A Java implementation of the RS1 algorithm using SQL

Robert H. Warren
warren@cs.uregina.ca

Julia A. Johnson
julia@cs.laurentian.ca

Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2

Technical Report TR-2000-03
ISBN 0-7731-0399-6

Abstract

This paper describes a Java implementation of the RS1 Rough Sets algorithm, that leverages the use of a Data Base Management System (DBMS) with Structured Query Language (SQL). DBMS use ensures that large information tables can be processed by the algorithm, while keeping the computational resource needs of the Java class low.
The algorithm is implemented within a single Java class, making it ideal not only for Rough Set research, but as an add-on to non Rough Set projects.

Keywords Java, Rough Set Implementation, Database Management System

1 Introduction

RS1 is a Rough Sets induction algorithm developed by Wong, Ziarko and Ye in 1986 for generating decision rules based on a table of inconsistent information.[1] A Java implementation of the RS1 algorithm is described. The algorithm logic was written in the Java language, while the actual data manipulation was performed through the Java DataBase Connectivity (JDBC) package. The relational database used to manage the actual data set was the open source Postgresql database package which supports the SQL query language. This Java implementation is being developed in the context of a diversity of applications of Rough Sets techniques for dealing with inconsistent and incomplete knowledge bases.[8, 6, 5, 4, 3, 2, 7]

2 Objective

The objective of this research is the implementation of an inductive Rough Sets algorithm to serve as a kernel from which additional Rough Sets research can be performed. While much work has been done, commercially available implementations of RS algorithms are few. Most implementations are experimental and monolithic. (eg: In the case of Rosetta[9], the implementation is a closed package who's capabilities cannot be extended.) In this implementation, the processing is done within a single class file, thus making it relatively simple to integrate into other applications.

Rough Sets is based on set theory and requires vast amounts of set operations which DBMS are ideally suited to perform. Traditionally, these set operations were performed locally within the algorithm development environment. This ensured a localized treatment of the set information, sometimes at the expense of resource efficiency, as the mechanics of data manipulation took second place to the primary objective of implementing Rough Sets.

The novelty of our work lies in the use of a DBMS to perform the data processing functions of the Rough Sets algorithm. In most implementations[9], DBMS use is limited to the import or export of data to and from the Rough Set implementation. This is unfortunate, because DBMSs are designed to handle large volumes of data and efficiently manipulate them based on both queries and constraints. Because the DBMS access is done through the JDBC package, the overhead of accessing information tables is kept at a minimum while ensuring that a maximum number of data sources can be used. This research is meant as a proof of concept as to the use of Data Base Management Systems (DBMS) with Rough Set algorithms.

3 RS1 algorithm description

The RS1 algorithm is an algorithm that functions by incrementally selecting a series of attributes around which to "pivot", generating rule sets of increasing complexity until all examples in the universe are covered.

At first, each attribute (A_i) is individually processed, and for each of it's possible values (V_{ij}), a subset (S_{ij}) of the universe (E) is generated(1). Each of these subsets can be part of the Upper Bound($\overline{Y}$), the Lower Bound ($\underline{Y}$) or not part of anything.

$$\sum_{i=1}^{m}\sum_{j=1}^{n_i} S_{ij} = subset(E, A_i = V_{ij}) \tag{1}$$

The set of all positive class examples is generated as a subset (S_+), and the attribute subset (S_{ij}) is part of the Lower Bound if it intersects with this class subset(3). Likewise, an attribute subset (S_{ij}) is part of the Upper Bound if it is included within this class subset(2).

$$S_{ij} \subseteq \overline{Y} \iff (S_{ij} \cap S_+) \tag{2}$$

$$S_{ij} \subseteq \underline{Y} \iff (S_{ij} \subseteq S_+) \tag{3}$$

Then a quality value represented by α is generated for each attribute (4). The attribute with the largest value of α then becomes the pivot attribute for the next iteration. The universe of possible elements is cleared of rows that are already covered by the rule set using the equation (5).

$$\alpha = 1 - \frac{|\overline{Y} - \underline{Y}|}{|E|} \tag{4}$$

$$E = E - [(E - \overline{Y}) \cup \underline{Y}] \tag{5}$$

Using the pivot attribute, the list of attribute is traversed again and new subsets are generated for each of the value combinations for pivot and attribute. The Lower and Upper bounds are again generated and the attribute with the best α is joined to the pivot, so that we now have a two attribute pivot.

The process is repeated again, adding attributes to the pivot, until we either run out of attributes or the universe becomes empty.

4 Implementation description

The implementation in Java is based on a slightly modified RS1 algorithm. The implementation is a restriction of the RS1 algorithm as only one positive decision class is currently supported and an unique identifying attribute is needed. In order to optimize for a large, real-world, application a DBMS was used to store and retrieve the the set information using SQL queries. The source code is not reviewed here for space considerations; instead the implementation of set operations using SQL is examined as this forms the cornerstone of the implementation.

