ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.4, April 2009, pp.918−929
doi: 10.3724/SP.J.1001.2009.03225

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于代价模型的不一致 XML 数据修复启发式计算*

吴爱华[1,2], 王先胜[1], 谈子敬[1+], 汪 卫[1]

[1](复旦大学 计算机信息与技术系,上海 200072)

[2](上海海事大学 计算机科学系,上海 200135)

A Cost-Based Heuristic Algorithm for Repairing Inconsistent XML Document

WU Ai-Hua[1,2], WANG Xian-Sheng[1], TAN Zi-Jing[1+], WANG Wei[1]

[1](Department of Computer Information Technology, Fudan University, Shanghai 200072, China)

[2](Department of Computer Science, Shanghai Maritime University, Shanghai 200135, China)

+ Corresponding author: E-mail: zjtan@fudan.edu.cn

Wu AH, Wang XS, Tan ZJ, Wang W. A cost-based heuristic algorithm for repairing Inconsistent XML document. *Journal of Software*, 2009,20(4):918−929. http://www.jos.org.cn/1000-9825/3225.htm

Abstract: Computing a repair for inconsistent XML documents is significant in applications. But getting an optimum repair is a NP complete problem, especially when XML documents violate both the function dependence and the key constraints. This paper proposes a cost-based heuristic algorithm, which can find a repair with the lowest cost in polynomial time. It first scans the original XML documents once to get the inconsistent data. Then it computes the general candidate repairs for each inconsistent data, and gets a whole document repair heuristically based on its cost. The experimental evaluation show that even when XML documents are large, with high percent of dirty elements, and against many different constraints, the algorithm can still run in less than $O(n^3)$ w.r.t. the size of inconsistent elements.

Key words: inconsistency; inconsistent data; repair; consistent answer; XML data cleaning; incomplete database

摘 要: 在实际应用中,为不一致的 XML 文档计算最优修复意义重大.但求解最优修复是一个 NP 完全问题,特别是在 XML 文档同时违反函数依赖约束和主键约束时.提出一个基于代价模型的、可以在多项式时间内完成的启发式修复求解算法.该算法首先借助索引表,在一遍扫描原始 XML 文档的情况下寻找不一致数据集,然后为每一类约束的不一致数据集构造候选修复,同时计算其修复代价,最后启发式地求解一个代价最小的修复方案.实验结果表明,该算法的时间复杂度不超过冲突类的 3 次方,即便是在不一致数据量很大、噪声比例很大以及涉及多类语义约束时,也能较快地完成修复.

关键词: 不一致性;不一致数据;修复;一致的查询回答;XML 数据清洗;不完整数据库

中图法分类号: TP311 **文献标识码**: A

数据完整性约束是准确数据必须遵循的准则,也是数据的语义所在.但在实际应用中,因各种原因,数据常违反其完整性约束,如错误操作、多数据源合并等.在 XML 下该问题尤为突出.因为:(1) 数据本身的问题,作为

* Supported by the National Natural Science Foundation of China under Grant No.60603043 (国家自然科学基金)
Received 2007-08-22; Accepted 2007-11-02

一种半结构化数据,XML 的数据模型复杂,语法灵活,特别是 W3C 缺乏 XML 约束方面的严格规范;(2) XML 广泛应用于数据交换和数据集成,而这类应用环境下,即便单数据源都一致,但当多数据源合并时,仍常发生数据冲突.因此,不一致 XML 文档的修复意义更为重大.

例 1:设图 1 中的 XML 数据是两数据源合并的结果,实线部分来自 *A*,虚线部分来自 *B*.描述的是某港机公司承接的项目,project 表示项目,projectName 能够标识一个 project,为完成某 project,须采购若干零部件(part),partID 标识 part,每个 part 具有来源(source)、颜色(color)和安装日期(date)等属性.而 source 又包含供应商名(comName)和该部件的售价(price).根据图 1 所示的约束,该 XML 数据中存在以下不一致的数据:

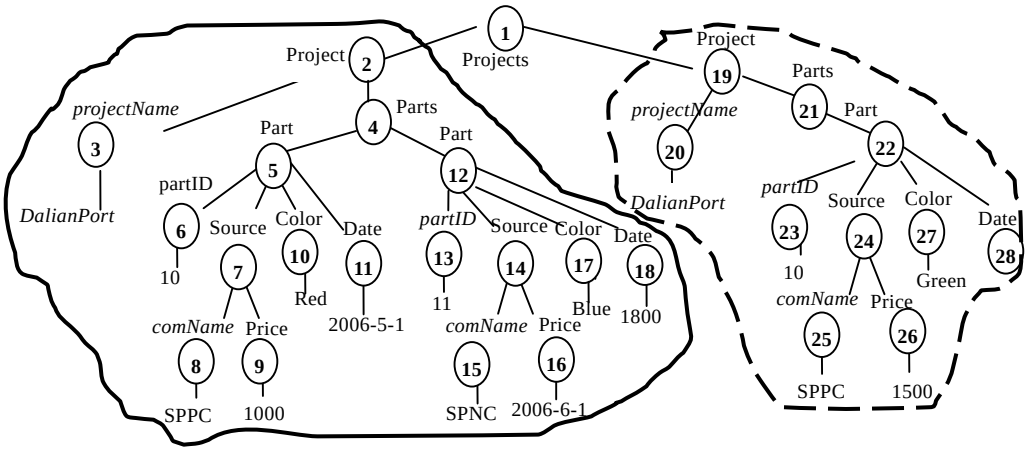
(1) 存在两个名字都叫 DalianPort 的 project;

(2) SPPC 公司生产的同一个零部件在实线子树中为 red,但在虚线子树中却为 green;

(3) SPPC 公司提供的 10 号零件,在实线中价格为 1 000,但在虚线中却为 1 500;

(4) 实线子树中,11 号零件在 2006-5-1 安装;

(5) 虚线子树中,10 号零件的安装日期为空.



Related constraints:

*ic*1: In the whole document, *projectName* identify a project (ε,*projects*/*project*,(*projectName*))([x_1],[x_2]⇒x_1!=x_2)

*ic*2: In a project, *partID identify a* part (*projects*/*project*,*parts*/*part*,(*partID*))([x_1],[x_2]⇒x_1!=x_2)

*ic*3: In the whole document, *comName* determine color of part (All parts of same company is of same color)
(ε,*projects*/*project*/*parts*/*part*,(*source*/*comName*,*color*))([*x*,y_1],[*x*,y_2]⇒y_1=y_2)

*ic*4: In the whole document, *partID* and *comName* determine price of par
(ε, *projects*/*project*/*parts*/*part*,(*partID*,*source*/*comName*,*source*/*price*)) ([(*x*,*z*),y_1],[(*x*,*z*),y_2]⇒y_1=y_2)

*ic*5: Install date can not be NULL, and No part can be installed on 2006-5-1
(ε,*projects*/*project*/*parts*/*part*,(*date*))([*x*]⇒*x*!=2006-5-1) (ε,*projects*/*project*/*parts*/*part*,(*date*))([*x*]⇒*x*!=*NULL*)

Fig1 An inconsistent XML section

图 1 一个不一致的 XML 数据片段

在图 1 这样的不一致数据源上提交查询,回答可能包含不正确信息,因此必须修复不一致数据源.根据文献[1],修复是没有违反完整性约束且和原数据距离极小的数据.不一致数据的修复在数量上巨大,求解其所有修复是一个 maxNP[2]难问题,时间消耗在指数级.而在实际应用中,求解所有修复和求解最优修复的意义基本一样.虽然后者也是一个 NP 完全问题(参见第 2 节),但可以使用启发式算法寻求到多项式时间内的解.

