BULL: a Library for Learning Algorithms of Boolean Functions

Yu-Fang Chen and Bow-Yaw Wang

Academia Sinica, Taiwan

Abstract. We present the tool BULL (Boolean fUnction Learning Library), the first publicly available implementation of learning algorithms for Boolean functions. The tool is implemented in C with interfaces to C++, JAVA and OCAML. Experimental results show significant advantages of Boolean function learning algorithms over all variants of the L^* learning algorithm for regular languages.

Web-site: `http://bull.iis.sinica.edu.tw/`

1 Introduction

BULL is the first publicly available implementation of learning algorithms for Boolean functions. Three algorithms are implemented in the library. The classical CDNF algorithm infers Boolean functions over a fixed number of variables. The incremental CDNF+ and CDNF++ algorithms infers Boolean functions over an indefinitely number of variables. The library is implemented in C with C++, JAVA, and OCAML interfaces. Sample codes of C, C++, JAVA, and OCAML are distributed with the library. Users can adopt BULL by modifying them.

What is it? Learning algorithms for Boolean functions can be viewed as an efficient procedure to generate a *target* Boolean function only known to a teacher. This type of learning algorithms assume a teacher who answers queries about the target Boolean function. The learning algorithms acquire information from the answers to queries and organize them in a systematic way. In the worst case, learning algorithms will infer a target Boolean function within a polynomial number of queries in the CNF and DNF formula sizes of the target function.

Learning in formal verification. Since the work in [9], algorithmic learning has been applied to formal verification techniques such as specification synthesis [9], automated compositional verification [5], and regular model checking [6]. Most applications are based on the L^* automata learning algorithm for regular languages. The learning algorithm enumerates states explicitly. Its applications are hence inherently explicit [5], or use explicit automata as implicit representations of state spaces [6].

Why use Boolean learning. Implicit algorithms (e.g., SAT-based model checking) can greatly improve the capacity of various verification techniques. Similar improvements have also been reported in applications of the CDNF learning algorithm for Boolean functions. In [3], the learning algorithm is adopted to infer implicit contextual assumptions in automated compositional reasoning. It is shown that learning implicitly can tackle certain hard problems unattainable by traditional explicit algorithms. The CDNF algorithm is also applied to loop invariant generation. The learning-based framework can be much more efficient than conventional static analysis algorithms [7].

For regular languages, the learning algorithms are available in veteran tools (such as libalf [1] and learnlib [12, 11, 10]). Implementations of learning algorithms for Boolean functions however are still missing. Since it would take a considerable amount of time to understand and implement learning algorithms for Boolean functions, lack of publicly available tools could be an obstacle to develop related techniques in the research community. In order to lower the barrier to entry, we decide to develop the BULL library.

The Position of the Paper The Boolean learning project starts in 2009 and since then we tested different variants of the algorithms and data structures. Several of them indeed dramatically improved the performance, e.g., non-membership queries are introduced partly for performance reasons. However, since boolean learning is a new technique to most people in the community. We decided to spend the pages for a general introduction instead of technical details.

2 The BULL Library

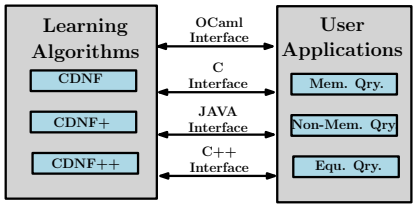

Fig. 1. System Architecture

Figure 1 shows the architecture of the BULL library. The core library contains three learning algorithms implemented in C. They are the CDNF [2], the CDNF+ [4], and the CDNF++ [4] algorithms. The CDNF algorithm assumes that the number of variables in the target Boolean function is known. The CDNF+ and the CDNF++ algorithms do not have this assumption. In addition to the learning algorithms, we also provide C++, JAVA (via JNI), and OCaml interfaces.

