基于二元关系消减的概念格维护算法

王春月 王黎明 张 卓
(郑州大学信息工程学院 郑州 450001)

摘 要 针对有限空间下如何快速维护概念格的问题,提出一种消减形式背景中冗余二元关系的概念格维护算法。传统的算法删除冗余关系后需要重新构造概念格,这种方式较为费时。而所提算法能够在原始概念格的基础上直接调整得到新概念格的方法,可以处理任意位置的二元关系消减的情况。它采用自底向上广度优先方式遍历格节点,首先根据当前节点是否同时包含冗余关系对象和冗余关系属性,将当前节点分为受影响的节点和不变节点;然后根据当前节点与父子节点的外延和内涵的关系,再将受影响的节点细分为 4 类,即减对象节点、减属性节点、分割节点、删除节点;最后根据父子节点的类型更新边。实验结果表明,在一定程度上与传统算法相比,所提算法能够获得更好的时间性能。

关键词 形式概念分析,概念格,二元关系消减,算法
中图法分类号 TP311 **文献标识码** A

Algorithm of Maintaining Concept Lattice Based on Binary Relation Decrement

WANG Chun-yue WANG Li-ming ZHANG Zhuo
(School of Information Engineering, Zhengzhou University, Zhengzhou 450001, China)

Abstract In view of the problem of how to maintain the concept lattice in limited space, a concept lattice maintenance algorithm was proposed to decrease redundant binary relation in formal context. Traditional algorithms need to re-construct the concept lattice after deleting redundant relations, this approach is more time-consuming. For the above, we proposed a concept lattice based on the original direct adjustment to obtain new concept lattice method, which can handle cases anywhere in the binary relations of decrement. It uses the bottom-up breadth first traversal grid nodes. Firstly, according to whether the current node contains redundant relational object or redundant relational attribute, the current node is divided into the affected nodes and the constant node. Then based on the relationship between the parent-child node of the current node and extent and intent, then the affected nodes subdivided into four categories, namely reducing object node, reducing attribute node, splitting nodes, deleting nodes. At last, according to the type of parent-child node update edges. Experimental results show that, to some extent, compared with the traditional algorithm, it can get better time performance.

Keywords Formal concept analysis, Concept lattice, Binary relation decrement, Algorithm

1 引言

概念格[1]作为数据分析和知识发现的有效工具,已经广泛地应用于 web 服务、网络安全管理、数据挖掘等众多领域中[2-4]。随着时间的推移,数据库中会产生大量的冗余信息,而构造概念格的时空复杂度会随着形式背景的增加呈指数增长。如何提高概念格的构造效率,降低概念格的规模是概念格应用的前提。国内外的研究人员对此进行了广泛的研究。现有的方法主要分为形式背景的简化、概念格构造过程中的改进以及应用过程中根据用户的需要对格结构进行整体简化。

形式背景简化主要按照一定的规则对背景进行约简,使得约简后的形式背景规模减少,从而降低构造格的时空复杂度。文献[5]提出形式背景属性约简的一种新的概念和计算方法,该方法通过引入交式可约元概念,在保持对象集不变的前提下得到属性约简个数计算的精确公式,改进了原有约简个数估计。文献[6]通过减少行与列的角度来简化形式背景,该方法定义包含约简概念,选择性地去掉了拥有属性较少的对象和拥有对象较少的属性,大大简化了形式背景。概念格构造过程中可以采用渐进式算法或是并行算法提高建造概念格的效率。文献[7]是基于对象渐增的概念格渐进式构造算法。该算法在原始概念格的基础上通过节点的父子关系快速找到标准产生子节点与更新节点进行修改,从而得到新形式背景下的概念格,避免了重新构格且节省了大量的时间。文献[8,9]提出消减单个冗余对象和属性的渐减式构造算法。当对象或属性减少时,能比传统算法节省大量的运行时间。

本文受国家青年科学基金项目(61303044)资助。
王春月(1991-),女,硕士生,主要方向为形式概念分析及应用、数据挖掘等,E-mail:iecywang@gs.zzu.edu.cn;王黎明(1963-),男,博士,教授,主要研究方向为现代软件工程技术、分布式人工智能和数据挖掘等;张 卓(1978-),男,博士,讲师,主要研究方向为形式概念分析及应用等,E-mail:iezhangzhuo@zzu.edu.cn。

目前也有多属性同步消减的概念格构造算法[10,11]，当需要减去多个属性时比文献[8]表现出更好的时间性能。文献[12]提出消减冗余二元关系的概念格维护算法，提出了父子概念对的概念，来确定概念格维护位置以及概念之间关系的调整，而二元关系消减算法不能处理当被减二元关系是某列属性中仅存的一个二元关系的情况，为此，提出了另一种属性消减的概念格维护算法进行处理。文献[13]提出一种基于负载均衡的并行构造模糊概念算法，该算法能够有效地提高模糊概念的构造效率。在概念格的实际应用中，对于已经构造好的概念格进行压缩，减少概念数目。文献[14]提出一种利用 K-means 聚类算法压缩概念格的方法，它通过定义对象和属性的重要度计算概念间的相似性，该方法不仅得到结构清晰且规模减少的概念格，并且压缩后的概念格也是完备格。

