Practice/Real-Life Applications of Computational Algorithms

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

喬波
0540017
Dec. 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 2, which is the central topic of this report.

This report will cover the new features added in the milestone 2, changes to its basic structure, the compiling options and the usage of the program, difficulties and problems during this stage of implementation and the plan for the next milestone.

1 New Features

At the milestone 2, the work is mainly about the conflict driven clause learning and non-chonological backtracking.

In milestone 1, the basic BCP and 2 literal watching is implemented. In this milestone, the conflict driven clause learning (CDCL) and non-chonological backtracking is implemented by adopting the First Unique Implication Point (FirstUIP) method. Since the clause learning method introduced new clauses into the existing clause database, the original data structure based on array with fixed size [1] is also changed. Besides, the Variable State Independent Decaying Sum (VSIDS) is also adopted in this milestone to provide more heuristic on branching from locality. However, for some test cases provided, the VSIDS leads to a worse performance, while the improvement of performance gains from other test cases is not significant. Hence, this new feature is disabled by default. In the detail of implementation, this VSIDS also contains bias on the shorter clause, with adopted the same heuristic in milestone 1.

2 Changes to Basic Structure

Most of the basic data structures remain unmodified since the last milestone, while the solver state related data structures are changed to supprt the dynamic clause database. In last milestone, clauses, assignment, literal watching status are all stored in fixed sized array. In this milestone, they are rebased on vector of pointer linking to dynamic allocated memory. Since the memory address are always the same regardless of the resize of the container, all of the data are used as pointer or iterator without dedundent copies. However, this implementation will cause more fragmented memory usage and increase the cache miss rate, which lead to some performance drop for small test cases used in milestone 1.

For the new features, a dedicated clause list in intruduced to manage different types of clause seperately. This clause list is also prepared for further restriction and elimation of clause, which is planed to implement in next milestone.

Some statistical counter is embedded into the solver to monitor the execution status, which can be toggled via compiling options.

Apart from the main solver program, another notable change is make to the parser to improve the compatibility for data files.

3 Compiling Options and Usage

This section list some new options introduced in milestone 2. In the "README.md" file in the submission pack-

[1] implemented with vector while depending on pointer to inner entry, which will be invalidate when the vector is resized.

age, a more complete list of usage of the compiling options and executabe is presented.

3.1 Compiling options

This program is tesed in g++ 5.4.0 environment [2] with "-std=c++14" 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 [3]. 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 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=                  \
     FLAGS_PRINT_STATIS=1            \
     FLAGS_LITERAL_WEIGHT_DECAY=1    \
     FLAGS_LITERAL_WEIGHT_UPDATE=1

# test the program with diffenent set of problem
make test-sanity
make test-tiny
make test-m2
make test-m2-hard

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

The detail of the compiling options are listed as follows:

- **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 "-O3", etc.). 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.

- **FLAGS_PRINT_STATIS**: toggle the print of statistic data. When enabled, the program will print some statistic data of execution state every 3000 descstion. Currently, the program will print the number of decision, the number of implication, the number of conflict, the number of current learnt clause, and the maxium level of backtracking.

- **FLAGS_LITERAL_WEIGHT_DECAY**: toggle the literal weight decaying mode. When enabled, the weight of literal will decay as the excution of the solver. Current decaying strantegy is multiple $\frac{3}{4}$ every 200 decision. This option is independent with the following weight update option.

- **FLAGS_LITERAL_WEIGHT_UPDATE**: toggle the literal weight update mode. When enabled, the weight of literal will be updated when new clause (the learnt conflict clause) is add into database.

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. To fully disable file writing, a feasible approach is redirecting the output to "null" virtual device.

4 Difficulties and Problems

Based on the first milestone, this milestone is mainly to implement some important "great ideas" including the CDCL and VSIDS. As the growth of the code amount, it becomes harder and harder to trace and resolve misbehaviored program when some special error can only be represented at large and complex test cases. Over 70% of the time is spent on debugging. Another problem is that the newly introduced features does not always direct to the improvement of performance.

For the given test case, there are still two of then cannot be solved within acceptable time, which may require some other techniques to resolve.

[2] on Ubuntu 16.04.1 platform

[3] may fail to list all of the source in other implementations of make utility, such as the default one on some BSD platform.

5 Next Milestone

For the milestone 3, some features are planed to implement including “random restart” and “preprocessing”. Some other plan for the next milestone including optimization memory usage or allocation and add some application of SAT problem by encoding some practical problem.