4.1 A rational for the use of SQL

Data Base Management Systems (DBMS) are fairly mature, robust and standardized systems. They are able to store and process large amounts of tabular data through the use of Structured Query Language (SQL).

The use of a DBMS greatly reduces the amount of code required to implement the RS1 algorithm because it allows the actual mechanics of information manipulation to be dispatched to the DBMS. Instead of fine-tuning the algorithm to the particulars of file I/O, we can rely on the engineering embedded within the DBMS to self-optimize the operations required to implement the RS1 algorithm.

Furthermore, offloading the table operations to the DBMS ensures that the memory consumption by the Rough Set algorithm will be low. Only the table and view names need to be kept in local memory by the Java class, the heavy I/O operations being handled by the DBMS. Most of these have built-in memory space management and internal query caching and optimization. This shelters the Java class from design decisions that are out of the scope of the Rough Sets abstraction, such as selecting an internal set storage method.

Finally, SQL is a sufficiently powerful language to support most set operations needed by a Rough Set algorithm including the subset of, intersection of and union of functions. It is relatively trivial to code these operations because SQL frees us from array and object-space considerations. The generation of subsets is achieved through the generation of temporary tables or views which can be discarded to reduce storage space utilization.

4.2 Implementing set operations using SQL

Within the DBMS, an element of a set is represented as a row within a table and SQL queries are used to manipulate the set elements as desired. The universe is represented by a master table that contains the data that is to be processed by RS1.

In the algorithm described in Section 3, two basic types of operations need to be performed. The generation of sub-sets based on attribute value constraints and the set operators $\cap$ and $\subseteq$.

4.2.1 Generating sub-sets:

In order to generate the subsets needed in (1), the possible values of all attributes must be known. To do this, we use the SQL query listed in Example 1, from which we can obtain the possible values for an attribute. This is repeated for each attribute, enabling the RoughSet class to generate all possible value combinations that need to be verified.

Example 1 *SELECT DISTINCT ATTRIB FROM TABLE*

To generate the sets we could use nested sub-queries. However, some SQL implementations only have limited support for nested sub-queries, which would

make portability an issue.

Instead, sub-sets can be generated from the data using either views or tables. Generating the subset using a table means creating a separate table to which rows are copied (Examples 2 and 3). This takes up disk space, and the time needed to copy the record. Because a view is table that is actually a query of another table, no additional disk space is needed(Example 4). However, a performance penalty occurs because a query is run internally by the DBMS.

Example 2 *CREATE TABLE TMP3976 () INHERITS (MAINTABLE)*

Example 3 *INSERT INTO TMP3976 SELECT * from MAINTABLE WHERE eyes='Blue' AND hair='Red'*

Example 4 *CREATE VIEW TMP3976 AS SELECT * FROM MAINTABLE WHERE eyes='Blue' AND hair='Red'*

4.2.2 Coding set functions:

After the sub-sets have been generated, both the intersection function and the inclusion functions need to be implemented in order to determine if (3) or (2) occur.

In the case of (2), the result needed is the presence of data in the intersection between two sets. This is implemented in Example 5 where the number of elements within the intersection are counted and returned to the Java class.

Example 5 *SELECT COUNT(ITEM) FROM TABLE1 WHERE ITEM IN (SELECT ITEM FROM TABLE2)*

A variation of this query is used in Example 6 to implement (3). In order for TABLE1 to be included in TABLE2, all the elements from it must be part of TABLE2. Therefore, the count returned from the SQL query most be 0.

Example 6 *SELECT COUNT(ITEM) FROM TABLE1 WHERE ITEM NOT IN (SELECT ITEM FROM TABLE2)*

5 Testing of algorithm

The implementation was tested on two sample data sets and the output compared with hand-derived expected results. The two data sets were those presented in [1] which have been reproduced in tables 1 and 2. The output results are provided in the remainder of this section. A partial trace of the algorithm for the data pesented in table 1 is provided in Appendix A.

item	height	hair	eyes	class
1	Short	Dark	Blue	-
2	Tall	Dark	Blue	-
3	Tall	Dark	Brown	-
4	Tall	Red	Blue	+
5	Short	Blond	Blue	+
6	Tall	Blond	Brown	-
7	Tall	Blond	Blue	+
8	Short	Blond	Brown	-

Table 1: Test Data Set 1

id	weight	sex	class
E1	Heavy	F	+
E2	Heavy	M	+
E3	Medium	M	-
E4	Medium	F	-
E5	Light	M	+
E6	Light	M	+
E7	Light	F	+
E8	Light	F	-
E9	Light	M	-
E10	Light	F	-

Table 2: Test Data Set 2

height='Short' → class='+' covers 3 row(s) (1 positive row(s)).
height='Tall' → class='+' covers 5 row(s) (2 positive row(s)).
hair='Blond' → class='+' covers 4 row(s) (2 positive row(s)).
eyes='Blue' → class='+' covers 5 row(s) (3 positive row(s)).