每一类约束都有其独特的修复特征,如:主键约束要求违反它的子树在主键上取不一样的值;而函数依赖约束的修复,则内在地要求将所有语义上违反了同一个函数依赖的子树的目标节点置等.若数据只违反一类约束,则寻找多项式时间内的修复方法也不是一件难事.但在实际应用中,数据的不一致性是综合的.只考虑一类约束或逐个修复不一致数据,都可能给出错误修复.因为一条记录的变动可能影响其他记录,导致新的冲突产生.这两类做法意义都不大.好的修复算法应该能够考虑到同一节点违反多约束的情况.而各类约束中,内在矛盾的是

置等和置不等两类约束,置等类约束要求所有违反同一约束实例的子树都必须在目标节点上取相同值,而置不等类约束则要求所有违反同一约束实例的子树在目标节点上取不同值.总之,好的修复方案应具有如下特点:(1) 能够适用尽可能多的常用约束类型;(2) 能够在多项式时间内完成;(3) 能够综合考虑所有不一致数据的修复,不会带来新的不一致;(4) 能够修复规模大和错误率高的 XML 文档;(5) 准确率高.

本文综合考虑了多类约束,包括:(1) 域约束;(2) 根据其他节点的取值来限定某个节点的取值;(3) 函数依赖;(4) 主键约束.涵盖了实际应用中的大部分约束.设计了一种基于代价模型的启发式算法,能够在多项式时间内寻找不一致的 XML 文档的一个最优修复.本文的贡献在于:

- 提出了适合 XML 文档修复的基于代价模型的启发式算法,能够在多项式时间内寻找整个 XML 文档的修复.实验证明,这种方法确实能够缩短时间;
- 提出了适合 XML 的代价模型,因为其树状特性,XML 的代价模型比关系数据的代价模型要更复杂,本文充分考虑了 XML 的特性,给出了节点修改、节点删除、节点插入、子树删除和子树插入的代价模型,而且在代价模型中还考虑到操作类别、距离和可信度 3 个要素;
- 能够同时解决置等和置不等约束,在扩大问题范围之后,仍能取得较好的时间性能;
- 扩展了冲突类的概念和冲突类索引表,并给出使用冲突类索引表检索 XML 文档中冲突类的算法,它能在一次遍历下,检索出 XML 中所有的不一致数据,这使得本文能够处理大型 XML 文档.

1 相关工作

关于数据一致性问题,关系数据库和数据交换领域已展开了广泛的研究,包括[1−6],但 XML 领域尚处在起步阶段.一致信息一般通过下面两种方法得到:(1) 修复,即寻找与原文档距离最小的一致数据;(2) 一致的查询回答,即找出能够在所有修复中出现的公共回答.本文使用的是第 1 种方法.

本文借鉴了关系领域中的基本概念和方法:将文献[1]提出的修复概念延伸到 XML 领域;将文献[7]提出的主键索引表修改并应用到 XML 领域;借鉴并丰富了文献[4,5]在关系数据清洗中提出的基于代价模型的启发式算法.文献[4]针对的是函数依赖和包含依赖,其代价模型要素包括数据源可信度和目标属性的字符编辑距离.提出了平衡类的概念,以归结所有应共享同一目标值的属性.算法的基本思路是,先扩展平衡类,再根据代价选择最优目标值.文献[5]针对的是数值类非主属性的修复,以目标属性的数值距离为基本代价,并提出了覆盖代价的概念.其基本思路是,找出所有相互矛盾的数据集(约束违反集),寻找一个能够解决矛盾且距离原值最小的目标值来修复约束违反集,同时计算其覆盖代价,根据它,启发式地计算整体最优解.但 XML 的结构、约束、查询和修改操作远比关系数据库要复杂,这些方法不能直接应用到 XML 领域.如:Repair 概念需重新修改以适应树状结构;文献[4,5]的代价模型对树状结构不适用;修复计算方法不能直接应用到 XML 领域.本文将平衡类和约束违反集的概念扩展为适合 XML 的冲突类,重新定义了适合 XML 且考虑多修复操作的代价模型,给出了适合 XML 的冲突类修复的计算标准.

就 XML 领域而言,已有相关工作可以归纳为:

第 1 类,不一致 XML 文档修复的整体框架.比如文献[8],它通过构造所有修复,再求交集来得到一致数据,复杂度高,耗时大.尽管两者考虑的约束模型类似,但本文求的是最优修复而不是所有修复,效率更高.

第 2 类,XML 数据集成和变换中的语义约束保持和查询结果一致问题.文献[9]研究的是如何将遵循原 DTD(document type definition)的 XML 文档重构为遵循目标 DTD 的 XML 文档,并如何进行相应的查询改写.文献[10]讨论了 XML 数据集成中如何保证函数依赖的一致.其应用背景和研究目的都和本文的不尽相同.

第 3 类,某类约束的 XML 数据一致性.文献[11]讨论了一组函数依赖上的不一致数据的修复问题,主要使用的操作是节点修改.文献[7]着重于 XML 键值重复的修复.本文的约束类型更宽泛,但其困难也更大.文献[12]解决的是违反 DTD 的不一致 XML 数据的修复问题,而不考虑 XML 上的完整性约束,这和本文的工作有所不同.

与已有 XML 修复工作相比,从约束类型上说,本文能够综合修复多类型约束.从修复操作上说,前人工作只支持节点删除和节点插入两种,本文支持删除、修改和插入,并引入操作系数,以优先某种操作.

2 系统模型和代价模型

XML 文档上的约束包括结构方面的约束(由 DTD 或 XML Schema 定义)和语义方面的约束(由 XML 语义约束表示),本文处理的是后者.本文的修复对象是无 DTD 或 XML Schema 的 XML 文档.至于如何保证 XML 文档上不存在违反其 DTD 或 Schema 的数据,文献[12]对其进行了研究.另外,本文假设所处理的 XML 文档已被先序编号,系统也已按(起始标号,结束标号)为 XML 文档建了索引.本文允许节点修改、子树(节点)的删除和插入 3 种操作.这里,我们约定删除以节点 n 为根的子树的操作,记作 $delete(n)$,在节点 n 下面插入一个根节点标签为 tag 的子树,记为 $insert(n,tag)$,而将子树 t_1 修复为子树 t_2,则记作 $repair(t_1,t_2)$.

