub_w:
    //遍历当前子处理流的首节点所依赖的节点
    for key, value in depend[sub[0]]:
      frontsub=nodesub[key]; //依赖的子处理流
      cursub= nodesub[sub[0]]; //当前子处理流
      subdepend[cursub][ frontsub]=1;
6) 输出: Sub_w, Sub_seq, subdepend
```

4 图存储算法依赖关系方法

在算法库的实际使用中, 通常我们遇到的场景有两种, 一种是构建算法库的图结构, 在有新的算法或者新的处理流程时, 修改和增加算法库图结构的节点和关系. 另一种是在使用过程中, 从算法库中提取由若干算法组成的处理流程.

4.1 图模型构建方法

图模型存储网络结构时具有天然优势[14,15], 例如图 4 表达了两个任务各自关联的处理流, 即完整的解决方案, 均只包含 1 条无分支子处理流, 子处理流 1、2 分别复用了算法模块节点 *A*、*B* 和 *C*. 为方便表示, 约定后文的子处理流均为无分支子处理流.

图 4 中算法模块 *A*、*B* 和 *C* 存在复用情况, 因此可对模型进行变形如图 5 所示, 既表达了算法间的依赖关系, 又去除了冗余信息.

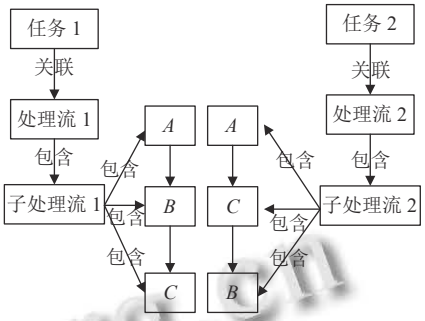


图 4 处理流直接存储图

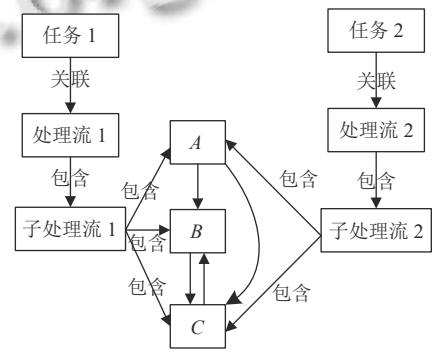


图 5 算法模块复用图

但算法模块 *B* 和 *C* 的拓扑关系在任务 1 和任务 2 中恰好相反, 因此在图 5 中所体现的关系不易理解. 为解决这个问题, 且同时突出算法模块依赖关系的主导作用, 本文提出“连接点”的概念. 连接点本质是将算法模块依赖关系由图数据库中的关系类型转换为实体类型, 辅助区分算法模块在不同处理流的作用. 连接点插入存在依赖关系的算法模块中间, 从处理流到连接点的边表示连接点属于该处理流, 比如图 6 中, 算法模块 *B* 在子处理流 1 的作用下后续节点是算法模块 *C*, 算法模块 *C* 在子处理流 2 的作用下后续节点是算法模块 *B*.

图 6 算法关系存储方式相比于图 4 算法关系存储方式, 增加了连接点, 减少了算法模块节点的数量. 随着子处理流复用频率增大, 图 6 存储方式的优势会更加明显. 例如当增加依次执行子处理流 1 和子处理流 3 的任务 3 时, 可以复用子处理流 1 内部信息, 如图 7. 在子处理流 1 和子处理流 3 中间添加连接点 3, 表示在处理流 3 下, 依次执行子处理流 1 和子处理流 3. 在算法模块 *C*、*D* 之间增加连接点 4, 表示处理流 3 下, 依次执行算法模块 *C*、*D*.

