The `libalf` Library

August 15, 2010

Contents

List of Figures

List of Tables

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 Nondeterministic Finite Automata (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 August 15, 2010, 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 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.

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

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., B?chi 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 implementation 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 a conjecture. As mentioned in the previous section, there exists two types of learning algorithms - *offline* and *online* algorithm.

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

The *offline* algorithm continues as stated below.

1. The knowledgebase is furnished with the set of words and their classifications (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 the conjecture in conformance with the samples.

(a) Algorithms

(b) Knowledgebase

(c) Filter & Normalizer

(d) Logger & Statistics

Figure 1.1: Components of `libALF`

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 possible events may occur.

   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.

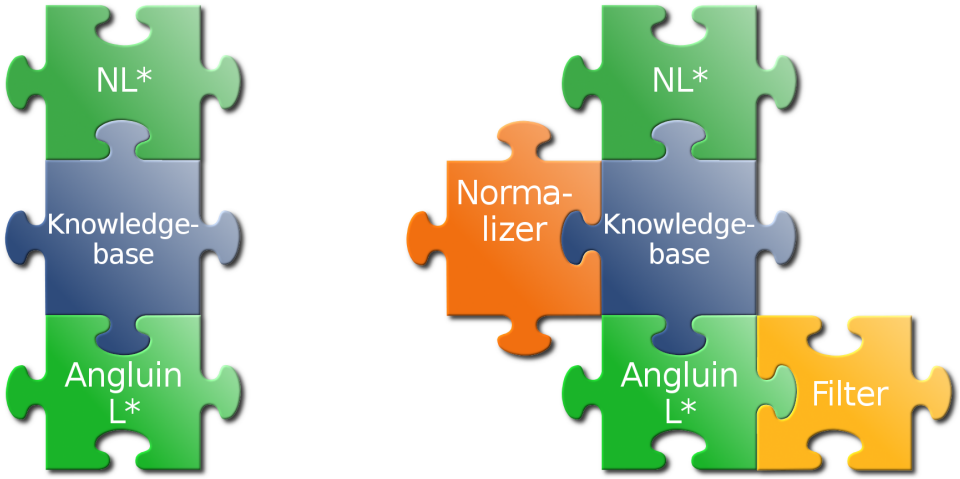

Figure 1.2: Pictorial Representation of Plug and Play support

1.2.3 Filters and Normalizers

The knowledgebase can be associated with a number of *filters*, which are used for domain-specific optimization. This implies that 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 additionally features 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.

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.

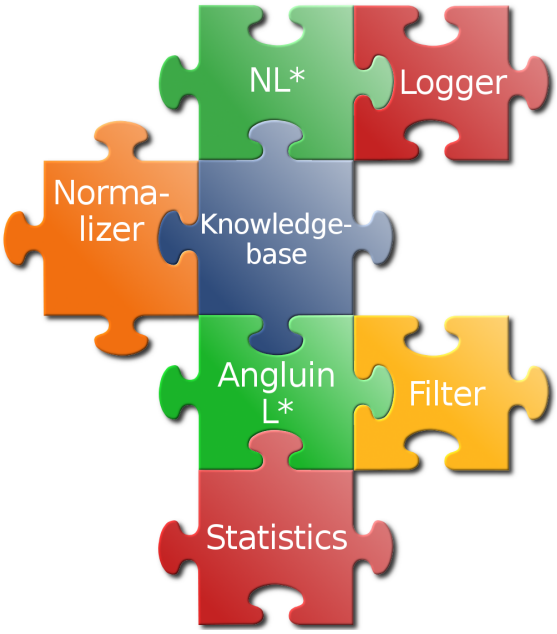

Figure 1.3: Addition of Loggers and Statistics in Plug and Play fashion

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 retrieve 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.

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()*** - Prompts 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 argument and presents

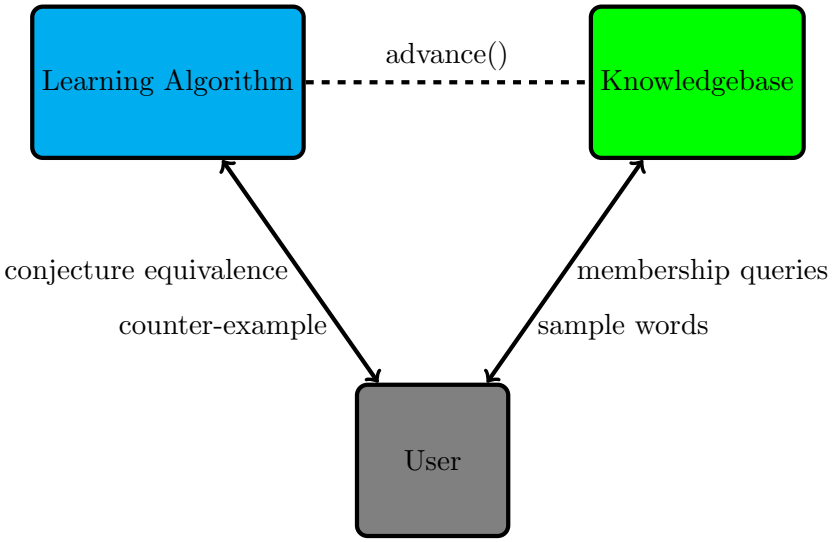

Figure 1.4: data flow of the `libALF` components

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 prompts the user to input the *alphabet size* of the Automaton.

2. An empty knowledgebase is now initialized 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).

2 Compiling and Installing libALF

This chapter will guide you through the compilation and installation of libALF on Linux and Windows.

libALF works on Linux and Windows on both 32- and 64-bit architectures. However, as libALF has no prerequisites, it is most likely that it also runs on various additional operating systems. If you want to compile libALF for another operating system, e.g. MacOS X, the guidelines for compiling libALF for Linux may be a good reference.

This document is organized in six sections: The first two sections describe how to obtain libALF (if you not already have) and what prerequisites need to be fulfilled. Sections 2.3 to 2.5 show how to compile and use libALF, jALF and dispatcher. The sixth section gives help on troubleshooting.

2.1 libALF Package Information

The library is freely available under the open source LGPL v3 license at the libALF website http://libalf.informatik.rwth-aachen.de. Download the libALF package and extract it to a folder of your choice. The package contains the following components:

- The libALF C++ library.
- jALF (libALF's Java interface).
- The dispatcher (a network-based libALF server).

This guide will demonstrate how to employ and use libALF in user applications through *examples* available at libALF's website. We recommend that you download and extract the example sources to a folder of your choice.

2.2 Prerequisites

The libALF library itself does not have any prerequisites, but some components have. To use the additional components, please ensure that the following requirements are satisfied.

- For compiling and using jALF you need a Java Development Kit (JDK) Version 6.0 or later installed. Moreover, we recommend using the *Ant* build tool downloadable from http://ant.apache.org/.
- The dispatcher requires a POSIX-compliant operating system. While there is no problem under Linux, the dispatcher will not compile under Windows.

Linux. For compiling the C++ sources in Linux, you require a C++ compiler (this document assumes that you use the *GNU C++ compiler*) and the *make utility*, which is used to automate the build process. Both tools should be installed by default on every Linux machine.

Windows. To compile the C++ sources on Windows, we recommend using the *Minimalist GNU for Windows (MinGW)* compiler and *MSYS*, a Unix-style shell for Windows. Both can be obtained from `http://www.mingw.org/`.

Please follow the instructions on the website to set up MinGW and MSYS properly. In particular, make sure that you install the MSYS make package (if not done automatically). Using MSYS gives you the advantage of following all instructions described in this document no matter whether you use Linux or Windows. However, please be careful with folder names that contain blanks; you may have to enclose them in quotes and replace every blank with a backslash followed by a blank (or, in the best case, you avoid them completely).

2.3 The libALF C++ Library

The section will describe how to compile and install the library as well as how to run applications that use the library.

2.3.1 Compiling libALF

You have the choice to compile the `libALF` library either as a *static* or as a *shared* library. If you do not know the difference or if you just want to use the library, you should compile a shared library as described below and follow the respective instructions for running your application.

Compiling a Shared Library

Compiling `libALF` is easy: simply change into the `libalf/src` folder and invoke the make utility by typing

```
make
```

The make utility automatically detects which operating system you are running and compiles the library accordingly. After the compilation you should find the binary file `libalf.so` (on Linux) or `libalf.dll` (on Windows) inside the `libalf/src` folder. However, if you experience problems, you can explicitly tell the make utility for which system you want to compile `libALF` by typing `make libalf.so` (under Linux) or `make libalf.dll` (under Windows).

Compiling a Static Library

You can compile a static library using the command (inside `libalf/src`)

```
make libalf.a
```

on both Linux and Windows.

However, make sure that you delete any shared library in the folder before you link your application with libalf as some operating systems (e.g. Linux) always prefer shared libraries if present.

2.3.2 Installing libALF

Installing `libALF` means to copy to the compiled shared library and `libALF`'s headers to a location where your operating system finds them.

Linux. To install `libALF` in Linux, first compile the library and type `make install`. You can uninstall `libALF` by using the command `make uninstall`. Please note that you need root privileges for both actions.

Windows. On Windows, you have to manually copy the compiled shared binary files to your `windows/system` directory. Unfortunately, there is no common place to put header files in. Thus, you have to specify the header's location every time you compile an application that uses `libALF` (see the section below).

