浙 江 大 學

本 科 生 毕 业 设 计 报 告

项目名称 基于大规模知识图谱的规则挖掘系统的实现

姓　　名 周自强

学　　号 3120101943

指导教师 陈华钧

专　　业 计算机科学与技术

学　　院 计算机学院

A Dissertation Submitted to Zhejiang University for the Degree of Bachelor of Engineering

TITLE:Rule Mining System in Large Scale Knowledge Bases

Author: Zhou Ziqiang

StudentID 3120101943

Mentor Chen Huajun

Major: Computer Scienceand Technology

College: Zhejiang University

Submitted Date: 2016-5-30

浙江大学本科生毕业论文（设计）诚信承诺书

1.本人郑重地承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。

2.本人在毕业论文（设计）中引用他人的观点和参考资料均加以注释和说明。

3. 本人承诺在毕业论文（设计）选题和研究内容过程中没有抄袭他人研究成果和伪造相关数据等行为。

4. 在毕业论文（设计）中对侵犯任何方面知识产权的行为，由本人承担相应的法律责任。

毕业论文（设计）作者签名：

_____年_____月_____日

摘要

知识库是有大量的世界实体以及它们之间的关系组成，以有向图的形式存储，并可以通过三元组的形式将其表示出来。然而知识库不是完整的，因此我们需要设计实现一个规则挖掘系统，从知识库中挖掘出规则，通过这些规则由已有的事实推断出更多的事实，对知识库进行完善。

AMIE 是一个能够从已有的知识库中挖掘出规则的系统，然而它也有局限性，即只能挖掘封闭规则，对于非封闭规则并不支持。如何对非封闭规则进行挖掘，实现一个较为完善的规则挖掘系统，就是我们这次研究的课题。

关键词　规则挖掘；非封闭规则；AMIE

Abstract

Large scale knowledge bases (KBs) are consists of a lot of real world entities and their relations. KB can be expressed in the form of a directed graph, which is stored with three tuples. However, the knowledge bases are not completed. So we need to design and implement a rule mining system excavating rules from the KBs and get more facts by the rules. Thus, we can improve the KB.

AMIE is a rule mining system which can mine rule from the existing KBs. However, it also has the limitation, that is, only the closing rules can be mined. How to mine opened rules to achieve a more perfect system, is the subject of our study.

Keywords rule mining; opened rules; AMIE

目录

第 1 章　项目背景

1.1 背景介绍

近几年来，许多知识库，例如 Yago、DBPedia、Freebase 开始兴起，这些知识库在各种实际应用例如自然语言问答、语义搜索引擎等方面都有重要作用。知识库针对某一领域或者多个领域，将各个互相联系的事实进行存储、组织、管理与使用。这些知识库由数百万的世界实体以及它们之间的关系所组成，以有向图的形式存储，结点表示实体，连接表示实体间的关系。尽管这样的知识库包含成千上万的实体，它仍然是稀疏的，即它在实体之间缺失了大量的关系。在大规模知识库中存储的实体（entity），包括了人、国家、河流、城市、大学、电影、动物等等信息；还有许多事实（fact），例如北京是中国的首都、美国的官方语言是英语、长江是中国的一条河流等。从这些知识库中可以知道许多信息，例如美国总统是谁、中国人讲什么语言，甚至可以知道奥巴马的妻子叫什么名字。

然而知识库并不是完整的。也就是说，知识库中缺失了很多信息。这时候，我们需要从已有的知识库中进行规则挖掘，进而找寻两个实体之间的关系，补充知识库的信息，使得知识图谱密集化。例如一条规则：livesIn(h, p) ^ marriedTo(h, w) =>livesIn(w, p)，它表示如果两个人是夫妻关系，那么他们（通常）住在同一个城市。

挖掘出这样的规则，我们能够利用它做什么？首先，我们可以进行知识库的补全，例如，我们知道甲有一个妻子乙，并知道甲居住的城市 A，那么我们就可以通过挖掘出的规则推断出妻子乙也居住的城市 A；第二，这些规则可以找出知识库中一些潜藏的错误信息，例如知识库中说甲的妻子乙居住在城市 B，我们就能够判断这条信息很有可能是错误的；最后一点，这些规则可以使我们更好地理解整个知识库，例如我们可以发现使用相同语言国家之间贸易往来频繁、婚姻是一种对称的关系等等。

1.2 项目介绍

1.2.1 项目内容

我们的目标是实现一个规则挖掘系统，用来分析大规模知识图谱。由于 AMIE 在大规模知识图谱的分析与规则挖掘中具有良好的效果与较高的挖掘效率，我们的系统基于 AMIE 实现。AMIE 算法在挖掘封闭规则这一方面已经有了较为成熟的理论支持与算法实现，但是在非封闭规则这一方面还有欠缺。因此我们需要实现的规则挖掘系统，不仅仅对封闭规则进行挖掘，对非封闭规则的挖掘也有一定程度的支持。

1.2.2 可行性分析

大规模知识库的规则挖掘这一课题有一定难度，但基本可行，原因主要基于以下几个方面：

1. 在大规模知识图谱中，存在成千上万的事实（fact），从这些已有事实中我们可以找到一些规则，用来描述这些事实。而通过挖掘出来的规则，我们可以进一步完善知识库。因此，对知识库进行相应的规则的挖掘是可行的。

2. 关联规则是反映一个事物与其他事物之间的相互依存和关联性，是数据挖掘中的一个重要技术。现在有许多关于关联规则挖掘的算法，例如 AMIE 在不完整知识库中的关联规则挖掘，或者使用桥接实体以及随机游走算法来进行规则的挖掘。并且 AMIE 已有相应的代码来实现关联规则挖掘的部分功能。由此可以看出，在知识库中进行关联规则的挖掘并不是不可能的。

3. 规则挖掘的算法越来越成熟，实际应用也越来越广泛，越来越多的人在研究学习规则挖掘，并提供了许多高质量的开源代码，这对于我们实现关联规则挖掘有很大的帮助。

4. 网络上提供了许多知识库数据例如 YAGO、DBpedia 的数据集用于测试，这对于我们实现并调试规则挖掘算法提供了便利。

第 2 章 项目实施方案

2.1 相关概念

AMIE 是一个数据规则挖掘系统，它能够实现从一个知识库中提取出逻辑规则的置信度等信息，从而进行规则的挖掘。我们在 AMIE 的基础上进行规则挖掘系统的开发。这里我们需要了解一些概念。

2.1.1 RDF 知识库

知识库中的一个 fact 可以使用一个三元组来表示，形式<x, r, y>，其中 x 为主语（subject），r 为关系或谓语（relation or predicate），y 为宾语（object）。这里我们表示成 r(x, y)。知识库就是这些三元组的集合。

2.1.2 函数（function）

在关系 r 中，对于每一个 subject，最多只有一个 object 阈值对应，这样的关系称为函数（function）。另外，我们使用标记 functionality，关系 r 的 functionality 是一个 0 到 1 的值，当 r 是一个 function 的时候值为 1。

$$fun(r) = \frac{\#x{:}\exists y{:}r(x,y)}{\#(x,y){:}r(x,y)}$$

2.1.3 规则（rule）

形式如 B1∧B2∧…∧Bn⇒r(x,y)，缩写为$\vec{B}\Rightarrow r(x,y)$

其中 B1、B2…是形如 r(x,y)的包含变量的三元组，每个都是一个 atom，式子右侧的 r(x,y)为 head atom，左侧的 B1∧B2∧…∧Bn 为 body atoms。