针对以上现有的方法，本文提出了一种新的简化概念格的方法，从而实现提高概念格的构造效率、降低概念格的规模的目的。该方法通过自底向上广度优先方式遍历格节点，并根据当前节点与冗余二元关系以及父子节点的内涵与外延之间的关系，识别出当前格节点的类型，最后根据父子节点的类型来更新边。本文算法能够处理形式背景中任意位置的二元关系消减的情况。实验结果表明，在删除冗余二元关系的情况下，本文算法要快于传统构造算法，具有较优越的时间性能。

2 相关定义

本节对于二元关系消减理论所涉及到的定义进行介绍，对于概念格的基本定义延用文献[1]中的表达方法，这里不再赘述。本文假设读者对于概念格的基本知识有一定的了解。

二元关系消减算法就是在给定的原形式背景 $K=(G,M,I)$所对应的初始概念格 $L(K)$以及消减 gIm 冗余二元关系的情况下，求解新形式背景 $K^*=(G,M,I^*)$所对应概念格 $L(K^*)$的过程，其中 $g\in G,m\in M,g$ 被称为冗余关系对象，m 被称为冗余关系属性，I^* 表示 I 集合减去 gIm 冗余二元关系后剩余关系的集合。对于 $L(K)$中每个概念节点，根据它和被减冗余二元关系(gIm)之间的关系，可以定义不同的类型，根据节点的不同类型调整格结构进而得到新形式背景下的新概念格。对于概念节点 C 的外延与内涵分别用 $Extent(C)$和 $Intent(C)$表示，父子节点分别用 $Parent(C)$和 $Child(C)$表示，父子概念关系用$\prec$表示。由形式背景 K 生成的所有概念节点的集合记作$CS(K)$。

定义 1(受影响节点) 如果 $L(K)$中某个格节点 C 满足 $g\in Extent(C)$且 $m\in Intent(C)$，则称 C 为受影响节点。

概念格 $L(K)$中除受影响节点外的其余节点统称为不变节点，根据受影响的节点和父子节点的外延与内涵的关系，可将受影响的节点类型进一步细分为以下 4 种情况。

定义 2(减对象节点) 如果 $L(K)$中某个格节点 C 满足：1)$g\in Extent(C)$且 $m\in Intent(C)$；2)若 $\forall C_1,C_2\in CS(K)$，其中 $C_1\prec C\prec C_2$，都有 $Extent(C_1)\neq Extent(C)-\{g\}$且 $Intent(C_2)=Intent(C)-\{m\}$。此时称 C 为减对象节点。

如果 C 是减对象的节点，则在 $L(K^*)$中 C 被更新为 $(Extent(C)-\{g\},Intent(C))$。

定义 3(减属性节点) 如果 $L(K)$中某个格节点 C 满足：1)$g\in Extent(C)$且 $m\in Intent(C)$；2)若 $\forall C_1,C_2\in CS(K)$，其中 $C_1\prec C\prec C_2$，都有 $Extent(C_1)=Extent(C)-\{g\}$且 $Intent(C_2)\neq Intent(C)-\{m\}$。此时称 C 为减属性节点。

如果 C 是减属性的节点，则在 $L(K^*)$中 C 被更新为 $(Extent(C),Intent(C)-\{m\})$。

定义 4(分割节点) 如果 $L(K)$中某个格节点 C 满足：1)$g\in Extent(C)$且 $m\in Intent(C)$；2)若 $\forall C_1,C_2\in CS(K)$，其中 $C_1\prec C\prec C_2$，都有 $Extent(C_1)\neq Extent(C)-\{g\}$且 $Intent(C_2)\neq Intent(C)-\{m\}$。此时称 C 为分割节点。

如果概念节点 $C=(A,B)$是分割节点，则产生新节点 $C''=(A,B-\{m\})$和 $C'=(A-\{g\},B)$。显然，$C'\prec C''$。对于分割节点的父子节点的边先做如下处理，如图 1 所示。

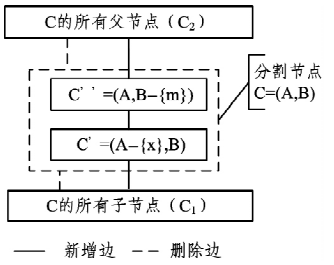


图 1 分割节点父子边变化示意图

为了方便以后对边的统一处理，在 $CS(K^*)$中，C'节点标记为减对象节点，C''节点被标记为减属性节点。

定义 5(删除节点) 如果 $L(K)$中某个格节点 C 满足$g\in Extent(C),m\in Intent(C)$且 $\exists C_1,C_2\in CS(K)$，其中 $C_1\prec C\prec C_2$。使得 $Extent(C_1)=Extent(C)-\{g\}$且 $Intent(C_2)=Intent(C)-\{m\}$，此时称 C 为删除节点。C_1 被称为 C 的删除父格节点，C_2 被称为删除子格节点，显然 C 与 C_1 以及 C 与 C_2 是一一对应的，且 C_1 和 C_2 均为不变节点。

定义 6(最小概念) 如果一个格节点 C 满足 $g\in Extent(C),m\in Intent(C)$且对于 C 的任意子节点都是不变节点，则 C 被称为最小概念。

3 二元关系消减算法的基本思想

本节提出一种在原始概念格的基础上得到新概念格的方法，新概念格中的节点可以通过原始概念格中的不变节点与除删除节点外的其余受影响节点更新得到。显然，求解 $L(K^*)$的关键就是找出 $L(K)$中的所有受影响的节点。由于受影响的节点均是最小概念的祖先节点，除最小概念的祖先节点外均不考虑，从而缩小节点的搜索范围。