2.3.3 Compiling Applications That Use libALF

When compiling an application that uses `libALF`, the compiler needs to find `libALF`'s headers and the compiled library. Please note that you do not have to provide this information if you have libalf installed on your system.

Otherwise, you have to use the GNU C++ compiler's `-I` parameter to specify `libALF`'s header locations (typically `libalf/include`) and the `-L` parameter to specify the location for the compiled library (which is `libalf/src`). You also have to link the application to `libALF` using `-lalf`.

We will consider the online-example to explain the compilation.

Compiling applications that links to shared library. To compile the online example that uses the shared library, type the following command.

```
g++ -I path_to_headers -L path_to_library online.cpp -lalf
```

Compiling applications that links to static library. If you want to link libalf statically into your application, you can do so by adding `-static` as additional parameter just before linking to libalf like below.

```
g++ -I path_to_headers -L path_to_library online.cpp -static
-lalf
```

In both cases, it is also a good idea to specify the name of the output file using the `-o` parameter, e.g. `-o online`.

Additional Parameter for Windows. Please note that on Windows the Winsock2 library has to be linked additionally to every program using `libALF`. You can do this by adding `-lws2_32`. Again it is crucial that you add this parameter after all input files.

2.3.4 Running Applications That Use `libALF`

An application *statically* linked to libalf can be executed as usual. However, if you run a program that uses `libALF` as a *shared library*, you need to specify where your operating system can find the library (again, you do not need to provide this information if you have installed `libALF` on your system).

Linux. On Linux, use the `LD_LIBRARY_PATH` variable to point to the location of the shared library. For instance, you can run the above compiled online example with the command

```
LD_LIBRARY_PATH=path_to_library ./online
```

Windows. Unfortunately, on Windows there is no direct way of telling the system where to find shared libraries. Instead, you have to add their locations to Windows' `PATH` variable or copy the library into the folder your application is executed from. Then, execute your application as usual.

For further details please refer to the examples' Readme and Makefile.

2.4 The `jALF` Java Library

`jALF` is the Java interface to `libALF`. It lets you access `libALF` via the dispatcher or via Java's *Native Inter-face JNI*. The latter way requires that you compile a second C++ library (some kind of wrapper), that obeys Java's naming convention and performs some basic conversions of internal data structures. However, if you want to use `jALF` only in connection with the dispatcher, it is enough to compile and use the Java sources.

In the following we assume that you are familiar with basics of compiling and running Java programs.

2.4.1 Compiling `jALF`'s Java sources

In order to compile `jALF`'s Java sources, change to the `libalf/jalf` folder and type

```
ant
```

This invokes the Ant build utility and produces the file `jalf.jar` containing all compiled class files inside the `libalf/jalf` folder. If you do not wish to use `jALF` via JNI, you can skip compiling `jALF`'s C++ sources.

Note that you can generate `jALF`'s *JavaDoc* also using Ant with the command `ant doc`. Thereafter, the JavaDoc can be found inside the `libalf/jalf/java/doc` folder.

2.4.2 Compiling jALF's C++ Sources

The jALF C++ library needs to be a shared library. However, you have the option to link libALF either dynamically or statically to jALF. The latter option is often preferred and enabled by default. Please remember to delete any shared library in libalf/src before you compile jALF's C++ sources (libALF is recompiled for you). You may use the command make -C libalf/src clean for this.

Compiling jALF's C++ sources is also automated by means of the make utility. However, as additional information the Java compiler requires the location of Java's JNI header files, which are contained in every JDK. Their location is passed on to the make utility using the JAVA_INCLUDE variable. Thus, to compile jALF's C++ sources, go to libalf/jalf/src and execute

```
JAVA_INCLUDE=path_to_jdk/include make
```

Again, the make utility should detect your operating system automatically, but you can also use the commands make libjalf.so (for Linux) and make jalf.dll (for Windows) to explicitly compile jALF for your desired operating system. After a successful compilation, the compiled binary is located in libalf/jalf/src.

Dynamic Linking. As mentioned, libALF is linked statically by default. If you want link libALF *dynamically*, you can use the commands make libjalf.so-dynamic (for Linux) and make jalf.dll-dynamic (for Windows).

2.4.3 Compiling Java applications that use jALF

Fortunately, the Java compiler does not need to know anything about the C++ libraries to compile your application and only needs access to jALF's Java class files. You specify this information by adding the jalf.jar file to Java's classpath. Our Java online example, for instance, can be compiled using the following command (first change into the folder containing the example sources):

```
javac -classpath "path_to_jalf/jalf.jar" Online.java
```

2.4.4 Running Java applications that use jALF

Besides the location of the jalf.jar, running a Java application that uses jALF requires telling Java where it can find the compiled jALF and libALF C++ libraries. (If you have installed libALF to your system or if you linked jALF statically to jALF, you do not need to bother about the latter.)

The place where Java looks for C++ libraries is controlled by Java's interval library path variable. This variable can only be changed at the start of the Java VM. You do so by setting the variable named java.library.path to the location of the jALF library (i.e, the jALF C++ binary which is typically libalf/jalf/src) using the -D parameter.

Linux. To run the online example on Linux, one has to execute the following command (inside the folder containing the compiled online example):

```
java -classpath "path_to_jalf/jalf.jar:."
     -Djava.library.path=path_to_jalf_library Online
```

If necessary, specify the location of the shared `libALF` library as described in Section 2.3.

Windows. Please recall that Linux and Windows use different ways of separating folders. While you must use a colon on Linux, you must use a semicolon on Windows; everything else is the same as before.

```
java -classpath "path_to_jalf/jalf.jar;."
     -Djava.library.path=path_to_jalf_library Online
```

For further details please refer to the examples' Readme.

2.5 Compiling and Using the Dispatcher

Please recall that the dispatcher only compiles and runs on a POSIX-compliant operating system such as Linux, but not on Windows.

2.5.1 Compiling the Dispatcher

To compile the dispatcher, first compile a shared `libALF` library as described Section 2.3.

Dynamic Linking. By default, the dispatcher is dynamically linked to `libALF`. To compile an executable linked statically, change into the folder `libalf/dispatcher` and execute

```
make
```

This creates the executable dispatcher in the same directory.

Static Linking. To link the dispatcher statically to `libALF`, use the following command

```
make dispatcher-static
```

Again, remember to remove any compiled shared library in `libalf/src` first.

2.5.2 Running the Dispatcher

The dispatcher is executed like any other executable on your system. However, remember to specify the location of the `libALF` shared library if necessary.

2.6 Troubleshooting

When experiencing troubles, the first thing you should try is to execute `make clean` in the `libalf` and `libalf/jalf` folders as well as `ant clean` in the `libalf/jalf` folder. This deletes all compiled files and solves most compiler and linker problems. However, if this does not work for you, you may find a solution for your problem in the list below:

- There are no known problems.

3 The Knowledgebase

The knowledgebase is the central repository of membership information. It is a database that stores words and their classifications. Apart from this basic functionality, the knowledgebase also offers number of other features thereby increasing the support for extensibility. These features are reflected in its implementation wherein the methods used in the knowledgebase can be divided into two categories - ***Methods important for Using*** `libALF` and ***Methods important for expanding*** `libALF` .

The chapter discusses these methods from both a user and a developer's perspective. The material will include adequate account of its structure, operations and implementation. The final section of this chapter will provide an appendix of all the methods used in programming the knowledgebase with a brief description corresponding to it.

3.1 The knowledgebase - A User's Perspective

This section deals with basic functionality of the knowledgebase and provide fundamental information that one would have to know to employ `libALF` in an application. It will explicate functional practice of the database and some elementary details of methods to better understand how all operations are carried out.

To discover more about the internal structure of the knowledgebase and the methods that help to extend it, you may refer to the next section on Developer's Perspective and Methods in Detail.

Basic Concepts

Definition of some key terms concerning the underlying concepts are listed below.

Alphabet An alphabet is a finite set of symbols which is usually denoted by Σ. Symbols can be numbers (0,1,...) or alphabets(a,b,...) and so on.
Alphabet size $\mid \Sigma \mid$ is the size of the set Alphabet. Since `libALF` uses integers as symbols, the largest symbol in the alphabet is one less than the alphabet size. Thus, when alphabet size of two is specified by the user, `libALF` uses the following symbols.

$$\Sigma = \{0, 1\}$$

$$\mid \Sigma \mid = 2$$

This implies, that an alphabet size of two results in the largest symbol being as "1".

Word A word ($w \in \Sigma^*$) is finite string formed by the concatenation of the symbols from Σ. Since `integer` type are used to represent symbols, the words are operated as a `list` of `integer`.

Language Language $L = \Sigma^*$ is a set of words formed by symbols, given the alphabet.

In the context of the previous example, "01101" is a word from the given set of alphabet and the L = { 01101, 11011} is a Language.

Words and Classifications

The knowledgebase of `libALF` is an efficient storage of words and their classifications. Words are represented as `list` of `integer`. Classification refers to a set of arbitrary values which that are mapped to the words. For instance, classification for a Finite Automata refers to `true` or `false`. Since the knowledgebase is a template class, arbitrary values can be used for storing the classification.

Queries and Answers

The next important function of the knowledgebase is to store queries to help the learning algorithm build the conjecture. As mentioned in the Introduction, a query is a word whose classification is unknown and needs to be retrieved from the user or teacher. In other words, the query must be *resolved* by the user. To resolve the query, user provides what is called an *answer*. Thus, when a learning algorithm is processing the membership information from the knowledgebase, it may give rise queries and are stored in the knowledgebase. These are later presented to the user to resolve them.

