J. Luke Scott

Selected Topics in Artificial Intelligence
MiCS Program
University of Luxembourg

Final Project
Friday 18th January, 2013

A Java Implementation of Cumulative Measures

This document describes a query and revision library for Cumulative Measures, as described in [5].

The library supports a number of useful features, most importantly:

- The ability to add arbitrary propositions to a knowledge base
- Revision by plain belief, static value, or by comparative value (by stating a desired relationship between two propositions)
- Querying the knowledge base to retrieve a set of permutations of propositions
- Querying the knowledge base for a total valuation of the results which match the query.

This library has been successfully tested using various use cases from the literature (see Section 1.4 for more information).

1 Functionality

The library implements a very intuitive Cumulative Measure knowledge base API which allows adding propositions, revising, and querying. The core component of the API is a `CumulativeMeasure` object, which contains the knowledge base data as well as an index which is used internally for querying.

After instantiating a `CumulativeMeasure` object, it then becomes the central point for command execution. Furthermore, for more fine-grained control of behavior, or to do more complex logic outside the knowledge base, the `Value`, `Query`, and `Constraint` objects can be used individually.

Another core object in the library is a `BeliefPermutation`. This object represents a unique combination of true/false assertions for each proposition in the knowledge base. Most operations in the library will perform logic over sets of `BeliefPermutations`.

1.1 Command Overview

There are primarily four commands which constitute the `CumulativeMeasure` functionality. In the examples here, I assume a `CumulativeMeasure` object named `cumulative`.

New propositions can be added to the knowledge base using `CumulativeMeasure.add()`. For instance, to add a proposition named `A`, one would call the command:

```
cumulative.add("A")
```

The strength of belief for new propositions added will be divided evenly between `true` and `false`.

`CumulativeMeasure.query()` is used for returning belief permutations from the knowledge base by passing in a query string, for example:

```
Set<BeliefPermutation> results =
    cumulative.query("a OR b");
```

would return all belief permutations where either `a` or `b` are true.

`CumulativeMeasure.getValue()` is similar to

`CumulativeMeasure.query()`, except that it returns the total value of the matching permutations. For instance:

```
Value v =
    cumulative.getValue("a OR b");
```

would return the total value of all belief permutations where either `a` or `b` are true.

`CumulativeMeasure.revise()` is used for modifying the belief values stored in the knowledge base. It modifies the knowledge base in-place and does not return any result. For example:

```
cumulative.revise("a >= (0, 0.6)");
```

would revise the knowledge base such that all belief permutations where `a` is `true` will add up to the value `(0, 0.6)` or greater.

1.2 Query Syntax

Queries can be divided into two categories: those that are used to *retrieve* sets of `BeliefPermutation` objects or total values thereof, and those that are used during revision to specify *constraints* over sets of these objects.

A *retrieval* query has only one part: the *query string*. This is a string in one of the following forms:

```
a AND -b [AND c [AND ...]]
 -a OR b [OR c [OR ...]]
    a AND (-b OR -c)
     a OR (b AND c).
```

Note that the - symbol can be used for negation. Also note that in last two forms, nested phrases can recursively contain other nested phrases.

To avoid ambiguity about the grouping of terms, at any single level of recursion only one of `AND` or `OR` can be used, but not both. This is to avoid arbitrary interpretations of a query such as: `a AND b OR c`.

A *constraint* query has three parts: a *query string*, an *operator*, and a *value string*. A constraint query might look like one of the following:

```
a = (1, 0.5)
a AND b <= c OR (d AND e).
```

The *query string* part is the same as above, and determines which belief permutations will be considered for revision.

The *operator* part is one of `<=`, `>=`, or `=`, and indicates how the value of the matched permutations should be limited.

Finally, the *value string* part is a phrase that evaluates to a Cumulative Measure valuation with a global rank and local probability. This can be either a query string (which will retrieve results as in a normal query, and then calculate the sum of those results) or a value as written directly in the form `(rank, probability)`. Ranks must be integers, but probabilities can be a decimal value. Probabilities in rank 0 must be in the range [0,1].

1.3 Setting Up Eclipse

To use this library, you will need to have a recent version of Eclipse installed which contains the Java libraries as well as JUnit 4.0. The easiest way to get this is to download the pre-configured "Eclipse IDE for Java Developers" at `http://www.eclipse.org/downloads/`.

The code will be delivered as a zip archive of a fully functional Eclipse project, which can be simply extracted and opened in Eclipse. To open the project in Eclipse, within the **File** menu select **Import...**. Expand the **General** folder and select **Existing Projects into Workspace**. Browse to the folder on your

computer where you extracted the archive and click OK.