规则一般是指封闭规则，即规则中的各个变量至少出现 2 次。在封闭规则中又有一类特殊的规则，包含了 Instantiated Atoms，即 subject 或者 object 有实例化的数据。

这里我们将规则进行一些扩展，即挖掘非封闭规则。非封闭规则比较广泛，我们将其进行一些限制。例如，对于一个 subject 变量 a，查看 a 的 relation，我们可以发现这样的规律：R1∧R2∧…∧Rn⇒R，这类似于关联规则的挖掘。这样，

我们可以挖掘出一条规则 R1(a,b)∧R2(a,c)∧…∧Rn(a,x)⇒R(a,y)。这里，我们不关心 object 是什么，我们关心的是 a 的关系之间的联系。

2.1.4 支持度（Support）

对于一个规则，需要在知识库中有对应的事实（fact）来支持。一般来说，支持度（support）是指满足一条规则的实例数目，但是为了使规则在增加 body 的 atom 时，支持度相应地不变或减少，我们用所有满足规则限制的实例中所有不同的 head（一般规则确定了 relation 实例，这里只指 subject-object 对）的数量来表示支持度。

$$support(\vec{B}\Rightarrow r(x,y)) = \#(x,y):\exists z1,\ldots,zm:\vec{B}\wedge r(x,y)$$

其中 z1,⋯zm 是除了 x、y 之外的变量。

2.1.5 HeadCoverage

为了对挖掘出的规则进行一些限制，我们只计算支持度是不够的。由于支持度是一个绝对数字，在这里我们使用 head coverage 来表示一个相对值：

$$headcoverage(\vec{B}\Rightarrow r(x,y)) = \frac{support(\vec{B}\Rightarrow r(x,y))}{size(r)}$$

其中 size(r):=#(x’,y’) : r(x’,y’)，表示关系 r 对应的 fact 的数量。

headcoverage 对挖掘出的规则进行了一些限制，低于 headcoverage 阈值的规则将被抛弃。在实际计算过程中，我们会根据 minHeadCoverage 与 size(r)的乘积计算出一个 minSupport，然后对它进行比较。

2.1.6 标准置信度（standard confidence）

对于挖掘出的规则，如何来衡量其可信度。这里我们使用标准置信度来表示：

$$confidence(\vec{B}\Rightarrow r(x,y)) = \frac{support(\vec{B}\Rightarrow r(x,y))}{\#(x,y):\exists z1,\ldots,zm:\vec{B}}$$

其中分子是支持度，分母是指满足规则中的 body 部分的 subject-object 对的数目。

2.1.7 PCA 置信度（partial completeness assumption confidence）

另一种计算规则可行度的方法是用 PCA 置信度。与标准置信度不同的是，分母中需要满足的条件中增加了 r(x, y')项目，用来表示满足规则或者不满足规则的所有 x- y'对，注意这里 x 与 head atom 中相同。因此，相较标准置信度的计算，少了一些无关项（无关项是指 subject 没有出现在 head atom 中的 body 项目）。PCA 置信度的计算方式如下：

$$confidence_{pca}(\vec{B}\Rightarrow r(x,y)) = \frac{support(\vec{B}\Rightarrow r(x,y))}{\#(x,y):\exists z1,\ldots,zm,y':\vec{B}\wedge r(x,y')}$$

2.2 AMIE 规则挖掘算法

输入知识库 KB，阈值 minHC，规则的最大长度 maxLen，最小置信度 minConf。

输出规则。

首先获取一个规则队列，初始包含所有的 head atom，且 size 为 1。

然后每次从队列中获取一个 rule，如果这个 rule 满足一定条件，则添加到输出队列；如果该 rule 的长度不超过最大长度 maxLen，则对该 rule 进行进一步完善，并将完善后的满足条件的 rule 添加到规则队列中。

伪代码：

```
function AMIE(KB K, minHC, maxLen, minConf)
    q = [r1(x, y), r2(x, y)…rm(x, y)]
    out = <>
    while ¬ q.isEmpty() do
        r = q.dequeue()
        if AcceptedForOutput(r, out, minConf) then
            out.add(r)
        end if
        if length(r) <maxLen then
            R(r) = Refine(r)
            q.enqueue(rc)
        end if
```

```
    end while
    return out
end function
```