定义 2.1. 给定一个 XML 模式 D,D 上的语义约束是这样的一个二元组$(R_1,R_2,(Q_1,Q_2,\ldots,Q_n))(X_1,X_2,\ldots,X_m\Rightarrow$逻辑表达式),其中,$R_1,R_2,Q_i(1\le i\le n)$是路径表达式[13],$R_1$ 是约束的作用范围,R_2 是目标节点,$Q_i(1\le i\le n)$是约束节点,X_1, $X_2,\ldots,X_m\Rightarrow$逻辑表达式是约束条件,逻辑表达式的每一个关系表达式都形如 $u\theta w$,其中,θ内置谓词,u 是变量或包含变量的表达式,而 w 是常量、变量或包含变量的表达式.整个约束表示在 R_1 范围内,所有的 R_2 节点上,如果 $Q_1,Q_2,\ldots,Q_n$ 满足 $X_1,X_2,\ldots,X_m$,就必须要使得条件为真.

XML 语义约束总是与其作用范围联系在一起,是一个相对概念.

定义 2.2. 形如$(R_1,R_2,(Q_1,\ldots,Q_n))([(X_1,\ldots,X_{n-1}),y_1][(X_1,\ldots,X_{n-1}),y_2]\Rightarrow y_1=y_2)$的约束称为置等约束.

图 1 的约束 $ic3$,$ic4$ 就是这类约束,它们内在要求找出所有具有相同的 $Q_i(1\le i\le n-1)$的目标节点在 Q_n 上的统一值.可以发现,函数依赖是一种置等约束.

定义 2.3. 形如$(R_1,R_2,(Q_1,\ldots,Q_n))(X_1,\ldots,X_m\Rightarrow u!=w)$类的约束称为置不等约束,其中,$u$ 和 w 都是变量.

图 1 的约束 $ic1$ 就是这种约束.可以发现,主键约束是一种置不等约束.

本文能够解决的实际应用中的大部分约束,具体来说,包括:

- 形如$(R_1,R_2,(Q))([x]\Rightarrow x\theta w)$的一维一元约束条件,维数指的是约束节点的个数,元指的是约束条件中“⇒”左边的分组数.
- 下面的一维二元或一维多元约束:要求这 n 棵子树在约束节点上的取值必须满足某种内在谓词约束.一维的置等和置不等类约束就是这类约束的一个特例.
- 下面两种多维一元约束:一种是当其他 $n-1$ 维的值是确定的,要求某一维上的值满足一定的内在谓词约束;第 2 种则是定义这 n 维的值之间满足一定的内在谓词约束.
- 维数小于 4 且目标节点只有一个的二元置等约束.
- 二元置不等约束,且“!=”两边都只涉及一个节点.

定义 2.4[1]. 给定 XML 文档 T 和 T' 及约束集 IC,如果 $T|\neq IC$,而 $T'|=IC$,且不存在这样的 T'':1) $T''|=IC$;2) $T'\subseteq T''\subseteq T$;3) $\Delta(T'',T)<\Delta(T',T)$,那么 T' 叫做 T 的一个修复.特别地,T' 中允许绑定到变量的节点.

定理 1. 设有一个 XML 文档 X 和其上的约束集 IC_s,修复 X 的每步操作都对应着一个代价 λ,那么寻找满足下列条件的 X'(简称为 XORP 问题)是一个 NP 完全问题:$X'|=IC_s$,且 $\Delta(X,X')$极小,且 $\Sigma\lambda$最小.

证明:显然,用最质朴的方法——回溯,可以求解得到 X',其复杂度为指数级,因此这是一个 NP 问题.进一步地,只要存在一个已知的 NPC 问题 Q,若通过多项式变换后,能将 XORP 问题转变为 Q,则命题成立.

最小权重集合覆盖问题(minimal weight set cover problem,简称 MWSCP):给定一个集合 U,S 是 U 的子集;$\cup S=U$(能覆盖 U),且 S 的每个元素都对应一个权重,那么能不能寻找 U 的一个可覆盖它的权重总和最小的子集集合.MWSCP 是一个 NP 完全问题[14].这里把 MWSCP 和 XORP 问题的多项式映射 f 定义为 $U=\{V_i|V_i=\{R|R$ 是冲突类 C_i 的一个候选修复,具有代价$\},1\le i\le|$冲突类集 $C|\}$,R_i 之间是或的关系,S 的代价是其元素的代价和.下面证明这确实是一个多项式映射.

首先,只要为每个冲突类构造其候选修复,就能最终形成 U,这一步的时间代价是多项式的,即映射的时间代价是多项式(参看第 4.1 节).它具有多项式映射的第 1 个性质.其次,设集合 SS 是 XORP 的解,也就是说,$\forall V_i\in U$,$\exists R\in SS,R\in V_i$,因为 V_i 的成员 R 间是或的关系,任意一个 R 都能代表 V_i,所以,SS 也是 MWSCP 的解.反过来,设集合 SS 是 MWSCP 的解,SS 必然能够修复 X,得到的 X'满足 $\Delta(X,X')$极小,且 $\Sigma\lambda$最小,因为任何多余的操作都意味着

代价的增加,所以 *SS* 也是 XORP 的解.因此,该映射也满足多项式映射的第 2 个性质. □

本文的代价模型由 3 部分组成:操作类别系数、节点的可信度以及原值和目标值之间的距离.其中,操作类别系数和节点的可信度作为参数,由用户指定.

定义 2.5. 两个字符串 A 和 B 之间的距离 $dist(A,B)=|\{n|A[n]!=B[n]\}|/\max(len(A),len(B))$.

假设 $\lambda_1,\lambda_2,\lambda_3$ 分别是修改、删除和插入的操作系数,R_n 是节点 n 的可信度,$dist(N,O)$是目标值 N 和原值 O 之间的距离,*Null* 是指空字符串,那么:

定义 2.6. 节点 n 的值修改代价 $Mncost=\lambda_1\times R_n\times dist(N,O)$.删除叶子节点 n 的代价 $Dncost=\lambda_2\times(\lambda_1\times R_n\times dist(null,O)+1)$.插入叶子节点 n 的代价 $Incost=\lambda_3\times(dist(N,null)+1)$.

叶子节点是节点和值的组合,所以,删除叶子节点实际上是先将它的值修改为空字符串,再删除它.相应的代价也由值修改和节点删除两部分组成,而删除一个节点的代价可以看作是一个单位时间.同理,插入叶子节点的代价也由值修改和节点插入两部分代价组成.

子树操作归根到底由上面 3 种原子操作组成,因此,一个以节点 n 为根节点且层高为 H 的子树的删除代价可以用公式(1)来表示.同理,在节点 n 下面插入一个标签为 *tag* 且层高为 H 的子树,其代价为公式(2):

$$Dtcost(n,H)=\lambda_2\times\begin{cases}k+\sum_{i=1}^{i\le k}Dtcost(n',H-1), & H>1\\ Dncost()/\lambda_2, & H=1\end{cases} \tag{1}$$

$$Itcost(n,tag,H)=\lambda_3\times\begin{cases}k+\sum_{i=1}^{i\le k}Itcost(n',tag',H-1), & H>1\\ Incost()/\lambda_3, & H=1\end{cases} \tag{2}$$

将子树 t 修改为子树 t',其修复代价可以用下面的公式表示:

$$Rtcost(t,t')=\sum Itcost+\sum Dtcost+\sum Mncost \tag{3}$$

即,子树的修复代价是将子树 t 修改为 t'所发生的子树删除、子树插入和节点修改的代价之和.

