The LibALF Library

November 18, 2009

Chapter 1

Introduction

1.1 LibALF Basics

The `libALF` library is an actively developed, stable, and extensively-tested library for learning finite state machines. It unifies different kinds of learning techniques into a single flexible, easy-to-extend, open source library with a clear and easy-to-understand user interface.

The `libALF` Library provides a wide range of *online* and *offline* algorithms for learning Deterministic (DFA) and Non Determinisitc Finite Automaton (NFA). *online* algorithm is a technique where the hypothesis is built by understanding the classification (whether accepted or rejected) of queries asked to some kind of a *teacher*. While, an *offline* algorithm builds an apposite hypothesis from a set of classified examples that were passively provided to it. As of November 18, 2009, the library contains seven such algorithms implemented in it which are listed in Table 1.1.

The central aim of `libALF` Library is to provide significant advantages through potential features to the user. Our design of the tool primarily focuses on offering high flexibility and extensibility.

Flexibility is realized through two essential features the library offers. The first being the support for switching easily between learning algorithms

Online Algorithms	Offline Algorithms
Angluin's L [2] (two variants)	Biermann [3]
NL [4]	RPNI [13]
Kearns / Vazirani [10]	DeLeTe2 [6]

Table 1.1: List of Algorithms Implemented

and information sources, which allows the user to experiment with different learning techniques. The second being the versatility of the tool. Since it is available in both C++ and `Java` (using the Java Native Interface), it can be used in all familiar operating systems (Windows, Linux and MacOS in 32- and 64-bit). In addition, the dispatcher implements a network based client-server architecture, which allows one to run `libALF` not only in local environment but also remotely, e.g., on a high performance machine.

In contrast, the goal of extensibility is to provide easy means to augment the library. This is mainly achieved by `libALF`'s easy-to-extend design and distributing `libALF` freely as open source code. Its modular design and its implementation in C++ makes it the ideal platform for adding and engineering further, other efficient learning algorithms for new target models (e.g., Büchi automata, timed automata, or probabilistic automata).

Other pivotal features of the library include, ability to change the *alphabet size* during the learning process, extensive logging facilities, domain-based optimizations via so-called normalizers and filters, GraphViz visualization.

1.2 Conceptual Details

The `libALF` consists of four main components, the Learning Algorithm, the Knowledgebase, Filters & Normalizers and Logger & Statistics. Figure 1.1 shows a characteristic view of the these components. Our implemention of these components allows for plug and play usage.

1.2.1 The Knowledgebase

The knowledgebase is an efficient storage for language information that accumulates every word and its associated classification. It allows storage of values of arbitrary types and in the forthcoming sections we will describe its implementation where a word is stored as a list or array of `Integers`. It forms the fundamental source of information for a learning algorithm. Using an external storage for the knowledgebase has the advantage of it being independent of the choice of the learning algorithm. This enables interchanging of learning algorithms on the basis of same knowledge available.

1.2.2 Learning Algorithm

A learning algorithm is a component that retrieves the desired information from the knowledgebase to construct an automaton. As mentioned in the previous section, there exists two types of learning algorithms - *offline* and *online* algorithm.

(a) Algorithms

(b) Knowledgebase

(c) Filter & Normalizer

(d) Logger & Statistics

Figure 1.1: Components of LibALF

The workflow of the algorithms begins with a common step wherein the algorithm is supplied with information about size of the alphabet for the automaton. Thereafter, the algorithms follow two distinct procedure to compute the automaton.

The *offline* algorithm continues as stated below.

1. The knowledgebase is furnished with the set of words and their classifications (typically provided by the user).

2. When all details have been supplied and is available in the knowledgebase, the learning algorithm is made to advance to compute hypothesis in conformance with the initial input.

An *online* algorithm proceeds in the following manner. The following two steps are repeated until a correct conjecture is determined.