2.1 How to Use the Package

In order to adopt the learning algorithms in BULL, users have to play the teacher and answer queries posed by the algorithms. For the sake of presentation, let us assume that $f(x, y, z) = (x \wedge \neg y) \vee (x \wedge z)$ is the target Boolean function over variables x, y, and z. Consider the following sample queries from the learning algorithms:

1. A *membership* query on a partial assignment $\{(x, \mathit{false})\}$. On a membership query, the teacher checks if the target is satisfiable under the given assignment. Here the teacher answers *no* since $f(\mathit{false}, y, z)$ is not satisfiable.
2. A *non-membership* query on a parital assignment $\{(y, \mathit{true})\}$. On a non-membership query, the teacher checks if the negation of the target is satisfiable under the assignment. For this example, the teacher answers *yes*.
3. An *equivalence* query on a conjecture $f'(x, y, z) = x \wedge y$. On an equivalence query, the teacher answer *yes* if the given formula is equivalent to the target. Otherwise, she returns an assignment as a counterexample. For this example, the teacher may return the assignment $\{(x, \mathit{true}), (y, \mathit{true}), (z, \mathit{false})\}$ since $f'(\mathit{true}, \mathit{true}, \mathit{false}) \neq f(\mathit{true}, \mathit{true}, \mathit{false})$.

Different learning algorithms pose different types of queries. Table 1 shows the differences among the three learning algorithms in BULL. The CDNF algorithm

	Num. of Vars.	Mem. Qry,	Non-Mem. Qry.	Equ. Qry.
CDNF	known	√		√
CDNF+	unknown	√		√
CDNF++	unknown	√	√	√

Table 1. Features of Algorithms

assumes the number of variables in the target Boolean function is known. The CDNF+ algorithm does not know the number of variables. Both algorithms only pose membership and equivalence queries. The CDNF++ algorithm does not presume the number of variables is known. It however poses membership, non-memberhip, and equivalence queries.

BULL defines the interfaces to the three types of queries. If all queries can be answered automatically, users can implement a mechanical teacher to answer queries through the interface. Learning algorithms in BULL will invoke the mechanical teacher and infer unknown target functions automatically. We refer interested users to Appendix A.2 or our web-site for a detailed demonstration of how to implement the above query functions and connect them to BULL.

2.2 Users of BULL

The BULL library targets the formal verification research community. As far as we know, several people in the field are interested in the applications of learning algorithms for Boolean functions. The library has already been used by the verification group in Oxford University (Learning-based Compositional Probabilistic Model Checking), the software trustability and verification group in Tsinghua University (Learning-Based Compositional Verification), and the static analysis group in Seoul National University (Loop Invariant Inference). Several other groups have shown their interests and asked for the source code.

2.3 Potential Applications

The CDNF algorithm has been applied to synthesize contextual assumptions in assume-guarantee reasoning. It has also been used to infer a loop invariant in program verification. These applications share common characteristics. First, computing contextual assumptions or loop invariants without learning is possible but expensive. It is however easy to verify if purported contextual assumptions or loop invariants work. Moreover, contextual assumptions or loop invariants are by no mean unique. It suffices to compute but one contextual assumption or loop invariant in these applications. From our experience, we believe that learning is most suitable for problems with the aforementioned characteristics.

For interested reader, a step-by-step tutorial of how to use the BULL library to find loop invariants is provided in Appendix A.2. We hope it may give some insights to more applications of the library.

3 Experimental Results

Since the target application of BULL is verification, in the first experiment, we decide to pick a classical example, n-bit counter, as the target for learning (Table 2). In Table 3, we show a different version where the n-bit counter model can be non-deterministically reset to 0 from any state. In the second experiment, we compare the performance of the Boolean learning algorithms using random 3SAT formulae of n variables. In those formulae, the ratio of the number of variables to the number of clauses is 1/4.[1] We use a timeout period of 10 minutes. In Figure 2, we show the average execution time of the first 50 non-trivial instances (satisfiable and all algorithms finished within the timeout period). In Table 4, we show the number of timeout cases out of 180 instances.