3 冲突类和冲突索引表

相互矛盾的数据叫做一个冲突类.修复的第 1 步就是找出 XML 文档中所有的冲突类.

定义 3.1[5]. 设 IC 是 XML 文档 x 上的约束集,如果 x 的节点集 $\delta\neq ic(ic\in IC)$且对任意 δ 的真子集 δ'都有 $\delta'|=IC$,那么,δ叫做 x 中 ic 的一个违反包.违反包的修复目标态是其终态 t,记作 $F(\delta)=t$.设 $I=\{(\delta,ic)|\delta\neq ic\wedge ic\in IC\wedge F(\delta)=t\}$,那么,$C=(I,t)$叫做冲突类.

例 4:图 1 中存在着下面的几个冲突类:$C_1=\{((2,19),ic1),t_1\}$,$C_2=\{((5,22),ic3),t_2\}$,$C_3=\{((5,22),ic4),t_3\}$,$C_4=\{((5),ic5.1),t_4\}$,$C_5=\{((22),ic5.2),t_5\}$,所有的终态将随着候选修复方案的确定而确定.

定义 3.2. 给定一个冲突类 $c(I,t)\in C$,c 的修复代价 $RCcost(c)$是将 I 的所有元素修复为 t 的代价和,即

$$RCcost(c)=\Sigma Rtcost(\delta,t)(\delta\in I).$$

本文使用扩展的冲突索引表[7]规约冲突类.见表 1,它有 5 层描述属性:约束名,表示当前节点可能违反的约束;范围节点是约束的作用范围;目标节点是约束作用的目标节点,也是约束节点的共同祖先节点;约束节点及其取值.同一个节点可能出现在多个节点中,同样,同一个约束也可能出现在多行中.

冲突类的规约算法在算法 1 和算法 2 中给出.其基本思想是,在 XML 文档遍历的过程中构造冲突类表.每当遍历到叶子节点时,就开始寻找匹配的约束,有的约束包含多条约束节点路径,而遍历到某条约束节点路径时,其他约束路径并未遍历,因此需将当前路径暂存在堆栈中,等遍历到最后一个约束时,才来填写冲突类表.一维约束只涉及一个节点,在构造冲突类表时,可判断这类冲突的存在,并写入冲突类表

表 1 是按照算法 1、算法 2 构造出来的图 1 的冲突类表的一部分.冲突类表构造成功以后,可以将它按照约束名、范围节点和目标节点进行排序,将违反同一个约束的相关节点排在一起,便于规约.因为所有相关冲突类必然具有相同的约束名和范围节点,且在约束节点上的取值违反了约束的定义.

Table 1 Part of the conflict class index table for example 1

表 1 例 1 的部分冲突类索引表

Constraint name	Sub. tree root	Target node	Constraint node	Value
*ic*1	1	2	*projectName*	*DalianPort*
*ic*2	2	5	*partID*	10
*ic*4	1	5	*partID*	10
			comName	SPPC
			price	1000
*ic*3	1	5	*comName*	SPPC
			color	Red
*ic*5	1	5	*date*	2006-5-1
*ic*2	2	12	*partID*	11

算法 1. *BuildConflictTable*(XML file F,IC_s)构造冲突类表.

输入:XML 文档 F 及其上的约束集合 Ic_s;

输出:冲突类表 *CT*.

//采用深度优先的方法遍历 XML 树,每次遍历到最底层后开始寻找匹配的约束,填写冲突类表.

P 指向 *root*,*s*=“ ”

while (存在未访问的 *root* 子节点)

{//遍历到叶子节点

q=p 的第 1 个未访问过的儿子

a) While (q→*type*==*element*){1. 记录路径;2. P=q;q=q→下一个未访问过的儿子.}

b) *link*(*s*,*K*) //将路径 *s* 推入链表 *K* 中

c) 寻找匹配的约束(从第 1 个约束开始)

d) 若没找到匹配的 *ic*,那么 *deleteFromLink*(*s*,*k*),然后执行 e),否则:

i. If (*ic* 还包括其他未在栈 *k* 中的约束节点路径),执行 e)

ii. 找出约束的范围节点、目标节点 n_2 和所有约束节点及其值,填写冲突表

iii. 如果约束的右侧是常量的话,若存在冲突,则归约出冲突类.

iv. 将栈中所有不出现在其他约束中的路径 *s* 删除.

e) 寻找下一个可以访问的路径}

算法 2. *GetConflictClass*(*CT*)归约冲突类.

输入:冲突类表 *CT*;

输出:冲突类栈 *CC*.

1. *sortCT*() //按约束名、范围节点、目标节点将冲突类表排序
2. 链表 *v* 置为空
3. for (*i*=0;未到 *CT* 的尾部;*i*++)

a) *push*(*ct*[*i*],*v*);*k*=*ct*[*i*];

b) while (*ct*[++*i*].约束名=*k*.约束名 && *ct*[*i*].范围节点=*k*.范围节点 && *ct*[*i*].约束节点=*k*.约束节点 && 未到 *CT* 的尾部) *link*(*ct*[*i*],*v*);

c) 将链表 *v* 中的冲突类表项按照约束定义进行比较,如果违反了约束,则归约出冲突项

d) 置链表 *v* 为空.

4 修复的启发式计算

规约冲突类后,需使用合适的方法修复它们,以最后修复 XML 文档.正如前文所述,冲突类间可存在错综复杂的关系:① 一个节点可能参与多个冲突类,不同的冲突类对其有不同的终态要求;② 一个冲突类可能涉及多

个节点,它们的取值相互关联.这些特点要求能够一次全面修复所有的冲突类.

本文将寻找修复的过程分成 3 步:(1) 计算每个冲突类的候选修复,候选修复的节点取值未必确定,可能用变量和该变量上的限制条件表示;(2) 计算每个候选修复的修复代价及其覆盖代价,并使用启发式算法确定不会和内含候选修复相冲突且代价最小的修复.整个算法做下来可能找不到解,因为有些文档本身不存在修复;本文允许修复中存在变量绑定,但有些节点的取值可以根据其他节点的值推算出来,如函数依赖类约束中的节点的取值就可以根据相互冲突的其他节点的值推算出来.同时,修复内各个候选修复之间可能可以合并,因此,(3) 合并能合并的候选修复,并将值能够确定的变量用常量替换,得到最后的修复.

4.1 冲突类候选修复的计算及其修复代价

定义 4.1. 给定一个冲突类 $C(I,t)$,冲突类的候选修复 rc 是能够修复 I 的操作序列.可以包括节点修改、子树(节点)删除和子树(节点)插入,并允许节点绑定到变量.

