Term Project: Yet Another SAT Solver (YaSat)
Milestone I

喬波
0540017
Nov. 23th, 2016

The term project: Yet Another SAT Solver (YaSat) is to build a program to solve boolean satisfiability (SAT) problem. More specifically, the project is to implement a program that processes a set of boolean clause consisting of boolean literals and represented in conjunctive normal form (CNF), then gives the result that states whether the SAT problem is satisfiable and for the satisfiable problem, gives a vaild solution.

The simplest idea of implementation is to enumerate all possible assignment combination of the each boolean literal, while this is impractical accounting to its complexity. However, by applying some kinds of processing techniques, a lot of unnecessory attempt can be avoided, thus the complexity will become acceptable, for generall cases. This term project is to build a solver that adopts some techniques to achieve a performance as high as possible.

This project is relatively complex, and several milestones are set to simplify the control of implementation. Currently implemented version of the project is in the milestone 1, which is the center topic of this report.

This report will cover the features of the milestone 1, its basic structure, the compiling options and the usage of the program, difficulties and problems during the implementation and the plan for the next milestone.

1 Features

At the milestone 1, the most important features implemented are the boolean constriant propagation (BCP), and 2-literal watching. Since the milestone 1 is to construct the framework of the whole project, other data structure and procedures are prepared for the implementation of other features.

The implemention of DPLL is based on a queue which stores the pending unique clause. After each decision, all affected clauses are checked and push the found unique clauses into the queue. Then iteratively process the queue until the queue is empty or a confict occurs. As for braching heuristic, the current adopted strantegy is to calculate some static attribute (the frequency in small clause), and store the sorted result before the main DPLL procedure. Besides, the decision and implication of literal assignment are stored in a stack instead of recursive function invocation, which is prepared for the implementation of the non-chronological backtracking.

2 Basic Structure

This project is implemented in C++ and divides the different parts of the solver into several C++ classes. The core solving procedures are listed in the “Solver” class and the core data structures including are arranged as seperate classes including “Bool” (boolean variable with extra unassigned state), “Literal” (represent a literal in clause), “Clause” (a clause), “ClauseWatching” (the state of the two watched literals of a related clause), “LiteralMeta” (the state related to specific literal including is assigned value, list of clause whose watching literals contians current literal) and “LiteralAssignment” (the assigment of literial including its type (decision or implcation), and the related literal).

These are basic elements of the program. As for the container to arrange the set or list of them, in this implementation, the contianers in Standard Template Library of C++ (STL) is used. In order to optimize the memory usage, most elements contains only one copy while store the pointer in other container. Since this requires that the original copy of the items are cannot be moved, some other data structure may requires to implement in later usage when the containers expanding their size and invalidate the pointers.

This program can also print detailed message about the execution state, which can be used to trace the behavior of this program during debugging process. Some color outputs are implemented to improve the readability of the execution messages. These debugging functions be opt on or off during compiling time.

Besides the mian program, to simplify the process of test, a small utility named “yasat-veri” is implemented to verify the solution given by the solver. This is integrated into the test target in the Makefile of project.

3 Compiling Options and Usage

At milestone 1, most features are implemented without alternations, and command line interface is not fully finished to tweak the various configurations of the core procedures. In this section, only some basic items are listed. In the "README.md" file in the submission package, ths usage of the compiling options and usages are also listed.

3.1 Compiling options

This program is tesed in g++ 5.4.0 environment [1] with "-std=c++11" option enabled (other options are not listed, which can refer to the Makefile). The compiling is controlled via Makefile, and tested "make" utility is GNU make [2]. The Makefile can not only compile the program, but also execution the test script. The test is done by use the SAT problem in the benchmark folder of the project. The test SAT problem are categorized into three groups, which are the sanity test, the tiny test, and the crafted test. The SAT problem in the first two groups are those problems that provided in the requirement, while the third group includes some hard and crafted problems from SAT Competition's benchmarks [3]. Currently, the solver cannot solve the problems in the third group in an acceptable time, while solving them is one of the goals to achieve in this implementation. Apart from the tests, the main function of the Makefile is setting up the sequence and options of each compiling command.

The usage of the Makefile of this project can be summarized as following.

```
# cd to the projec directory before compiling

# take defualt configuration
make

# set the option to non-empty value to enable
make FLAGS_DEBUG= FLAGS_COLOR=1 FLAGS_VERBOSE=

# test the program with diffenent set of problem
make test-sanity
make test-tiny
make test-crafted

# use both the sanity and tiny problem
make test
```

Currently, there are only 3 options in compiling staging and all of them are implemented by setting macro definition in source code. The options can be enabled by passing a non-empty value and disabled by passing an empty value. The functions of these macros are as listed below.

- **FLAGS_DEBUG**: toggle the debug mode. When enabled, the debug informations will be embedded into the binary (via "-ggdb3" option) and will not enable higher level compilier optimization (via "-O2", "-O3"). Disabled by default.
- **FLAGS_COLOR**: toggle the color output. When enabled, the output message will be formatted with color in supported teriminals. However, when redirect the message to file for later check, the ANSI control sequence will be seen. In this scenario, diable the option or read output file via "cat" command. Enabled by default.
- **FLAGS_VERBOSE**: toggle the verbose mode. When enabled, the program will print detailed information about the execution procedure. This can dramatically decrease the performance of the problem and transform it from CPU bounded application to IO bounded application. This is useful to find out the source of error during debugging. Disabled by default.

3.2 Executable Usage

Owing to the limited number of features implemented at milestore one, the mere messsages which requires to be specified are the input SAT problem and output solution. The input can be either a file or standard input from piped output of other program or mamual input, the latter can be handy to perform some test. The output can be a file or standard output, similarly. The usage of the problem can be summarized as following.

```
# specify the input and output
./yasat [in-file|--stdin] [out-file|--stdout]

# print help message
./yasat --help
```

Currently, when use the "--stdout" option to print to result to screen, the message will still generate and will write to "message.log" file in the work directory during execution. And parsing warning and error message will also be printed to standard output.

4 Difficulties and Problems

As the first milestone which setting up the framework of the project, initially, I had no idea about the implementation of those great ideas introduced in class, which made me keeping postpone the project and led to the overdue milestone. For me, the most tough point of this milestone is how to arrange all of or some of those important ideas together. After several simulation of the whole procedure by hand while managing to perform the operations which are directed supported by the computer, I gradually found some imcomplete structures and finally achieved the milestone 1.

Another tough problem is to trace the solving process to find the exactly point that some error occurs. Since this program contians lots of execution branch and the error can be cause by either the "Solver" procedure or the basic data structures that have not been fully tested, especially the data structure to store current watched literals. This is solved by

[1] on Ubuntu 16.04.1 platform
[2] may fail to list all of the source in other implementations of make utility
[3] refer to: http://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=downloads

printing some clear message of each operation and comparing the output of the problem step by step with the step of manual manipulation of the same problem with the same strantegy. However, this step is a great exhaustion of time and patience.

5 Next Milestone

As the first milestone of this project, only limited number of fearures are implemented. In the next milestone, the planned features including “1UIP based conflict driven clause leraning and non-chronological backtracking” as well as the “random restart”. Besides these features, other planned improvement including extendsion to the command line options to control more detialed features, reimplementation of the output of debugging message for control based on levels, gathering statistical information of the solver to monitor the performance.