Table 3: Upper bound rules for Table 1.

5.1 Results after processing table 1

The following results were consistent with the hand-derived results for the information contained in table 1.

5.2 Results after processing table 2

The following results were consistent with the hand-derived results for the information contained in table 2.

6 Conclusion

The implementation of the RS1 Rough Sets Inductive Algorithm with a Database Management System was successful. The expected results presented in [1] cor-

hair='Blond' ∧ eyes='Blue' → class='+' covers 2 positive row(s).
hair='Red' → class='+' covers 1 postive row(s).

Table 4: Lower bound rules for Table 1, without the Upper bound.

weight='Light' → class='+' covers 6 row(s) (3 positive row(s)).
sex='F' → class='+' covers 5 row(s) (2 positive row(s)).
sex='M' → class='+' covers 5 row(s) (3 positive row(s)).

Table 5: Upper bound rules for Table 2.

weight='Heavy' ∧ sex='F' → class='+' covers 1 positive row(s).
weight='Heavy' ∧ sex='M'→ class='+' covers 1 positive row(s).

Table 6: Lower bound rules for Table 2, without the Upper bound.

responded to those returned by our RS1 Java implementation, and are listed in Sections 5.1 and 5.2.

A Sample program trace

/*** Initialise ***/

```
RoughSet: Connecting to database.
RoughSet: Loaded column:[item, height, hair, eyes, class, ]
RoughSet: Checking for Class Column [class].
RoughSet: Checking for ID Column [item].
```

/*** Generate a sub-set of all positive class rows. ***/

```
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP7574 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP7574 SELECT
              * FROM roughtest WHERE class='+'].
createSubSet: Inserted [3] rows into table.
```

/*** Find first pivot. ***/

```
generateRules: Working on Attribute [height].
generateRules: Using table [TMP3769] to store Lower Bound.
generateRules: Using table [TMP1806] to store Upper Bound.
```

/*** For each possible value of the attribute, generate a subset of rows. ***/
/*** For each subset decide if it belong to the upper or lower bound. ***/

```
generateRules: Working on Attribute [height] with value [Short].
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP1311 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP1311 SELECT
```

```
              * FROM roughtest WHERE height='Short'].
createSubSet: Inserted [3] rows into table.
generateRules: Working on Attribute [height] with value [Tall].
createSubSet: Will write out a subset to table [TMP2118].
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP2118 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP2118 SELECT
              * FROM roughtest WHERE height='Tall'].
createSubSet: Inserted [5] rows into table.
```

/*** Generate alpha based on upper and lower bound. ***/

```
generateRules: Attribute [height] has an alpha value of [0.0].
generateRules: This Alpha of [0.0] is better than best Alpha of [-1.0].
```

/*** Try with next attribute. ***/

```
generateRules: Working on Attribute [hair].
generateRules: Using table [TMP6874] to store Lower Bound.
generateRules: Using table [TMP2093] to store Upper Bound.
```

/*** For each possible value of the attribute, generate a subset of rows. ***/
/*** For each subset decide if it belongs to the upper or lower bound. ***/

```
generateRules: Working on Attribute [hair] with value [Blond].
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP1991 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP1991 SELECT
              * FROM roughtest WHERE hair='Blond'].
createSubSet: Inserted [4] rows into table.
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP1866 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP1866 SELECT
              * FROM roughtest WHERE hair='Dark'].
createSubSet: Inserted [3] rows into table.
generateRules: Working on Attribute [hair] with value [Red].
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP7007 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP7007 SELECT
              * FROM roughtest WHERE hair='Red'].
createSubSet: Inserted [1] rows into table.
```

/*** Generate alpha based on upper and lower bound. ***/

```
generateRules: Attribute [hair] has an alpha value of [0.5].
generateRules: This Alpha of [0.5] is better than best Alpha of [0.0].
```

/*** Try with next attribute. ***/

```
generateRules: Working on Attribute [eyes].
generateRules: Using table [TMP2053] to store Lower Bound.
generateRules: Using table [TMP1910] to store Upper Bound.
```

/*** For each possible value of the attribute, generate a subset of rows. ***/
/*** For each subset decide if it belong to the upper or lower bound. ***/

```
generateRules: Working on Attribute [eyes] with value [Blue].
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP9680 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP9680 SELECT
              * FROM roughtest WHERE eyes='Blue'].
createSubSet: Inserted [5] rows into table.
generateRules: Working on Attribute [eyes] with value [Brown].
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP1701 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP1701 SELECT
              * FROM roughtest WHERE eyes='Brown'].
createSubSet: Inserted [3] rows into table.
```