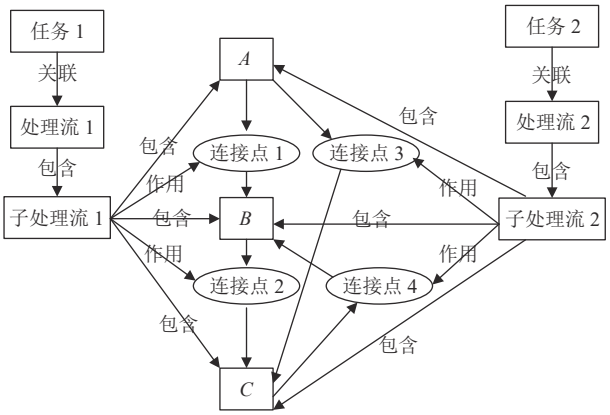


图 6 带有连接点的处理流存储图

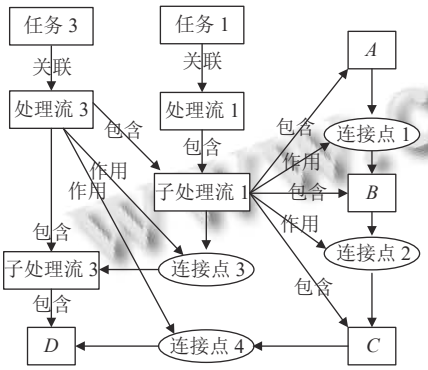


图 7 子处理流复用存储图

传统方式存储任务 1、任务 2 和任务 3 的处理流, 会存储 3 次算法模块 *A*, 本文存储方法仅需存储 1 次算法模块 *A*. 任务 1 和任务 3 中算法模块 *A* 至算法模块 *B* 链路的复用, 使得寻找算法模块 *A* 的后续节点时, 也少一次匹配. 设算法模块被所有任务复用 *t* 次, 所在的某个子处理流被复用 *w* 次, 则匹配该算法模块依赖节点或后续节点次数为 *t*–*w*+1, 与直接存储处理流方法相比, 提升了寻找算法模块的依赖节点和后续节点的性能.

在图模型中扩增算法关系时, 很可能出现新任务的某条子处理流包含旧子处理流的情况. 严格来说, 情况分为两类, 第 1 种如图 7, 任务 1 和任务 3 都复用子处理流 1, 第 2 种情况是新任务的子处理流除了包含旧子处理流, 还需要有其他算法模块.

第 2 种情况可以直接在图模型中添加新的子处理流节点 4, 如图 8 所示. 子处理流 4 相对于子处理流 1 复用了算法模块 *A*、*B*、*C*, 连接点 1、2, 新增了算法模块 *E*、连接点 5, 以及子处理流 4 衍生出去的关系, 图模型存储时, 关系以指针形式保存, 相对于存储节点, 只需很小的空间.

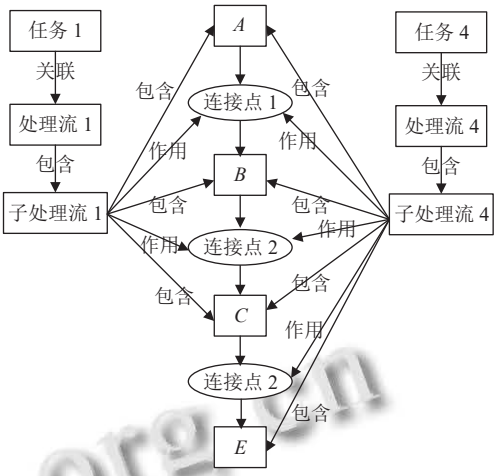


图 8 共用部分模块的子处理流存储图

4.2 处理流自动构建和提取方法

任务的处理流程可能包含多个相同的算法模块, 描述一个任务需要用到任务流程的临时节点 *id*、临时节点 *id* 之间的顺序关系, 以及临时节点 *id* 和算法模块唯一 *id* 的映射关系.

将新的任务加入图模型, 一种方法是先将临时节点 *id* 集合、描述临时节点依赖关系的哈希表 *depend*、描述临时节点后续关系的哈希表 *next*, 通过算法 1, 并设置算法 1 的初始化子处理流起始序号 *subid* 为当前图模型中最大的子处理流序号, 输出得到子处理流集合 Sub_w、子处理流序号集合 Sub_{seq}、描述子处理流依赖关系的哈希表 *subdepend*.