1. The algorithm is made to advance.

2. Here one of the following two possibile events may occur.

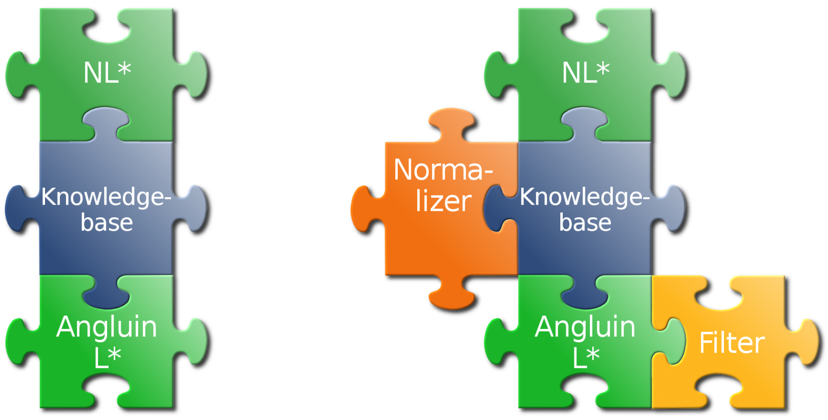

Figure 1.2: Pictorial Representation of Plug and Play support

(a) If no hypothesis is created, "membership queries" that require associated classification are resolved (by the *teacher*) and added to the knowledgebase.

(b) If a hypothesis was created, the "equivalence query" is answered by the teacher. If the conjecture is incorrect a counter example is rendered by the teacher.

An insight into the working of the two algorithms is given in Section 1.3.

1.2.3 Filters and Normalizers

A knowledgebase can be associated with a number of *filters*, which are used for domain-specific optimization. By that, we mean the knowledgebase makes use of domain-specific information to reduce the number of queries to the teacher. Such filters can be composed by logical connectors (and, or, not). In contrast, *normalizers* are able to recognize words equivalent in a domain-specific sense to reduce the amount of knowledge that has to be stored.

1.2.4 Loggers and Statistics

The library additionaly features such as means for statistical evaluation or loggers. A logger is an adjustable logging facility that an algorithm can write to, to ease application debugging and development. The modularity of our approach in developing `libALF` facilitates these components to be added in an easy plug and play fashion and that is shown in Figure 1.2 and Figure 1.3.

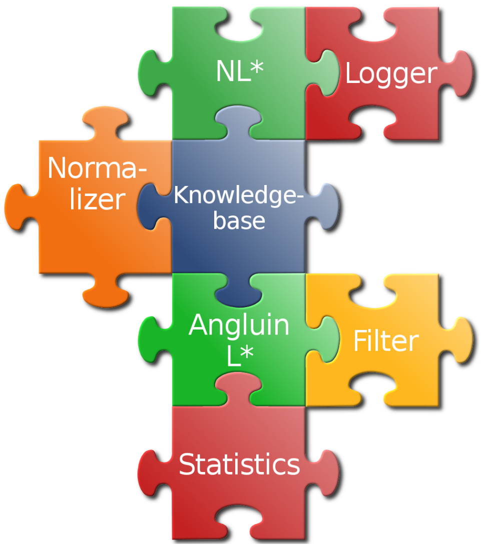

Figure 1.3: Addition of Loggers and Statistics in Plug and Play fashion

1.2.5 Connections of the Components

The primary aspect in describing the working would be to outline the data flow between a learning algorithm, the knowledgebase and the user (or *teacher*) as sketched in Figure 1.4.

The learning algorithm and the knowledgebase share information with user. The knowledgebase, as stated earlier, is the fundamental information for the learning algorithm to develop an automaton. The learning algorithm advances with whatever knowledge is available. The learning algorithm connects with the user to collect relevant information such as equivalence of a conjecture or to retrive counter-example. When the learning algorithm creates more membership queries, they are stored in the knowledgebase leading to it initiating a communication with the user who is required to answer membership queries (or input sample words in case of an offline algorithm). All such information extended by the user are stored in the knowledgebase.