Serialize and Deserialize

`libALF` allows serialization and deserialization of the knowledgebase. This feature is most advantageous in offering portability. Serialization allows user to save the current work done with `libALF` as a linear representation. User can save the knowledgebase into the hard disk, share it over the internet, carry it and use it another machine and so on. Deserialization converts the linear form back to the data structure that can be processed by the learning algorithm.

GraphViz Visualization

The knowledgebase allows one to generate a GraphViz Visualization of all available information. User can create a ".dot" file of the knowledgebase which can be executed by the GraphViz tool for a pictorial representation.

Merging Knowledgebases

Another essential feature of the knowledgebase is the ability to be merged with another knowledgebase preserving the consistency. These features are elaborated in forthcoming

sections.

3.1.1 Methods in Detail

The section describes mostly the methods important for using `libALF`. The description pertains to support understanding how the knowledgebase works and how it can be employed in an application.

Creating the Knowledgebase

The knowledgebase is built on a class named `knowledgebase`.

- **knowledgebase::knowledgebase()**

  The constructor of the `knowledgebase` class creates the knowledgebase.

Adding Knowledge to the Knowledgebase

- **knowledgebase::bool add_knowledge(list<int> & word, answer acceptance)**

  The method is used to add membership information to the knowledgebase. The parameter "word" represents the sample word and "acceptance" represents the classification of the word.
  The method returns `true` if the knowledge was added successfully. Otherwise, returns `false`.

Handling Queries

When an *online* algorithm produces queries, they are stored in the knowledgebase and can be retrieved and resolved by the user. The following methods are used for related operations.

1. **knowledgebase::knowledgebase * create_query_tree()**

   The method creates a knowledgebase containing only the queries. The return type, is therefore, set as `knowledgebase`.

2. **knowledgebase::list<list<int>> get_queries()**

   This method returns the list of all the queries present in the knowledgebase.

Alphabet in Knowledgebase

At any point of time, one can retrieve the largest symbol being processed in the knowledgebase using the following method.

- **knowledgebase::int get_largest_symbol()**

  The method returns the largest symbol that exists in the knowledgebase which is one less than the alphabet size. `libALF` uses integers to store symbols. The method, however, recognizes only increment in the alphabet size. A decrease in the size of alphabet is not reflected.

- **knowledgebase::int check_largest_symbol()**

  The method performs a check on the knowledgebase and realizes the largest symbol that is currently available. Thus, a decrease in the alphabet size can be recorded by this method.

Iterators

The knowledgebase uses the `list` of `integer` to represent the words. Consequently, a `list` of `list` of `integer` is used to represent many words in a sequence. This particularly is used when viewing the whole data or all the queries present in the knowledgebase. To iterate over these lists and process the words, the following methods are used.

1. **knowledgebase::iterator begin()**

   The method returns an iterator that begins at the root node.

2. **knowledgebase::iterator end()**

   The method returns the final or the end node for the iterator.

3. **knowledgebase::iterator qbegin()**

   The method returns an iterator that begins at the first query present in the knowledgebase.

4. **knowledgebase::iterator qend()**

   It returns the end node for the iterator.

The iteration over the words is performed by overloading the "+" operator. Given below is an example of its usage.

```
1 iterator ki;
2 list<list<int> > ret;
3 for(ki = this->qbegin(); ki != this->qend(); ++ki)
4         ret.push_back(ki->get_word());
```

Here, the iterator begins at the first query present in the knowledgebase and uses the "get_word()" function to retrieve the query and adds it to "ret".

Displaying the knowledgebase

This refers to various types of representation of the knowledgebase.

1. **knowledgebase::string tostring()**

   The method creates a `String` representation of the entire knowledgebase.

2. **knowledgebase::string generate_dotfile()**

   The method is used to create a GraphViz Visualization of the entire knowledgebase. The `String` returned by the method can be saved as a ".dot" file and executed by the GraphViz tool for obtaining a graphical representation of the knowledgebase.

Serialization and Deserialization

The feature increases the portability of `libALF`. It is performed using the following methods.

1. **knowledgebase::basic_string<int32_t> serialize()**

   The method converts the entire knowledgebase into a linear representation as a `String` composed of integers.

2. **knowledgebase::bool deserialize(basic_string<int32_t>::iterator &it, basic_string<int32_t> ::iterator limit)**

   This method converts the serialized linear form of the knowledgebase to the data structure recognized and operable by the learning algorithm.

Merging Knowledgebases

- **bool merge_knowledgebase(knowledgebase & other_tree)**

  The method returns `true` after merging two consistent knowledgebases. Two knowledgebases are said to be consistent only if they contain similar words and answers. The method returns `false`, if the knowledgebases are inconsistent.
  However, the method ignores the queries and merges only all the answered words from the two knowledgebases.

Other Methods - Retrieving Memory Usage

- **unsigned long long int get_memory_usage()**

  As an additional feature, the method returns the memory used by the knowledgebase.

Concluding Notes

3.2 Structure of the Knowledgebase - A Developer's Perspective

The data structure of the knowledgebase is designed to offer flexibility and expandability of `libALF` library.

3.2.1 Representation of a word in the Knowledgebase

The knowledgebase is a prefix tree with nodes representing words. Consider a word formed by N symbols/characters. In principal, the knowledgebase does not *store* the word at the node but stores the i^{th} symbol of the word (where `i`>0 and `i`<N). Figure 3.1 gives a pictorial representation of how a word is represented in the knowledgebase.

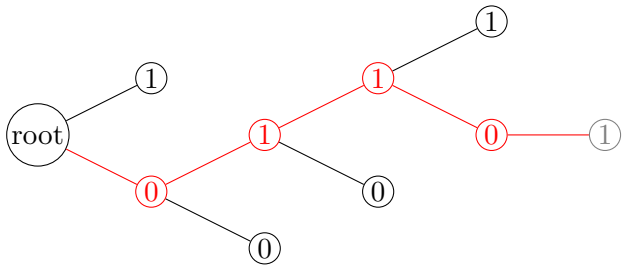

Figure 3.1: Representation of a word "01101" in knowledgebase

Consider the word `01101` as marked in the tree above. When this word is added to the knowledgebase, it is not stored as a single `String` at a node but the symbols of the word are stored at consecutive nodes. Sequence of these symbols at depth six (starting from the root) constructs the word. Hence it is termed that the node circled gray *represents* the word `01101`. In the tree, the symbol `0` is a child of the root, symbol `1` is the child of `0` and so on. The node representing the word can be reached by traversing from the root to the last child and accumulating the symbols that every node contains. Alternatively, the word can also be retrieved by ascending from a node to the root and reversing the word obtained. `libALF` uses the latter technique.

3.2.2 Description of the Structure

The knowledgebase is as a template class enabling the use of arbitrary values for storing membership information. The class `knowledgebase` contains a class `node` consisting variables necessary for the node. `node` also holds internal methods which are important for both using and expanding `libALF`.

The constructor of the class `node` creates the root node with the following values to its attributes.

- `parent`

  A pointer variable of type `node` that points to the parent of the node.

- `label`

  An `integer` variable that stores the symbol represented by the node.

- `status`

  The variable **status** indicates whether the classification of the word represented by the node is *required*, *answered* or can be *ignored*.
  Since the knowledgebase is a prefix tree, it stores not only the supplied words but also the prefixes of the word. However, the classification of these prefixes may not be of interest and can be *ignored*. On the other hand, learning algorithm creates queries that are to be resolved. Status of such words are set as *required*. Hence, the variable differentiates these words. For this purpose, it is an `enum` type variable that can take one of three values "NODE_IGNORE", "NODE_REQUIRED" and "NODE_ANSWERED".

- `ans`

  The variable stores the answer (or the acceptance criteria / classification) of the node.

3.2.3 Methods in Detail

This section describes methods that are more useful for extending `libALF`. These methods mostly emphasize on operations that can be performed over the node, methods on handling words, classifications and queries. A complete list of methods and their explanation is provided in the next section.

Creating the Root node

The root node is initialized when the knowledgebase is created. The constructor of class `node` defines the following properties for its attributes.

- **node::node(knowledgebase * base)**

  The method sets the following properties to the root node.

  1. parent = NULL ; root node does not have a parent.
  2. label = -1 ; which is equivalent to epsilon
  3. status = STATUS_IGNORE ; the acceptance rule or classification of the root node is initially not necessary.

Working with Nodes

The child nodes are structured as `vector` type `node` variable called `children`. Methods operating on the nodes are mostly internal methods and cannot be accessed publicly.

1. **node::node* get_next(node * current_child)**

   The method returns the next node or the next child of the current node (which is passed as an argument).

2. **node::node * get_parent()**

   The method returns the parent of the current node.

3. **node::list<int> get_word()**

   The method returns the word that the current node represents. The method traverses backwards in the tree (ascending from child to parent) and reverses the sequence obtained to build the correct word.

4. **node::int get_label()**

   The method returns the label of the node.

5. **node::node * find_child(int label)**

   The method finds the child node with the specified label.

6. **node::node * find_descendant(list<int> :: iterator infix_start, list<int>:: iterator infix_limit)**

   The method finds the child node specified by a word and returns it. It traverses through the tree based on the iteration over the word to find the path that generates the required word.