接着根据子处理流节点之间关系、子处理流节点和临时节点之间关系、临时节点和算法模块映射关系在图模型中构造处理流, 具体实现如算法 2 所示.

算法 2. 处理流构造算法

输入: 子处理流集合 Sub_w、子处理流序号集合 Sub_{seq}、描述子处理流依赖关系的二维哈希表 *subdepend*、临时节点 *id* 和算法模块唯一 *id* 的映射表 *Mt*

1) 初始化任务序号 *x*、处理流序号 *y*

2) 在图模型中创建任务节点 *x*、处理流节点 *y*, 并设置任务节点指向处理流节点的关系为关联

3) for *i* in range (0, $Sub_w.size()$):

在图中创建子处理流节点 $Sub_{seq}[i]$, 设置处理流节点 *y* 至子处理流节点 $Sub_{seq}[i]$的关系名为包含;

//遍历子处理流 $Sub_w[i]$

for *t*_id range (0, $Sub_w[i].size()$):

aseq=*Mt*[*t*_*id*]; //获得算法模块序号

if(图模型中算法模块节点 *aseq* 不存在)

创建算法模块节点 *aseq*;

设置子处理流节点 $Sub_{seq}[i]$至算法模块节点 *aseq* 关系为包含;

if(*t_id*>0)
if(算法模块 *Mt*[*t_id*]和算法模块 *Mt*[*t_id*–1]之间没有连接点)
在算法模块 *Mt*[*t_id*]与 *Mt*[*t_id*–1]中间创建连接点, *Mt*[*t_id*]指向连接点, 连接点再指向 *Mt*[*t_id*–1];
设置子处理流节点 *i* 至连接点关系为作用;
4) //遍历存在依赖关系的子处理流
for *key*, *desub* in *subdepend*:
for *dekey*, *value* in *desub*:
图模型中找到子处理流节点 *key* 包含的首个算法模块并用 *first* 表示, 寻找子处理流节点 *dekey* 包含的最后的算法模块并用 *final* 表示;
if(*final* 和 *first* 之间没有连接点)
在 *final* 和 *first* 之间建立连接点
设置处理流节点 *y* 至该连接点的关系为作用
5) 输出: 包含新任务算法关系的图模型

支持往图模型自动添加任务算法关系后, 设计从图模型中提取任务算法关系的算法, 主要分为 3 步. 首先, 获取该处理流包含的子处理流节点, 此刻子处理流亦为无分支子处理流, 接着, 收集子处理流节点包含的算法模块拓扑关系, 最后收集不同子处理流之间算法模块的拓扑关系, 得到完整的处理流信息. 处理流提取算法模块关系的具体实现如算法 3 所示.

算法 3. 处理流提取算法

输入: 处理流节点 *W*
1) 把 *W* 包含的子处理流节点存入集合 $Nodes_w$
2) 初始化映射集 *Mt*, *Mt* 记录算法模块临时 *t_id* 与图模型算法模块 *id* 映射关系
3) 初始化算法模块临时 *t_id* 依赖关系集 *Dt*
4) 设置算法模块临时 *t_id* 初始值为 1
5) 初始化 Hs_w 记录子处理流初始节点 *t_id*, Ts_w 记录子处理流尾节点 *t_id*
6) for *si*, *s* := range $Nodes_w$:
由节点 *s* 获取该子处理流下包含的算法模块 *id*, 并按照连接点逻辑顺序将算法模块 *id* 依次存入 *arry*
7) for *j*, *id* := range *arry*:
键值对 (*t_id*, *id*) 加入 *Mt*;
t_id++;
if(*j*==0) 键值对 (*si*, *t_id*) 加入 Hs_w
if(*j*== *len*(*arry*)–1) 键值对 (*si*, *t_id*) 加入 Ts_w
if(*j*>0) 键值对 (*t_id*, *t_id*–1) 加入 *Dt*
8) 依据 *W* 下的处理流之间的连接点找到输入、输出子处理流, 再根据 *Tsw*、*Hsw* 找到输入子处理流尾节点 *t_id*1 和输出子处理流首节点 *t_id*2, 将键值对 (*t_id*2, *t_id*1) 加入 *Dt*
9) 输出: 任务关联的某个处理流 *W* 的 *Mt* 和 *Dt*

考虑到图模型存储时, 处理流的若干子处理流可能共用同一算法模块, 算法 3 为处理流包含的每个算法模块节点生成唯一临时 *t_id*, 用集合 *Mt* 存储 *t_id* 与图中算法模块 *id* 映射关系, *Dt* 记录临时节点的依赖关系, *Mt* 和 *Dt* 共同表示完整的任务处理流程. 在计算资源足够充沛的前提下, 通过并发执行当前时刻所有无依赖的算法模块, 进而保证任务计算效率取当前时刻最佳.