1.3 Demo Application

In this section we describe the working of the *offline* and *online* algorithms with reference to the demo code in C++ available at our website `http://libalf.informatik.rwth-aachen.de/`. Demo programs of the algorithms in `Java` are also available there.

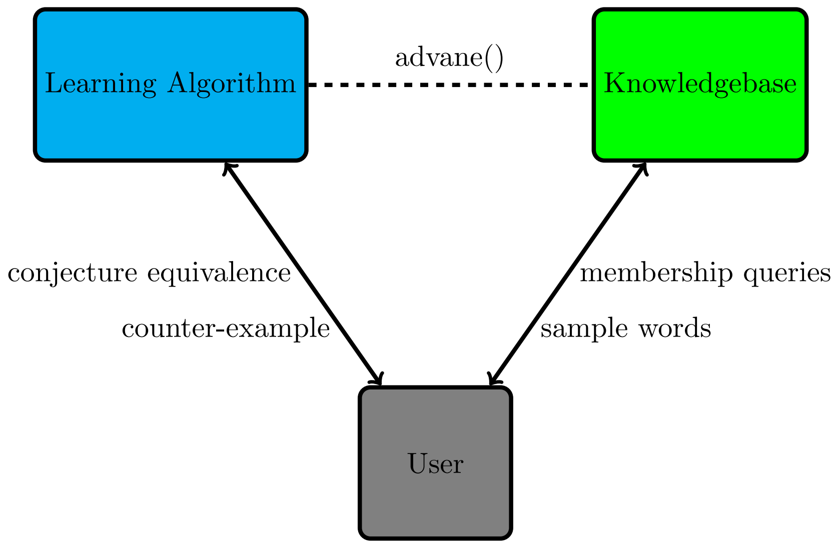

Figure 1.4: data flow of the `libALF` components

The following code snippet briefly demonstrates how to employ the `libALF` library in a user application. It is important that you become familiar with the auxilliary methods used in the program and hence their operations are explained first.

- ***get_AlphabetSize()*** - Promts the user to provide information about the size of alphabet and stores it as an `Integer`.
- ***answer_Membership(li)*** - Takes the list of queries as an arguement and presents it to the user to classify them. It returns `true` when the word is to be accepted and returns `false` when it is to be rejected.
- ***check_Equivalence(cj)*** - Presents the computed conjecture to the user who marks it as correct or incorrect. Returns `true` or `false` respectively.
- ***get_CounterExample(alphabetsize)*** - Requests the user to input the counter-example and returns the word as a list (array in `Java` implementation) of integers. It takes the alphabetsize as a parameter for validation purposes.
- ***get_Samples(alphabetsize)*** - Retrieves the sample word from the user. The alphabetsize is passed as a parameter for validation purposes.
- ***classification = get_Classification()*** - Retrieves the classification of the sample word from the user. Returns `true` when the word is to be accepted and returns `false` when it is to be rejected.

- ***enough_Samples()*** - Requests the user to specify whether all samples have been provided by the user. Returns "y" if user desires addition of more samples or "n" if all samples have been provided already.

1.3.1 Online Algorithm

An *online* Algorithm, as mentioned in the previous section, formulates the conjecture by putting forth "queries" to the *teacher*.

```
 1 void main(int argc, char**argv) {
 2 int alphabetsize = get_AlphabetSize();
 3 knowledgebase<bool> base;
 4 angluin_simple_table<bool> algorithm(&base,
 5 NULL, alphabetsize);
 6 do {
 7  conjecture * cj = algorithm.advance();
 8  if (cj == NULL)
 9  {
10    list<list<int> > queries = base.get_queries();
11    list<list<int> >::iterator li;
12    for(li = queries.begin(); li != queries.end(); li++)
13    {
14       bool a = answer_Membership(*li);
15       base.add_knowledge(*li, a);
16    }
17  }
18  else
19  {
20    bool is_equivalent = check_Equivalence(cj);
21    if (is_equivalent) result = cj;
22    else
23    {
24       list<int> ce = get_CounterExample(alphabetsize);
25       algorithm.add_counterexample(ce);
26    }
27  }
28 }while (result == NULL);
29 cout<<result->visualize();
30 }
```