流程图如下：

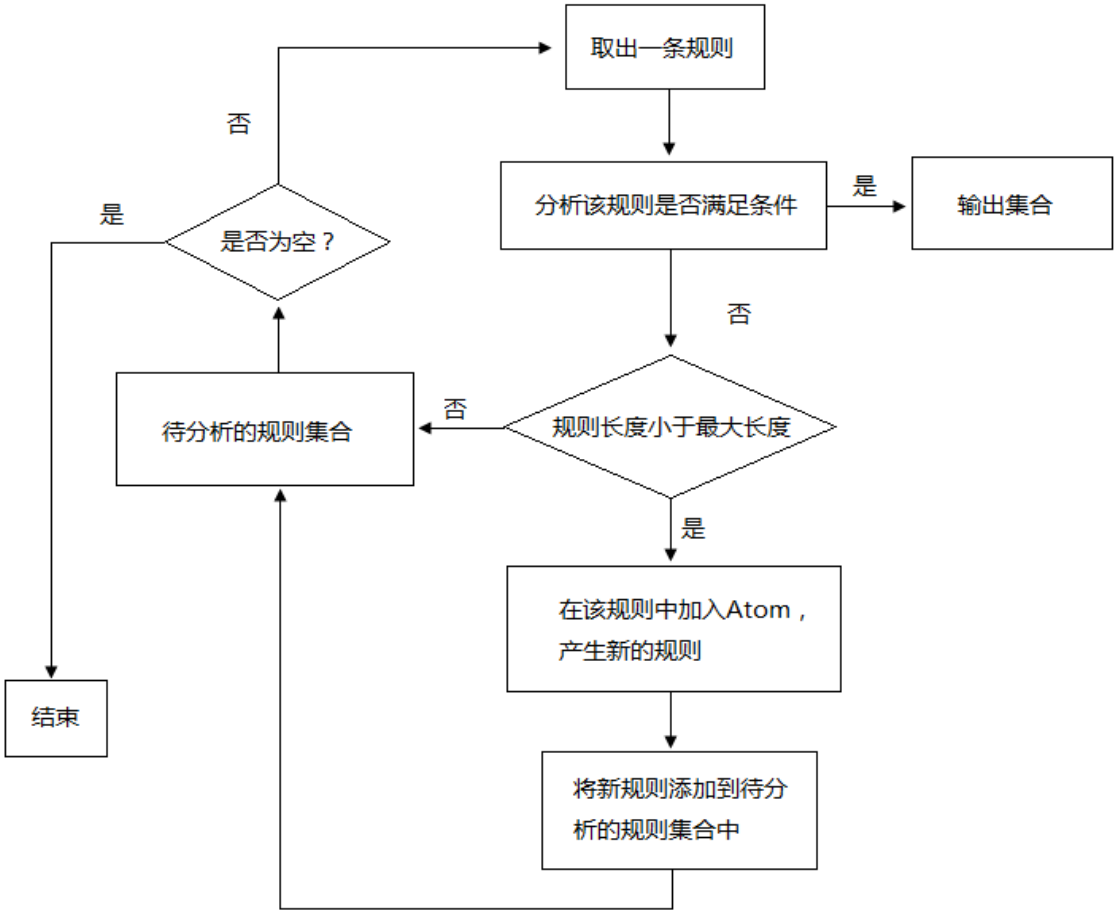

图 2-1规则挖掘流程

2.3 项目实施计划

3 月 1 日~3 月 15 日：熟悉 AMIE，了解封闭规则挖掘算法。

3 月 16 日~3 月 31 日：熟悉 AMIE 代码，阅读相关论文，了解 AMIE 的整体架构与算法实现。

4 月 1 日~4 月 7 日：获取一些知识库，使用 AMIE 进行知识库中规则的挖掘，将结果进行统计对比。

4 月 8 日~3 月 15 日：在 AMIE 的基础上进行系统开发，实现封闭规则挖掘。

4 月 16 日~4 月 23 日：编写代码，进一步完善规则挖掘算法的细节，并使用提供的知识库数据进行挖掘测试。

4 月 24 日~4 月 30 日：优化算法，使用多种方法提高算法效率，进行规则挖掘，并进行结果分析与对比。

5 月 1 日~5 月 7 日：在实现基本的规则挖掘功能的基础上，提供系统对中文知识图谱规则挖掘的支持。

5 月 8 日~5 月 12 日：学习非封闭规则挖掘的相关知识，获取理论基础。

5 月 13 日~5 月 23 日：在实现封闭规则挖掘的基础上，进一步进行非封闭的规则挖掘，并进行数据测试与比较。

5 月 24 日~5 月 28 日：完善一些细节，修复各类 bug，完成毕业设计。

2.4 本章小结

这章主要介绍的 AMIE 算法中的一些基本概念，我们规则挖掘过程中需要用到的一些公式，还有规则挖掘的基本流程以及项目的进度计划。这里我们对规则挖掘过程进行了一个总览，后面的章节就是对这个挖掘过程的细化。

第 3 章　在项目中负责的具体工作

3.1 AMIE 基础上的代码编写及修改

AMIE 基础上的代码编写及修改。算法的流程之前已经介绍过了，这里讲一下具体实现。

3.1.1 知识库存储及数据处理

fact 是以三元组的形式存在的。这里我们沿用 AMIE 的存储形式（HashMap），在知识库类 KB 中，有：

```
Map<String, Map<String, Map<String, Int>>> subjectRelationObjectMap;
Map<String, Map<String, Map<String, Int>>> subjectObjectRelationMap;
Map<String, Map<String, Map<String, Int>>> relationSubjectObjectMap;
Map<String, Map<String, Map<String, Int>>> relationObjectSubjectMap;
Map<String, Map<String, Map<String, Int>>> objectSubjectRelationMap;
Map<String, Map<String, Map<String, Int>>> objectRelationSubjectMap;

Map<String, Int> subjectSize = new HashMap<String, Int>();
Map<String, Int> relationSize = new HashMap<String, Int>();
Map<String, Int> objectSize = new HashMap<String, Int>();
```

使用 6 个 map 是为了更快地对 fact 进行定位。另外 3 个 map 是存储单个变量对应的 fact 数量。其中 Int 是自己定义的类，内部是 long 型。有了这些 map，我们对知识库的搜索就变得非常简单了。

KB 中有很多重要的方法，这些方法在 AMIE 中已经很好地实现，这里只是简单地讲一下。

countProjectionBindings 方法：

```
public Map<String, Int> countProjectionBindings(String variable, String[]
projectionTriple, List<String[]> otherTriples)
```

方法查询满足条件的 variable 的实例，返回这个 variable 实例以及它对应的 projectionTriple 的数量。这里需要满足的条件是指满足 projectionTriple 以及 otherTriples 规则的 variable。

selectDistinct 方法：

```
public Set<String> selectDistinct(String variable, List<String[]> query)
```

方法查询满足 query 条件的 variable 的实例，返回 variable 实例集合。

resultsOneVariable 方法：

```
protected Map<String, Int> resultsOneVariable(String... projectionTriple)
```

对 projectionTriple 中的单个变量进行实例化，并查找这样的实例的数目。

resultsTwoVariables 方法：

```
protected Map<String, Map<String, Int>> resultsTwoVariables(String[]
triple, int pos1, int pos2)
```

对 triple 中的 2 个变量进行实例化，并查找这样的实例的数目。

resultsThreeVariable 方法：

```
protected Map<String, Map<String, Map<String, Int>>>
resultsThreeVariables(String[] triple, int pos1, int pos2, int pos3)
```

对 triple 中的 3 个变量进行实例化，并查找这样的实例的数目。

3.1.2 规则存储与计算

规则类 Rule，使用三元组 list 来存储规则：

```
private List<String[]> triples = new ArrayList<String[]>();
```

3.1.2.1 Rule 类的一些重要的属性

support(long)：support 表示满足该规则的实例数目（支持度）。
stdBodySize(double)：STD 置信度=support/stdBodySize。
pcaBodySize(double)：PCA 置信度=support/pcaBodySize。
ancestor(List<Rule>)：该规则的祖先，即该规则是从哪些规则衍生出来的。

opened(boolean)：标示该规则是否是非封闭规则。

3.1.2.2 Rule 类的一些重要的方法

getAllVars

```
public List<String> getAllVars()
```

返回规则中的所有变量。

getOpenVars

```
public List<String> getOpenVars()
```

返回规则中的所有不封闭的变量（不封闭指只出现 1 次）。

getFixedVars

```
public List<String> getFixedVars()
```

返回非封闭规则中的固定变量（即在非封闭规则中的封闭的变量）。

containsLevel2RedundantSubgraphs

```
public boolean containsLevel2RedundantSubgraphs()
```

检测规则是否包含两层相同的子规则。

isClosed

```
public boolean isClosed()
```

检测规则是否封闭（封闭是指所有变量至少出现 2 次）。

hashCode

```
@Override
public int hashCode()
```

hashCode 函数在添加规则到规则队列之类的操作中用到，用于计算规则的哈希值。这里我们尽量使不同的规则有不同的 hashcode，以减少下面 equals 的调用次数，提高程序运行效率。这里 hashcode 是通过支持度、规则长度、head 中的实例化的值来计算。另外，对于非封闭规则，我们需要进行一些特殊处理，用来和封闭规则区分开。具体实现：

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (int) support;
    result = prime * result + (int) getRealLength();
    result = prime * result + ((getHeadKey() == null) ? 0 :
getHeadKey().hashCode());
    if (this.opened) {
        result = prime * result * this.getTriples().size();
    }
    return result;
}
```

equals