Once the project is opened, find the file `AllTests.java` (under `src/test`). Right-click on it and select **Run As...** > **JUnit**. If all the tests pass, then that means the project has been set up correctly and everything works.

To start experimenting with the code, open the file `SandboxTests.java` (which is also under `src/test`), and start writing your own tests. Re-run `AllTests.java` at any point to see the effects of the tests you write.

Continue to Section 1.4 to understand how the existing code and test cases work, or jump ahead to Section 2 for more information on writing your own tests.

1.4 Test Cases

The following is a code excerpt showing a common use case in the unit tests included with the library:

```
cumulative.add("a");
cumulative.add("b");

assertTrue(value("a", 0, 0.5));
assertTrue(value("a AND b", 0, 0.25));
assertTrue(value("-a AND -b", 0, 0.25));

cumulative.revise("a >= (0, 0.6)");

assertTrue(value("a", 0, 0.6));
assertTrue(value("a AND b", 0, 0.3));
assertTrue(value("-a AND -b", 0, 0.2));
```

First, a cumulative measure is populated with propositions (`a` and `b` in this case), and some initial assertions are made about the initial belief values of the propositions.

The initial assertions work as a sanity check to ensure that the test is set up as the developer expects, and also serve to very clearly document what is being tested.

Then, a revision is made to the cumulative measure, and finally, more assertions are made to verify the revision had the intended effect. Someone reading the test can easily see a transition from a starting state to a modified state by one single revision command.

The following test cases are included with the library to demonstrate its functionality, and are located in `src/test`:

AllTests.java This is a test suite, which runs all of the other test cases. Run this to demonstrate that everything is working properly.

Literature_*.java These test cases demonstrate examples taken from the literature on belief revision, specifically [1], [2], [3], [4], [5].

ProjectCompletionTests.java This is a test case which demonstrates a series of revision steps given by Dr. Weydert to me in an email on Monday 1st April, 2013. I have used this as a litmus test for completion of the initial coding portion of the assignment.

QueryTests.java This test case illustrates the Query API and asserts that it catches errors properly and recognizes valid query syntax. See 1.2 for more information on query syntax.

QueryTreeNodeTests.java This is a low-level test case which tests an underlying library object used for parsing queries. See 1.2 for more information on query syntax.

QueryValueTests.java This is a simple test case for demonstrating proper functioning of the `CumulativeMeasure .getValue()` method.

RevisionTests.java This simple test case demonstrates that the `Cumulative-`

`Measure.revise()` method is working properly.

SandboxTests.java This is a test case that anyone can use to experiment with using the Cumulative Measures query and revision library. See Section 2 for more information.

TestBase.java This is an abstract base class which houses the logic common to all the test cases.

TokenListTests.java This is a low-level test case to test an underlying library object used for parsing queries. See 1.2 for more information on query syntax.

ValueTests.java This demonstrates the functionality of the `Value` object, which represents a Cumulative Measure value encompassing a global (integer) rank and a local (real-valued) probabilistic valuation.

2 Writing Your Own Tests

The test case called `SandboxTests.java` can be used to create your own tests and experiment with the Cumulative Measures query and revision API.

Use Section 1.1 as a guide to experiment with adding propositions to the knowledge base, applying revision operations, and querying the results to verify the results you expect.

Using the **Eclipse debugger** is extremely useful! You can use it to stop execution during a test and inspect the values of the knowledge base and other variables. To debug a test, first set a breakpoint by double-clicking in the margin (by the line number) of the line of code where you would like to stop execution. Then right-click on `AllTests.java` and select **Debug As...** > **JUnit**.

When the debugger stops on a line of code, use your mouse to hover over the variable that you are interested in (hovering over the `cumulative` variable is a good place to start) and Eclipse will display a pop-up with the value of that variable. Pressing F6 will allow execution to continue to the next line, so you can observe the effects of each command as it is run.

References

[1] A. Darwiche and J. Pearl. On the logic of iterated belief revision. *Artificial intelligence*, 89(1):1–29, 1997.

[2] N. Friedman and J.Y. Halpern. Plausibility measures: a user's guide. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 175–184. Morgan Kaufmann Publishers Inc., 1995.

[3] S.M. Glaister. Symmetry and belief revision. *Erkenntnis*, 49(1):21–56, 1998.

[4] Y. Jin and M. Thielscher. Iterated belief revision, revised. *Artificial Intelligence*, 171(1):1–18, 2007.

[5] E. Weydert. General belief measures. In *Proceedings of the Tenth international conference on Uncertainty in artificial intelligence*, pages 575–582. Morgan Kaufmann Publishers Inc., 1994.