7. **knowledgebase::node* get_rootptr()**

   The method returns the pointer to the root node of the knowledgebase.

8. **knowledgebase::node* get_nodeptr(list<int> & word)**

   The method returns the pointer to the current node specified by the word as the parameter.

Words and Classification

The primary purpose of the knowledgebase is centered on storing words and classifications which constitutes the first source for a learning algorithm to compute a conjecture. Methods related to this function are listed below.

1. **node::node * find_or_create_child(int label)**

   This function returns the child node given the label. If the node does not exist, it creates the child node with the specified label.

2. **node::node * find_or_create_descendant(list<int>::iterator infix_start, list<int>::iterator infix_limit)**

   This function behaves almost similar to the previous one. The difference is that it does not operate on a single label but on a word (which is a list of `integer`).

The method `add_knowledge` simply calls the method `find_or_create_descendant` so that the knowledge will be added only information about the word does not already exist in the knowledgebase.

Handling Queries

In addition to the already discussed query handling methods, the ones discussed below is of most interest for expanding `libALF`. As already mentioned, query handling mainly depends on the `status` of the word. The following methods operate on this aspect.

1. **node::bool mark_required()**

   The method returns `true` if the acceptance or the classification of the node is *required* (i.e, "status" is NODE_REQUIRED). It returns `false` if the classification is already known.

2. **node::bool is_required()**

   The method returns the "status" as `NODE_REQUIRED`. It is used to set this status to a particular node under consideration.

3. **node::bool is_answered()**

   The method returns the "status" as `NODE_ANSWERED`. It is used to set this status to a particular node under consideration.

4. **node::answer get_answer()**

   The method returns the *answer* (classification of the node) stored for the node.

The following methods describe the operations associated with queries.

1. **knowledgebase::int add_query(list<int> & word, int prefix_count = 0)**

   The method is primarily used to add a query to the knowledgebase. When a query is generated, the method first checks if that word already exists with its classification in the knowledgebase (by using the `find_or_create_child(int label)` method). Hence, if the classification of the word is unknown and does not already exist, the corresponding node will be created and eventually added in the query tree (since this method is used by `create_query_tree(...)`.

2. **knowledgebase::bool resolve_query(list<int> & word, answer & acceptance) and bool resolve_or_add_query(list<int> & word, answer & acceptance)**

These two methods can be described together as their functionality is almost similar and differ only in one aspect. Both methods return `true` if the classification of the word is already known and `false` if it is unknown. While "resolve_query()" only returns `false`, "resolve_or_add_query()" marks the status of this word as required and then returns `false`. Naturally, the former uses "find_descendant()" and the latter makes use of "find_or_create_descendant()".

3. **knowledgebase::void clear_queries()**

   This method is used to remove all the nodes that are identified or marked as a query.

Alphabet in Knowledgebase

The section gives an extended view of the concept of alphabet size in the knowledgebase. In principal, the knowledgebase does not store the alphabet size of the conjecture specified by the user. The knowledgebase does not work only based on the alphabet size specified by the user and is capable of constructing the tree even if symbols outside the alphabet set are input. The knowledgebase can identify the rise in the alphabet size and record the largest symbol in the tree. But, such improper input will lead the learning algorithm to compute incorrect conjecture.

1. **knowledgebase::int get_largest_symbol()**

   The method has already been described in the section on user's perspective. Deepening into the concept behind it, the method simply returns what is available in the variable "largest_symbol". It does not check whether the alphabet size has been modified. A good way to do that would be to use the methods listed below.

2. **knowledgebase::int check_largest_symbol()**

   The method performs a check on the knowledgebase and realizes the largest symbol that is currently available. Hence, if there was a change in the size of the alphabet at some point of time, it is automatically adjusted when this method is called.

3. **bool cleanup()**

   The method cleans the knowledgebase by removing all the unnecessary branches i.e., branches that consists only of IGNORE as the status. This is an example for a method that can cause a change in the largest symbol. If the branches containing a particular symbol are removed in the clean up, it subsequently causes a change in the largest symbol.

Displaying the knowledgebase

Having described two methods in the previous section on the same topic, what is listed below is another method that is mostly of interest to a developer.

- **void print(ostream &os)**

  This method prints the knowledgebase to any kind of an output stream. It prints the word, its status and the answer of all the words available in the knowledgebase.

The methods described above summarizes the most important methods from the view of a developer.

3.3 Methods Specifications

The section supplies a comprehensive list of all the methods used in the knowledgebase along account of their particulars. The section is divided based on on the class that the methods belong to.

3.3.1 Class - `node`

1. **get_next**

   `node* get_next(node * current_child)`

   parameters:

   current_child - The current child

   description:

   Returns the next node

2. **Constructor**

   `node(knowledgebase * base)`

   parameters:

   base - the name of the knowledgebase

   description:

   Creates the root node of the knowledgebase and sets its `parent` to NULL

3. **get_parent**

   `node * get_parent()`

   parameters:

   –

   description:

   Returns the parent of the node that calls this function

4. **find_child**

```
node * find_child(int label)
```

parameters:

label - the label that the child node must contain

description:

Returns the node that contains the label specified as the parameter

5. **find_descendant**

```
node * find_descendant(list<int>::iterator infix_start,
list<int>::iterator infix_limit)
```

parameters:

infix_start - the word or symbol that is the starting point
infix_limit - the word which is to be found

description:

Returns the node that represents the word infix_limit

6. **find_or_create_child**

```
node * find_or_create_child(int label)
```

parameters:

label - the label that the child node must contain

description:

Returns the node that contains the label specified as parameter. If not found, it creates such a node and returns it.

7. **serialize_subtree**

```
void serialize_subtree(basic_string<int32_t> & into)
```

parameters:

`into` - the string that contains the serialized trees of the knowledge-base

description:

Converts the subtree into `String` and appended to `into`. This method is used during serialization

8. **deserialize_subtree****

```
bool deserialize_subtree(basic_string<int32_t>::iterator & it,
basic_string<int32_t>::iterator limit, int & count)
```

parameters:

`it` - iterator to iterate over the string containing the word
`limit` - the last word that the subtree contains `count` - `integer` to count the subtrees

description:

Returns `true` after deserializing the subtrees. Returns `false` if `it` and `limit` are equal.

9. ## get_selfptr

`node * get_selfptr()`

parameters:

–

description:

A self pointer that returns its own node.

10. ## max_child_count

`int max_child_count()`

parameters:

–

description:

Returns the maximum number of children existing in the knowledgebase. If it returns "n", it implies that there may exist [0..n] suffixes.

11. ## has_specific_suffix

`bool has_specific_suffix(answer specific_answer)`

parameters:

`specific_answer` - the answer that needs to be compared

description:

Checks if a specific suffix/word has a specific answer. Returns `true` if such a case exists, otherwise returns `false`.

12. ## get_label

`int get_label()`

parameters:

–

description:

Returns the label of this node (which is the last symbol of the word that this node represents).

13. ## get_word

`list<int> get_word()`

parameters:

–

description:

Returns the word that this node represents.

14. ## mark_required

`bool mark_required()`

parameters:

–

description:

Returns `true` if answer to this node is required. Returns `false` if the answer is already known.

15. ## is_required

`bool is_required()`

parameters:

–

description:

Returns `true` if this node is marked unknown and required, `false` otherwise.

16. ## is_answered

`bool is_answered()`

parameters:

–

description:

Returns `true` if this node is already answered, `false` otherwise.

17. ## set_answer

`bool set_answer(answer ans)`

parameters:

`ans` - answer that must be set.

description:

Returns `true` if the node is already answered and the answer is same as `ans`, otherwise `false`. If the node is not answered already, then it returns `true` after setting `ans` as the answer.

18. ## get_answer

```
answer get_answer()
```

parameters:

–

description:

Returns the answer of this node.

19. ## no_subqueries

```
bool no_subqueries(bool check_self = true)
```

parameters:

`check_self` - Set to true. Used to check if the status of this node is marked required.

description:

Returns `true` if there are any queries with this node as the prefix, `false` otherwise. Also returns `false` if this node is not answered and marked required.

20. ## different

```
bool different(node * other)
```

parameters:

`other` - a node whose answer needs to be compared to the current node.

description:

Returns `true` if this node and `other` node have the same answer, `false` otherwise.

21. ## recursive_different

```
bool recursive_different(node * other, int depth)
```

parameters:

`other` - a node whose answer needs to be compared to the current node. `depth` - the depth of the tree that indicates the length of the word.

description:

Compares the answers of the two nodes and their children up to a level specified by `depth`. Returns `true` if there are no inconsistencies in the answers, `false` otherwise.

22. ## is_prefix_of

`bool is_prefix_of(node*other)`

parameters:

`other` - a node whose answer needs to be compared to the current node.

description:

Returns `true` if this node is a suffix of the `other` node.

23. ## is_suffix_of

`bool is_suffix_of(node*other)`

parameters:

`other` - a node whose answer needs to be compared to the current node.

description:

Returns `true` if this node is a prefix of the `other` node.

24. ## get_memory_usage

`unsigned long long int get_memory_usage()`

parameters:

–

description:

Returns the size of the memory used by this subtree.

25. ## ignore

`void ignore()`

parameters:

–

description:

Changes the status of this node to NODE_IGNORE.

26. ## cleanup

`bool cleanup()`

parameters:

–

description:

Returns `true` after deleting all the branches that has status as NODE_IGNORE.