```
@Override
public boolean equals(Object obj)
```

由于 hashCode 并不能将规则完全区分开来，这时候在 map 中添加规则的时候，遇到相同的哈希值时，会继续调用 equals 来比较。如何判断两条规则是否等价是个关键问题。

首先我们来看什么是等价规则（规则中的变量都以“?”开头）。

例如有两条规则如下：

规则 1：?a <R1> ?b ∧ ?b <R2> ?c ⇒ ?a <R3> ?c

规则 2：?f<R2> ?e ∧ ?d<R1> ?f⇒ ?d<R3> ?e

这两条规则就是等价的。也就是说，两条规则 relation 相互对应，相应的变量相互对应（变量不一定要相同），他们就是等价的。

对规则进行等价性判断：一条规则可以表示成有向图图，可以通过图的同

构判断两个规则是否相同。而有向图的同构判断算法又比较复杂，有一种出入度序列法，可以较快地判断（接近多项式时间复杂度），但是在最坏的情况下仍有指数的时间复杂度。

要准确判断两个规则是否等价比较困难，但是较精确地判断还是比较容易实现的。为了提高规则挖掘的效率，相较于 AMIE，我们重写了规则比较算法。在比较过程中，先对规则进行简单的判断（例如，判断两个规则的长度、head atom 是否相同等），若相同，接着将规则转化成有向图，使用图中每个结点的出度、入度以及各条边的值（这里是指 Relation），计算出该规则的一个哈希值，然后对规则的哈希值进行判断。虽然这样不能达到非常准确的要求，但是计算速度快，效率高。

代码如下（RuleGraph.java），遍历图中所有的有向边，其中对 head 的有向边进行特殊处理

```
// head edge
edge.hashValue *= 31;

// all edges
for (int i=0; i<edgeNum; i++) {
    Edge edge = edgeList[i];
    edge.hashValue += edge.in.in * DEFAULT_HASH_VALUE_IN;
    edge.hashValue += edge.in.out * DEFAULT_HASH_VALUE_OUT;
    edge.hashValue = edge.hashValue * 31 + edge.str.hashCode();
    edge.hashValue += edge.out.in * DEFAULT_HASH_VALUE_IN;
    edge.hashValue += edge.out.out * DEFAULT_HASH_VALUE_OUT;
    this.hashValue += edge.hashValue;
}
```

另外，为了提高比较的效率，我们提前 new 出 RuleGraph 的实例，之后的 equals 过程直接调用这些实例，而不是每次都新建两个 RuleGraph 对象。

3.1.3 规则挖掘过程

为了提高 CPU 利用率，提升规则挖掘效率，使用多个线程进行挖掘。初始默认 4 个线程。规则挖掘大致步骤如下：

a) 从规则队列中取出规则
b) 判断规则是否满足输出条件，满足则输出
c) 对规则进行进一步推演，得到更多的规则，并将这些规则添加到规则队列中。

代码如下所示：

```
// 线程运行函数，规则挖掘的整体流程
@Override
public void run() {
```

```
    while(true) {
        Rule r = null;
        // 取出一条规则
        synchronized (q) {
            Iterator<Rule> iterator = q.iterator();
            if (iterator.hasNext()) {
                r = iterator.next();
                iterator.remove();
            } else {
                break;
            }
        }
// 判断规则是否满足输出条件（主要是计算置信度）
        if(miningAssistant.acceptForOutput(r)) {
            synchronized (out) {
                out.add(r);
                ruleConsumer.addRule(r, true);
            }
        }
// 对规则进行推演，得出更多的规则，并添加到规则队列
        if(!r.isPerfect() && r.length() < miningAssistant.getMaxLen()) {
            Collection<Rule> R = miningAssistant.refine(r);
            synchronized (q) {
                q.addAll(R);
            }
        }
    }
}
```

3.1.4 规则队列的初始化

在线程开始运行之前，我们需要将规则队列初始化，也就是说要先找出长度为 1 的规则，即只包含 head atom 的规则。我们先来看 AMIE 中的获取方法：使用 countProjectionBindings，传入三变量的 projectionTriple，然后对 relation 变量进行查询。这样得到的结果就是 relation 的一个 map。然后建立新的规则，长度为 1，包含 subject 变量和 object 变量，以及 relation 实例。

我们在这里将其拓展。

首先是 Instantiated Atom 的添加。Instantiated Atom 在这里指除了 relation 以外，还有另一个变量也被实例化。在这之前，AMIE 已经寻找到了 head atom 并将其添加到规则队列中，我们对这些寻找到的 head atom 进一步处理。

对于每一个 head atom，将其三元组传入 countProjectionBindings，将

subject 变量（object 变量）作为变量传入，这样就得到了 subject 变量（object 变量）的实例 map，之后对 head atom 中的 subject 变量（object 变量）进行实例化，得到新的 head atom，这时，这个 atom 就是 Instantiated Atom，检验其支持度，满足条件后将其加入规则队列。部分代码如下：

```
// succedent为AMIE挖掘出的head atom
Map<String, Int> objects = this.kb.countProjectionBindings(succedent[2],
succedent, otherProjectionTriples);
for (Entry<String, Int> object : objects.entrySet()) {
//检验其support参数
   if (object.getValue().value >= minSupportThreshold) {
      succedent = succedent.clone();
//实例化object
      succedent[2] = object.getKey();
      candidate = new Rule(succedent, cardinality);
//添加到规则队列
      out.add(candidate);
   }
}
```

实例化 subject 的过程同上。

另外，由于我们添加了对非封闭规则的支持，为了挖掘非封闭规则，初始化过程中对非封闭规则进行一些额外的处理。这里我们只用在之前挖掘出的二变量的规则的基础上细化。我们拷贝这些规则，在上面打上非封闭规则的标签，然后添加到规则队列中。

部分代码如下：

```
public Collection<Rule> getInitialAtomsOpened(Collection<Rule> in,
double minSupportThreshold) {
   Collection<Rule> out = new LinkedHashSet<Rule>();
// 查询是否允许非封闭规则挖掘
   if (!this.getAllowOpenedAtoms()) {
      return out;
   }
   for (Rule rule : in) {
      if (rule.getTriples().size() != 1) {
         continue;
      }
// 只对二变量的head atom进行处理
      if (KB.numVariables(rule.getHead()) != 2) {
         continue;
      }
      Rule cand = new Rule(rule, rule.getSupport());
// 添加非封闭规则标签
      cand.setOpened(true);
```

```
        out.add(cand);
    }
    return out;
}
```

3.1.5 refine 过程

refine 过程就是规则推演的过程，即从一条长度为 L 的规则，推出多条长度为 L+1 的规则。我们来看 refine 的过程：

```
public Collection<Rule> refine(Rule in) {
//通过headcoverage阈值计算出对应的该规则的support阈值
    double threshold = this.getThreshold(in);

//查询达到threshold阈值的从输入规则in衍生出来的所有规则
    Collection<Rule> out = new LinkedHashSet<Rule>();
    Collection<Rule> danglingAtoms = getDanglingAtoms(in, threshold);
    Collection<Rule> instantiatedAtoms = getInstantiatedAtoms(in,
danglingAtoms, threshold * this.getThresholdConstMulti());
    Collection<Rule> closingAtoms = getClosingAtoms(in, threshold);
    Collection<Rule> openedAtoms = getOpenedAtoms(in, threshold *
this.getThresholdOpenedMulti());

//将查询出的规则添加到规则队列
    out.addAll(danglingAtoms);
    out.addAll(instantiatedAtoms);
    out.addAll(closingAtoms);
    out.addAll(openedAtoms);
    return out;
}
```

可以看到，在获取 Instantiated Atom 以及 Opened Atom 的时候，支持度阈值乘了一个倍数。这是为了对规则进行更大的限制，从而挖掘出质量更高的规则。

3.1.5.1 DanglingAtom

新建 atom，包含三个新变量（subject、relation、object），将其中一个变量（subeject 或 object）修改为规则中的已有变量，一般是指规则中的非封闭变量，另一个则为 dangling 变量，再将 relation 进行实例化。下面就是查询满足条件的 relation：

将新的 Atom（三变量）添加到原有的规则中（之后把规则还原），规则长度增加 1，然后调用查询函数 countProjectionBindings，查询变量为 relation 变量，新的规则作为限制条件。这样得到了一个 map，存储变量实例-

支持度数据对。之后遍历这些 relation 实例，过滤出满足 headcoverage 的规则，检查规则是否冗余，满足这些条件后，将该规则添加到规则队列。

简单代码如下：