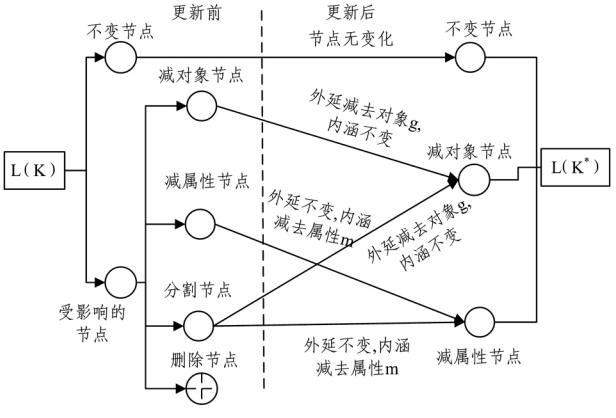


图 2 原始概念格 $L(K)$与新概念格 $L(K^*)$节点之间的映射关系

本文的冗余二元关系消减理论是对基于属性消减的概念格生成算法[8]理论的扩展。通过研究发现概念格消减冗余二元关系后的节点的变化规律如图 2 所示。显然，依据上节定义可知，$L(K^*)$中任意概念节点在 $L(K)$中除删除节点外都存在唯一的概念节点与之对应。

当渐减式删除一个冗余二元关系后，格节点之间的边是否发生变化依赖于父节点的外延变化时子节点的内涵是否也同时发生变化，同时需要考虑删除节点的父子节点之间是否需要新增边。

命题 1 设在 $L(K)$中，C_1 为减对象节点，C_2 为减属性节点，在 $L(K^*)$中两个节点相应的更新为 C_1'和 C_2'。如果在 $L(K)$中，$C_2 \prec C_1$，则在 $L(K^*)$中 C_2'和 C_1'中不存在偏序关系。

证明：设 $C_1=(A_1,B_1)$，$C_2=(A_2,B_2)$，在 $L(K)$中，因为 $C_2 \prec C_1$，且 $g \in A_2$ 以及 $g \in A_1$，所以 $A_2 \subseteq A_1$，$B_2 \supseteq B_1$。在 $L(K^*)$中父子节点更新为 $C_1'=(A_1-\{g\},B_1)$，$C_2'=(A_2,B_2-\{m\})$，显然更新后的两个节点 $Extent(C_2')$不包含于 $Extent(C_1')$。因此，在 $L(K^*)$中 C_2'和 C_1'中不存在偏序关系。

命题 1 说明，当形式背景消减关系 gIm 后，原始概念格中 C_1 父节点为减对象节点，C_2 子节点为减属性节点的边需要删除。此时需要考虑父节点 C_1 与 $Child(C_2)$以及子节点 C_2 与 $Parent(C_1)$是否新增边，若 C_1 与 $Parent(Child(C_2))$都不存在偏序关系，则 C_1 与 $Child(C_2)$需要新增边；若 C_2 与 $Child(Parent(C_1)))$都不存在偏序关系，则 C_2 与 $Parent(C_1)$需要新增边，如图 3 所示。

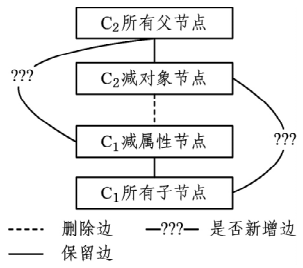


图 3 父节点为减对象及子节点为减属性边变化示意图

命题 2 设 $C_1=(A_1,B_1)$为删除节点，如果 $C_c=(A_c,B_c)$，使得 $C_c \prec C_1$ 成立，则 C_c 必然为下列 3 种情形之一：

1）C_c 为 C_1 的删除父格节点；

2）C_c 为删除节点；

3）C_c 为减属性节点。

证明：设 $C_2=(A_2,B_2)$为 C_1 的删除父格节点，则 $C_2 \prec C_1$，$A_2=A_1-\{g\}$。

1）由定义 5 可知，C_2 为不变节点，$g \notin A_2$，$m \in B_2$。需要证明当 C_c 为不变节点时，若 $C_2 \neq C_c$，则 $C_c \prec C_1$ 不成立即可。由于 $C_c \prec C_1$ 且 C_c 为不变节点，则有 $A_c \subseteq A_1 \Rightarrow A_c \subseteq A_1-\{g\}=A_2 \subseteq A_1$。若 $C_2 \neq C_c$，有 $C_c \prec C_2 \leqslant C_1$，因此 $C_c \prec C_1$ 不成立。

2）反证法，假设 C_c 为减对象节点，由于 $C_c \prec C_1$，$C_2 \prec C_1$，因此 C_2 和 C_c 不存在偏序关系，再由 $C_c \prec C_1$，有 $A_c \subseteq A_1 \Rightarrow A_c-\{g\} \subseteq A_1-\{g\}=A_2$，因此节点更新后，在 $L(K^*)$中，有 $C_c \prec C_2 \Leftrightarrow A_2 \subseteq A_p$，由于在 $L(K^*)$中，C_c 概念的内涵并不改变，矛盾，假设不成立。