3.3.2 Class - `iterator`

1. ## iterator

`iterator()`

parameters:

–

description:

Sets this node to NULL and initializes an iterator to the last node that is marked required.

2. ## iterator

`iterator(const iterator & other)`

parameters:

–

description:

Sets the values of this node to those of the `other` node.

3. ## iterator**

`iterator(bool queries_only, typename list<node*>::iterator currentquery, node * current, knowledgebase * base)`

parameters:

`queries_only` - a boolean variable that is `true` if queries exist and `false` if there are no queries in the knowledgebase `currentquery` - iterator to iterate over queries `current` - the node under consideration `base` - the knowledgebase that is being processed.

description:

Sets the values of this node to those specified in the argument.

4. **operator++**

   `iterator & operator++()`

   parameters:

   –

   description:

   Operator overloading applied to "++". Creates an iterator that points to the next query and returns the node

5. **operator++ ****

   `iterator operator++(int foo)`

   parameters:

   –

   description:

   –

6. **is_valid()**

   `bool is_valid()`

   parameters:

   –

   description:

   Returns `true` if the current iterator is not NULL, `false` otherwise.

7. **operator***

   `node & operator*()`

   parameters:

   –

   description:

   Returns the pointer to the current iterator.

8. **operator->**

   `node * operator->()`

   parameters:

   –

   description:

Returns the current iterator.

9. **operator=**

`iterator & operator=(const iterator & it)`

parameters:

`it` - an iterator

description:

Creates an iterator with the values of iterator `it` and returns this.

10. **operator==**

`bool operator==(const iterator & it)`

parameters:

`it` - an iterator

description:

Returns `true` if the current iterator is same as `it`, `false` otherwise.

11. **operator!=**

`bool operator!=(const iterator & it)`

parameters:

`it` - an iterator

description:

Returns `true` if the current iterator is not equal to `it`, `false` otherwise.

3.3.3 Class - `knowledgebase`

1. **knowledgebase**

`knowledgebase()`

parameters:

–

description:

Creates a knowledgebase with root as NULL.

2. **knowledgebase**

`~knowledgebase()`

parameters:

–

description:

Deletes the knowledgebase by deleting the root.

3. ## clear

`void clear()`

parameters:

–

description:

Deletes the existing knowledgebase and creates a new knowledgebase containing only the root.

4. ## clear_queries

`void clear_queries()`

parameters:

–

description:

Deletes all the nodes that are marked as queries.

5. ## undo**

`bool undo(unsigned int count)`

parameters:

`count` -

description:

Used to undo the last operation.

6. ## get_memory_usage

`unsigned long long int get_memory_usage()`

parameters:

–

description:

Returns the memory used by the knowledgebase.

7. ## is_answered

`bool is_answered()`

parameters:

–

description:

Returns `true` if there are no nodes marked required, `false` otherwise.

8. **is_empty**

`bool is_empty()`

parameters:

–

description:

Returns `true` if there are no nodes marked required and answered (the tree is empty), `false` otherwise.

9. **count_nodes**

`int count_nodes()`

parameters:

–

description:

Returns the number of nodes present in the knowledgebase.

10. **count_answers**

`int count_answers()`

parameters:

–

description:

Returns the number of nodes that are already answered in the knowledgebase.

11. **count_queries**

`int count_queries()`

parameters:

–

description:

Returns the number of nodes that are marked required in the knowledgebase.

12. **count_resolved_queries**

`int count_resolved_queries()`

parameters:

–

description:

Returns the number of answered nodes that were once marked required.

13. **reset_resolved_queries**

`void reset_resolved_queries()`

parameters:

–

description:

Resets the number of resolved queries to zero.

14. **get_largest_symbol**

`int get_largest_symbol()`

parameters:

–

description:

Returns the largest symbol that is present in the knowledgebase. Essentially, this returns the number which is one less than the alphabet size.

15. **check_largest_symbol**

`int check_largest_symbol()`

parameters:

–

description:

Adjusts the largest symbol present in the knowledgebase and returns it.

16. **print**

`void print(ostream &os)`

parameters:

`os` - an output stream.

description:

Prints the knowledgebase on the screen. Prints the word, status and the answer of all the words stored in the knowledgebase.

17. ## tostring

`string tostring()`

parameters:

–

description:

Used to return a string stream for serialization.

18. ## generate_dotfile

`string generate_dotfile()`

parameters:

–

description:

Generates the ".dot" file of the knowledgebase for graphical representation.

19. ## serialize

`basic_string<int32_t> serialize()`

parameters:

–

description:

Returns a string which represents the complete knowledgebase.

20. ## deserialize

`bool deserialize(basic_string<int32_t>::iterator &it, basic_string<int32_t>::iterator limit)`

parameters:

`it` - iterator to iterate over the string containing the word. `limit` - iterator that points to the last word

description:

Returns `true` if the string was deserialized to knowledgebase successfully, `false` if the deserialization failed.

21. deserialize_query_acceptances

```
bool deserialize_query_acceptances(basic_string<int32_t>::iterator
&it,
basic_string<int32_t>::iterator limit)
```

parameters:

`it` - iterator to iterate over the string containing the word. `limit` - iterator that points to the last word

description:

Returns `true` after answering all the queries in the knowledgebase from a single serialized data.

22. create_query_tree

```
knowledgebase * create_query_tree()
```

parameters:

–

description:

Returns a tree created by adding to this tree, all the words marked as required in the knowledgebase.

23. get_queries

```
list<list<int> > get_queries()
```

parameters:

–

description:

Returns a list of list of `integer` that consists of all the queries existing in the knowledgebase.

24. merge_knowledgebase

```
bool merge_knowledgebase(knowledgebase & other_tree)
```

parameters:

`other_tree` - the tree to be merged.

description:

Returns `true` if the current tree could be merged with the `other_tree`, `false` otherwise. Two trees can be merged only if they are consistent. This method merges only answered information, it does not merge the queries.

25. ## add_knowledge

`bool add_knowledge(list<int> & word, answer acceptance)`

parameters:

`word` - the word to be added to the knowledgebase. `acceptance` - the classification of the word.

description:

Returns `true` if the knowledge for this word does not already exist and is successfully added to the knowledgebase, `false` otherwise.

26. ## add_query

`int add_query(list<int> & word, int prefix_count = 0)`

parameters:

`word` - the word/query to be added to the knowledgebase. `prefix_count` - initialized to zero. It is the count of all the prefixes that can be formed with the word.

description:

Creates the query and the necessary prefixes (which will also be marked as a query) and returns the total number of queries formed.

27. ## resolve_query

`bool resolve_query(list<int> & word, answer & acceptance)`

parameters:

`word` - the word/query to be added to the knowledgebase. `acceptance` - the classification of the word.

description:

If the word is already known and is answered, then the answer is assigned to `acceptance` and returns `true`, otherwise returns `false`.

28. ## resolve_or_add_query

`bool resolve_or_add_query(list<int> & word, answer & acceptance)`

parameters:

`word` - the word/query to be added to the knowledgebase. `acceptance` - the classification of the word.

description:

Returns `true` if the word is already known and answered, else marks the word as required and returns `false`.

29. **get_nodeptr**

`node* get_nodeptr(list<int> & word)`

parameters:

`word` - a word.

description:

Returns the node that represents this word.

30. **get_rootptr**

`node* get_rootptr()`

parameters:

–

description:

Returns the root.

31. **begin**

`iterator begin()`

parameters:

–

description:

Returns an iterator that begins at the root node.

32. **end**

`iterator end()`

parameters:

–

description:

Returns an iterator which is used to point to the last node.

33. **qbegin**

`iterator qbegin()`

parameters:

–

description:

Returns an iterator that begins at the first node that is marked required.

34. ## qend

`iterator qend()`

parameters:

–

description:

Returns an iterator which is used to point to the last node that is marked required.

4 Learning Algorithms

Learning algorithms try to construct a conjecture from available information stored in the knowledgebase. A conventional way to distinguish learning algorithms is to group them into *online* and *offline* algorithms. Online learning techniques are capable of actively asking queries to some kind of teacher who is able to classify these queries. Offline algorithms, on the other hand, are passively provided with a set of classified examples from which they have to build the conjecture.

Online Algorithms

The online learning algorithm follows the most common model of Minimally Adequate Teacher (MAT) that involves some kind of a teacher to *resolve* two types of queries *membership queries* and *equivalence queries*. Online learning algorithms build the conjecture by actively asking queries to a *teacher* (i.e. a user application).
The teacher is required to resolve a membership query by providing the classification of the given word. Equivalence queries check whether a derived conjecture is an equivalent description of the target language to be inferred.

- The algorithm runs on iteration which begins at making an *advance* where in the algorithm tries to compute a conjecture with information available in the knowledgebase.
- This leads to the rise of membership queries if no conjecture was created. These queries are resolved by the teacher, answer added to the knowledgebase and the algorithm continues the iteration.
- On the other hand, if a conjecture was computed, it is presented to the teacher. The algorithm terminates if the conjecture is correct. Otherwise, the iteration continues after the teacher renders a counter example.

Offline Algorithms

Offline algorithms, in contrast to the online variant, finds the smallest DFA consistent with a given set of classified words. The algorithm is provided with a set S of classified words (called samples) and the algorithm derives a conjecture which conforms to these samples.

The working of this algorithm follows a simple two step procedure.

- The knowledgebase is furnished with all samples.