```
// 获取三元组（包含三个新变量）
String[] newTriple = rule.getTriplePattern();
// 将其中一个变量设置为规则中的已有变量
newTriple[0] = rule.getJoinVariable();
// 或者 newTriple[2] = rule.getJoinVariable();
// 向规则中添加 atom，用于之后的查询
rule.getTriples().add(newTriple);
// 查询满足条件的 relation 实例，返回该实例及对应的 headatom 的数量，用以表示支持度
Map<String, Int> promisingRelations = this.kb.countProjectionBindings(newTriple[1], rule.getHead(), rule.getBody());
// 遍历 relation，过滤出满足条件的规则，并添加到规则队列
for (Entry<String, Int> relation : promisingRelations.entrySet()) {
   long cardinality = relation.getValue().value;
   if(cardinality < minCardinality) {
      continue;
   }
   newTriple[1] = relation.getKey();
   Rule candidate = new Rule(rule, newTriple, cardinality);
   if (!candidate.isRedundantRecursive()) {
      candidate.setHeadCoverage(candidate.getSupport() / this.headCardinalities.get(candidate.getHeadRelation()));
      candidate.setSupportRatio(candidate.getSupport() / this.kb.getSize());
      candidate.setParent(rule);
      output.add(candidate);
   }
}
```

3.1.5.2 Insantiated Atom

获取 Instantiated Atom，可以从 Dangling Atom 出发，进一步实例化刚添加进的 atom 中的非封闭变量。为了获取这些实例，我们使用 countProjectionBindings 进行查询，查询变量为非封闭变量。之后，再对查询到的所有实例进行遍历，过滤出满足条件的规则。

部分代码如下：

```
// 查询变量为刚从dangling atom过程中添加进的atom中的非封闭变量
Map<String, Int> constants =
this.kb.countProjectionBindings(danglingEdge[danglingPosition],
candidate.getHead(), candidate.getBody());
// 遍历所有查询到的实例
for (Entry<String, Int> constant : constants.entrySet()) {
   long cardinality = constant.getValue().value;
// 比较headcoverage阈值
   if (cardinality >= minCardinality) {
      String[] lastPatternCopy =
candidate.getTriples().get(lastTripplePatternIndex).clone();
      lastPatternCopy[danglingPosition] = constant.getKey();
      Rule cand = candidate.instantiateConstant(danglingPosition,
constant.getKey(), cardinality);
      if (cand.getRedundantAtoms().isEmpty()) {
         cand.setHeadCoverage((double)cardinality /
headCardinalities.get(candidate.getHeadRelation()));
         cand.setSupportRatio((double)cardinality /
(double)getTotalCount(candidate));
         cand.setParent(candidate);
// 添加到规则队列
         output.add(cand);
      }
   }
}
```

3.1.5.3 Closing Atom

Closing atom 针对那些非封闭变量数量小于等于 2 的规则。新建 Atom，将 subject 与 object 变量修改为规则的非封闭变量，使将要产生的规则变为封闭。relation 作为查询变量。接下来调用 countProjectionBindings，查询出满足条件的 relation 实例，遍历这些实例，过滤出规则并添加到规则队列。

部分代码如下：

```
//向规则中添加新的atom，更新查询条件
rule.add(newTriple);
//查询满足条件的relation
Map<String, Int> promisingRelations =
this.kb.countProjectionBindings(newTriple[1], rule.getHead(),
rule.getBody());
//恢复原有规则
rule.getTriples().remove(nPatterns);
```

```
for(String relation: promisingRelations.keySet()){
   long cardinality = promisingRelations.get(relation).value;
//支持度检查
   if (cardinality < minCardinality) {
      continue;
   }

   newTriple[1] = relation;
//新建规则
   Rule candidate = new Rule(rule, cardinality);
   candidate.add(newTriple);
//冗余检查
   if(!candidate.isRedundantRecursive()){
      candidate.setHeadCoverage((double)cardinality /
(double)this.headCardinalities.get(candidate.getHeadRelation()));
      candidate.setSupportRatio((double)cardinality /
(double)this.kb.getSize());
      candidate.setParent(rule);
//添加到规则队列
      output.add(candidate);
      candidate.flag = new String("ClosingAtoms");
   }
}
```

3.1.5.4 Opened Atom

Opened Atom 针对那些 opened 属性为 true 的规则，即非封闭规则。我们在非封闭规则内部添加的 opened 属性，用来跟封闭规则做区分。

在 2.1 中我们已经讲过要挖掘的非封闭规则的形式。对于一个输入规则，我们首先要获取到该规则的固定变量 fixedVar，即规则中所有 Atom 共有的变量。这里有一个特殊情况，就是当规则长度为 1，也就是说规则只有 head atom 时，固定变量可以是 subject 和 object 两个变量中的任意一个。得到固定变量后，我们开始使用 countProjectionBindings 查询，查询变量为 relation。之后对 relation 所有实例进行遍历，检验是否满足条件。这里有一点不一样的是，对于非封闭规则，计算支持度的方式与封闭规则不同。封闭规则计算支持度是对 head atom 进行计数，这里我们对 head atom 中的固定变量进行计数。这里调用 countDistinct，查询变量为固定变量，查询条件为新的非封闭规则。得到支持度后，再检查是否大于支持度阈值。条件都满足后，将新的非封闭规则加入规则队列，并标记为非封闭规则。

部分代码如下：

```
Map<String, Int> relations = this.kb.countProjectionBindings(newTriple[1],
in.getHead(), in.getBody());
in.getTriples().remove(nPatterns);
for (Entry<String, Int> entry : relations.entrySet()) {
   if (entry.getValue().value < minCardinality) {
      continue;
   }
   if (entry.getKey().equals(in.getHead())) {
      continue;
   }
   String tmp2 = newTriple[1];
   newTriple[1] = entry.getKey();
   Rule candidate = new Rule(in, newTriple, entry.getValue().value);
   newTriple[1] = tmp2;
   long sup = this.kb.countDistinct(fixedVar, candidate.getTriples());
   if (sup < minCardinality) {
      continue;
   }
   if (!candidate.getRedundantAtoms().isEmpty()) {
      continue;
   }
   candidate.setSupport(sup);
   candidate.setParent(in);
   output.add(candidate);
   candidate.flag = new String("OpenedAtoms");
   candidate.setFixedVar(fixedVar);
}
newTriple[fixedPos] = tmp;
```

3.1.6 acceptForOutput

acceptForOutput 过程用来检查一条规则是否满足输出条件。

对于包含 Instantiated Atom 的规则，我们这样规定，规则中 Instantiated Atom 的数量至少为 2。这是因为很少会有只包含一个 Instantiated Atom 的规则。

然后是计算规则的置信度。这里我们只计算规则的标准置信度。由标准置信度的计算公式 StdConfidence=support/bodySize，支持度 support 我们在挖掘规则的过程中已经得到了，bodySize 对于不同的规则计算方式也不同。

对于 head 不是 Instantiated Atom 的封闭规则（head 中 subject 与 object 都为变量），bodySize 计算方式在 2.1.6 中提到，查询规则的 head 中的两个变量对的实例数目，实例应满足规则中的 body 部分的限制。

对于 head 为 InstantiatedAtom 的封闭规则（head 中 subject 或者 object 被实例化），由于 head 只有一个变量，我们只需要对这一个变量进行查询，查询应满足规则中的 body 部分的限制。

对于非封闭规则，尽管 head 中有两个变量，但是其中一个变量（指非固定变量）并没有出现在 body 中，因此我们没有办法查询这两个变量的满足 body 条件限制的实例。这里我们忽略这个非固定变量，再联系之前所述，非封闭规则在计算支持度的时候，只使用了固定变量来进行查询，那么这里计算 bodySize 的时候，我们同样只使用固定变量。这里我们用到另一个查询函数 countDistinct，用来查询满足条件的变量的实例个数。输入固定变量和规则的 body 部分，我们就可以查询到非封闭规则的 bodySize。计算 bodySize 的代码如下：