3）反证法，假设 C_c 为分割节点，由于 $C_2 \prec C_1$，因此 $A_2=A_1-\{g\}$，又由于 $C_c \prec C_1$，因此 $A_c \subseteq A_1$，故 A_c 与 A_2 的交集存在两种结果等于空集或不为空，若结果为空集，说明 $Extent(C_c)=\{g\}$，由于 $g \notin A_2$，因此 C_c 不是最大下界，那么 C_c 的子节点中存在外延为空集的最大下界节点，由定义 6 可知，与分割节点的定义矛盾，所以假设不成立；若结果不为空，那么 C_c 与 C_2 有一个共同子节点且外延为 $B_c-\{m\}$，由定义 6 可知，与分割节点的定义矛盾，所以假设不成立。证毕。

由命题 2 可知，删除节点的子节点有 3 种情况：唯一的不变节点即删除父格节点、减属性节点和删除节点。

命题 3 设 $C_1=(A_1,B_1)$为删除节点，如果 $C_p=(A_p,B_p)$，使得 $C_1 \prec C_p$ 成立，则 C_p 必然为下面 3 种情形之一：

1）C_p 为 C_1 的删除子格节点；

2）C_p 为删除节点；

3）C_p 为减对象节点。

证明：由于概念格内涵与外延的对偶性，由命题 2，同理可证该命题的正确性。

由命题 3 可知，删除节点的父节点有 3 种情况：唯一的不变节点即删除子格节点、减对象节点和删除节点。

当删除节点被删除时，与父子节点的边也应该被删除。同时，需要考虑父子节点之间是否需要新增边，对于删除节点的某个父节点来说，若删除节点子节点的所有父节点都与其不存在偏序关系时，此时需要新增边。删除节点的父节点有以下两种情况：不变节点时，需考虑子节点类型为减属性节点与不变节点是否新增边；减属性节点时，只需考虑子节点为不变节点时是否新增边，如图 4 所示。

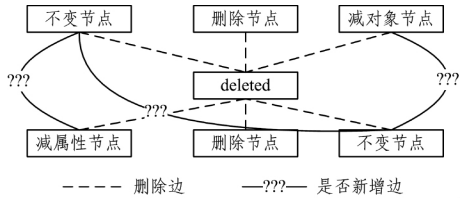


图 4 删除节点父子边关系变化示意图

综上，边的更新只考虑以上两种情况，其余的边则保持不变。即维护删除节点的父子节点之间边的关系以及维护父节点为减对象节点，而子节点为减属性节点的边的关系。显然依据上述理论更新概念节点和边关系，能够保证修改后的格仍然是一个完备格。

3.1 算法描述

依据上述理论，算法 DelRelation 的主要思想如下：首先从最大下界出发自底向上找到最小概念，若最小概念不是最大下界节点，那么对 $Inf(L(K))$节点单独进行处理。然后从最小概念出发自底向上以广度优先方式遍历最小概念的祖先节点并判断其类型进而做相应处理，最后更新父子节点之间边的关系。DelRelation 算法的伪代码如下。

算法 1 DelRelation

```
输入：形式背景 K 以及对应的概念格 L(K)，被减二元关系 gIm
输出：新形式背景下的概念格 L(K*)
1. Begin
2.   FindMinConcept(g,m,inf(L(K)),L(K));
3.   if minConcept≠inf(L(K)) then
4.     DealWithInf(g,m,L(K));
5.   end if
```

```
6.  UpdateNodes(g,m,L(K),affected-Queue);
7.  UpdateEdges(L(K),edgeChanged,ReSet);
8. End
```

FindMinConcept 过程从最大下界 $inf(L(K))$ 出发自底向上寻找格的最小概念，找到后放入 $affectedQueue$ 队列尾部，过程结束。$affectedQueue$ 队列保存受影响的节点。该过程伪代码描述如下。

过程 1　FindMinConcept

```
输入：原始概念格 L(K)，inf(L(K))，冗余二元关系 gIm
输出：affectedQueue 队列
1. Begin
2.  将 inf 添加至 tempQueue 的队列尾部；
3.  While tempQueue 不为空 do
4.    C:=tempQueue 队首元素出队列；
5.    for each Cp∈Parent(C) do
6.      if g∈Intent(Cp) and m∈Intent(Cp)
7.        then flag:=true;
8.        将 Cp 添加至 affectedQueue 队尾；
9.        break;
10.     else
11.       将 Cp 添加至 tempQueue 的队尾；
12.   if flag=true then break;
13. End
```

若找到的最小概念不是最大下界节点，需要对最大下界节点 $inf(L(K))$ 进行处理，即 DealWithInf 过程。当被减二元关系是某列属性中的最后一个二元关系时，此时最大下界节点的内涵将会更新为 $Intent(inf(L(K)))-\{m\}$，若 inf 节点的父节点中存在相同的内涵，则 inf 即为删除节点，将其放入 $edgeChanged$ 的队尾处等待边更新操作。DealWithInf 的算法伪代码如下。

过程 2　DealWithInf

```
输入：形式背景 K 以及对应的概念格 L(K)，被减二元关系 gIm
输出：节点边需要更新的队列 edgeChanged
1. Begin
2.  objSet:=具有属性 m 的所有对象的集合；
3. if objSet 集合有且仅有一个对象 g then
4.  Intent(inf(L(K))):=Intent(inf(L(K)))-{m};
5.  for each C∈Parent(inf) do
6.    if Intent(inf)=Intent(C) then
7.      inf 的 isDeleted 标记域置为 true;
8.      添加 inf 到 edgeChanged 队列尾部；
9. End
```