- The algorithm is then made to advance to compute the conjecture conforming to the samples.

List of Algorithms

As of August 15, 2010, `libALF` implements seven algorithms for both deterministic and non deterministic automata as listed below.

Online Algorithms

1. **Angluin L*** [**?**] [**?**] [**?**] (..)
2. **NL*** [**?**] [**?**] (..)
3. **Kearns/Vazirani** [**?**] [**?**] [**?**] (..)

Offline Algorithms

1. **RPNI** [**?**] [**?**] [**?**] [**?**] (..)
2. **Biermann** [**?**] (..)
3. **DeLeTe2** [**?**] (..)

4.1 Methods - User Perspective

In this section, we describe methods that are important for using `libALF`. The following material elaborates on initializing the learning algorithm, its association with knowledgebase and building the conjecture. Like the knowledgebase, learning algorithm also supports serialization and deserialization. Other methods that enable the user to work on the statistics are also described.

Initializing the Learning Algorithm

Given below is an example of the RPNI algorithm is initialized.

- **RPNI(knowledgebase<answer> * base, logger * log, int alphabet_size)**

  learning_algorithm(*parameters*)

  The constructor of all the learning algorithms follow such an initialization. It sets the pointer to the knowledgebase, the logger and the alphabet size.

Note: Other algorithms may need more than the above mentioned arguments since they depend on the working of the algorithm itself. However, the above example shows the minimal set of arguments required for any learning algorithm.

Working with Alphabet Size

1. **learning_algorithm::void set_alphabet_size(int alphabet_size)**

   The method sets the alphabet size for computing the conjecture. This method is used only during the initial setting of the learning algorithm.

2. **learning_algorithm::void increase_alphabet_size(int new_asize)**

   The method increases the size of the alphabet to a new value.

3. **learning_algorithm::int get_alphabet_size()**

   Returns the alphabet size of the conjecture.

Knowledgebase in Learning Algorithm

1. **learning_algorithm::void set_knowledge_source(knowledgebase <answer> * base)**

   The method sets the source (the knowledgebase) which consists of all the membership information to the learning algorithm.

2. **learning_algorithm::knowledgebase<answer> * get_knowledge_source()**

   Returns the pointer to the knowledgebase which is currently the source of membership information.

Advancing

- **learning_algorithm::conjecture * advance()**

  The method returns a conjecture if enough information is available in the knowledgebase to construct one. If not, it returns NULL but produces membership queries that are stored in the knowledgebase.

Adding Counter-example

- **virtual bool add_counterexample(list<int>)**

  The method is used by *online* algorithms when a computed conjecture is declined by the teacher, i.e. when the equivalence query is answered negative. The counter example provided is first processed by the learning algorithm which marks it as a membership query and is added to the knowledgebase.
  This method is used only by an *online* algorithm. For *offline* algorithms, this method is a stub.

Synchronization with Knowledgebase

As discussed in the previous chapter, the knowledgebase supports *undo* operation. When an undo operation has been performed, the learning algorithm must be synchronized to the knowledgebase, failing to which may generate erroneous output. We use the following method to synchronize the learning algorithm with the knowledgebase. This method must be called after each undo operation.

- **learning_algorithm::bool sync_to_knowledgebase()**
  The method checks the knowledgebase and changes its internal data to be synchronized with the knowledgebase and returns `true`. If it returns `false`, the algorithm is in an undefined state and must not be used anymore.
  Note: This method should be called after each undo operation performed in the knowledgebase.

On the other hand, the knowledgebase need not necessarily allow the undo operation. Hence we use the following method to check the same.

- **learning_algorithm::bool supports_sync()**
  Returns `true` if undo operations on the knowledgebase are allowed, otherwise returns `false`.

Working with Loggers and Normalizers

1. **virtual void set_logger(logger * l)**

   If the value of "l" is not NULL, then it is set as the logger. Otherwise, the logger is set to "ignore" which implies that no logger exists.

2. **virtual void set_normalizer(normalizer * norm)**

   Sets the normalizer to the one pointed by the argument "norm".

3. **virtual void unset_normalizer()**

   Sets the normalizer to NULL.

Working with Statistics

1. **virtual memory_statistics get_memory_statistics()**

   Returns the memory statistics.

2. **virtual timing_statistics get_timing_statistics()**

   Returns the timing statistics which is stored in the variable "current_stats".

3. **virtual void enable_timing()**

   Enables the maintenance of timing statistics by setting the "do_timing" variable to be `true`.

4. **virtual void disable_timing()**

   Disables the maintenance of timing statistics by setting the "do_timing" variable to be `false`.

5. **virtual void reset_timing()**

   The "current_stats" is reset.

Serialize and Deserialize

1. **virtual basic_string<int32_t> serialize()**

   The method returns a `String` composed of `integer` containing the serialization of the state of the learning algorithm. The data can be loaded with the following method.

2. **virtual bool deserialize(basic_string<int32_t>::iterator & it, basic_string <int32_t>::iterator limit)**

   Restores the data of a serialized learning algorithm. The current state of the learning algorithms is discarded.
   The method returns `true` if the deserialization was successful. Otherwise, returns `false`.

4.2 Methods - Developer's Perspective

A developer's perspective of learning algorithm mainly centers on the how the algorithm advances and builds the conjecture. The section describes "advance()" and its associated methods. B

- **virtual conjecture * advance()**

  The most important method towards building the conjecture. The method first gathers all knowledge available and tries to derive a conjecture. If a conjecture is formulated, it is returned. If a conjecture was not produced, the knowledge having unknown classification is marked *required* and NULL is returned.

The following code snippet shows how advance works. The methods used internally are described below.

```
1 virtual conjecture * advance()
2 {{{
3         conjecture * ret = NULL;
4         //When no knowledgebase is found.
5         if(my_knowledge == NULL) {
6           (*my_logger)(LOGGER_ERROR, "learning_algorithm::advance(): no knowledg
```

```
7                                       return false;
8             }
9
10          start_timing(); // For statistics
11          if(complete()) {
12            ret = derive_conjecture();
13                // When a conjecture could not be dervied
14                if(!ret)
15                 (*my_logger)(LOGGER_ERROR, "learning_algorithm::advance(): d
16            }
17          stop_timing();
18          return ret;
19 }}};
```

1. **virtual bool complete()**

   The method is used by the learning algorithm to complete their internal data structures such that a conjecture can be derived from it. Returns `true` if all internal data is available. Returns `false` if there is missing knowledge.

2. **virtual conjecture * derive_conjecture()**

   The method derives a conjecture from the given data structure available in the knowledgebase.

One other method used by learning algorithms is the conjecture_ready method to check if a conjecture can be derived successfuly or not.

- **virtual bool conjecture_ready()**

  Returns `true` if a conjecture can be constructed without any further queries. Otherwise, returns `false`.

5 Loggers & Statistics

5.1 logger

To ease application development and debugging, `libALF` provides two components - Loggers and Statistics.

A **Logger** is an adjustable logging facility that an algorithm can write to. One may insert different category of messages in the logger and is of substantial use in application debugging. When a learning algorithm is initialized, a logger is associated with it along with the knowledgebase. `libALF` provides flexible logger implementations for the user. A learning algorithm can either use an output stream or a buffer as the logger. On the contrary, one may also choose to work without a logger.

Statistics refers to the statistical data that can be acquired by evaluating the learning procedure. Information about the memory usage, queries produced, time taken for computing conjecture and other details may serve as base for analysing the learning algorithm in various cases.

5.2 Loggers

A learning algorithm may write different types of messages to the logger. A logger has to be associated with the knowledgebase during the initialization of the learning algorithm.

Example of how to set a logger

The following code snippet shows to how to create an instance of a logger and associate it with the learning algorithm along with knowledgebase and alphabet size.

```
1 // getting the alphabet size
2 int alphabet_size = get_AlphabetSize();
3
4 // create instance of knowledgebase
5 knowledgebase<bool> base;
6
7 // create instance of a buffered logger.
8 buffered_logger bufflog;
9
10 // Create learning algorithm (Angluin L*)
11 // with a logger - bufflog and alphabet size - alphabet_size
12 angluin_simple_table<bool> algorithm(&base, bufflog, alphabet_size);
```

5.2.1 The concept of Loglevel

To achieve a good organization of these messages written to the logger, the messages are categorized with different labels. A `loglevel` is a marker indicating the priority of types of messages to be written into the logger. Setting up a `loglevel` ensures only messages having priority of that level and higher are written to the logger while messages having lower priority are discarded or skipped.

The category of messages is initialized with a *enum* type variable `logger_loglevel`.

- **LOGGER_INTERNAL=0** ; (An internal method)
- **LOGGER_ERROR = 1** ; All log messages that describe a non-recoverable error are marked with this.
- **LOGGER_WARN = 2** ; Messages describing a state or command that is erroneous but may be ignored under most conditions.
- **LOGGER_INFO = 3** ; Any information that does not describe an erroneous condition.
- **LOGGER_DEBUG = 4** ; Messages that may help debugging of libalf.(Most likely removed before release version).
- **LOGGER_ALGORITHM = 5** ; (Do not use this as minimal loglevel)

For instance, setting up a `loglevel` of "2" will make the learning algorithm write warning and error messages (level 1 and 2) to the logger while messages labelled with lower priority (3 and 4) and discarded.

5.2.2 The Logger class

The main class that consists of attributes and methods to implement the logger.