```
// antecedent为待计算规则的body部分
// head为待计算规则的head部分
if (!body.isEmpty()){
   try{
      if(KB.numVariables(head) == 2){
         // head中包含两个变量
         String var1, var2;
         var1 = head[KB.getFirstVarPos(head)];
         var2 = head[KB.getSecondVarPos(head)];
bodySize = (double) computeStdBodySize(var1, var2, rule);
      } else {
         // head中只包含一个变量（head为Instantiated Atom）
bodySize = (double) this.kb.countDistinct(rule.getFunctionalVariable(),body);
      }
      rule.setStdBodySize((long)bodySize);
   }catch(Exception e){

   }
}
.....

// 计算bodySize的computeStdBodySize部分代码
if (!query.getOpened()) {
// 封闭规则对head中两个变量进行查询，查询条件为规则的body部分
   result = this.kb.countDistinctPairs(var1, var2, query.getBody());
} else {
// 获取非封闭规则的固定变量
   String fixedVar = query.getFixedVar();
   if (KB.isVariable(fixedVar)) {
// 对固定变量进行查询，查询条件为规则中的body部分
```

```
        result += this.kb.countDistinct(fixedVar, query.getBody());
    }
}
```

3.2 中文支持

原 AMIE 算法中，对知识库三元组的存储是使用 ByteString（javatools 的一部分），内部使用 byte 数组存储。ByteString 通过构造方法将知识库文件中的 String 类型转化为 ByteString 类型，我们看一下转化过程：