每个冲突类都有一组自己的候选修复.其具体求解方法如下:

置等类约束,设冲突类 cc 中有 $n_1,n_2,\ldots,n_k$ 个节点,$n_i.U(1\le i\le k)$相等,但 $n_i.V(1\le i\le k)$不相等,分别是 $m_1,m_2,\ldots,m_k$,则 cc 的候选修复可以是:RC_1 修改 $n_i.V(1\le i\le k)$,使得 $n_i.V(1\le i\le k)=x(x=m_1$ 或 m_2 或…或 $m_k)$,x 取其他值意义不大;RC_2 修改 $n_i.U(1\le i\le k)$,使得 $n_i.U(1\le i\le k)!=n_j.U(1\le j\le k,j!=i)$;$RC_3$ 删除 $n_1,n_2,\ldots,n_k$ 中的任意 k–1 个节点.最坏情况下,设 m_1~m_k 互不相同,U 是 L 个属性的属性集,那么,RC_1 类候选修复有 k 个,RC_2 类候选修复有 k^L 个,RC_3 类候选修复有 k 个.由于 $L\le 2$(处理的置等类约束的维<4),置等类约束的冲突类的候选修复最多有 k^2+2k 个,是$|I|$的平方级.

置不等类约束,因为不等约束只针对一个约束节点,设为 A,设冲突类 cc 有 $n_1,n_2,\ldots,n_k$ 个节点,$n_i.A(1\le i\le k)$相等,设为 M,cc 的候选修复可以是:RC_1 修改 $n_i.A(1\le i\le k)$,使得 $n_i.A(1\le i\le k)!=n_j.A(1\le j\le k,j!=i)$;$RC_2$ 删除 $n_1,n_2,\ldots,n_k$ 中的任意 k–1 个节点.RC_1 类候选修复有 k 个,RC_2 类候选修复有 k 个,共有 $2k$ 个候选修复.

其他约束的候选修复为:(1) 一维一元约束,设冲突类的目标节点为 n,约束是 $n.A\theta w$,则候选修复为:RC_1 修改 $n.A$ 的值,使得 $n.A=x(x\theta w)$;RC_2 删除 n.共有两个候选修复;(2) 一维二元约束,假设约束条件有两个变元 x,y,其中,$x=N$(N 是常量),此时,约束要求 $y\theta w$,则修复可以有:RC_1 修改 x,使得 $x!=N$;RC_2 修改 y,使得 $y\theta w$;RC_3 删除 x 所在的子树;RC_4 插入子树,使得该子树在约束节点上的取值满足约束.设约束条件涉及两个变量,如 $x>y$,那么,RC_1 修改 x,使得 $x>y$;RC_2 修改 y,使得 $x>y$;RC_3 删除 x 所在的子树;RC_4 插入子树,使得该子树在约束节点上的取值满足约束.无论哪一种,修复的数量≤4;(3) 一维多元约束,基本思想和一维二元的类似,是限定每一元变量的取值,使得约束能够得以满足.假设有 k 个变量 $x_1,x_2,\ldots,x_k$,那么修复有:RC_1 修改第 $i(1\le i<k)$个变元的值,使得它的值不等于其原来的值;RC_2 修改 x_k 的值,使得它满足约束;RC_3 删除值取作 $x_i(1\le i<k)$的子树;RC_4 插入子树并使其在约束节点上的取值满足约束.总的候选修复的个数≤$2k$;(4) 多维一元约束,假设冲突类的目标节点为 n,约束节点是 $n.U$,$n.A$,约束是 $n.U=N$,$n.A\theta w$,那么,候选修复为:RC_1 修改 $n.A$ 的值,使得 $n.A=x(x\theta w)$;RC_2 修改 $n.U$ 的值,使得 $n.U!=N$;RC_3 删除节点 n.假设约束是 $n.A\theta n.U$,那么 RC_1 修改 $N.U$ 的值,使得 $n.A\theta n.U$ 满足;RC_2 修改 $n.U$ 的值,使得 $n.A\theta n.U$ 满足;RC_3 删除节点 n 的这棵子树.设 $n.U$ 中有 k–1 个属性,那么无论哪一种,候选修复的个数≤k+1.

表 2 是图 1 中部分冲突类的候选修复.使用变量绑定后,冲突类的候选修复是有限的.

Table 2 Part of candidate repair and its cost for example 1

表 2 例 1 的部分冲突类候选修复及其代价

	Candidate repair	Cost
c_1	2. *projectName*=*x*(*x*!=*Dalianport*) && 19. *projectName*=*x*(*x*=*Dalianport*)	0.8
	19. *projectName*=*x*(*x*!=*Dalianport*) && 2. *projectName*=*x*(*x*=*Dalianport*)	0.5
	Delete (2)	220.8
	Delete (19)	111
c_2	5. color=green && 22. color=green	0.8
	22. color=red && 5. color=red	0.5
	5. *comName*=*x*(*x*!=SPPC) && 22. comName=*x*(*x*=SPPC)	0.8
	22. *comName*=*x*(*x*!=SPPC) && 5. *comName*=*x*(*x*=SPPC)	0.5
	Delete (5)	96
	Delete (22)	84

某个候选修复可能覆盖多个冲突类,如:假设在修复 C_1 时,将以 19 号子节点为根节点的子树删除,那么将能同时修复除 c_4 之外的所有冲突.所以,候选修复的实际修复代价应是 *RCcost*/*n*(其中,*RCcost* 是冲突类的修复代价,*n* 是其所能覆盖的冲突类数),也叫做覆盖代价.

4.2 XML文档修复的启发式计算

XML 文档的修复是所有冲突类的修复组合,最直观的修复方法是通过回溯,找出所有可能的冲突类候选修复组合.但这样做非常耗时,且只能处理小容量 XML 文档.而实际应用只需最优修复.根据前文:① 冲突类的修复代价越大,说明该候选修复越不应该被考虑;② 某冲突类的候选修复可能覆盖其他冲突类.最优修复应该和原 XML 文档距离最近、可信度最高、信息丢失最少.候选修复的代价之和可以成为衡量修复方案好坏的标准,因此,最佳修复就转变为代价最小的修复.针对这一问题,本文提出了启发式算法,见算法 3.

算法 3. *GetFRepair*() 模糊修复的启发式算法.

输入:冲突类 *CC*;

输出:模糊修复 *Frepair*.

1. //为冲突类集 *CC* 中的每个冲突类 *cc* 构造候选修复 *rc*,并计算 *rc* 的修复代价 *w*,将(*i*,*rc*,*cc*,*w*,*t*)压入 *RC* 堆栈

   for (*i*=0;*i*<|*CC*|;*i*++){*RC*=*ComputeRC*(*i*,*CC*[*i*])}

2. //寻找每个候选修复的覆盖冲突,*include* 函数可判断两个操作在语意上是否相互覆盖.