5 算法管理应用实例

目前, 主流的开源图数据库[15]有 Neo4j、JanusGraph、HugeGraph、Dgraph 等, 其中 Neo4j[16]采用了原生的属性图模型, 具有查询邻接数据快、多维度存储、支持可视化等特点, 长期受业界青睐[17], 本文选用 Neo4j 为工具存储算法关系.

以作者参与的科研项目为例, 该项目首先通过使用 NDVI、EVI 和 SAVI 等 6 种遥感指数[18,19]历史时序数据分别训练 LSTM 神经网络获取各个遥感指数的年际变化规律, 然后根据变化规律对监测期的数据趋势做出较为准确的预测, 并与实际观测提取数据进行对比, 若预测数据与观测数据差异超出一定的阈值范围则判定为异常, 基于 SVM 把异常变化情况分类为生态环境异常变化和非生态环境异常变化, 最终给出生态变化的判断. 接下来对上述算法关系进行存储.

存储管理上述诊断系统的算法关系, 首先参照遥感技术标准体系[20], 建立遥感算法的层级关系, 如图 9 所示. 在图 9 的基础上, 可根据相关标准继续划分遥感算法层级关系, 再将算法程序归类到算法层级, 并且, 还可将算法程序的升级版本归类到相关层级.

本文的算法程序是由 Docker[21]打包的镜像文件, 内含程序文件、运行时环境以及操作系统, 任务的处理流程可兼容 C++、Python、Matlab 等各类语言编写的程序, 图模型用图节点表示算法程序实例.

生态环境异常遥感诊断系统首先并发执行 “NDVI→LSTM→阈值法异常判定”“EVI→LSTM→阈值法异常判定”“SAVI→LSTM→阈值法异常判定”“NBR→LSTM→阈值法异常判定”“MSAVI→LSTM→阈值法异常判定”“NDMI→LSTM→阈值法异常判定” 6 条子处理流, 接着将 6 个中间结果输入 SVM 算法模块获得最终结果.

以“NDVI→LSTM→阈值法异常判定”“EVI→LSTM→阈值法异常判定”两条链路为例, 两条链路共用“LSTM→阈值法异常判定”子链路, 第 1 条链路可以划分为两个子处理流, 子处理流 1 只包含 NDVI, 子处理流 2 包含“LSTM→阈值法异常”判定, 子处理流 3 包括 SVM, 同理, 设置子处理流 4 和子处理流 5 来表示“EVI→LSTM→阈值法异常判定”第 2 条链路, 如图 10 所示.

无论是本文算法关系存储方法还是原始算法关系存储方法, 子处理流节点数量几乎一致, 比如, 考虑到链路“NDVI→LSTM→阈值法异常判定”潜在的语义关系, 原始算法关系存储方法也可将其存储为两条子处理流, 此时也需要增加子处理流节点. 当然, 本文算法关系存储方法和直接存储算法关系方法也可选择不进一步拆分“NDVI→LSTM→阈值法异常判定”链路, 本文方法还可减少子处理流之间的一个连接点.