/*** Generate alpha based on upper and lower bound. ***/

```
generateRules: Attribute [eyes] has an alpha value of [0.375].
```

/*** End of first iteration, dump the pivot attribute's rules ***/
/*** to the results table and prune the universe data table. ***/

```
generateRules: Best Attribute was [hair] with an Alpha of [0.5],
               storing lower bound and recursing.
storeRules: Storing lower bound rule(s) over [1] columns from table
            [TMP6874] to table [RULEL].
storeRules: Will send [SELECT DISTINCT  hair, class  FROM TMP6874]
            as SQL Soup.
storeRules: Will send [INSERT INTO RULEL ( hair, class) VALUES ( 'Red', '+')]
            as SQL Soup.
storeRules: Storing upper bound rule(s) over [1] columns from table
            [TMP6874] to table [RULEH].
storeRules: Will send [SELECT DISTINCT  hair, class  FROM TMP6874]
            as SQL Soup.
storeRules: Will send [INSERT INTO RULEH ( hair, class) VALUES ( 'Red', '+')]
            as SQL Soup.
storeRules: Storing upper bound rule(s) over [1] columns from table
            [TMP2093] to table [RULEH].
storeRules: Will send [SELECT DISTINCT  hair, class  FROM TMP2093]
            as SQL Soup.
storeRules: Will send [INSERT INTO RULEH ( hair, class) VALUES ( 'Blond', '+')]
            as SQL Soup.
storeRules: Will send [INSERT INTO RULEH ( hair, class) VALUES ( 'Blond', '-')]
```

```
            as SQL Soup.
storeRules: Will send [INSERT INTO RULEH ( hair, class) VALUES ( 'Red', '+')]
            as SQL Soup.
createSubSet: Creating table with SQL SOUP [CREATE TABLE TMP4285 () INHERITS
              (roughtest)].
createSubSet: Will insert rows using SQL SOUP [INSERT INTO TMP4285 SELECT
              * FROM roughtest WHERE class='+'].
createSubSet: Inserted [3] rows into table.
```

/*** Generate a sub-set of all positive class rows. ***/

```
generateRules: Created table [TMP4285] as positive reference class.
generateRules: Pivoting on at least [hair].
```

/*** Using hair as a pivot, repeat process for ***/
/*** both left-over attributes and pick best alpha. ***/

```
generateRules: Working on Attribute [height].
(...)
generateRules: Attribute [height] has an alpha value of [0.5].
generateRules: This Alpha of [0.5] is better than best Alpha of [-1.0].
(...)
generateRules: Attribute [eyes] has an alpha value of [1.0].
generateRules: This Alpha of [1.0] is better than best Alpha of [0.5].
generateRules: Best Attribute was [eyes] with an Alpha of [1.0],
               storing lower bound and recursing.
```

/*** Second best attribute is eyes, dump the pivot ***/
/*** attribute's to the results table and prune the Universe data table.***/

```
generateRules: No rows left in universe, exit.
*** Finished processing ***
```

References

[1] Wong, S. K. M., Ziarko, W., Ye, R. L., "Comparison of rough-set and statistical methods in inductive learning", International Journal of Man-Machine Studies (1986) 24, pp. 53-72

[2] J.A. Johnson and H. Li. Rough set approach for deadlock detection in Petri nets. In *Proc. International Conference on Artificial Intelligence IC-AI*, 2000. Las Vegas, Nevada, USA, June.

[3] J. Johnson and M. Liu. The language REFINE. In *Proc. 4^{th} Joint Conference in Information Science JCIS'98*, volume II, pages 367–370, 1998.

[4] J. Johnson and M. Liu. Rough sets for informative question answering. *Journal of Computing and Information*, 3(1), 1998. Available on-line http://www.jci.trentu.ca/jci/vol.3.

[5] A. Liang, M. Maguire, and J.A. Johnson. Rough set based Webct learning. In *Proc. First International Conference on Web-Age Information Management WIAM*, 2000. Shanghai, China, June.

[6] J. Johnson. Rough mereology for industrial design. In L. Polkowski and A. Skowron, editors, *Rough Sets and Current Trends in Computing*, pages 553–556. Springer, 1998.

[7] J.A. Johnson. Rough scheduling. In *Proc. 5^{th} Joint Conference in Information Science JCIS 2000*, volume 1, pages 162–165, 2000.

[8] J.A. Johnson, X. D. Yang, and Q. Hu. Word sense disambiguation in the rough. In *Proc. Fourth Symposium on Natural Language Processing SNLP*, 2000. Chiangmai, Thailand, May.

[9] http://www.idt.unit.no/ aleks/rosetta/features.htm