   初始化数组 *a* 的所有元素为 0

   for (*i*=0;*i*<|*RC*|;*i*++)

   { *k*=*RC*[*i*].*i*,*a*[*i*][*k*]=1

   for (*j*=0;*j*<|*RC*|,*j*++)

   if(*i*!=*j* && *include*(*RC*[*i*].*CC*,*RC*[*j*].*cc*)){*k*=*RC*[*j*].*i*,*a*[*i*][*k*]=1}

   RC[*i*].*t*=*RC*[*i*].*w*/Σ*a*[*i*][*j*](0≤*j*<|*CC*|)}

3. //选择代价最低的 *rc*,若存在多个代价相同的最小修复,则选择和模糊修复 *FRepair* 中已有 *rc* 不冲突的 *rc*,推入 *FRepair*,同时,其他所有覆盖 *CC* 的候选修复不再是 *CC* 的候选修复.置相应的 *a*[*i*][*j*]为 0,并从 *RC* 中删除其中只覆盖 *CC*′的候选修复,重新计算 *RC*[*i*].*t*.

   push (*select*(min*T*(*RC*)),*FRepair*)

   a[*m*][*n*]=0(*m* 任意,*n* 满足下列条件:设 *m*′为 *rc* 所在的行,有 *a*[*m*′][*n*]=1)

   delete those *rc*′ in *RC*, all *a*[*m*][*n*] (*m* 任意,*n* 为 *rc*′.*i*) is 0

   RC[*i*].t=*RC*[*i*].*w*/Σ*a*[*i*][*j*](0≤*j*<|*CC*|)//重新计算各候选修复的代价,已删除的 *RC*[*i*]不算

4. 重复 3,直到 *RC* 为空.

算法首先计算冲突类集 *CC* 中的每个冲突 *cc* 的候选修复 *rc* 及其修复代价 *w*(当节点取值不确定时,其修改代价取最大值 1),并将五元组(*i*,*rc*,*cc*,*w*,*t*)压入 *RC* 堆栈,其中,*i* 是该 *cc* 在 *CC* 中的序号,*t* 是 *rc* 能够覆盖的冲突类的数量(暂时设置为 0).这一步的时间代价是|*RC*|.

算法第 2 步寻找每个候选修复的覆盖冲突类,假设 *CC* 中共有 *M* 个冲突类,这 *M* 个冲突类共有 *N* 个候选修复,设数组 *a*[*N*][*M*]的每个元素 *a*[*i*][*j*]的值为 0 或 1,0 表示第 *i* 个候选修复不能覆盖第 *j* 个冲突类,1 表示能覆盖.给定候选修复 *RC*[*i*](0≤*i*<|*RC*|)和 *RC*[*j*](0≤*j*<|*RC*|,*i*!=*j*),如果 *RC*[*i*].*rc* 和 *RC*[*j*].*rc*:(1) 是插入节点,*RC*[*i*].*rc* 是 *insert*(*n*), *RC*[*j*].*rc* 是 *insert*(*m*),*m* 和 *n* 指的是同样的节点,或者 *m* 是 *n* 的子孙后代节点,那么 *RC*[*i*].*rc* 覆盖了第 *RC*[*j*].*i*+1 个冲突;(2) 删除节点,*RC*[*i*].*rc* 是 *delete*(*n*),*RC*[*j*].*rc* 是 *delete*(*m*),*m* 和 *n* 指的是同样的节点,或者 *m* 是 *n* 的子孙后代节点,那么 *RC*[*i*].*rc* 覆盖了第 *RC*[*j*].*i*+1 个冲突;(3) 是修改节点值,*RC*[*i*].*rc* 是 *A*=*x*(*x*θ*w*$_1$),*A* 是目标节点.约束节点,*RC*[*j*].*rc* 是 *B*=*x*(*x*θ*w*$_2$),*B* 也是目标节点.约束节点,如果 *A*=*B*,那么如果 *x*θ*w*$_1$ 和 *x*θ*w*$_2$ 相互包含,那么,*RC*[*i*].*rc* 覆盖了第 *RC*[*j*].*i*+1 个冲突.

两个条件之间是否包含的问题也有很多相关工作,数值型的条件,因为是连续空间,可以用区间包含的方法

来解决;而对于字符串类型的条件,尽管其取值为离散型的,但也能比较 w_1 和 w_2 的关系,也就是可以判断这两个条件是否能够相互包含.当然,对于形如 $A!=x$ 的修复,道理也是一样的.

一个候选修复不会同时覆盖某个冲突类的多个候选修复,因此不存在重复计算问题.所有覆盖都计算完毕后,候选修复的 t 值就能计算出来.这一步的时间代价为$|RC|^2$.

算法第 3 步、第 4 步是整个算法的核心,每次都寻找 w/t 值最小且不会和已选定候选修复冲突的候选修复 minRC,推入模糊修复堆栈,然后将所有也能覆盖 minRC 所覆盖的冲突类的其他 RC 标志为不能覆盖.然后检查修改以后的 RC 集合,将不能覆盖任何冲突类的 rc 从集合中删除,重新计算其他 rc 的 t 值.最后寻找下一个 w/t 值最小的候选修复,直到所有冲突类都被覆盖.如果当中某一步找不到一个不会和已选定候选修复冲突的 minRC,那么算法宣告无解.这一步的算法复杂度为$|CC|\times|RC|\times(|CC|+1)$.

4.3 修复精化

模糊修复中的节点取值不确定,根据节点绑定条件的区间运算可以确定其中某些节点的取值,或者缩小变量的取值范围,也可以根据操作的包含性减少冗余操作,这个过程叫做修复精化.

模糊修复的各候选方案之间可能相互包含,如两个删除子树操作 A 和 B,A 删除的子树包含了 B 删除的子树,那么 AB 可以合并为 A.同理,删除子树和修改节点间、子树插入间也能合并,但子树插入和节点修改间以及子树插入和子树删除间不能合并.节点修改之间的合并是这里的重点.任何两个候选修复 A 和 B,设它们都包含对节点 N 的修改,则可以使用区间合并的方法进一步确定 N 的取值范围,甚至 N 的值.如 A 和 B 中 N 的值限制条件分别是 $x\theta w_1$ 和 $x\theta w_2$,且 AB 中的 θ 均为“>”或“≥”且 $w_1>w_2$,则合并后 N 的限制条件是 $x\theta w_1$.如果区间合并中出现矛盾,则摒弃该模糊修复.如果所有模糊修复都被否定,那么整个文档不能修复.根据候选修复的产生方法,节点值修改的候选修复可能是:a) $N.A=x(x\theta w_1)$或 b) $N_1.A=x(x\theta w_1)$ && $N_2.A=x(x\theta w_1)\ldots N_n.A=x(x\theta w_1)$.两个 a)形式的候选修复合并,结果仍然是 a)形式的候选修复;a)形式的候选修复和 b)形式的候选修复合并,结果仍是 b)形式的,但相关节点约束已被合并后的节点约束所代替;两个 b)形式的候选修复 RC_1 和 RC_2 合并,结果仍是 b)形式的,是 RC_1 && RC_2,且 RC_1 和 RC_2 中相同节点的约束已合并.精确修复以后,某个节点的取值约束是确定的或具有一定的取值范围.

