Reduce And prIoritize SuitEs: A Tutorial

This tutorial will assist the user in acquiring, installing, and using the RAISE package. Section sec:acq explains how to get the project. Section 2 shows how to set up the package on your machine so that it may be used in your source code. Section 3 presents the necessary steps for using the reduction and prioritization algorithms including data structures, source code examples, and evaluation of the results.

1 Acquisition

SVN Checkout: The RAISE sourcecode is hosted by Google in an svn repository. You may obtain an anonymous checkout from the subversion repository with the following command.

```
svn checkout http://raise.googlecode.com/svn/ <destination-folder>
```

2 Installation

Depenencies: There are two requirements that must be fulfilled before the RAISE package can be built.

- RAISE requires at least java version 1.5.
- The source code comes with an ANT build.xml file, so ANT (ant.apache.org) must be installed in order to build the package using the ANT build file.

Compile: All ANT commands must be executed from the root directory of the project. The root directory contains the 'src', 'lib', 'build', 'doc', and 'data' directories.

To compile the source code enter the following code in the command line.

```
ant compile
```

After compilation, you can execute the RAISE test suite with the following command.

```
ant testSetCover
```

Install: To use the package from anywhere on your machine, add the full path to the raise/src directory to your classpath environment variable. Also you will need to add the raise/lib/xstream-1.1.3.jar file to your classpath.

1	0	0	1	2
0	1	0	1	2
1	0	1	0	2
2	1	1	2	0

Figure 1: An example coverage matrix.

Name	Time
1	4.0
2	1.0
3	2.0
4	3.0

Figure 2: An example time file.

3 Using the Package

3.1 Creating a Test Suite Representation

The RAISE package reduces and prioritizes test suites. A test suite is represented by the SetCover object. In your source code create a new test suite with the line,

```
SetCover sc = new SetCover()
```

Test suites are stored on disk as a combination of coverage matrix and time information files. The coverage matrix files contain a binary matrix whose columns represent tests and rows represent requirements. A 1 at row `i` and column `j` indicates that test `j` covers requirement `i`. The last row and column of each file contains the summary data for that row or column. The number in the last column of each row represents the total number of tests that cover the requirement represented by the row. The number in the last row of each column represents the total number of requirements that the test represented by the column covers. Figure 1 shows an example coverage matrix file. In this example the first column represents a test that covers the first and third requirements, but not the second. The fact that the test covers two requirements is represented in the last row for the column. The first row shows a requirement that is covered by the first and fourth tests, but not the second and third. The summary column shows that the requirement is covered by two tests.

Each matrix file is accompanied by a file that stores the timing information for each test. The first column contains the heading 'Name' and then the index of each test. The second column first has the heading 'Time' and then a time value for each test. In Figure 2, test 1 executes for 4.0 seconds, test 2 executes for 1.0 seconds, test 3 executes for 2.0 seconds, and test 4 executes for 3.0 seconds.

You may create your own matrix and time files or you can experiment with the files in the data directory. Build the SetCover object using the static method shown below. The first parameter is the string path to the matrix file and the second is the string path to the time file. The method returns a SetCover object that is defined by those files.

```
sc = SetCover.constructSetCoverFromMatrix(String matrixFile, String
                                          timeFile)
```

3.2 Reducing and Prioritizing:

After the SetCover object has been created, there are 4 different algorithms to choose from that perfrom reduction and prioritization. Each algorithm takes a String paramter that represents the greedy choice metric to be used. This can be `ratio`. `time`, or `coverage`. The Harrold Gupta Soffa algorithm also takes an integer `numberOfLooksAhead` which is used to break ties in the algorithm. For complete explanations of each algorithm, please consult the SAC 2009 paper in the doc directory.

- Harrold Gupta Soffa
  - `reduceUsingHarroldGuptaSoffa(String metric, int numberOfLooksAhead)`
  - `prioritizeUsingHarroldGuptaSoffa(String metric, int numberOfLooksAhead)`
- Delayed Greedy
  - `reduceUsingDelayedGreedy(String metric)`
  - `prioritizeUsingDelayedGreedy(String metric)`
- Greedy
  - `reduceUsingGreedy(String metric)`
  - `prioritizeUsingGreedy(String metric)`
- 2-Optimal
  - `reduceUsing2Optimal(String metric)`
  - `prioritizeUsing2Optimal(String metric)`

So for example, to use the Harrold Gupta Soffa algorithm for reduction with the `ratio` greedy choice metric with 3 level look ahead on a SetCover `sc` you would use the statement,

```
sc.reduceUsingHarroldGuptaSoffa("ratio",3);
```

4 Evaluating Results

Currently, the reduction and prioritization algorithms destroy the SetCover objects when they are executed. To compare the results to the original test suite, the values that will be compared must be saved before the reduction or prioritization algorithm is run, or the `restoreSetCover()` class method can be called to restore the SetCover in such a way that the data can be retreived. This restoration does not garauntee that another algorithm may be run succesfully. In order to execute more reduction and prioritization techniques, the SetCover must be fully reconstructed from the matrix and time files.