```
/** Use of() */
protected ByteString(CharSequence s) {
data = new byte[s.length()];
for (int i = 0; i < s.length(); i++) {
data[i] = (byte) (s.charAt(i) - 128);
}
    hashCode = Arrays.hashCode(data);
}
```

可以看出，将 2 字节的 char 类型转化为 1 字节的 byte 类型的过程中丢失了信息，这导致了它无法处理中文字符。为了避免这个问题，我们使用支持 unicode 的 String 类型，而不是 ByteString，虽然效率可能会降低一些。

3.3 本章小结

本章详细介绍的规则挖掘系统的实现，包括规则挖掘的流程、不同规则挖掘的实现细节、规则的等价性判断以及规则的置信度的计算。这里重要的一点就是这个规则挖掘系统对非封闭规则提供的支持，能够挖掘出部分非封闭规则。另外，与 AMIE 相比，我们的规则挖掘系统提供的对中文知识库的支持。

第 4 章　项目成果

4.1 测试环境与系统配置

这里我们对规则挖掘系统进行一些测试。

测试环境：

操作系统：CentOS release 6.7 (Final)

CPU：Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz (8 processors)

内存：32G

测试数据：来自 https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/amie/

表 4-2测试用知识库

知识库	Fact 数量
yago2core.10kseedsSample.compressed.notypes	46654
yago2core.10kseedsSample.compressed.notypes_zh_cn	46654
yago2core_facts.clean.notypes	948358
persondata+mappingbased_properties.decoded.compressed.noliterals.en	7035092
persondata+mappingbased_properties.decoded.compressed.noliterals.notypes.en	11024066

4.2 规则挖掘系统测试

规则挖掘前我们需要先设置一些参数，包括知识库文件路径、挖掘规则的最大长度（maxad）、线程数（numThread）、是否支持 Instantiated Atom（const）、是否支持非封闭规则（open）。另外，还需要两个参数，单独对 Instantiated　Atom 与非封闭规则的 minHeadCoverage 进行限制，防止产生大量限制较为宽松的这两类规则。另外，下面的时间是指规则挖掘时间，不包括知识库读取时间。“/”表示超出测试时间。

4.2.1 简单封闭规则挖掘

简单封闭规则指不包含 Instantiated Atom 的封闭规则。即规则中，所有的 Atom 除了 relation 部分是实例化的之外，subject 与 object 都是变量。参数设置：

-maxad2 -numThread 4

测试结果如下：

表 4-2 封闭规则挖掘结果

Dataset	Fact num	Max depth	Rules mined	Time(s)
yago2core.10kseedsSample.compressed.notypes	46654	2	5	0.265
		3	28	2.411
		4	342	8.827
		5	3231	86.253
yago2core_facts.clean.notypes	948358	2	6	1.419
		3	29	25.256
		4	291	568.635
		5	/	/

可以看到，随着规则长度的增加，挖掘时间迅速增加，为了更加直观一点，我们看一第一个数据集的规则数量和挖掘时间在规则长度增长下变化的折线图：

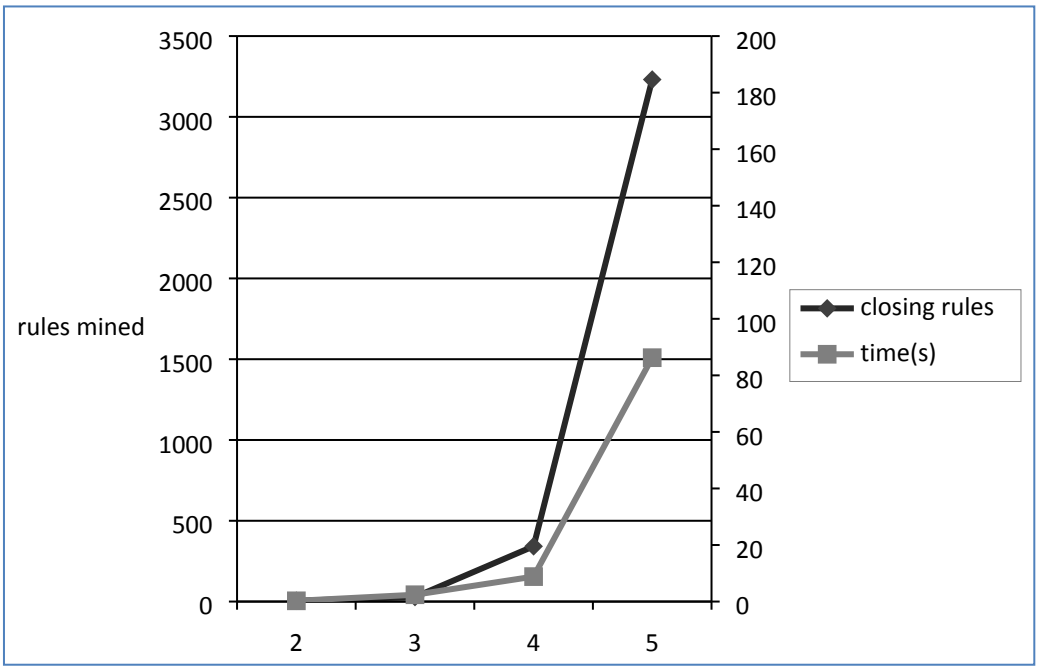

图 4-1封闭规则挖掘结果折线图

从折线图中可以看出，随着规则长度的增加，挖掘时间和挖掘数量都接近指数增长。

4.2.2 包含 Instantiated Atom 的封闭规则挖掘

测试包含 Instantiated Atom 的封闭规则。即规则中，存在至少 2 个 Atom 除了 relation 部分是实例化的之外，subject 或 object 也被实例化。参数设置：

-maxad 2 -numThread 4 -const 5

测试结果如下：

表 4-3 包含 Instantiated Atom 的封闭规则挖掘结果

Dataset	Fact num	Max depth	Closed Rules	Inst. Rules	All Rules	Time(s)
yago2core.10kseedsSample.compressed.notypes	46654	2	5	3	8	1.996
		3	28	307	335	18.468
		4	344	7132	7476	164.406
yago2core_facts.c	94835	2	6	3	9	3.304

lean.notypes	8	3	29	308	337	43.389
		4	289	9060	9349	1581.143

可以看到，增加了 Instantiated Atom 之后，挖掘时间和规则数量都大量增加，我们来看一下第一个数据集的折线图：

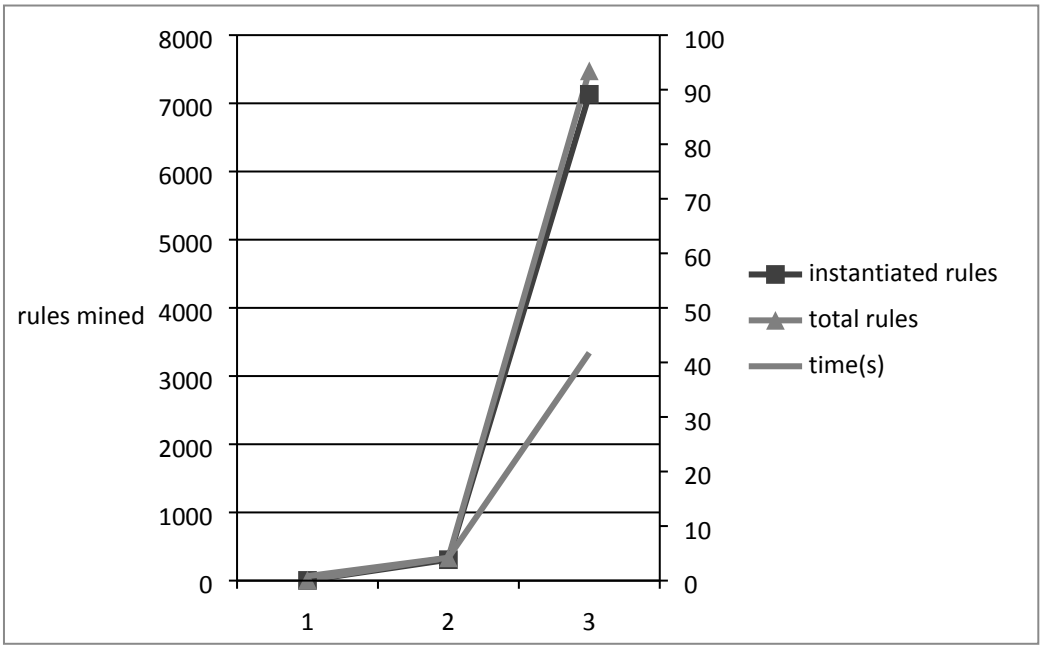

图 4-2 封闭规则挖掘结果折线图（含 Instantiated Atoms）

从图中可以看出，在挖掘出的所有规则中，包含 Instantiated Atoms 的规则占据了绝大部分。挖掘时间也大量消耗在挖掘这类规则中。因此我们在参数 const 中添加了限制，减少 Instantiated 规则的输出，同时也提高规则的质量。

4.2.3 非封闭规则挖掘

简单封闭规则指不包含 Instantiated Atom 的封闭规则。即规则中，所有的 Atom 除了 relation 部分是实例化的之外，subject 与 object 都是变量。这里的测试包括了封闭规则与非封闭规则的挖掘，但不包括 Instantiated Atoms。参数设置：

-maxad 2 -numThread 4 –const 5 –open 20

测试结果如下：

表 4-4 非封闭规则挖掘结果

Dataset	Fact num	Max depth	Closed Rules	Opened Rules	All Rules	Time(s)
yago2core.10kseedsSample.compressed.notypes	46654	2	5	38	43	0.541
		3	28	100	128	2.489
		4	345	165	510	8.865