UpdateNodes 主要负责处理 $affectedQueue$ 队列中受影响的节点，标记每个节点所属类型并对节点的外延与内涵做相应处理，UpdateNodes 过程伪代码如下。

过程 3　UpdateNodes

```
输入：L(K)，affectedQueue 队列，被减二元关系 gIm
输出：edgeChanged 队列，ReSet 集合
1. Begin
2.  While affectedQueue 队列不为空 do
3.    C=affectedQueue 队首元素出队列；
4.    Extent(C):=Extent(C)-{g};
5.    eqlExt:=false;
6.    for each Cc∈Child(C) do
7.      if Extent(Cc)=Extent(C) then
8.        eqlExt:=true;break;
9.      end if
10.   end for
11.   Intent(C)=Intent(C)-{m};
12.   eqlInt=false;
13.   for each Cp∈Parent(C) do
14.    if Intent(Cp)=Intent(C) then
15.      eqlInt=true;
16.    end if
17.    if Cp.vs=false and m∈Intent(Cp) then
18.      将 Cp 添加至 affectedQueue 队尾；
19.      Cp.vs=true;
20.    end if
21.   end for
22.   if eqlExt=true and eqlInt=true then
23.      C.isDeleted=true;
24.      将 C 添加至 edgeChanged 队尾；
25.      continue;
26.    end if
27.    if eqlExt=true and eqlInt=false then
28.      C.isDelAtrr= true;
29.      Extent(C)=Extent(C)∪{g};
30.      将 C 放入 ReSet 集合中；
31.      continue;
32.    end if
33.    if eqlExt=false and eqlInt=true then
34.      C.isDelObj=true;
35.      Intent(C)=Intent(C)∪{m};
36.      将 C 添加至 edgeChanged 队尾；
37.      将 C 放入 ReSet 集合中；
38.      continue;
39.    end if
40.    if eqlExt=false and eqlInt=false then
41.      添加新概念 Cnew=(Extent(C)∪{g},Intent(C));
42.      将 Cnew 与 C 所有父节点均添加边；
43.      将 C 与其父节点的边均删除；
44.      C 与 Cnew 节点之间添加边；
45.      Intent(C)=Intent(C)∪{m};
46.      C.isDelObj=true;
47.      Cnew.isDelAtrr=true;
48.    将 C 与 Cnew 放入 ReSet 集合中；
49.    end if
50.  end while
51. End
```

该过程中，$eqlExt$ 用于标志当前节点外延减去对象 g 后与子节点外延是否相等；$eqlInt$ 用于标记当前节点减去属性 m 后与父节点内涵是否相等。为了加快节点类型的判断速度，每个节点有 4 个标记位：vs 用于标记节点是否被处理过，$isDeleted$ 用于标记节点是否为删除节点，$isDelAtrr$ 用于标记节点是否为减属性节点，$isDelObj$ 用于标记节点是否为减对象节点。$ReSet$ 集合用于保存删除节点和更新节点，方便最后对这些节点进行标志复位，从而可以进行其他二元关系的删除。

算法第 4－10 行判断当前节点减去对象 g 的外延与其子节点外延是否相等；第 11－21 行判断当前节点内涵减去属性 m 与其父节点内涵是否相等，若父节点未被访问过且内涵包含属性 m，则将该待处理节点放置 *affectedQueue* 队尾等待处理；第 22－26 行表示当前节点表示删除节点，将该节点放置 *edgeChanged* 队尾，等待修改父子节点边；第 27－32 行判断当前节点是减属性节点；第 33－39 行表示该节点为减对象节点，并放置 *edgeChanged* 队尾；第 40－49 行处理当前节点 C 为分割节点的情况，添加新概念 $C_{new}=(Extent(C), Intent(C)-\{m\})$，且 C 节点更新为 $Extent(C)-\{g\}$，内涵不变。此时，分割节点可以理解为被分成减对象节点与减属性节点两种类型的节点，方便以后节点的统一处理。即 C_{new} 的 *isDelAttr* 标记域被置为 true，C 的 *isDelObj* 标记域被置为 true。整个 while 循环结束后，将得到新概念格中的所有概念集合 $CS(K^*)$。

UpdateEdges 过程主要负责两类边的更新操作，第一类是父节点为减对象节点、子节点为减属性节点时边更新的情况；第二类是删除节点的父子节点的边更新情况。UpdateEdges 过程的伪代码如下。