Attributes

It consists of two attributes that every type of logger makes use of.

1. **enum logger_loglevel minimal_loglevel** - a minimal setting of the loglevel.
2. **bool log_algorithm** - An boolean variable to indicate if a logger is to be associated with an algorithm or not.

Methods

The following methods are defined in the class.

1. **void set_minimal_loglevel(enum logger_loglevel minimal_loglevel)**
   Sets the minimum logger level using the `loglevel` attributes.

2. **void set_log_algorithm(bool log_algorithm)**
   The method sets logger for the algorithm if the argument is `true`. It ignores the logger if the parameter is `false`.

3. **virtual void operator()(enum logger_loglevel, string&)** and **virtual void operator()(enum logger_loglevel, const char* format, ...)**
   The method takes the logger type and message as parameters for entry to the logger. If other variables also need to be used, it can be done so using the second method.

Example of a learning algorithm writing message to the logger - Developer's View

```
1 if(my_knowledge == NULL)
2 {
3 (*my_logger)(LOGGER_ERROR, ''learning_algorithm::advance()
4       no knowledgebase was set!\n'');
5 return false;
6 }
```

The above code snippet is an extraction from the "advance()" method of a learning algorithm. When no knowledgebase is set to the algorithm, it enters the message "learning_algorithm::advance(): no knowledgebase was set" to the log "my_logger" and marks it as an error with "LOGGER_ERROR".

All three types of loggers are implemented with the respective classes, `ignore_logger`, `ostream_logger` and `buffered_logger`. All the classes inherit the `logger` class.

5.2.3 Types of Loggers

ignore_logger

A class that does not consist of any methods. In this case, the logger simply discards all messages.

ostream_logger

The class consists of method to write the message to an output stream.

- **ostream_logger::ostream_logger(ostream *out, enum logger_loglevel minimal_loglevel, bool log_algorithm = true, bool use_color = true)**

The method creates an output stream for the logger. The parameter `*out` points to the output stream and the minimal loglevel indicates the initial setting. The parameter `log_algorithm` is set to `true` by default since the logger will be used by the algorithm. The parameter `use_color` is set to `true` so that on a console output, you may view the messages in different colors!

buffered_logger

The class consists of methods for setting a buffer as a log (typically a string). It should be noted that the messages passed to the buffer will not be available until it is received and flushed explicitly.

1. **buffered_logger::buffered_logger(enum logger_loglevel minimal_loglevel, bool log_algorithm = true)**
   The method sets the buffered logger with the minimal loglevel. `log_algorithm` is set to `true`.

2. **buffered_logger::string * receive_and_flush()**
   The method receives and flushes the buffered stream.

5.3 Statistics

The Statistics component provides variable statistic types through a set of classes which consist of methods to map string to a value. Using this component, any statistical information can be generated and the value can be of types: int, double, bool or string. Casting between any of the above forms is also allowed but a wrong reference will give out an exception. The component also supports serialization and deserialization. The classes and methods from this component are described below.

statistic_type

The global enum variable `statistic_type` defines the values that can be used. It consists of `UNSET`, `INTEGER`, `DOUBLE`, `BOOL`, `STRING`.

statistic_data_bad_typecast_e

The constructor of this class throws the exception when typecast error occurs.

Class Statistic_Data

The class provides many functions for working with the `statistic_type` variable. The methods provided are given below.

1. **Get and Set methods**
   The class provides these methods to set and get appropriate values on the variables.

The methods are of the form `get_type()` and `set_type()` where *type* is one of *integer*, *double*, *bool* or *string*.

2. **Operator overloading functions** There are methods that overload operators for assigning values to the variables. when a typecast error occurs, exception is thrown.

generate_statistic

The class extends `map<string, statistic_data>` and provides methods to map the `statistic_data` to a key string. In essence, a statistical value is mapped to a statistical property (which is named as *key*) The methods are as follows.

1. **Get and Set methods**
   The class provides methods to set value to a property or get the value from the property. The method is of the form `inline void set_type_property(const string & key, type value)` where the type is one of *integer*, *double*, *bool* or *string*. Naturally, these methods use the set and get methods from `statistic_data` class.

2. **inline void remove_property(const string & key)**
   The method is used to remove the property *key* completely.

3. **inline void unset_property(const string & key)**
   The method unsets this property but does not remove from memory.

6 Filters & Normalizers

A knowledgebase can be associated with **filters** which can exploit domain specific properties and by that actively reduce the number of queries to the teacher during the learning phase. Such filters can be composed by logical connectors (and, or, not).
In contrast, **Normalizers** recognize equivalent words in a domain-specific sense to reduce the amount of knowledge that has to be stored.
Both components can be serialized and deserialized.

6.1 Filters

A knowledgebase can be associated with more than one filter through the logical connectors. A filter essentially works on the word and tries to resolve its classification. Thus, it reduces the number of queries that are asked to the teacher. You may also connect the filters with more than one logical connectors and, or and not.

A simple example Lets assume that you have associated two filters with learning algorithm. You wish that the classification of word is 1 (accepted) if only both the filters give positive results and is 0 (rejected) if even one of the filters give a negative result. `libALF` lets you connect these filters with logical connector "and" so that the above operation is precisely performed.
In the following material, we discuss the methods associated with Filters are implemented and how to work with it.

6.1.1 Class - filter

It is the main class that defines the types of filter.

Attributes - filter types

An *enum* type variable `type` is used to define the filter type.

- **FILTER_NONE = 0** ; No filter associated
- **FILTER_AND = 1** ; Filter type *and*
- **FILTER_OR = 2** ; Filter type *or*
- **FILTER_NOT = 3** ; Filter type *not*
- **FILTER_ALL_EQUAL = 4** ; Filter type for equal words

- **FILTER_REVERSE = 100** ; Filter type handling reverse of a word.
- **FILTER_IDENTITY = 200** ; Identity filter **

6.1.2 Class - filter_subfilter_array

The subfilter is an array of filters that is used to associate more than one filter to the knowledgebase. The class inherits the filter class.

Attributes

- **list<filter<answer>*> subfilter_array**
  A list of all the subfilters associated with the knowledgebase.

Methods

- **virtual void free_all_subfilter()**
  The method to erase all subfilters.
- **virtual void add(filter<answer> *f)**
  Method to add a filter into the array.
- **virtual void remove(filter<answer> *f)**
  Method to remove a filter from the array.

6.1.3 Important Methods of all Filters

All the filters (including the logical connectors) are associated with the following important methods that execute its feature.

1. **filter::void free_all_subfilter()**
   Method to erase all the subfilters (logical connectors) associated with the knowledgebase.
2. **filter::virtual enum type get_type()**
   The method returns the type of filter.
3. **filter::virtual bool evaluate(knowledgebase<answer> & base, list<int> & word, answer & result)**
   The main method to evaluate the word with the associated filter. It returns `true` if the word was evaluated successfully and the answer is stored in the parameter `result`. It returns `false` otherwise.

6.1.4 Types of Filters

1. **filter_and**
   It is the logical connector `and` that can be associated with any two filters. The `evaluate` method returns `true` only if the answer of the word can be determined by both the associated filters and the answer obtained is the same.

2. **filter_or**
   It is the logical connector `or` that can be associated with any two filters. It returns `true` if the word if at least one of the filters provide an answer for the word.

3. **filter_not**
   It is the logical connector `not` that can be associated with a filter. It returns `true` if the word was answered by the filter and sets the `result` to the `not` of the derived answer.

4. **filter_all_equal**
   Filter to check if the result is the same in all filters. Returns `true` if the subfilter array is non-empty, and if all filters can evaluate a word and produce the same answer.

5. **filter_reverse**
   The filter reverses the word and sends to all subfilters. Returns `true` if the reversed word can be evaluated by the subfilters and the answer is stored to the `result`.

6. **filter_identity**
   This is a filter that tries to identify if the answer to the word is already available in the knowledgebase. It returns `true` if it exists and is answered already (after setting the answer to `result`), `false` otherwise.

6.2 Normalizers

As mentioned before, Normalizers are means to reduce memory consumption during the learning phase. A normalizer defines a domain-specific equivalence relation (...) over all words and only stores data for one representative of each equivlance class. This means that the data for equivalent queries is only queried and stored once. Apart from reducing the memory consumption, the number of queries are also reduced. By subtyping the respective interface, a user can easily define her own domain-specific optimizations. Normalizers are extensively used by Angluin Algorithm.

6.2.1 Working Overview

Normalizers are based on the concept of Message Sequence charts (MSC). An MSC consists of a set of processes (or nodes) P, set of messages M, a set E of events which is partitioned into a set S of *send* events and a set R of *receive* events In the normalizer, the send and receive events are organized as odd and even pairs respectively. Odd

represents the send event of the sending process and even represents the receive event of the receiving process. The processes pass the symbols of the word as the messages and the normalizer tries to identify if the word belongs to the MSC's language. If yes, it returns the normalized word available in the knowledgebase. In this way, the need to store another word is eliminated.
Normalizer also supports the serialize and deserialize features.

6.2.2 Methods

The methods governing this component is described below.

User Perspective

From the user perspective, the two important methods that perform the normalization are given below. You can simply associate a normalizer with your Angluin learning algorithm.

- **list<int> normalizer_msc::prefix_normal_form(list<int> & w, bool & bottom)** This is the method that normalizes the given input. The parameter *w* is the word. The method creates the MSC and then attempts to normalize the word. If successful, it returns the normalized word. Otherwise *bottom* is set to true and it returns the BOTTOM_CHAR.