		5	3231	207	3438	86.764
yago2core_facts.clean.notypes	948358	2	68	23	29	3.116
		3	894	47	76	26.283
		4	293	66	359	566.243
		5	4589	72	4661	16467.04

这里我们将封闭规则挖掘（不含 Instantiated Atoms）与非封闭规则挖掘的数据进行对比：

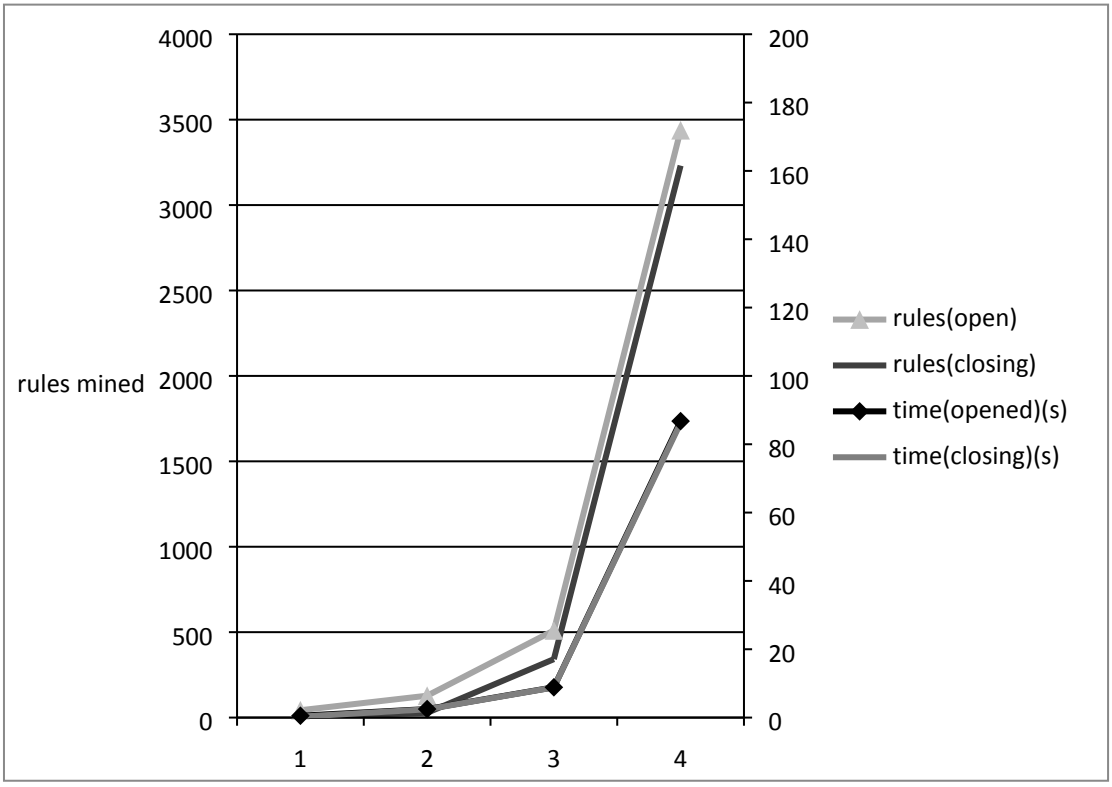

图 4-3 封闭规则（不含 Instantiated Atoms）与非封闭规则挖掘结果对比

分析图表中的数据，可以看出，在规则长度不超过 3 时，非封闭规则与封闭规则数量相差不多，但是在规则长度超过 3 时，规则中封闭规则占了多数。从这一点也可以看出，挖掘非封闭规则所用的时间在总挖掘时间中所占的比例极小，对整体造成效率的影响几乎可以忽略不计。

4.2.4 中文支持测试

使用中英文混合语言知识库

yago2core.10kseedsSample.compressed.notypes_zh_cn

测试结果截图：

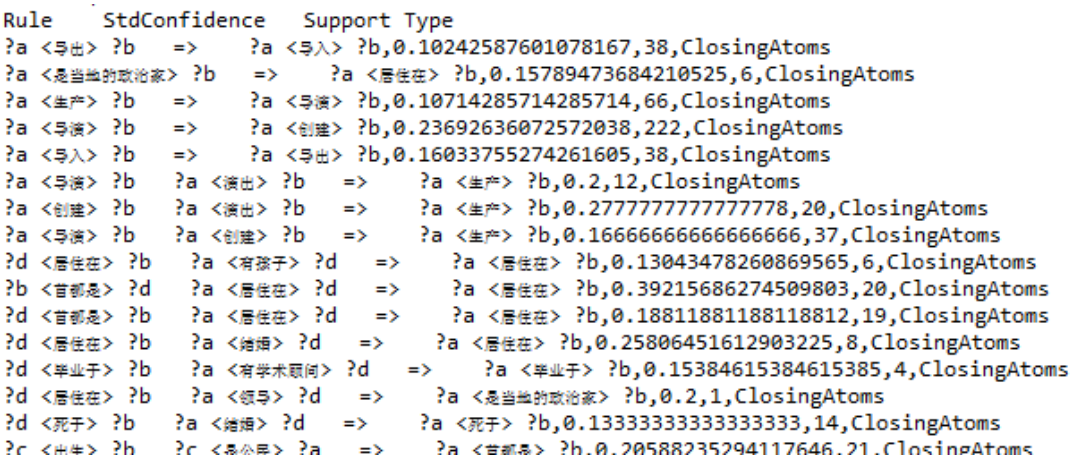

图 4-4 中文测试结果

4.3 项目总结

从以上规则挖掘的结果来看，规则长度不超过 3 时，挖掘时间还是比较短的，但是随着规则长度的增加，时间大量增加。另外，在处理 90w 的知识库时，内存占用大约是 2-3G，这也是限制挖掘规则长度的一个因素。

在挖掘规则类型方面，非封闭规则的挖掘对整体的挖掘速度影响并不大，影响较大的是包含 Instantiated Atom 的规则，数量较多，并且占用了大量的时间。如何减少这个时间占用，提高效率，这是一个重要的问题。

在知识库的规模方面，这里测试使用的是 94 万三元组的知识库，占用内存较多，对于更大的知识库，可能需要分布式处理。

总之，我们的规则挖掘系统实现了对非封闭规则的挖掘，提供了中文的支持，完善了 AMIE 的不足。

4.4 本章小结

本章是对实现的规则挖掘系统的测试，测试了封闭规则与非封闭规则的挖掘效率，并进行了各方面的对比，同时也指出了一些不足，希望在以后能够有机会进行改进。

参考文献

[1] Luis Galárraga, Christina Teflioudi, Katja Hose, Fabian M. Suchanek. AMIE: Association Rule Mining under Incomplete Evidencein Ontological Knowledge Base. Max-Planck Institute for Informatics, Saarbrücken, Germany, Aalborg University, Aalborg, Denmark. 2013

[2] Luis Galarraga, Christina Teioudi, Katja Hose, Fabian M.Suchanek. Fast Rule Mining in Ontological Knowledge Bases with AMIE+.Vldb Journal. 2015, 24(6):707-730

[3] Bhushan Kotnis, Pradeep Bansal, Partha Talukdar. Knowledge Base Inference using Bridging Entities. Indian Institute of Science.2015

[4]Ni Lao, Tom Mitchell, William W. Cohen. Random Walk Inference and Learning in A Large Scale Knowledge Base. Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing. 2011, 529-539

[5] Ziawasch Abedjan, FelixNaumann. Improving RDF Data Through Association RuleMining. Springer-Verlag Berlin Heidelberg. 2013

[6] K.Karthikeyan, Dr.V.Karthikeyani. Association Rule Mining Based Extraction of Semantic Relations Using Markov Logic Network. International Journal of Web & Semantic Technology (IJWesT) Vol.5, No.4. 2014

[7] Chen Liang, Kenneth D.Forbus. Learning Plausible Inferences from Semantic Web Knowledge by Combining Analogical Generalization with Structured Logistic Regression. Twenty-Ninth AAAI Conference on Artificial Intelligence. 2015

[8] 何月顺.关联规则挖掘技术的研究及应用. 南京航空航天大学. 2010

[9] 王爱平, 王占凤, 陶嗣干, 燕飞飞. 数据挖掘中常用关联规则挖掘算法计算机技术与发展. 2010

[10] 李锋, 商慧亮. 有向图的同构判定算法出入度序列法. 应用科学学报. 2002

致谢

这次的毕业设计是在我的导师陈华钧老师的指导下完成的，从毕业选题到设计完成，陈老师都给予了我许多指导与帮助。这里我对陈老师表示由衷地感谢。

感谢陈曦师兄，在整个毕业设计过程中，陈曦师兄一直悉心指导我，帮助我解决各种困难。在陈曦师兄的帮助下，这次的毕业设计才能够顺利完成。

感谢所有在我遇到困难的时候帮助我的同学们、朋友们，你们对我的鼓励与支持，是我前进的最大动力。

本科生毕业论文（设计）任务书

一、题目：基于大规模知识图谱的规则挖掘系统的实现

二、指导教师对毕业论文（设计）的进度安排及任务要求：

围绕规则挖掘，研究相关的技术成果，并且学习 AMIE 规则挖掘算法，完成规定数量的中英文文献阅读，并在此基础上完成 3000 字以上的文献翻译和 3500 字以上的开题报告。

实现知识库规则挖掘系统，实现非封闭规则的挖掘。

起讫日期 200 年 月 日至 200 年 月 日

指导教师（签名）＿＿职称＿＿

三、系或研究所审核意见:

负责人（签名）

年 月 日

毕 业 论 文（设计） 考 核

一、指导教师对毕业论文（设计）的评语：

指导教师(签名）

年 月 日

二、答辩小组对毕业论文（设计）的答辩评语及总评成绩：

成绩比例	中期报告占（10%）	开题报告占（20%）	外文翻译占（10%）	毕业论文（设计）质量及答辩占（60%）	总评成绩
分值					

答辩小组负责人（签名）

年 月 日