The workflow of the program is as described below:

1. At line 2, the program promts the user to input the *alphabet size* of the Automaton.

2. An empty knowledgebase is now intialized at line 3. (The knowledgebase stores the words as a list of `Integers`)

3. Now, a learning algorithm is created by providing three parameters - the knowledgebase, NULL for a logger, the Alphabet Size.

4. After having initialized the learning algorithm, the program is subjected to a loop where the algorithm is made to *advance* (Line 7). The result of this is stored in a `conjecture` type variable `cj`.

5. If there was no sufficient information available in the knowledgebase to construct a conjecture, then `cj` is NULL and the algorithm enters the condition at line 8. The algorithm produces the *membership queries* that needs to be resolved by the user (or *teacher*). The queries are obtained using the method `get_queries`. (Note: the queries are obtained in "list of list of `Integers`" since words are stored as `Integers` and there may be more than one query)

6. The queries produced are presented to the user who classifies it as *accepted* or *rejected*. This is done at Line 13 with `answerMembership` function. Subsequently, This information is added to the knowledgebase and the iteration of the loop continues.

7. However, if a conjecture was computed at line 7, (implying that enough information was available in the knowledgebase), then algorithm enters the condition at line 18. The conjecture is presented to the user by `check_equivalence` function at line 20.

8. If the conjecture is equivalent, the user marks it correct and the conjecture is stored in variable `result`. Iteration ends and the `result` is displayed in line 32.

9. If it is not equivalent, the user is now prompted to provide a counter example (line 27) and the program continues with the iteration. Typically, the counter example would influence the learning algorithm to invoke more *membership queries* that is to be resolved during the next *advance* of the algorithm.

1.3.2 Offline Algorithm

An *offline* algorithm, as mentioned in the previous section, computes a conjecture from a set of passively provided input samples with their classifications.

```
1 int main(int argc, char**argv)
2 {
3   int alphabetsize = get_AlphabetSize();
4   string input = "y";
5   list<int> words;
6   bool classification;
7   knowledgebase<bool> base;
8   while (input == "y")
9   {
10    words = get_Samples(alphabetsize);
11    classification = get_Classification();
12    base.add_knowledge(words, classification);
13    input = enough_Samples();
14  }
15  RPNI<bool> algorithm(&base, NULL, alphabetsize);
16  conjecture *cj = algorithm.advance();
17  cout <<cj->visualize();
18 }
```

The workflow of the above program is as follows:

1. At line 2, the program prompts the user to input the *alphabet size* used for the automaton (coded in the function `get_alphabetsize`)

2. Variables for storing the sample words and their classifications are described subsequently. An empty knowledgebase is now initialized.

3. The program then passes over a loop which recursively performs the action of reading the sample (line 10) and its classification (line 11) from the user. As and when the user inputs this information, it is continually added to the knowledgebase (line 13). The loop ends when the user indicates that the desired number of samples have been entered as coded in line 13 (its for this purpose that the `String input` is first initialized to "y").

4. Now, a learning algorithm (RPNI Offline Algorithm) is created by providing three parameters - the knowledgebase, NULL for a logger, the Alphabet Size (line 15)

5. Having initialized the algorithm, it is now made to *advance* which produces a conjecture that pertains to the user's specification of samples. (line 16)

6. Finally, the conjecture is printed as coded in line 17 of the program.

In both the command line programs implemented in C++ and `Java`, the program outputs the ".dot" file which contains the code that builds the conjecture graphically. (This file may be executed using the GraphVIZ tool).