- **list<int> normalize_msc::suffix_normal_form(list<int> & w, bool & bottom)** This method performs the same operation as the previous method except that a reversed MSC is created.

Developer Perspective

The MSC is stored in the form of a graph. The important attributes are as follows.

1. **vector<int> total_order** - Denotes the total order. A total order is nothing but the temporal order of the messages that are available for the events in the buffer.

2. **vector<int> process_match** - A relation that matches the events to a process.

3. **vector<int> buffer_match** - A relation matching event to a buffer. The messages are queued in send and receive buffers.

4. **int max_buffer_length** - The maximum number of messages in a buffer.

5. **list<msc::msc_node*> graph** - Variable for storing the graph

6. **queue<int> * buffers** - The buffers.

7. **unsigned int buffercount** - The count of number of buffers used.

8. **unsigned int label_bound** - The label bound is essentially the alphabet size known by the normalizer. A message must be in [0, label_bound).

The important methods are given below.

1. **void graph_add_node(int id, int label, bool pnf)**
   The method is used to add node to the graph. This method is used by the prefix and suffix normal form methods. There are two things to be noted here. First, the process connection for prefix_normal_form (PNF), a connection is made to the node from other youngest node with same process that is not connected. For suffix_normal_form (SNF), a connection is made from node to other youngest node with same process that is not connected. The connection between the messages is again ruled by PNF or SNF. For PNF, a receiving event is connected to oldest corresponding send-event that is not connected and for SNF, a sending event is connected from oldest corresponding send-event that is not connected.

2. **void normalizer_msc::clear_buffers(list<int> word)**
   Clears all buffers that this word has touched.

3. **bool normalizer_msc::check_buffer(int label, bool pnf)**
   The method checks if the message (label) can be put into its buffer or taken from its buffer. If yes, returns `true`. On the otherhand, if its buffer is full or another message is at the head of the buffer, it returns `false`.

4. **int normalizer_msc::graph_reduce(bool pnf)**
   This is the actual method used by the PNF and SNF methods for normalization.

5. **inline void connect_buffer(msc_node * other)**
   Method to connect the buffers.

7 jALF Java Library

The jALF Java library, as you may have come across in various sections in the previous chapters, is the Java implementation of libALF. However, jALF is not a standalone library. It is implemented as calls pointing to the C++ libALF objects.
The jALF library can be used either locally through JNI or remotely from a server using the dispatcher. The important point to be noted here is that a few features in jALF are not identical to libALF. The differences exist at different levels and will be described in this section along with how to use jALF and its developer's perspective.

7.1 Source Code Structure

The jALF implementation can be found in /libalf/jalf folder of the libALF package. The jALF package information is as follows.

- src - The folder contains C++ methods of JNI calls that forwards the calls to libALF.
- include - The header files generated using javah command.
- java/src/de/libalf - The files in this folder are the interfaces to the native methods.
- java/src/de/libalf/jni - The java native methods for JNI.
- java/src/de/libalf/dispatcher - The java native methods for dispatcher.

7.2 jALF- User Perspective

In this section we will introduce how to use jALF and explain its features.

Data Structures

The data structure used in jALF is mostly similar to that of libALF. An important difference lies in the data structure of the knowledgebase. While it is possible to store arbitrary value types for classification in libALF, it is possible to use only boolean values (true or false) can be used for storing classification information in jALF.
However, other differences in data structures are handled entirely by the jALF itself and the task does not burden on the user. For instance in libALF, words were represented as list of integers. Similarly jALF uses integer arrays, or more precisely, jintArray to

represent the words and `LinkedList` for list of words. `jALF` automatically performs the conversion during the execution.

The jALF Factory

Unlike `libALF` the components are not entirely free but belong to what is called a Factory class. From an abstract point of view, this factory can be imagined as a roof under which the components can be declared and used. The following code snippet shows an example of how to use the factory class. (Note: refer to the examples provided in the `libALF` website for the full program)

```
1  int[] words
2  boolean classification;
3  LibALFFactory factory = JNIFactory.STATIC;
4  alphabetsize = get_AlphabetSize();
5
6  //Factory created
7  Knowledgebase base = factory.createKnowledgebase();
8
9  /* Code to add knowledge to knowledgebase here */
10
11 LearningAlgorithm algorithm = factory.createLearningAlgorithm(
12                                    Algorithm.RPNI, base, alphabetsize);
13 //The algorithm is advanced
14 BasicAutomaton automaton = (BasicAutomaton) algorithm.advance();
15
16 //Output displayed
17 make_OutputFile(automaton.toDot());
```

As one can observe from above, the objects for knowledgebase and learning algorithm are created only through this factory class. Loggers and Normalizers also belong to the factory and must be initialized in the same way as the knowledgebase. After declaring the components under this factory, they can be used normally like in the C++ program. In this context, the user must understand and remember that `jALF` is a library that points to the objects of the C++ `libALF` library. Which means that although the components are declared under the factory, each component maintains a separate pointer to its corresponding C++ object. The object can be destroyed by calling the `destroy()` method and an exception is thrown when trying to access a destroyed object. And thus, the learning algorithm class provides you two extra methods compared to `libALF` C++ part, which are `remove_logger` and `remove_normalizer` which destroys the objects of logger and normalizer. The same can be created at a later point of time and can be attached to the learning algorithm through `set_logger` and `set_normalizer`. However, one has to note that `jALF` does not support IO logger. Therefore, you may choose to use only a buffered logger or ignore logger completely.
`jALF` also supports exceptions that helps user for debugging. For instance, an exception

is thrown when trying to add a counter example for an *offline* algorithm or when enough information is not provided during the creation of learning algorithm. The dispatcher can additionally give a protocol exception. (??)

7.3 jALF- Developer Perspective

The `jALF` library, in essence, are methods that forward the calls to the `libALF` library. (A short intro summary here - will be done after finalizing this part. Points about javah will be added in this part)

7.3.1 Naming Conventions

Before going into the details of the `jALF`, the text below briefs on the naming of the methods. The C++ part of the native methods, as a result of the `javah` command are written as

Java_de_libalf_jni_JNI[ClassName]_name_1of_1the_1method(***parameters*** **)**

The parameters consist of the JNI Environment variable, the java object and the parameters of the original method along with a pointer to the object of this method. For example, the method `void resolve_or_add_query` is coded as

Java_de_libalf_jni_JNIKnowledgebase_resolve_1or_1add_1query(JNIEnv *env, jobject obj, jintArray word, jlong pointer)

Here, *env* is the JNI environment variable, *obj* is the jobject, *word* represents the knowledge to be either resolved or added to the knowledgebase, pointer is the pointer to the C++ object.

7.3.2 JNIObject

The `JNIObject` is the root of all classes representing the JNI `libALF` C++ objects. Each JNIObject stores a 64 bit pointer variable that points to memory location of the C++ object. This ensures memory access is allowed on both 32 and 64 bit systems. Each `native` method call on C++ objects via the JNI interface has to provide a pointer to locate the object. This class is not initialized but its subclasses provide an `init` method to initialize a C++ object via the JNI interface and returns the memory address of the object. For instance, the native method `private native long init()` of the knowledgebase invokes the JNI interface to initialize a new C++ knowledgebase object without any parameters and returns the pointer to this object. The same exists for the dispatcher as `DispatcherObject.java`.
The `JNIObject` extends `LibALFObject` which is the interface that initializes the factory

and creates pointer to the C++ objects. And hence, the classes under the factory (knowledgebase, learning algorithm, logger and normalizer) implement a `destroy()` method to remove the pointer to the respective C++ object.

7.3.3 Automaton Tools

Two classes that are important for working with the automaton are described below.

- **BasicAutomaton**
  The BasicAutomaton class represents a deterministic or nondeterministic finite automaton as it is generated by the LibALF library.The automaton essentially consists of the set of *states* which is represented by `integer` between `0` and `numberOfStates` that work over an Alphabet set, set of *initial states* and *final states*. This class only stores the automaton but does not provide any functionality.
- **BasicTransition**
  Creates a new transition from source to destination, given the label of this transition.

7.3.4 Exceptions

As mentioned earlier, `jALF` supports exceptions that is derived from the interface `AlfException`. `jALF` throws an exception if an object has already been destroyed. This is handled by methods derived from the interface `AlfObjectDestroyedException`. To add more exceptions, simply include the methods in the corresponding interface and use it in the classes.

7.3.5 JNItools

The `jni_tools` provide methods useful especially for converting variables to JNI data structures. The methods provided are as follows.

- **jintArray basic_string2jintArray_tohl(JNIEnv *env, basic_string<int32_t> str)** The method is used to convert basic_string to jintArray. The function uses `ntohl` to convert the integer to host byte order.
- **jintArray basic_string2jintArray(JNIEnv *env, basic_string<int32_t> str)**
  Method to convert basic_string to jintArray.
- **jintArray list_int2jintArray(JNIEnv *env, list<int> l)**
  The method converts list of integers to jintArray.
- **jobject create_transition(JNIEnv* env, int source, int label, int destination)**
  The method creates an edge between the source node and the destination node with the prescribed label.

- **jobject convertAutomaton(JNIEnv* env, bool is_dfa, int alphabet_size, int state_count, set<int> & initial, set<int> & final, multimap<pair<int, int>, int> & transitions)**
  (??)