Evaluating Reduction: After the SetCover has been reduced, a LinkedHashSet of the new test suite can be obtained using the SetCover class method `getCoverPickSets()`. This collection of tests is garaunteed to cover all of the requirements that were covered by the original test suite. The size of the LinkedHashSet object can be retrieved using

`<linkedHashSetInstance>.size()`. This can be compared to the size of the original test suite size which can be obtained (either before the algorithm is run, or after restoring the SetCover) by using `<SetCoverInstance>.getTestSubsets().size()`.

Evaluating Prioritization: To evaluate the effectiveness of a prioritization, RAISE uses coverage effectiveness (CE) (See the SAC2009 paper for details). The SetCover class method `getCE(int[] order)` returns the CE of the prioritized test suite. The int array parameter represents the order of the new test suite. So the integer value at `order[0]` is the index of the test to be run first, the integer value at `order[1]` is the test to be run second, and so on. The order array for the initial order can be obtained using the line `SetCover.getIndecesFromSingleTestSubsetList(<SetCoverInstance>.getTestSubsets())` before the SetCover has been prioritized. After the SetCover has been restored, the statement `SetCover.getIndecesFromSingleTestList(<SetCoverInstance>.getPrioritizedSets())` will give the order array for the prioritized test suite.

5 Example

Listing 5 creates a SetCover, stores qualities of the SetCover for comparison, reduces, prioritizes, and evalutates the results of reduction and prioritiziation. To compile and use this code, you must change the paths to the matrix and time files so that they are correct for your installation.

```
// Import the package
import raise.reduce.*;

// Import LinkedHashSet
import java.util.LinkedHashSet;

public class test
{
  public static void main(String[] args)
  {
    // Store the paths to the matrix and time files.
    String matrix = "/home/adam/raise-read-only/data/raise/reduce/setCovers/RPMatrix.dat";
    String time = "/home/adam/raise-read-only/data/raise/reduce/setCovers/RPTime.dat";

    // Create the SetCover object
    SetCover sc = new SetCover();

    // Build the SetCover object
    sc = SetCover.constructSetCoverFromMatrix(matrix,time);

    // Get the original set of tests to compare to the
    // reduced set.
    LinkedHashSet<SingleTestSubset> tests = sc.getTestSubsets();

    // This integer represents the original size of the test suite.
    int originalSize = tests.size();

    // This array holds the original order of the test suite
    int[] originalOrder = SetCover.getIndicesFromSingleTestSubsetList(tests);

    // Get the original CE
    float originalCE = sc.getCE(originalOrder);
```

```
    // Reduce the test suite using the Greedy algorithm
    sc.reduceUsingGreedy("ratio");

    // This variable holds the new size of the test suite
    int newSize = sc.getCoverPickSets().size();

    // Rebuild the test suite for prioritization
    sc = SetCover.constructSetCoverFromMatrix(matrix,time);

    // Save the SetCover
    sc.savePristeneCopyByteArray();

    // Prioritize using the Greedy algorithm
    sc.prioritizeUsingGreedy("ratio");

    // Restore the SetCover so CE can be calculated
    sc.restoreSetCover();

    // This array represents the new order
    int[] newOrder = SetCover.getIndicesFromSingleTestList(sc.getPrioritizedSets());

    // Calculate the new CE.
    float newCE = sc.getCE(newOrder);

    // Print the results
    System.out.println("Original Size:\t"+originalSize +
      "\nReduced Size:\t"+newSize+"\nOriginal CE:\t"
      +originalCE+"\nPrioritized CE:\t"+newCE);
  }
}
```

The output of this class is

```
Original Size: 66
Reduced Size: 23
Original CE: 0.72134763
Prioritized CE: 0.97047114
```

Email comments, questions, and concerns to Adam at ams292@cs.pitt.edu.