	2	3	4	5	6	7	8	9	10	11	12
CDNF	0.02	0.02	0.05	0.11	0.35	1.03	2.29	4.3	9.8	23.6	66.2
CDNF+	0.01	0.02	0.04	0.09	0.27	0.77	1.5	2.4	5.7	14.1	40.3
CDNF++	0.01	0.02	0.04	0.09	0.25	0.77	1.5	2.4	5.6	13.8	39.8

Table 2. Comparison of Boolean function learning algorithms: using n-bit counter as the example

At the first glance, CDNF learning algorithm has the best performance among the three. However, it is not a fair interpretation for two reasons. First, CDNF makes use of some information (number of variables in the target function) that is not known by the other two algorithms. More importantly, in particular for the case of randomly generated formulae, almost all the variables will be

[1] This ratio is very close to satisfiability threshold of 3SAT formulae. Hence the chance of getting a satisfiable formula is 50%

	2	3	4	5	6	7	8	9	10	11	12
CDNF	0.00	0.02	0.07	0.24	0.75	2.83	12.13	32.01	112	451	1374
CDNF+	0.01	0.02	0.06	0.21	0.67	2.63	12.1	36.8	144	637	1671
CDNF++	0.01	0.02	0.06	0.21	0.62	2.63	12.08	36.88	145	582	1632

Table 3. Comparison of Boolean function learning algorithms: using n-bit counter with non-deterministic reset as the example

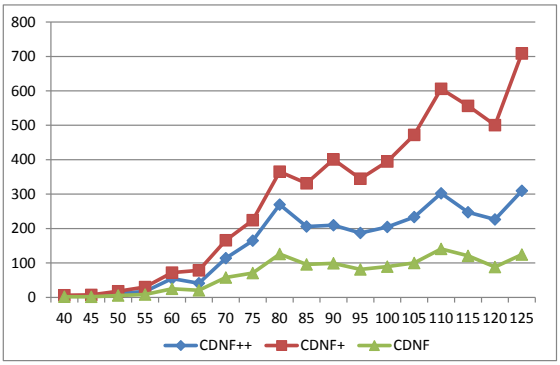

Fig. 2. Comparison of Boolean learning algorithms, using random 3SAT formulae as the benchmark. The vertical axis is the average execution time in seconds and the horizontal axis is the number of variables in the formula. Each point is the average results of 50 instances.

added to the final result. Hence the benefit obtained from incremental learning is not significant in such type of examples. In fact, the CDNF+ and CDNF++ algorithms are particularly useful in formal verification applications [4] such as those based on predicate abstraction and interpolation-based refinement. Typically in these applications, a boolean variable is used to indicate the truth of a predicate in certain points of program executions. Since the number of predicates in use would increase in each refinement step, there is no *a prior* known upper bound of needed variables.

Num. of Var.	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120	125	130	135
CDNF	0	0	0	0	0	0	1	2	0	14	7	24	28	31	48	51	70	76	90	93
CDNF+	0	0	0	0	0	0	1	6	5	21	19	42	48	51	83	80	88	99	118	122
CDNF++	0	0	0	0	0	0	2	4	6	19	16	32	40	45	69	69	82	90	106	109