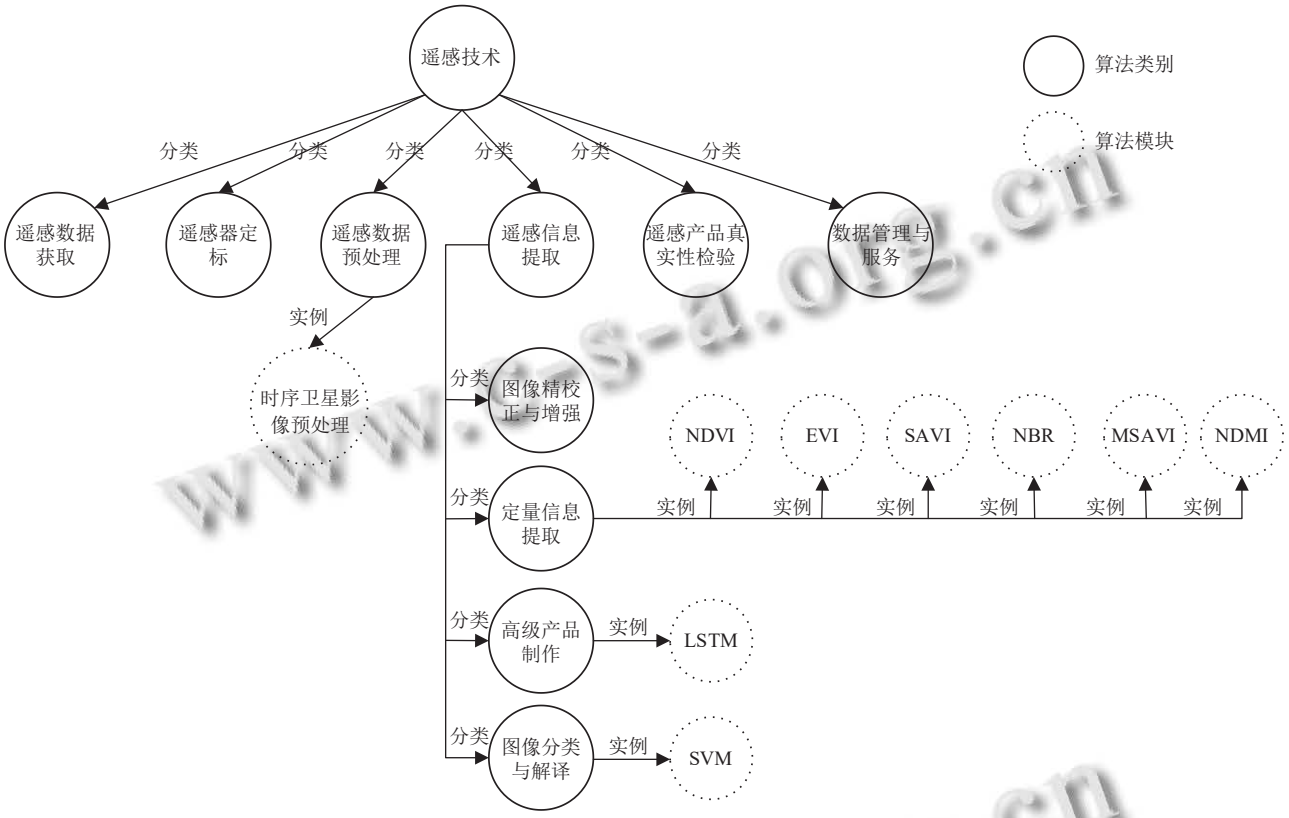


图 9 遥感算法层级分类关系

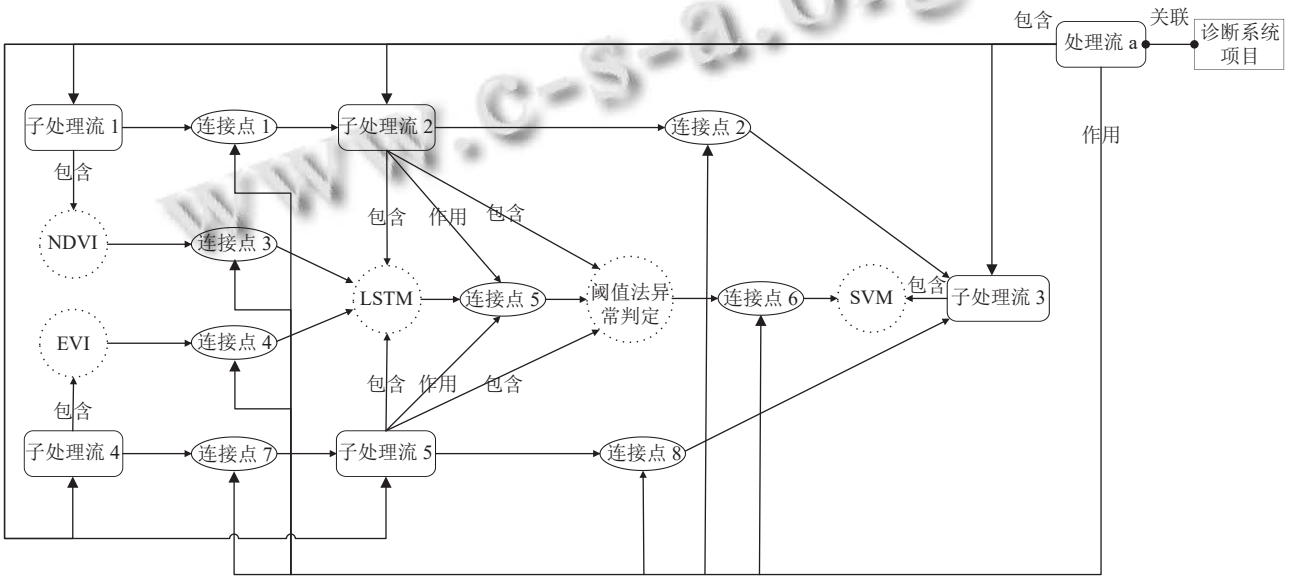


图 10 诊断系统处理流存储图

根据图 10, 表 2 对存在较大差异的算法模块数量和连接点数量进行统计.

表 2 诊断系统节点数目表

链路数	本文存储方法		直接存储	
	算法模块	连接点	算法模块	连接点
1	4	5	4	0
2	5	8	7	0
3	6	11	10	0
4	7	14	13	0
5	8	17	16	0
6	9	20	19	0

本文算法关系存储方法相比算法关系直接存储方法, 随着链路增加, 多次复用 LSTM、阈值法异常判定等算法模块节点, 但是连接点的数量也会增加, 甚至增加的节点总数还多于算法关系直接存储方法. 究其原因, 当前场景下每增加一条链路, 只有 2 个算法模块多一次复用, 却增加 3 个连接点, 因此如果复用的算法模块数量大于 3, 本文算法关系存储方法添加新链路的代价将更低.

分析一般情况, 往图模型中添加新任务的子处理流时, 复用长 m 的子处理流的统计频率为 p, 增加长 n 的新子处理流统计频率为 $(1–p)$, 例如, 在图模型中添加包含 10 条子处理流的一个任务, 复用了原来的 7 条子处理流, 新建了 3 条子处理流, 此时统计频率 p 为 0.7.

本文算法关系存储方法在复用子处理流时, 不增加算法模块, 只需要在复用的子处理流首尾添加常数级别的连接点, 而在增加新的子处理流时, 新增 n 个算法模块, 算法模块之间还需要新增规模 n 的连接点; 算法关系直接存储方法在复用子处理流、添加新的子处理流时, 分别增加 m 和 n 个算法模块. 则本文算法关系存储方法与算法关系直接存储方法需要新增的节点差值如式 (1) 所示.

$$\begin{aligned} diff &= (1-p)\times 2n-(p\times m+(1-p)\times n) \\ &= (1-p)\times n-p\times m \end{aligned} \tag{1}$$

令 $diff<0$ 得式 (2).