Table 3 First round of the greedy compute of Fig.1

表 3 图 1 的启发式计算初图

i	rc	w	w/t	C_1	C_2	C_3	C_4	C_5
1	rc_1	0.8	0.8	1	0	0	0	0
	rc_2	0.5	0.5	1	0	0	0	0
	rc_3	220.8	55.2	1	1	1	1	0
	rc_4	111	27.75	1	1	1	0	1
2	rc_5	0.8	0.8	0	1	0	0	0
	rc_6	0.5	0.5	0	1	0	0	0
	rc_7	0.8	0.4	0	1	1	0	0
	rc_8	0.5	0.25	0	1	1	0	0
	rc_9	96	32	0	1	1	1	0
	rc_{10}	84	28	0	1	1	0	1

4.4 算法的复杂度

根据前面的分析,设冲突类的个数为 m,约束平均涉及的约束节点个数为 k,则候选修复的个数最多为 $m\times(k^2+2k)$.一般情况下,k 的值不会太大,因此可以认为 k^2+2k 是一个整数常量 K.取得最坏值的情况是,当所有的约束都是置等类约束,且约束节点的个数为 N 时,根据启发式算法,计算所有候选修复的时间代价为 $m\times N$,为 m 初始化数组 a 并计算代价 w/t 的时间消耗最坏在$(m\times N)^2$.而后的循环过程中,因为共需覆盖 m 个冲突类,所以循环的趟数≤m,而候选修复可能覆盖多冲突类,故每次循环需比较的 RC 样本空间都比上次少.a 值修改也只针对部分样本,整个循环后需重新计算 $a(m\times N$ 行、m 列),代价是 $m^2\times N$.修改 a 的同时就能修改代价 w/t,也能决定是否删除该样本,但还需一遍遍历 RC 集,找出代价最小的元素,并拿它和 $FRepair$ 堆栈中的其他 rc 比较.假设 $FRepair$

和 *RC* 中的元素个数始终取最大值 *m* 和 *m*×*N*,则时间代价为 *m*×*m*×*N*+*m*×*m*.因此,整个算法的时间复杂度为 *m*×*N*+(*m*×*N*)^2+*m*^{2}×*N*+*m*^{2}×(*N*+1),也就是((*N*+1)^2*m*+*N*)*m*,在冲突类个数的平方级.但如果某节点只涉及一个冲突类,那么它的候选修复就无须与其他 *rc* 比较,且 *FRepair* 中的元素是不断增加的,而 *RC* 中的元素不断减少,所以实验的时间比这里分析的要少.

5 实 验

实验环境.实验使用的计算机的配置如下:Intel Core2 1.8GHZ,667MHZ 的 2G 内存;160G,8MB 缓存的硬盘.操作系统为 Windows XP+SP2,程序使用 C#编写.

我们在真实数据和人造数据上都进行了实验.

噪声.实验数据中人为加入的错误数据称为噪声.噪声比例是错误节点占总节点数的比重,实验对人造数据加入比例不同的噪声以测试性能.而对真实数据,则没有人为添加噪声数据,而是构建若干约束,以致原文档中一定包含违反它们的不一致数据,并以此来测试性能.

实验数据.真实数据来自 http://dblp.uni-trier.de/xml/的dblp_bht.xml,文档大小为 73.4MB,实验自定义了 4 条约束(参见下文),对该文档进行修复,修复的过程中没有考虑 DTD.人造数据则以例 1 的数据模式为模式,使用 XML Generator 分别生成大小不一的 XML 文档,且在数据中加入不同比例的噪声数据,然后根据例 1 的 5 条约束进行修复.实验首先生成了噪声比例在 3%的 5 类规模不同的数据,其大小分别为 125KB,495KB,1.168MB,4.604MB 和 11.987MB;然后又生成噪声比例不一、大小相似的数据,见表 4.

Table 4 One group of the experimental data

表 4 实验数据配置之一

Size of XML (KB)	495	495	494	493	491
Ratio of noise data (%)	1	3	6	10	20

冲突类规约算法性能.我们使用两组实验来衡量冲突类规约算法的时间性能:第 1 组实验测试了噪声比例在 3%的大小规模不同的 5 个 XML 文档上,使用例 1 的 6 个约束进行冲突类规约的时间消耗;第 2 组实验则测试了 500K 左右的 XML 文档,5 个噪声比例上,使用例 1 的 6 个约束进行冲突类规约的时间消耗.结果如图 2、图 3 所示.

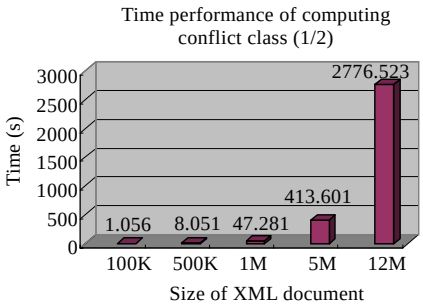


Fig.2 Time for conflict class when noise is 3%

图 2 噪声比例为 3%时的冲突类规约时间

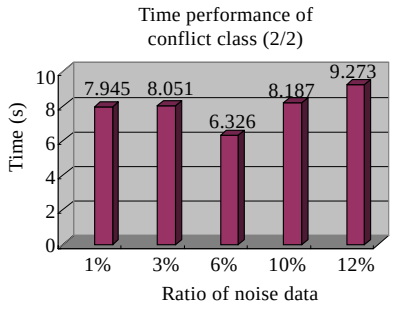


Fig.3 Time for conflict class for 500KB XML

图 3 500KB 左右的原文档的冲突类规约时间

实验结果表明,冲突类规约所需时间受原文档大小影响更大,而受噪声比例的影响更小.这是因为冲突类规约耗时在于一遍扫描原 XML 文档、后来的冲突类索引表排序和一遍扫描,其中,扫描原文档是最耗时的.同样的噪声比例下,XML 文档越大,扫描它的时间越长,生成的冲突类索引表的表项越多,对冲突类索引表的排序和扫描耗时越大,整个耗时也就越大.当然,同样的 XML 文档下,噪声比例越大,冲突类会越多,对排序后的冲突类索引表处理的时间越长,但原文档的扫描时间不变,冲突类索引表的表项也不变.因为冲突类索引表的表项取决于与约束路径匹配的路径数量,所以正如图 3 所示,同样大小的 XML 文档,不同噪声比例下,其冲突类的规约时间差

别也不大.

启发式算法的性能.为了验证 XML 修复的时间性能,我们也在上述两组数据的基础上分别进行了实验,结果如图 4、图 5 所示.实验结果表明,启发式算法的时间消耗与原 XML 文档及噪声比例都有很大的关系.事实上,启发式算法的时间主要与候选修复的多少有关,而候选修复的多少则取决于冲突类的数量和约束的类别.同样噪声比例下,原文档和不一致节点数量基本成正比.从图 4 可以看出,时间消耗确实在不一致节点的立方以内,如 500K 的 XML 文档耗时是 100K 的 6.2 倍,在平方以内;而 5M 的耗时是 1M 的 10.8 倍,也在平方以内.只有 1M 的耗时在 500K 的耗时的立方以外,这是因为添加噪声数据时,1M 的噪声节点都是主键约束和函数依赖约束节点,其候选修复比其他类别的多很多.因此,虽然噪声节点是 500K 的 2 倍,但候选修复是其 10 倍,耗时在立方以外.而同样的 XML 文档,噪声比例增加 1 倍,冲突类可能增加 2 倍甚至更多.因此,图 5 中除了 3%的时间消耗在 1%的 27 倍以内,其他耗时都在立方以外.但噪声比例和冲突类之间的增长关系受噪声分布和约束类别的影响,难以给出统一的公式.启发式算法的时间消耗基本上在候选修复的平方级别.