Table 4. The number of timeout cases out of 180 instances

References

1. B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. libalf: The automata learning framework. In *CAV*, LNCS, pages 360–364. Springer, 2010.
2. N. H. Bshouty. Exact learning Boolean function via the monotone theory. *Information and Computation*, 123(1):146–153, 1995.
3. Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang. Automated assume-guarantee reasoning through implicit learning. In *CAV*, LNCS, pages 511–526. Springer, 2010.
4. Y.-F. Chen and B.-Y. Wang. Learning boolean functions increamentally. In *CAV*, LNCS, pages 55–70. Springer, 2012.
5. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.
6. P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In *ENTCS*, pages 21–36, 2005.
7. Y. Jung, S. Kong, B.-Y. Wang, and K. Yi. Deriving invariants in propositional logic by algorithmic learning, decision procedure, and predicate abstraction. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI*, LNCS, pages 180–196. Springer, 2010.
8. Y. Jung, W. Lee, B.-Y. Wang, and K. Yi. Predicate generation for learning-based quantifier-free loop invariant inference. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *LNCS*, pages 205–219. Springer, 2011.
9. O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995.
10. M. Merten, B. S. Falk Howar, S. Cassel, and B. Jonsson. Demonstrating learning of register automata. In *TACAS*, LNCS, pages 466–471. Springer, 2011.
11. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation learnlib. In *TACAS*, LNCS, pages 220–223. Springer, 2011.
12. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. Learnlib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.

A Appendix

The presentation consists of two parts. It will begin with an introduction to the contents of the library and follow by a detailed tutorial of how to use BULL to find a loop invariant for a Boolean program.

A.1 The BULL Library

The package can be obtained via SVN:
`svn co http://project-bull.googlecode.com/svn/trunk/ project-bull-read-only`
(there is a space between "`...trunk/`" and "`project-bull-read-only`"). It consists of the following folders:

Src/core
: The core library code in C.

Src/c, Src/cpp, Src/java, Src/ocaml
: Examples of C, C++, JAVA, and OCAML implementations of an oracle that learns a given boolean formula. The **Src/c** also contains an example oracle that learns a loop invariant of a given boolean program (will be described in detail in Section A.2).

Src/solvers
: SAT Solvers used by the oracles. Currently we use minisat as the default solver for C++ and OCaml, SAT4J as the default for JAVA.

dimacs
: CNF formulae that can be used as the target functions, most are taken from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`.

Compilation We use `{BULL-dir}` to denote the root folder of BULL. The easiest way to compile the package is to run the `./build` script in the folder `{BULL-dir}`. It will automatically compile the solvers, the core library, and oracles implemented in different programming languages

Execution Below we show how to run learning algorithms to learn a given boolean formula using the example oracles implemented in C, C++, OCaml, and JAVA. We use `{Exec}` to denote the location of the executables.

- For C, `{Exec} = {BULL-dir}/Src/c/learn`
- For C++, `{Exec} = {BULL-dir}/Src/cpp/learn`
- For JAVA, `{Exec} = {BULL-dir}/Src/java/learn.sh`
- For OCaml, `{Exec} = {BULL-dir}/Src/ocaml/learn.btye`

Users can specify which learning algorithm to use via input arguments.

- `{Exec} 0` : using the CDNF algorithm
- `{Exec} 1`: using the CDNF+ algorithm
- `{Exec} 2`: using the CDNF++ algorithm

The executable reads a target boolean formula (in the form of a CNF formula in DIMACS format) from standard input. For example, the command `{Exec} 0 < ../../dimacs/small.cnf` will use the CDNF learning algorithm to learn a target function that equals the formula in `small.cnf`.

Output The final output of the program is shown in Figure 3. The CDNF algorithm uses 50 membership queries and 45 equivalence queries to get the result. The result is a formula in the form of conjunction of formulae in disjunctive normal form (CDNF). Each number in the result formula denote a literal.

```
Number of Variables Used : 10
{ { { 8 & 3 } | { 8 & 4 & 2 } | { 9 & 5 & 2 } | { 9 & 5 & 3 } | { 10 & 5
& 2 } | { 8 & 4 & 1 } | { 9 & 5 & 4 & 1 } | { 10 & 5 & 4 & 1 } } & { {
-6 } | { 9 & 5 } | { 10 & 5 & 2 } | { 10 & 5 & 4 & 1 } } & { { 9 } | { 10
& 2 } | { -5 } | { 10 & 4 & 1 } } & { { -10 } | { 2 } | { 4 & 1 } } &
 { { -1 } | { 4 } | { 2 } } }
```

Fig. 3. The final output.