过程 4　UpdateEdges

输入：L(K)，edgeChanged 队列，ReSet 集合

输出：新形式背景 K* 下的概念格 L(K*)

```
1. Begin
2.   while edgeChanged 队列不为空 do
3.     C=affectedQueue 队首元素出队列;
4.     if C 为减对象节点 then
5.       for each Cc∈Child(C) do
6.         if Cc 是减属性节点 then
7.         将 Cc 与 C 节点间的边删除;
8.         for each Cp∈Parent(C) then
9.           if Cp 不是减对象节点 then
10.              addNewEdges=false;
11.              for each Cpc∈Child(Cp) then
12.                if Cpc≠C and
13.                  Extent(Cc)⊆Extent(Cpc) then
14.                  addNewEdges=true;
15.                  break;
16.              if addNewEdges=false then
17.                Cp 与 Cc 节点之间新增边;
18.            for each Ccc∈Child(Cc) do
19.              if Ccc. isDelAttr=false then
20.                addNewEdges=false;
21.                  for each Cccp∈Parent(Ccc) do
22.                    if Cccp≠Cc and
23.                      Extent(Cccp)⊆Extent(C) then
24.                      addNewEdges=true;
25.                      break;
26.                  if addNewEdges=false then
27.                    节点 C 与 Cccp 之间新增边;
28.   else if C 为删除节点 then
29.     for each Cp∈Parent(C) do
30.       if Cp. isDelObj=true then
31.         for Cc∈Child(C) do
32.           addNewEdges=false;
33.           if Cc. vs=false then
34.             for each Ccp∈Parent(Cc) do
35.               if Ccp≠C and
36.                 Extent(Ccp)⊆Extent(Cp) then
37.                 addNewEdges=true;
38.                 break;
39.             If addNewEdges=false then
40.               节点 Cp 与 Cc 之间新增边;
41.       else if Cp. vs=false then
42.         for Cc∈Child(C) do
43.           if Cc. isDeleted=false then
44.             addNewEdges=false;
45.               for each Ccp∈Parent(Cc) do
46.                 if Ccp≠C and
47.                   Extent(Ccp)⊆Extent(Cp) then
48.                     addNewEdges=true;break;
49.             if addNewEdges=false then
50.                 Cp 与 Cc 节点之间新增边;
51.             将 C 与其子节点 Cc 的边移除;
52.           将 C 与其父节点 Cp 的边移除;
53.         将节点 C 从概念格 L(K)中移除;
54.   for each C∈ReSet do
55.       C 节点个标记域均置为 false，包括 vs;
56.       isDeleted，isDelAtrr 以及 isDelObj 域;
57. End
```

本算法是更新边的主要过程，其中第 4－27 行处理第一类边的情况；第 28－53 行则处理第二类边的情况，更新边的过程如图 3、图 4 所示，这里不再赘述；第 54－56 行是对受影响的节点统一恢复标记位，方便接下来对其他冗余二元关系进行删除。算法执行结束，将生成新概念格对应的 Hasse 图。

3.2　时间复杂度分析

概念格二元关系消减维护算法的时间复杂度主要由两部分组成：1）节点更新所用时间；2）边更新所耗时间。

节点更新的算法时间复杂度主要取决于主 while 循环的执行次数以及内层循环 6－10 行和 13－21 行 for 循环的执行次数。主循环依赖于 *affectedQueue* 队列中删除节点与更新节点的总个数，由定义 5 可知，概念格中的删除节点与更新节点的个数最多不超过 $\|L\|/4$ 个。内层循环 6－10 行的执行次数取决于子节点的个数，它的上界为 $\|M\|$ 个。第13－21 行内层循环的次数取决于父节点的个数，其上界用 $\|G\|$ 表示，则在最坏情况下节点更新算法的时间复杂度为 $O(\|L\|/4\times(\|G\|+\|M\|))$。

边更新的算法时间复杂度取决于主 while 循环与第 5－27 行双层嵌套循环以及第 29－53 行三层循环的执行次数。主循环执行次数依赖于被放入 *edgeChanged* 队列中的减对象节点与删除节点的总个数最多有 $\|L\|/4$ 个。第 5－27 行双循环次数为遍历减对象节点的所有父节点的所有子节点和减属性节点的所有子节点的所有父节点次数，不超过 $2\times\|G\|\times\|M\|$ 次；第 29－53 行执行次数为 $\|G\|^2\times\|M\|$ 次。所以最坏情况下的时间复杂度为 $O(\|L\|/4\times(2\times\|G\|\times\|M\|+\|G\|^2\times\|M\|))$。

综上所述，概念格二元关系消减维护算法在最坏情况下的时间复杂度为 $O(\|L\|/4\times(\|G\|+\|M\|))+O(\|L\|/4\times$

$(2\times\|G\|\times\|M\|+\|G\|^2\times\|M\|))=O(\|L\|\times\|G\|^2\times\|M\|)$。

4 删除单个二元关系的概念格更新示例

下面用简单的实例来说明概念格的维护过程。表 1 列出了一个简单的形式背景 $K=(G,M,R)$，图 5 为对应的概念格。

表 1 形式背景示例

	a	b	c	d	e
1	1	1	1	1	0
2	0	1	1	1	1
3	0	0	0	0	1
4	1	1	0	0	0
5	0	0	1	1	1

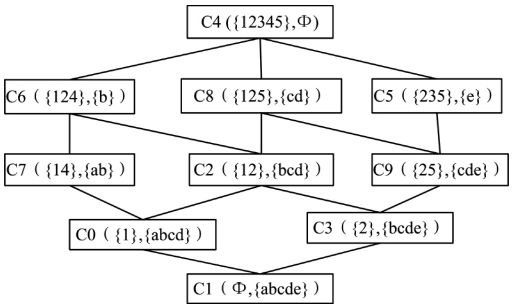


图 5 表 1 形式背景所对应的概念格

求解形式背景在删除冗余二元关系 1Ib 之后所对应的概念格。首先从最大下界节点 C1 出发自底向上找到最小概念 C0 节点，显然 C1 不是最小概念，此时需要对 C1 节点单独处理。由形式背景 K 可知，objSet 集合 $=\{1,2,4\}$，因为 objset 集合存在多个对象，所以当减去 1Ib 关系时对 C1 节点无影响。