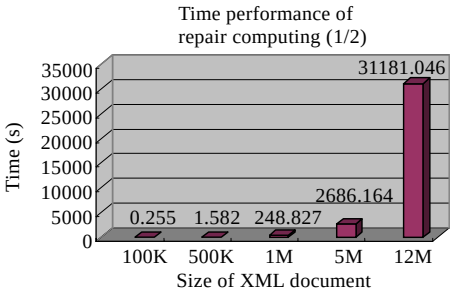


Fig.4 Time for repair compute when noise is 3%

图 4 噪声比例为 3%时的修复计算时间

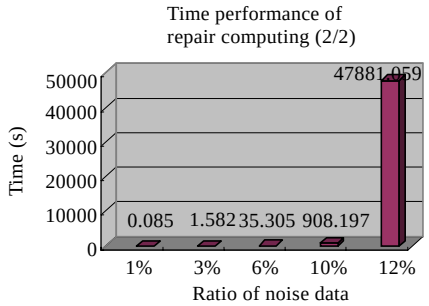


Fig.5 Time for repair compute when XML about 500KB

图 5 500KB 左右的原文档的修复计算时间

真实数据上的结果.我们在真实数据上也做了上述 3 方面实验,实验中的 4 条约束为:

(1) 一个 *dblp*/*bht* 内,*cite* 的 *key* 能够唯一决定该 *cite*,即(*dblp*/*bht*,*cite*,(@*key*))([x_1][x_2]⇒x_1!=x_2);

(2) 整个文档范围内 *footer* 的 *name* 不能为空,即(*ε*,*dblp*/*bht*/*footer*,(@*name*))([*x*]⇒*x*!=*NULL*);

(3) 整个文档范围内,*key* 值相同的 *cite* 具有相同的 *style*,即(*ε*,*dblp*/*bht*/*cite*,(@*key*,@*style*))([*x*,y_1][*x*,y_2]⇒y_1=y_2);

(4) 整个文档范围内,*bht* 的 *key* 能够唯一决定该 *bht*,即(*ε*,*dblp*/*bht*,(@*key*))([x_1][x_2]⇒x_1!=x_2).

在这 4 条约束下,不一致节点有近 5 万个,而实验结果(见表 5)表明,其时间消耗仍能接受.

Table 5 Repair computing performance of *dblp_bht.xml*

表 5 *dblp_bht.xml* 上的修复性能

Time of conflict class computing (s)	Time of heuristic algorithm (s)
5 475.862	27 293.328

6 总结和未来工作展望

求解不一致 XML 文档修复的复杂度大、耗时高,但在实际应用中又非常必要.本文提出一种启发式算法,可在多项式时间内找到 XML 文档的一个修复.具体来说,本文的贡献在于:

- 提出了适合 XML 文档修复的基于代价模型的启发式算法;
- 提出了适合 XML 的代价模型,给出了节点修改、节点删除、节点插入、子树删除和子树插入的代价公式;
- 能够同时解决置等和置不等约束;
- 扩展了冲突类的概念和冲突类索引表,并给出了使用冲突类表检索 XML 文档中所有冲突类的算法.

实验结果表明本文的启发式计算方法确实有效.

如何在 XML 的完整性约束和结构性约束(比如 DTD)下展开修复求解,是我们下一步工作之一.

References:

[1] Arenas M, Bertossi LE, Chomicki J. Consistent query answers in inconsistent databases. In: Proc. of the 18th ACM Symp. on PODS. Philadelphia: ACM, 1999. 68−79.

[2] Bertossi L, Bravo L, Franconi E, Lopatenko A. Complexity and approximation of fixing numerical attributes in databases under integrity constraints. In: Proc. of the 10th Int'l Symp. on Database Programming Languages. LNCS 3774, Springer-Verlag, 2005. 262−278.

[3] Andritsos P, Fuxman A, Miller RJ. Clean answers over dirty databases: A probabilistic approach. In: Proc. of the 22nd Int'l Conf. on Data Engineering (ICDE 2006). Atlanta: IEEE Computer Society, 2006. 30.

[4] Bohannon P, Fan WF, Flaster M. A cost-based model and effective heuristic for repairing constraints by value modification. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Baltimore: ACM, 2005. 143−154.

[5] Lopatenko A, Bertossi L. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In: Proc. of the 11th Int'l Conf. of Database Theory. LNCS 4353, Barcelona: Springer-Verlag, 2007. 179−193.

[6] Lopatenko A, Bravo L. Efficient approximation algorithms for repairing inconsistent databases. In: Proc. of the 23rd Int'l Conf. on Data Engineering (ICDE 2007). Istanbul: IEEE Computer Society, 2007. 216−225.

[7] Chen Y, Davidson SB, Zheng YF. XKvalidator: A constraint validator for XML. In: Proc. of the 2002 ACM CIKM Int'l Conf. on Information and Knowledge Management. McLean: ACM, 2002. 446−452.

[8] Tan ZJ, Zhang ZJ, Wang W, Shi BL. Consistent data for inconsistent XML document. Information & Software Technology, 2007, 49(9-10):947−959.

[9] Arenas M, Libkin L. XML data exchange: Consistency and query answering. In: Proc. of the ACM Symp. on Principles of Database Systems. Baltimore: ACM, 2005. 13−24.

[10] Wilfred NG. Repairing inconsistent merged XML data. In: Proc. of the 14th Database and Expert Systems Applications (DEXA 2003). LNCS 2736, Prague: Springer-Verlag, 2003,

[11] Flesca S, Furfaro F, Greco S, Zumpano E. Repairs and consistent answers for XML data with functional dependencies. In: Proc. of the Database and XML Technologies, the 1st Int'l XML Database Symp. LNCS 2824, Heidelberg: Springer-Verlag, 2003. 238−253.

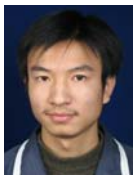
[12] Staworko S, Chomicki J. Validity sensitive querying of XML databases. In: Proc. of the 10th Int'l Conf. on Extending Database Technology. LNCS 4254, Munich: Springer-Verlag, 2006. 164−177.

[13] Buneman P, Davidson SB, Fan WF, Hara C, Tan WC. Keys for XML. Computer Networks, 2002,39(5):473−487.

[14] Lund C, Yannakakis M. On the hardness of approximating minimization problems. Journal of the Association for Computing Machinery, 1994,45(5):960−981.

吴爱华(1976－),女,江西鹰潭人,博士生,讲师,主要研究领域为 XML 存储和查询处理,数据清洗,知识发现,数据库系统.

谈子敬(1975－),男,博士,副教授,主要研究领域为 XML 数据库,数据清洗,知识发现,数据库系统.



王先胜(1985－),男,硕士,主要研究领域为 XML 数据库.

汪 卫(1970－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为 XML 数据库,生物信息,空间数据库,数据挖掘,数据库系统.