A.2 Use BULL to Infer Loop Invariants of a Boolean Program

Loop Invariant for a Boolean Program Given the following fragment of a Boolean program (a program with only Boolean variables): $\{pre\}$ while c do S $\{post\}$, where S is a sequence of program statements, pre is the precondition, c is the loop condition, and $post$ is the postcondition. Note that pre, c, and $post$ are encoded as propositional formulae. A loop invariant I satisfies the following conditions: (a) $pre \rightarrow I$, (b) $c \wedge I \rightarrow \text{Pre}(I, S)$, and (c) $\neg c \wedge I \rightarrow post$. In condition (b), the function $\text{Pre}(\phi, S)$ returns a *weakest precondition* function of ϕ before the execution of S. Precisely, the execution of the statement S from any satisfying assignment of $\text{Pre}(\phi, S)$ ends in a satisfying assignment of ϕ. A concrete example is given in Figure 4.

Input: bool x, y
bool t;
$\{pre : x \neq y\}$
while x **do**
  $t \leftarrow x$;
  $x \leftarrow y$;
  $y \leftarrow t$;
end
$\{post : x = \mathsf{false} \wedge y = \mathsf{true}\}$

For a loop invariant I over variables x and y, the following three conditions should hold:

1. $x \neq y \rightarrow I$
2. $x \wedge I \rightarrow Pre(I, t \leftarrow x; x \leftarrow y; y \leftarrow t)$
3. $\neg x \wedge I \rightarrow \neg x \wedge y$

Recall that the weakest precondition of I w.r.t. an assignment $x \leftarrow y$ equals $I[x \leftarrow y]$, where the notation $[x \leftarrow y]$ replaces all free occurrences of x in I with y.

Fig. 4. A sample Boolean program and its loop invariant

Below we demonstrate how to use BULL to find a proper loop invariant for the program in Figure 4. Since the number of variables in the invariant is known, here we choose the CDNF learning algorithm. The CDNF algorithm requires a teacher that answers equivalence queries and membership queries. An implementation can be found in the files `Src/c/tacas.h` and `Src/c/tacas.c`. Below we explain how the implementation is done. We first implement two functions, one for answering equivalence queries and the other for membership queries.

Equivalence Queries For equivalence queries, the teacher only needs to check if a conjecture function I satisfies the above three conditions. In case that I does not satisfy any of the above three conditions, a counterexample assignment is returned to the learning algorithm to refine the next conjecture. A pseudo code for the equivalence query can be found in Algorithm 1. The implementation can be found in the file `Src/c/tacas.c`.

Input: A boolean formula I
if $\neg(x \neq y \rightarrow I)$ *is satisfiable* **then**
  return a satisfying assignment for $\neg(x \neq y \rightarrow I)$
else if $\neg(\neg x \wedge I \rightarrow \neg x \wedge y)$ *is satisfiable* **then**
  return a satisfying assignment for $\neg(\neg x \wedge I \rightarrow \neg x \wedge y)$
else if $\neg(x \wedge I \rightarrow I[y \leftarrow t][x \leftarrow y][t \leftarrow x])$ *is satisfiable* **then**
  return a satisfying assignment for $\neg(x \wedge I \rightarrow I[y \leftarrow t][x \leftarrow y][t \leftarrow x])$
else Terminate with the answer I is a proper invariant for the loop;

Algorithm 1: Equivalence query

Membership Queries For a membership query on an assignment v, if v is an satisfying assignment of formula *pre*, v is also a satisfying assignment of a correct invariant due to the condition (a). Thus the teacher returns *yes* to the learning algorithm. On the other hand, if v is not a satisfying assignment of the formula $c \vee post$, it cannot be a satisfying assignment of a correct invariant because of the condition (c). The teacher hence should return *no*. If v does not fall in the above two cases, the teacher cannot decide whether the assignment is included in a correct invariant or not. It thus returns a yes-or-no answer randomly. A pseudo code for the membership query can be found in Algorithm 2. The implementation can be found in the file `Src/c/tacas.c`.