$$\ln p+\ln\left(\frac{m}{n}+1\right)>0 \tag{2}$$

不等式 (2) 左项表示为式 (3), 只要 $f(p, m, n)>0$, 本文算法关系存储方法即更优.

$$f(p,m,n)=\ln p+\ln\left(\frac{m}{n}+1\right) \tag{3}$$

$\exists p_1,m_1,n_1$, 使得 $f(p_1, m_1, n_1)=0$, 如图 11 平面, 根据函数单调性, 无论是增大新增任务子处理流复用频率 p, 复用的子处理流长度 m, 还是减小新增的子处理流长度 n, 即 (p, m, n) 落在平面的右边部分, 都将扩大本文算法关系存储方法优势.

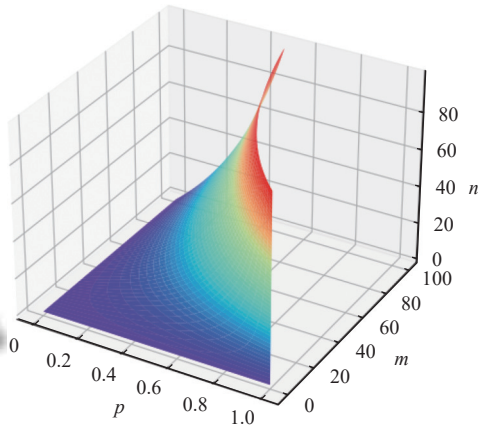


图 11 算法性能判别图

随着图模型存储的算法关系愈加丰富, 统计频率 p 的值会逐渐增加, 又因为当前新增的子处理流会成为下次任务复用的候选项, 因此 m、n 的值将趋于相等. 一种理想的情况是当 p、m/n 接近于 1, 此时 $f(p, m, n)$ 总是大于 0, 如式 (4) 所示, t 表示图模型拓展的次数.

$$\lim_{t\to\infty} f(p,m,n)=\lim_{t\to\infty}\left(0+\ln\left(\frac{m}{n}+1\right)\right)=\ln 2>0 \tag{4}$$

在算法关系表示方面, 以 LSTM 算法模块为例, 本文算法关系存储方法在图 10 中先检索 LSTM 节点, 再通过输出至该节点的连接点 3、4, 在小范围内找到依赖的模块 NDVI 和 EVI, 并且, 以算法模块为中心, 通过“包含”关系, 可以找到使用该算法模块的任务. 但是以任务为单位直接存储算法关系的方法需要遍历每个任务, 再判断该算法模块是否被任务所使用, 以及算法模块依赖的其他模块. 由此可见, 本文的算法关系存储方法可帮助研究人员更好地组织、发现不同领域下大量任务的算法关系, 从而提高科学工作的效率.

6 结束语

本文结合遥感领域的算法实例, 开展算法关系管理研究. 首先, 本文从项目工程经验中提取算法关系, 构建算法元组概念集, 具体包括算法类别、项目、处理流、算法模块, 概念集的关系主要有层级关系和拓扑依赖关系. 接着, 针对项目所涉及的处理流程存在复杂拓扑关系的问题, 本文提出处理流的拆分方法. 然后,

本文通过传统存储拓扑关系方法在算法关系管理方面的不足, 提出以图模型存储算法关系的方法. 最后, 在图数据库 Neo4j 中验证了本文算法关系管理方法.

本文的算法关系管理方法支持根据历史经验搜寻算法模块的后续节点, 基于算法深层关系, 研究如何在构建任务解决方案时为处理流的某个环节推荐具体的算法, 是未来充分发掘与应用算法组织关系能力的重要方向.

参考文献

1 覃雄派, 王会举, 李芙蓉, 等. 数据管理技术的新格局. 软件学报, 2013, 24(2): 175–197. [doi: 10.3724/SP.J.1001.2013.04345]

2 王浩宇, 郭耀, 马子昂, 等. 大规模移动应用第三方库自动检测和分类方法. 软件学报, 2017, 28(6): 1373–1388. [doi: 10.13328/j.cnki.jos.005221]

3 赵亮, 陈志奎. 大数据算法库教学实验平台设计与实现. 实验技术与管理, 2020, 37(6): 197–201, 206.