然后从最小概念 C0 出发自底向上遍历它的祖先节点寻找受影响的节点，即执行节点更新过程。执行后的概念格如图 6 所示。加粗的节点表示受影响的节点，节点中的下滑线表示节点去除的对象或属性。C2 标记为删除节点，C6 标记为减对象节点，C0 标记为减属性节点，C7 为分割节点，并被分为两个节点 C7 与 C10。为了方便边的更新，C7 被标记为减对象节点，C10 被标记为减属性节点。

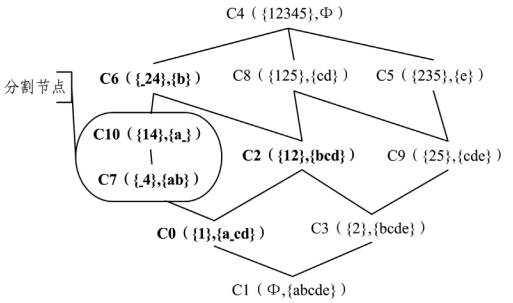


图 6 删除冗余二元关系 1Ib 后节点更新示意图

最后执行边更新节点算法，需要修改的边只有两种情况：1)删除节点的父子节点的边，如 C2 与 C6，C8 的边以及 C2 与 C0，C3 的边需要删除，同时考虑父子节点之间是否新增边；2)父节点为减对象节点，子节点为减属性节点的边。如 C6 与 C10 以及 C7 与 C0 的边被删除，同时考虑 C10 与 C6 的父节点是否新增加边以及 C7 与 C0 的所有子节点是否新增边。算法执行后得到相应的 Hasse 图，如图 7 所示，图中斜体字表示概念节点被删除。

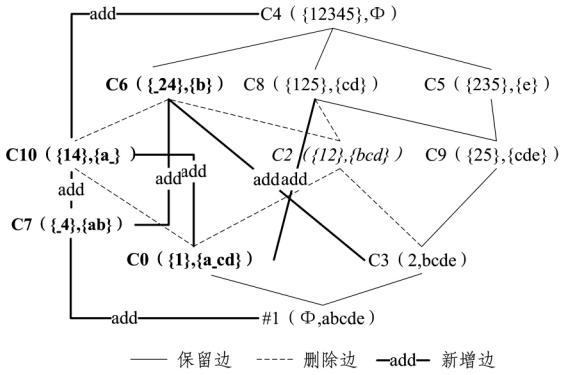


图 7 删除冗余二元关系 1Ib 后边更新示意图

5 实验结果分析

本节采用 Java 语言实现并测试本文算法与 Godin，NextClosure 以及 AddIntent 传统算法。实验平台运行在单处理器多核(AMD Athlon(tm) ×4)的计算环境上。除操作系统外，无其他程序同时运行。实验分为两个部分：1)在 UCI 公共数据集上进行完备性实验；2)在随机生成的数据集上，比较传统算法和本文算法 DelRelation 的运行效率。

5.1 完备性分析

完备性实验选择 3 个 UCI 公共数据集[15] Steel Plates Faults，Cloud 以及 Leaf 进行验证。首先对这些数据集进行预处理生成形式背景，然后在随机删除形式背景中的某个二元关系后，在新形式背景上分别使用 Goidn，AddIntent 算法构格，再使用 DelRelation 算法针对原始概念格进行调整产生新概念格，最后对比 3 个算法所产生的格结构。3 种算法所产生的概念格的数量如表 2 所列。

表 2 形式背景所产生概念格的数量

Algorithms \ Dataset	Cloud	Leaf	Steel Plates Faults
Godin	57	123	8624
AddIntent	57	123	8624
DelRelation	57	123	8624

实验结果进一步表明，本文算法 DelRelation 能够产生完备的概念，即概念的个数、内涵和外延以及上下界与其他算法所产生的格均相同，同时也说明了本文算法所依据的理论的正确性。

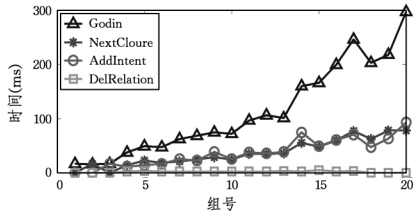
5.2 运行效率分析

本节测试 DelRelation 算法的性能，并与 Godin，NextClosure 以及 AddIntent 算法的构造效率进行比较。实验分为两组且每组实验数据均是由 matlab 自动生成的随机数据集。由于计算环境资源的限制，对比算法运行时间较长，因此实验数据选择规模相对较小的随机数据集。下面分别对两组实验进行具体说明。

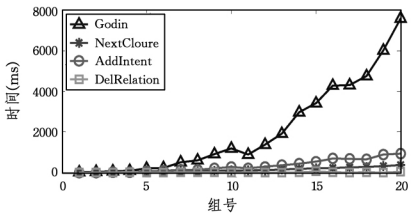
(1)在稀疏和稠密数据集下 DelRelation 算法和传统算法进行性能对比。

该实验中，关系概率分别选择 30%，50%，80%。在每个关系概率下，属性个数固定 $|M|=15$，对象个数从 10 开始，每次增加 10 个，直到 200 为止，一共产生 20 组形式背景。首先在每个形式背景中随机去除某个二元关系，然后利用 Godin，

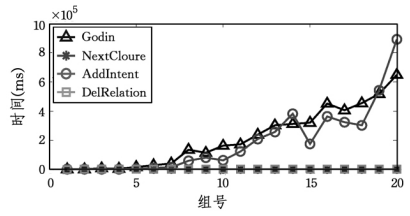
NextCloure 以及 AddIntent 算法在新形式背景上构造概念格并记录构造时间，最后利用本文算法在原有概念格的基础上进行调整得到新的概念格并记录运行时间。实验结果如图 8 所示，横坐标表示组编号，纵坐标表示算法运行时间。