Input: An assignment v
if v *is a satisfying assignment for* $(x \neq y)$ **then**
  return *yes*
else if v *is not a satisfying assignment for* $x \vee (\neg x \wedge y)$ **then**
  return *no*
else Randomly answer *yes* or *no*;

Algorithm 2: Membership query

To be more concrete, taking the BULL C interface as an example. The signatures of the functions implementing the two queries are described in Figure 5 (they are defined in `Src/core/query.h`).

```
membership_result_t mymemqry (void *info, bitvector *v);
equivalence_result_t* myequqry (void *info, uscalar_t num_vars,
                                boolformula_t* c);
```

Fig. 5. Function signatures of membership and equivalence queries. The field `void *info` is used for passing additional information to the queries, e.g., when running several different instances of BULL at the same time, the field can be used to identify the difference between instances. The field `uscalar_t num_vars` is used for passing in the number of variables of the candidate boolean function. In this example, this number should be fixed to a constant 2. `membership_result_t` is a boolean value while `equivalence_result_t` contains a boolean value for the yes-or-no answer and a bitvector to store a counterexample assignment.

The Learner Once the functions for the two kinds of queries are given, we use BULL to find an invariant for the given loop. This is done by calling a function `learn` using the two query functions as input parameters. To be more specific, in the C interface, one should call the function `learn` described in Figure 6 (defined in `Src/core/cdnf.h`).

```
boolformula_t *learn (void *info, uscalar_t num_vars,
            membership_t membership, membership_t comembership,
            equivalence_t equivalence, int mode);
```

Fig. 6. Signature the learner function. The field `void *info` is used for passing additional information to the queries and `uscalar_t num_vars` is the number of variables in the target function. The fields `membership_t membership`, `membership_t comembership`, `equivalence_t equivalence` are pointers to the implemented functions for membership query, comembership query, and equivalence query, respectively. For CDNF, CDNF+, CDNF++ learning algorithms, the values of `mode` should be set to CDNF, CDNF_Plus, CDNF_PlusPlus, respectively.

The function `learn` either returns a correct answer or a value `NULL` to indicate conflict answers received during the learning process. Conflict can happen because some membership answers are given randomly, an assignment may be satisfying and unsatisfying in different queries. When such a conflicting assignment is observed, the whole procedure restarts. An implementation of this can be found in Figure 7. Since the number of variables and hence the number of queries is bounded, the simple random teacher will find a correct invariant with probability 1 should such an invariant exist.

```
int main(int argc, char *argv[]){
    int mode=CDNF;
    boolformula_t* c=NULL;
    while(c==NULL){
        c=learn (NULL, 2,mymemqry,NULL, myequqry, mode);
    }
    printf(stderr,"\nFinished!\nResult:");
    boolformula_print(c);
    boolformula_free(c);
    return 0;
}
```

Fig. 7. The main function of the C implementation.

Output The final output of the program is shown in Figure 8. Two membership queries and four equivalence queries were made before getting this invariant. We map each literal to a number, e.g., in Figure 8, x is mapped to 1 and y to 2. The notation `{ { { 2 } | { 1 } } }` denotes a formula $(((y) \vee (x)))$.

```
Number of Variables Used : 2

Finished! Number of Mem. Q. =2, Equ. Q. = 4
Result:{ { { 2 } | { 1 } } }
```

Fig. 8. The final output.

Further Extension The above can be extended to support more general programs (e.g., with integer variables) through predicate abstraction and automated predicate refinement [8]. For such applications, the number of predicates needed for verification is unknown *a priori*. Algorithms such as CNDF+ and CDNF++ are therefore most suitable for them.