4 郑毅, 郑苹. 基于 C++标准模板库的 STL 数据拓扑重建. 工程设计学报, 2013, 20(6): 522–528. [doi: 10.3785/j.issn.1006-754X.2013.06.013]

5 陈诗军, 王慧强, 吕宏武, 等. 以构造为中心的底层软件复用方法. 计算机工程与设计, 2019, 40(5): 1370–1375.

6 王冰, 黄华国, 王景旭, 等. 耦合 ENVI-met 与 RAPID 模型模拟 3 维异质植被场景亮温分布. 遥感学报, 2020, 24(2): 126–141.

7 汪家亮, 赵银娣, 韩天庆. 结合分割线网络的遥感影像镶嵌及 ENVI/IDL 实现. 遥感信息, 2018, 33(6): 92–96. [doi: 10.3969/j.issn.1000-3177.2018.06.013]

8 Liu YJ, Wang SG, Zhao QL, *et al*. Dependency-aware task scheduling in vehicular edge computing. IEEE Internet of Things Journal, 2020, 7(6): 4961–4971. [doi: 10.1109/JIOT.2020.2972041]

9 Zhang QC, Zuo M, Cao Q, *et al*. Research on SoftMan component dynamic evolution system and its distributed multi-tasks collaboration mechanism based on game theory. Chinese Journal of Electronics, 2018, 27(4): 783–791. [doi: 10.1049/cje.2018.01.004]

10 陈杰, 文艳军, 王戟. 一种软构件依赖关系的拓扑布局算法. 计算机工程与科学, 2008, 30(5): 56–58. [doi: 10.3969/j.issn.1007-130X.2008.05.018]

11 许栋梁, 赵健, 王小宇, 等. 基于有向邻接矩阵的配电网拓扑检测与识别. 电力系统保护与控制, 2021, 49(16): 76–85.

12 Bhunia P, Bag S, Paul K. Bounds for eigenvalues of the adjacency matrix of a graph. Journal of Interdisciplinary Mathematics, 2019, 22(4): 415–431. [doi: 10.1080/09720502.2019.1630938]

13 唐德权, 张波云. 基于路径的频繁子图挖掘算法研究. 计算机工程与科学, 2019, 41(12): 2223–2230. [doi: 10.3969/j.issn.1007-130X.2019.12.018]

14 刘春江, 李姝影, 胡汗林, 等. 图数据库在复杂网络分析中的研究与应用进展. 数据分析与知识发现, 2022, 6(7): 1–11.

15 王鑫, 邹磊, 王朝坤, 等. 知识图谱数据管理研究综述. 软件学报, 2019, 30(7): 2139–2174. [doi: 10.13328/j.cnki.jos.005841]

16 郭家鼎, 王鹏. 基于数据仓库的典型图查询处理技术. 计算机工程, 2023, 49(9): 32–42.

17 Kamal SH, Elazhary HH, Hassanein EE. A qualitative comparison of NoSQL data stores. International Journal of Advanced Computer Science and Applications, 2019, 10(2): 330–338.

18 范德芹, 赵学胜, 朱文泉, 等. 植物物候遥感监测精度影响因素研究综述. 地理科学进展, 2016, 35(3): 304–319.

19 颜庆. 基于 RNN/LSTM 导航卫星反射信号特征融合的植被指数 NDVI 反演建模方法[硕士学位论文]. 徐州: 中国矿业大学, 2022.

20 李传荣, 刘照言, 王新鸿, 等. 国家级遥感技术标准体系逐步完善. 卫星应用, 2021(6): 16–20. [doi: 10.3969/j.issn.1674-9030.2021.06.005]

21 应毅, 刘亚军, 俞琰. 利用 Docker 容器技术构建大数据实验室. 实验室研究与探索, 2018, 37(2): 264–268. [doi: 10.3969/j.issn.1006-7167.2018.02.064]

(校对责编: 孙君艳)