(a)关系概率 30%



(b)关系概率 50%



(c)关系概率 80%

图 8　不同关系概率下的 4 种算法时间性能对比

由图 8(a)、(b)可知，在固定属性数时，随着对象数目的增大，本文算法 DelRelation 与其他算法相比波动幅度较小。说明在原有概念格中进行对象与属性间二元关系的删除要比从形式背景中重新构造概念格节约大量时间。由于 NextCloure 算法不生成格结构，而本文算法生成格结构，因此在图 8(c)中，稠密数据集下 NextCloure 算法与本文算法表现出的时间性能接近。从图 8(a)—图 8(c) 3 个图观察对比得知，随着关系密度的增加，算法的整体时间有所增加，而本文算法始终保持相对稳定的优势，说明了本文算法的有效性。

(2)固定对象数，增加属性数时，DelRelation 算法和传统算法进行性能对比。

该实验的对象个数为 25，关系密度为 30%，属性数目为 25～500，共产生 20 组形式背景，每组相差 25 个属性。随机删去形式背景的某个冗余二元关系，Goidn，NextCloure 和 AddIntent 算法在新形式背景下重新构格，而本文算法 DelRelation 则在原始概念格上进行调整得到新概念格结构，并记录每个算法的构造格结构时间。实验结果如图 9 所示。

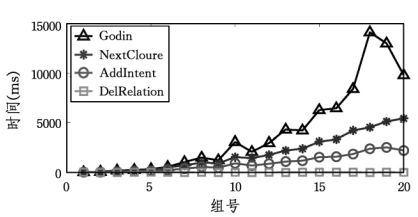


图 9　在固定对象数 $|G|=25$ 时，4 种算法时间性能对比

由图 9 可知，在固定对象数的情况下，随着属性个数的增加，本文算法 DelRelation 相比传统算法运行时间增加缓慢，增加幅度较小，说明对同等规模的概念格删除某个冗余二元关系时，本文算法的时间性能优越。

结束语　本文提出了一种删除冗余二元关系的概念格维护算法。该算法针对原始概念格进行调整得到新概念格，避免重新构格，从而节约了大量的时间。同时本文算法可以处理任意位置的二元关系消减的情况，相对于对象或属性的消减维护，本文算法的优势在于处理那些不在同行或同列的二元关系。实验结果说明所提算法的正确性与有效性，与传统算法相比节省了大量的运行时间。

参 考 文 献

[1] Will R. Restructuring lattice theory: An approach based on hierarchies of concepts[C]// Rival I. ed. Ordered Sets. Dordecht-Boston: Reidel, 1982: 445-470

[2] Azmeh Z, Huchard M, Tibermacine C, et al. Using Concept Lattices to Support Web Service Compositions with Backup Services [C]// 2010 Fifth International Conference on Internet and Web Applications and Services (ICIW). IEEE, 2010: 363-368

[3] Mouliswaran S C, Kumar Ch A, Chandrasekar C. Modeling Chinese Wall Access Control Using Formal Concept Analysis[C]// 2014 International Conference on Contemporary Computing and Informatics (IC3I). IEEE, 2014: 811-816

[4] Mouliswaran S C, Kumar Ch A, Chandrasekar C. Representation of Multiple Domain Role Based Access Control Using FCA[C]// 2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT). IEEE, 2015: 1-6

[5] 李进金，张燕兰，吴伟志，等. 形式背景与协调决策形式背景属性约简与概念格生成[J]. 计算机学报，2014，37(8)：1768-1774

[6] Wei Ling, Sun Y W, Wang D H. Concept Lattice Expansion and Recovery Based on Reduced Formal Context[C]// 2011 International Conference on Electric Information and Control Engineering (ICEICE). IEEE, 2011: 4048-4051

[7] Merwe D V D, Obiedkov S, Kourie D. AddIntent: A New Incremental Algorithm for Constructing Concept Lattice[M]// Concept Lattices. Springer Berlin Heidelberg, 2004: 205-206

[8] 张磊，张宏莉，等. 概念格的属性渐减原理与算法研究[J]. 计算机研究与发展，2013，50(2)：248-259

[9] Zhang L, Zhang H L. An Incremental Algorithm for Removing Object from Concept Lattice[J]. Journal of Computational Information Systems, 2013, 9(9): 3363-3372

[10] 马垣，马文胜. 概念格多属性渐减式构造[J]. 软件学报，2015，16(12)：3162-3173

[11] 姜琴，张卓，王黎明. 基于多属性同步消减的概念格构造算法[J]. 小型微型计算机系统，2016，37(4)：646-652

[12] 智慧来，智东杰. 概念格维护原理与算法[J]. 计算机工程与应用，2014，50(6)：96-101

[13] 张卓，杜鹃，王黎明. 基于负载均衡的模糊概念并行构造算法[J]. 控制与决策，2014，29(11)：1935-1942

[14] Wei L, He M. Concept Lattice Compression based on K-means [C]// 2013 International Conference on Machine Learning and Cybernetics (ICMLC). IEEE, 2013: 802-807

[15] Frank A, Asuncion A. UCI machine learning repository [EB/OL][2016-1-10]. http://www.ics.uci.edu