Learnlib Practicum

March 7, 2017

Abstract

The goal of this assignment is to practice with the LearnLib framework for automata learning algorithms. You can do this assignment with the same partner as the previous assignment.

Preliminaries: Installing stuff.

Optinal: Install Eclipse For this assingment we reccomended the use of Eclipse. You are however free to compile the code by command line :)

Optional: Install GraphViz Learlib provides the option to export the learned models to the GraphDOT format so that they can visualized by the GraphViz application. If you want to visualize the learned models (reccomended), you need to install GraphViz. This can be done from here: http://www.graphviz.org/Download..php. If you do not install GraphViz, you will see the models in the (textual) GraphDot format.

Sidenote: For Windows users, please be aware that you will need to add GraphViz to your PATH variable (as explained on the download page, but easily looked over).

Part 1. Learning a model of your Wolf-Goat-Cabbage implementation

The assingment. We will start off this practicum by automatically learning a model of your Wolf-Goat-Cabbage implementation. We will provide you with some basic Learlib settings for this task. The only task you will need to perform is to write a wrapper for your implementation so that Learnlib will know how to interact with it. This will be done by implementing Learnlibs SUL interface.

Getting to work. Download the provided basic framework from GitHub. You can import this project into eclipse. Next to the basic framework for using Learnlib (the learner class and the (empty) SUL wrapper class), this also includes a .jar file with the entire Learlib project. (Including a jar file in eclipse: right click project name ->properties ->Java Build Path ->Add External JARs ->Select learnlib jar). Make sure that you can successfully run the learner. It should return a model with only one state, as the wrapper has not yet been implemented.

Sidenote. In the learner class indicate if you have installed graphDOT, so it will produce a graphical model.

Next up. Include your Wolf-Goat-Cabbage implementation into the project (Including a jar file in eclipse: right click project name->properties ->Java Build Path ->Add External JARs ->Select implementation jar). Now start to implement the SUL wrapper.

We assume the same input and outputs as used in the lecture. So the inputs are one of the set ("cabbage?", "goat?", "nothing?", "wolf?") and each input should return a response that is contained in the set ("eaten!", "finished!", "init", "ok!", "retry!").

Part 2. SUL Wrapping

While it is pretty cool to be able to learn a model by interacting with an implementation, the success of this method depends in practice will also be dependent on its performance. Is it still feasible to learn the model of a larger implementation in a reasonable amount of time? Increasing the performance of this method is currently an active area of research.

In this part we want you to get some insight into one of the bottlenecks of the process: interaction with the SUL. Each interaction with the SUL is considered expensive, and therefore if possible we want to limit the amount of interaction.

First we want to gain some insight into the interaction taking place in the learning process. Therefore we want to wrap our SUL in a SymbolCounterSUL[1] that counts the number of total steps taken during the learning process, and then pass this SymbolCounterSUL to the membershipOracle. After the experiment, obtain the data using the getStatisticalData() method. Observe the output.

Now we want to see if we can improve this. Learnlib offers a caching mechanism for SULs that might just help. Read the documentation to see what the cache wrapper[2] does:

Since SULCache's constructor method is deprecated, we recommend that you use the createCache method from the SULCaches[3] class to wrap your SUL with this caching mechanism.

Think about whether you should wrap the SUL directly or whether you should wrap the SymbolCounterSUL. Hint: the SymbolCounterSUL counts every call of the step() method before it passes on the call to the SUL it wraps. If you cannot determine the correct order, you can always experiment to find out :)

See if, after wrapping the SUL in a cache, you can notice a significant change in the number of executed steps.

[1] https://github.com/LearnLib/learnlib/blob/develop/core/src/main/java/de/learnlib/oracles/SymbolCounterSUL.java
[2] https://github.com/LearnLib/learnlib/blob/develop/filters/cache/src/main/java/de/learnlib/cache/sul/SULCache.java
[3] https://github.com/LearnLib/learnlib/blob/develop/filters/cache/src/main/java/de/learnlib/cache/sul/SULCaches.java